PB152 Operační systémy

Petr Ročkai

Část A: Organizace

1 Prerekvizity Tento kurz předpokládá jen minimum znalostí: spoléhá zejména na základní znalost fungování počítače, tak jak byla probrána v předmětech PB150 nebo PB151. Také je potřeba, abyste měli představu, jak počítače vykonávají programy, a tedy byste měli mít nějakou elementární zkušenost s programováním. Překvapivě důležitým požadavkem tohoto předmětu je schopnost soustředit se na psaný text: jak při studiu těchto poznámek, tak u zkoušky.

- princip fungování počítače (PB150, PB151)
- základy programování (IB111, IB113)
- porozumění psanému textu

2 Studijní materiály Operační systémy jsou zásadním stavebním prvkem výpočetní techniky, a programátoři se jimi zabývají už řadu desetiletí. Tomu odpovídá množství literatury, která na toto téma existuje.

Operační systémy plní mnoho úloh, a každá z nich představuje poměrně rozsáhlý obor studia. V tomto kurzu se budeme zabývat pouze těmi nejdůležitějšími prvky, a to navíc v relativně omezené míře: hlubší znalosti můžete získat studiem literatury, a zejména analýzou existujících operačních systémů. Existuje řada systémů, ke kterým jsou dostupné zdrojové kódy, které můžete jednak studovat, jednak upravovat a zkoumat jak se taková změna projeví. Znalosti takto získané lehce předčí vše, co se vůbec lze naučit čistě teoretickou cestou (ať už v tomto předmětu, nebo studiem odborné literatury).

Knihy zde odkazované mají svoji vlastní strukturu, a jejich objem dalece převyšuje limity semestrálního kurzu. Přesto Vám doporučujeme si je prolistovat a relevantní sekce přečíst – abyste probíraným tématům skutečně porozuměli, je důležité prozkoumat je z více úhlů.

- poznámky ve studijních materiálech v ISu
- literatura
  - Modern Operating Systems (A. Tanenbaum)
  - OSTEP (Remzi & Andrea Arpaci-Dusseau)
  - dlouhá řada knih a online zdrojů

3 Ukončení Tento předmět je ukončen **zápočtem**. Máte-li studijním programem předepsánu zkoušku, tato je vedena pod samostatným kódem PB152zk (musíte si ji tedy zapsat na začátku semestru). Předměty jsou hodnoceny zcela nezávisle (hodnocení PB152zk se do tohoto předmětu **nepřenáší** – musíte složit zápočtový test).

Forma zkoušky a forma zápočtového testu bude stejná, liší se ve dvou parametrech: obtížnost otázek a bodové hranice. V obou případech dostanete 12 otázek (každá otázka odpovídá jedné kapitole, tzn. jedné přednášce) a každá otázka bude sestávat z 5 tvrzení, ze kterých právě 2 jsou pravdivá.

Tvrzení u zápočtu budou jednoduchá faktická tvrzení, např. „virtualizace paměti je na úrovni procesoru realizována stránkovací jednotkou“ nebo „stránkové tabulky, a tedy virtuální paměť, si spravuje každý proces ve vlastní režii“. V případě zkoušky se bude jednat o komplikovanější tvrzení rozsahu jednoho odstavce textu.

Termíny budou společné pro zápočet i zkoušku, můžete tak obojí absolvovat zároveň. Na zápočet bude 30 minut, na zkoušku 2.5 hodiny.

- PB152 zápočtem; zkouška → PB152zk (nezávisle)
- v obou případech 12 otázek po 5 tvrzeních
  - právě 2 tvrzení jsou pravdivá
  - vyberete 2: +4/0/-1 (2/1/0 správně)
  - vyberete 1: +1/-1 (správně/špatně)
  - vyberete jiný počet: 0
- zápočet: 32 bodů
- zkouška: A ≥ 44, B ≥ 40, C ≥ 36, D ≥ 32, E ≥ 28

4 Seminář Operační systémy mají samozřejmě i praktickou stránku: víceméně každý program, který kdy napíšete (nebo budete používat), bude fungovat v kontextu operačního systému a naprostá většina těchto programů s ním bude nějak interagovat. Jako ve všech oblastech programování, zvládnutí interakce s OS vyžaduje určitý cvik. Seminář z operačních systémů Vám dá příležitost ho získat.

Protože přímá interakce s operačním systémem je ve většině případů realizována na úrovni jazyka C, bude většina semináře probíhat právě formou programování v C. Je proto důležité, abyste si tento jazyk osvojili (například v kurzu PB071).

- praktická interakce s OS
- v podzimním (dalším) semestru
- předpokládá znalosti z této přednášky
- předpokládá znalost jazyka C

1. virtualizace výpočetních zdrojů
2. souběžnost
3. vnitřní struktura

5 Přehled semestru Kurz je rozdělen na 3 velké tematické celky. První měsíc se budeme zabývat virtualizací (zejména) výpočetních zdrojů: procesoru, paměti, perzistentních datových úložišť a konečně dalších periferií (zejména síťového a grafického hardware).
Druhý čtyřtýdenní blok bude zkoumat souběžnost, která je pro operační systémy velmi důležitá, a s ní související témata: synchronizace, paralelizmus, latence, komunikace a přerušení.
V posledním bloku nahlédneme trochu více „pod pokličku" – z úrovně obecných abstrakcí se přesuneme na konkrétní implementační otázky. Bude nás zajímat programovací rozhraní (API), komunikace s vnějším světem, architektura jádra a nakonec se budeme stručně zabývat virtualizací operačního systému.

Část B: Základní pojmy a definice

Tato kapitola předchází prvnímu bloku a zavádí pojmy a koncepty, které budeme potřebovat během celého semestru. Bude-li některý hrát v pozdější přednášce obzvláště důležitou roli, jeho definici si připomeneme. Jinak ale budeme v dalším předpokládat, že tyto pojmy znáte.

B.1: Abstrakce

- umožňuje stavbu rozsáhlých systémů
- založena na skrývání detailů
- známé vnější chování
- neznámá vnitřní struktura
- příklad: cihly (na stavbu zdí)

B.1.1 Co je abstrakce? Je to centrální koncept (a to nejen pro operační systémy), který nám zejména umožňuje stavět složité a rozsáhlé systémy z jednodušších stavebních kamenů. Abstrakce je založena na **skrývání detailů** – na to, abychom mohli s nějakým prvkem pracovat, potřebujeme znát jeho **vnější chování**, ale ideálně nemusíme znát jeho vnitřní strukturu. Je-li toto vnější chování výrazně jednodušší, než jeho vnitřní realizace, abstrakce nám znatelně zjednoduší přemýšlení o **kompozici** několika takových abstrakcí.

> **Příklad**: Potřebujeme postavit zeď. Na to se nám budou hodit cihly (a malta). Vnější chování cihly je jednoduché: je to přibližně kvádr pevných rozměrů, který má nějakou minimální pevnost v tlaku. Vnitřně je cihla překvapivě složitá: cihla se vyrábí z vhodné hlíny vypálením za správných podmínek. Hlínu je potřeba vytěžit, zpracovat, namíchat s případnými aditivy, vylisovat, nařezat, vysušit a vypálit. Cihla navíc nemusí být nutně pálená, může být vyrobena jiným procesem, má-li potřebné vnější vlastnosti, a třeba ani nemusí být vyrobena z hlíny.
> Nic z toho nás při stavění zdi ale nemusí moc trápit. Zeď vznikne vhodným složením a slepením cihel a není až tak důležité, jak přesně cihla vznikla, jaká přesně byla použita hlína a při jaké teplotě nebo jak dlouho jsme ji vypalovali. Cihla je pro zedníka **abstrakcí** nějakého vnitřně složitého a variabilního objektu.
> Při stavbě zdi se musí nicméně dbát na správné provázání jednotlivých cihel, na svislost a rovinnost zdi, atp. Z pohledu architekta (nebo obyvatele) ale není příliš důležité, jak přesně zeď vznikla, jaká se použila malta, jestli je z plných cihel nebo dutých, pálených nebo nějakých jiných. Zeď je další **abstrakce**: je rovná, svislá, má nějakou nosnost, izolační schopnost, otvory na okna a dveře a můžeme si na ni třeba zavěsit obraz.[1]

- základní jednotka abstrakce software
- odděluje rozhraní od implementace
- modul lze levně vyměnit za jiný podobný

B.1.2 Modul Nejběžnější abstrakcí (se kterou se budeme setkávat na každém kroku) je softwarový **modul**,[2] který má **rozhraní** (vnější chování) a **implementaci** (vnitřní struktura, mechanismus). O systému říkáme, že je modulární, je-li vystavěn z vhodných abstrakcí: jednotlivé části ve své **implementaci** spoléhají pouze na **vnější rozhraní** ostatních částí. Modularita nám umožňuje jednoduše vzít některý modul a vyměnit jej za jiný modul, s jinou vnitřní strukturou, ale se stejným vnějším chováním.

> **Příklad**: Změníme-li dodavatele cihel během probíhající stavby, pravděpodobně se nic moc nestane, i přesto, že nový dodavatel těží hlínu jinde (a ta má tedy trochu jiné složení) a třeba pak vypaluje cihly při jiné teplotě. Díky abstrakci (modularitě) jsou staré a nové cihly záměnné.

Porušení modularity (a tedy **principu abstrakce**) má za následek opačný efekt: potřebujeme-li vyměnit nějaký modul ve větším celku, je nutné zároveň upravit řadu dalších modulů tak, aby s náhradou pracovaly správně. To může mít kaskádový efekt na další a další moduly.

---

[1] Jakmile se pokusíme do zdi vyvrtat díru, abychom onen obraz mohli zavěsit, zjistíme, že abstrakce zdi není zcela dokonalá – abstrakce tzv. **propouští** (leaks) – některé vnitřní detaily se projevují i navenek, jsou abstrakcí propuštěny do vnějšího chování. Je-li například zeď vyrobena z klasických cihel a omítnuta, může se stát, že vyvrtáme díru mezi cihlami a hmoždinka nám nebude ve zdi držet. Navíc různé „implementace" zdí vyžadují různé typy hmoždinek, atp.

[2] Není náhoda, že moduly se často přirovnávají k cihlám. Někdy se jim dokonce přímo říká „stavební kameny" nebo „building blocks".

B.2: Zdroje

B.2.1 Počítač Počítač je složitý elektronický stroj – **vnitřní detaily** jeho fungování jsou daleko mimo náš současný obzor. Díky **abstrakci** se ale můžeme na počítač dívat jako na relativně jednoduché **výpočetní zařízení**, které má jednotné **vnější chování** – bez ohledu na to, kdo vyrobil paměťové moduly nebo procesor, jaké tento obsahuje tranzistory, jaké je chemické složení polovodičů v těchto tranzistorech, atd.

- von Neumanova architektura
- výpočetní jednotka provádí instrukce
- operační paměť ukládá data
- paměť je adresovatelná

Obecně používanou abstrakcí pro počítač je tzv. **von Neumanova architektura**[3], která má několik modulů, které lze rozdělit do dvou kategorií. Tou první jsou **základní výpočetní zdroje**, totiž:

- **výpočetní jednotka** (procesor, CPU, procesorové jádro, atp.) je zařízení, které vykonává **instrukce** (elementární příkazy, pomocí kterých počítač programujeme),
- **operační paměť** (obvykle jednoduše „paměť“, případně RAM, pracovní paměť) je zařízení, které si **pamatuje data** (čísla), se kterými program při svém výpočtu pracuje, a které je schopno tato data předávat výpočetní jednotce a na její pokyn je měnit.

Instrukce, které výpočetní jednotka vykonává, jsou uloženy, podobně jako pracovní data programu, v operační paměti. Operační paměť je **adresovaná**: to znamená, že je složena z **očíslovaných buněk**, kde každá buňka si pamatuje jeden **bajt** (číslo v rozsahu 0 až 255). Buňky jsou očíslovány po sobě jdoucími celými čísly, obvykle počítáno od nuly. Instrukce mohou vyžádat načtení čísla z libovolně vypočtené adresy[4] – tím se paměť liší od **registrů**, které mají pevná jména (program nemůže „spočítat“ které dva registry má sečíst). Jedná se o podobný rozdíl (opět nikoliv náhodou), jako mezi celočíselnou **proměnnou** a **seznamem**.[5]

Pro jednoduchost budeme program, který je ve von Neumannově modelu uložen v operační paměti stejně jako data, považovat za neměnný. Zejména tedy neuvažujeme programy, které do paměti zapisují a pak spouštějí nové instrukce.

B.2.2 Periferie Mohlo by se zdát, že programy provádí řadu jiných činností, které nejsou výpočetní: interagují s uživatelem, kreslí obrázky, posílají data po síti, atp. Ve skutečnosti jsou to ale všechno pouze skryté výpočty (schované za důmyslnou abstrakcí) – vstup z klávesnice je řada čísel, signál pro monitor je řada čísel, signál na síťovém rozhraní je řada čísel. Zařízení, která posílají resp. přijímají data (v podobě sekvencí čísel), a převádí je na nějaký viditelný efekt ve fyzickém světě, budeme v naší abstrakci označovat za **periferie** a blíže si je popíšeme v příslušných kapitolách.

- periferie přijímají a odesílají data
- data pro periferie jsou výsledkem výpočtu
- data od periferie jsou vstup pro výpočet
- rozšíření výpočtu o synchronizaci

V této chvíli je důležité, že veškerá činnost programu skutečně sestává z manipulace s čísly – z výpočtů. **Program** tedy můžeme také považovat za **abstrakci**: je to sekvence instrukcí, které popisují nějaký výpočet. Ve druhém bloku pak k tomuto chápání programu ještě přidáme **synchronizaci** – interakci výpočtu s periferiemi, nebo s jinými souběžně probíhajícími výpočty.

B.2.3 Virtualizace Jeden takto zavedený von Neumanův **počítač** vykonává právě jeden **program**. To představuje určitý problém: je žádoucí, aby jeden fyzický počítač mohl (alespoň zdánlivě) provádět několik různých programů zároveň. Zde do hry vstupuje **operační systém**, který fyzické výpočetní zdroje **virtualizuje** a umožňuje tak každému programu pracovat, jako kdyby měl svůj vlastní počítač.

- jeden počítač = jeden program
- řešení → virtualizace zdrojů
- každý program vidí „vlastní“ počítač
- realizuje OS + hardware

Úkolem systému virtualizace je tedy vytvořit několik nezávislých **virtuálních kopií** jednoho fyzického zařízení. Toto je umožněno **abstrakcí**:

1. **fyzické** výpočetní zdroje, které poskytuje fyzický počítač, a
2. **virtuální** výpočetní zdroje, které poskytuje systém virtualizace,

jsou z pohledu programu záměnné (mají stejné **vnější chování**, které přibližně odpovídá výše popsanému von Neumanovu počítači), i když je jejich vnitřní realizace odlišná.

Virtualizace je realizována částečně hardwarově (procesorem) a částečně softwarově (operačním systémem). Virtualizací základních výpočetních zdrojů se budeme blíže zabývat v prvních dvou kapitolách prvního bloku. Zbytek bloku se pak bude zabývat virtualizací periferií.

---

[3] Existují i jiné abstrakce počítače, tato je ale nejběžnější, pro naše potřeby v této chvíli dostatečná, a také celkem dobře odpovídá fungování skutečných počítačů.

[4] Je obvyklé, že adresy sice označují jednotlivé bajty paměti, ale procesor pracuje s většími celky – **slovy**. Slova jsou v paměti uložena po bajtech, a narážíme tak na první skulinku v této jinak zatím docela pěkné abstrakci: není určeno, jak přesně se větší číslo zakóduje do sekvence menších (zejména není jasné, v jakém pořadí). Skutečně, různé počítače tento problém řeší různě.

[5] Viz také třetí kapitolu sbírky IB111 Základy programování.

B.3: Souběžnost

Předmětem druhého bloku bude časově závislá interakce výpočtu s jinými výpočty (prováděnými ve stejnou dobu) a s periferiemi. Budou nás zejména zajímat případy, které nelze chápat jako jednoduché jednosměrné předávání dat.

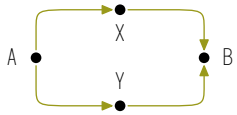
- událost = jev v čase
- obecné uspořádání událostí → předcházení
- neuspořádané události jsou souběžné

B.3.1 Událost a předcházení Klíčovým pojmem zde bude **událost** – jev, který nastane v čase (ne nutně pevně určeném nebo známém), který můžeme pozorovat, a o kterém můžeme říct, že nastal před nebo po nějaké jiné události, případně že s ní nastal souběžně. Relaci uspořádání, která tuto chronologii popisuje, budeme říkat **předcházení** (anglicky „happens before“). O dvou různých událostech, nazvaných třeba A a B, tedy může platit právě jedna z těchto možností (plyne z vlastností uspořádání):

1. A předchází B (tzn. A nastalo první),
2. B předchází A (tzn. B nastalo první),
3. A nepředchází B ani B nepředchází A – události jsou souběžné.

Opět se zde jedná o **abstrakci**: tentokrát skrýváme vnitřní detaily procesů (dějů odehrávajících se v čase), které se mohou stát v různém pořadí díky náhodným vlivům, a snažíme se jejich **vnější chování** popsat pomocí zmiňované relace. Vnější chování nějakého modulu závisí pouze na jeho relaci předcházení, nikoliv už na tom, jak přesně budou v čase rozloženy konkrétní události.

Příklad: Uvažme množinu událostí $U = \{A, B, X, Y\}$ na kterých je zavedena relace předcházení $P = \{(A, X), (X, B), (A, Y), (Y, B)\}$:

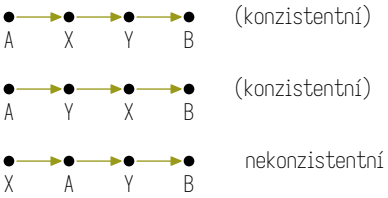


Události X a Y jsou **souběžné**, mezi ostatními dvojicemi je uspořádání zjevné.

- časový sled = lineární uspořádání událostí
- lze chápat jako přidělení časových razítek
- kompatibilita s relací předcházení
- odlišná kompatibilní chování → hazard souběhu

B.3.2 Časový sled, hazard souběhu **Časovým sledem** událostí rozumíme **lineární** uspořádání událostí (tedy pro **každou** dvojici A, B událostí platí buď A předchází B nebo B předchází A). Časový sled si můžeme představit i jako přiřazení **časového razítka** každé události takové, že žádné dvě události nenastanou ve stejné chvíli.

Příklad: Na výše uvedené množině U můžeme uvažovat tyto dva časové sledy, které jsou oba konzistentní s relací P – je zřejmé, že na pořadí X a Y nezáleží. Naopak třetí uvedený časový sled s relací P konzistentní **není**:



Je-li tato abstrakce porušena – tedy **vnější** chování systému se liší ve dvou různých časových sledech, které jsou ale oba konzistentní s požadovanou relací předcházení – mluvíme o tzv. **hazardu souběhu**, **race hazard** nebo nejčastěji **race condition**[6].

B.4: Operační systém

1. umožňuje běh aplikací
2. přístup k HW a SW zdrojům
3. běhová podpora
4. současný běh několika programů
5. izolace programů a uživatelů

B.4.1 Vnější rozhraní Nikoho asi nepřekvapí, že samotný pojem **operační systém** je abstrakcí. Operačních systémů existuje celá řada a svým vnitřním fungováním jsou značně rozmanité. Společným jmenovatelem všech operačních systémů jsou tyto tři pilíře:

1. umožňují běh uživatelských programů,
2. zprostředkují jim přístup k hardwarovým a softwarovým zdrojům,
3. poskytují jim běhovou podporu a přidružené služby.

[6] Česky někdy označované také jako **souběh** – my tento pojem používat nebudeme, protože by mohlo lehce dojít k záměně pojmů **souběh** a **souběžnost**, přičemž se jedná o zcela odlišné jevy.

V tomto kurzu se budeme prakticky výlučně zabývat tzv. **víceúlohovými** a **víceuživatelskými** operačními systémy, které navíc:

4. umožňují současný běh několika uživatelských programů,
5. poskytují **izolaci** současně spuštěných programů tak, aby žádný nemohl narušovat běh jiného.

Zejména body 4 a 5 jsou realizovány již zmiňovanou **virtualizací** výpočetních (a jiných) zdrojů.

B.4.2 POSIX Pro praktické účely je ale abstrakce „víceúlohový, víceuživatelský operační systém" stále velmi hrubá (popisuje vnější chování takového systému jen ve hrubých rysech). Naštěstí pro tento typ systémů existuje mnohem přesnější a zároveň obecně uznávaná abstrakce, standardizovaná pod názvem POSIX (Portable Operating System Interface) organizací The Open Group.
Stěžejní částí standardu POSIX je tzv. API (Application Programming Interface) popsané jako řada funkcí jazyka C. Toto rozhraní umožňuje uživatelským programům využívat řadu služeb, které operační systémy běžně poskytují.
Rozhraní předepsaného standardem POSIX se sice nedrží všechny víceuživatelské operační systémy,[7] ale je přesto široce používané. Standardem POSIX a rozhraním, které poskytuje uživatelským programům, se budeme zabývat v deváté kapitole.
Konečně, přestože lze velké části operačního systému úspěšně „schovat" za abstrakci (např. zde zmiňovaný POSIX), je důležité znát i jeho vnitřní mechanismy, resp. obecné principy a vzory, na kterých staví.

- mnohem obsáhlejší rozhraní
- standardizované nezávislou organizací
- popisuje zejména C API
- široká podpora v operačních systémech

B.4.3 Jádro Běžný návrh operačního systému sestává z **jádra** a **uživatelského prostoru**. V tomto modelu má jádro **privilegovaný přístup** k výpočetním zdrojům, zejména procesoru. Je mu tak umožněno řídit hardwarové prostředky virtualizace – přidělovat paměť, procesor atd. ostatním součástem systému a zejména uživatelským programům. Zároveň tím jádro dostává roli strážce integrity: to, co jádro nepovolí, není uživatelským procesům umožněno.
Přestože operační systémy bez jádra (případně s jádrem, které není jasně odděleno od zbytku systému) existují, jsou natolik vzácné, že nemá velký smysl se jimi v základním kurzu blíže zabývat. Blíže se implementací a vnitřní strukturou jádra operačního systému budeme zabývat v jedenácté kapitole.

- jádro vs uživatelský prostor
- privilegovaný přístup k hardwaru
- realizuje a řídí virtualizaci zdrojů
- systémy bez jádra → mimo záběr

Poznámka: Protože slovní spojení „víceúlohový, víceuživatelský operační systém" je značně neohrabané, a protože jinými systémy se budeme zabývat jen zcela okrajově, v dalším budeme spojením „operační systém" vždy myslet systém tohoto typu, nebude-li uvedeno jinak. Tyto operační systémy mají prakticky bez výjimky architekturu s jasně vyčleněným jádrem.[8]

B.4.4 Virtualizace OS Jak jsme již naznačili, virtualizace zdrojů je jednou z hlavních úloh operačního systému. Přesto má smysl uvažovat také o virtualizaci operačního systému jako celku, a to hned ze dvou důvodů (tyto důvody spolu pravda úzce souvisí):

1. Izolace – operační systém sice poskytuje izolaci procesů a uživatelů, zároveň ale poskytuje i poměrně rozsáhlé sdílené zdroje (blíže se na ně podíváme v pozdějších kapitolách, počínaje tou třetí). Tyto představují určité riziko vzájemné „kontaminace" různých úloh – je tedy žádoucí takovým úlohám vytvořit podle možnosti samostatné instance OS.
2. Zjednodušení správy – operační systém je v praxi základní jednotkou správy počítačových systémů, je proto často výhodné rozdělit úkoly mezi větší počet instancí OS (částečně souvisí i se zmiňovanou izolací). Tradičně to často znamenalo více fyzických počítačů – virtualizace OS umožňuje mezi těmito instancemi fyzický počítač sdílet. Další velkou výhodou pro správce je možnost **migrovat** (třeba i běžící) operační systém na jiný hardware.

- více instancí OS na jednom počítači
- větší míra izolace než procesy
- zjednodušení správy
- migrace, a zejména živá migrace

B.4.5 Přenositelnost O přenositelnosti lze, v kontextu operačních systémů, mluvit ve dvou kontextech:

1. přenositelnost samotného operačního systému na jiné technické vybavení (jiný procesor, jiné periferie, atp.),
2. přenositelnost aplikačního software mezi různými operačními systémy.

Oba tyto případy ale mají zároveň mnoho společného. Přenositelnost je umožněna abstrakcí: je-li technické vybavení od většiny operačního systému odděleno vhodnou abstrakcí (rozhraním),

- využití abstrakce a modularity
- přenášíme program do nového prostředí
- pro operační systém: různý HW
- pro uživatelské programy: různé OS

[7] Hlavní výjimku zde tvoří OS Windows, ale i na tomto systému existuje hned několik metod, jak alespoň nějaké míry kompatibility s tímto standardem dosáhnout.

[8] Proto se bude pojem jádra objevovat v celém kurzu jako implicitní součást operačního systému. V systémech, které jádro nemají, nebo alespoň není jasně odděleno od uživatelského prostoru, tradiční úlohy jádra plní jiné komponenty (jsou-li příslušné úkoly vůbec pro daný systém relevantní).

je mnohem snazší upravit pouze část systému, která se nachází mezi samotným hardwarem a touto abstrakční bariérou. Naopak, v případech, kdy velká část systému spoléhá na konkrétní vnitřní vlastnosti počítače, je mnohem těžší tento počítač vyměnit za jiný, s jinými vnitřními vlastnostmi.
Aplikační software je na tom podobně: operační systém poskytuje abstraktní rozhraní (více o něm si povíme ve třetím bloku), ale zároveň může aplikace využívat služeb a vlastností, které jsou specifické pro konkrétní implementaci, a které tedy nejsou skryty abstrakcí. Taková aplikace pak bez větších zásahů nebude na jiném operačním systému fungovat.

Část 1: Virtualizace paměti

Tato kapitola se bude zabývat pamětí a mechanismy, které umožňují operačnímu systému do jednoho fyzického adresního prostoru „naskládat“ více programů tak, aby si tyto vzájemně nepřekážely.

Doporučené čtení: A. Tanenbaum, H. Bos – Modern Operating Systems (4th Ed.): § 3.2.1 The Notion of an Address Space ◇ § 3.3 Virtual Memory ◇ § 3.5.4 Separate Instruction and Data Spaces ◇ § 3.5.5 Shared Pages ◇ § 3.5.7 Mapped Files ◇ § 3.6.2 Page Fault Handling ◇ § 10.4 † Memory Management in Linux.

1.1: Paměť

- speciální instrukce pro čtení/zápis
- load – načte data z adresy do registru
- store – zapíše data z registru na adresu
- adresa = celé číslo

1.1.1 Přístup k paměti Na fundamentální úrovni interaguje program s pamětí použitím dvou základních operací:

- instrukce tvaru `load reg_addr reg_data` **načte** hodnotu z paměti a uloží ji do registru; `reg_addr` je registr, ve kterém je uložena **adresa**, kterou program požaduje,
- instrukce tvaru `store reg_addr reg_data` naopak hodnotu do paměti **uloží**, a to opět na **adresu** zadanou registrem `reg_addr`.

V obou případech platí, že adresa je nějaké **číslo**. Libovolné složitější instrukce, které by procesor mohl poskytovat, lze rozdělit na sekvenci instrukcí `load`, výpočetních instrukcí, které pracují pouze s registry, a instrukcí `store`.

- množina platných adres (čísel)
- rozdělen do souvislých bloků
- jedna adresa = jeden bajt dat
- větší objekty → několik sousedních adres

1.1.2 Adresní prostor Každá adresa je **číslo**, ale ne každé číslo je platná adresa. Adresním prostorem tedy budeme nazývat zejména množinu **platných adres**, ale můžeme v tomto pojmu zahrnout i případnou další zajímavou strukturu paměti (např. to, že nějaká část adres má jiné vlastnosti než ty ostatní). Adresní prostor se obvykle skládá ze souvislých bloků, kde jednotlivé bloky obsahují mnoho po sobě jdoucích adres (řádově miliony).
Je obvyklé, že každé adrese odpovídá právě jeden bajt[9], a že pro větší celky udáváme **nejnižší** adresu, a jednotlivé bajty jsou uloženy na sousedních adresách.

Příklad: Adresní prostor se skládá ze 3 bloků:

- `0x010000` – `0x01ffff` (64 KiB),
- `0x030000` – `0x03ffff` (64 KiB),
- `0x100000` – `0xffffff` (960 KiB).

Čtyřbajtové slovo, o kterém řekneme, že je uloženo na adrese `0x010030` je ve skutečnosti uloženo na adresách `0x010030`, `0x010031`, `0x010032` a `0x010033`. Na adrese `0x50` nic uloženo být nemůže, protože se nejedná o platnou adresu. Stejně tak nemůžeme na adresu `0x01fffd` uložit čtyřbajtové slovo (poslední bajt by měl adresu `0x020000`, která podobně není platná).

- paměť dostupná programu → adresní prostor
- pracovní paměť – pro libovolné použití
- kód – instrukce které tvoří program
- zásobník – informace o aktivních podprogramech
- může být dynamický – adresy přibývají/mizí

1.1.3 Paměť programu Program má při svém běhu k dispozici nějaký adresní prostor, který může používat k ukládání dat. Krom pracovní paměti (do které si program ukládá libovolná data jak uzná za vhodné) je součástí adresního prostoru programu také **kód** – adresy, na kterých jsou uloženy instrukce, ze kterých program samotný sestává, a obvykle také hardwarový **zásobník**, který se používá k realizaci podprogramů (volání funkcí).

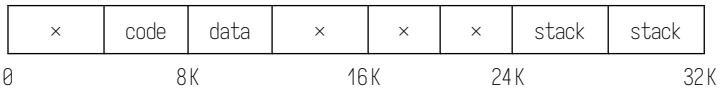
Příklad: Představme si například program, který rekurzivně počítá, jestli lze číslo 100 zapsat jako součet tří druhých mocnin. Je to jednoduchý program, který lze zapsat pomocí několika

[9] Některé exotičtější počítače používají slabiky jiné velikosti než osm bitů, např. devět. Případně vůbec nepoužívají dvojkovou soustavu, a tedy bity. Vše ostatní, zejména koncept adresního prostoru, pro ně ale platí.

desítek instrukcí, a ani nebude potřebovat příliš mnoho paměti. Jeho adresní prostor by mohl vypadat například takto:

- **kód** je uložen na adresách 0x1000 – 0x1fff,
- **data** jsou uložena na adresách 0x2000 – 0x2fff,
- **zásobník** je uložen na adresách 0x6000 - 0x7fff.

Představuje-li každá buňka 4 kilobajty a × označuje neplatné adresy, adresní prostor vypadá takto:



Nutno ještě poznamenat, že skutečné programy na moderních počítačích používají adresní prostory, které jsou **mnohem** větší. Třeba typická velikost zásobníku je 4 nebo 8 MiB.

Adresní prostor procesu může být **dynamický** – na žádost programu lze do adresního prostoru přidat nové platné adresy, nebo některé stávající odebrat. Mohou se v něm objevit také další speciální bloky, o kterých si více povíme později.

1.1.4 Fyzická paměť Fyzické adresy (tedy adresy, které patří fyzickému adresnímu prostoru) přímo pojmenovávají fyzické **paměťové buňky** hardwarových zařízení, zejména paměti RAM. Přestože se jedná o jeden adresní prostor, podobně jako v případě paměti programu může mít další strukturu. Největší a nejdůležitější blok (případně několik bloků) fyzických adres zpravidla patří operační paměti (paměti s přímým přístupem, random access memory, RAM). Toto je paměť v klasickém smyslu tohoto slova, a slouží především k ukládáni pracovních dat programů.
Fyzické adresy ale mohou odpovídat i jiným zařízením, např. grafickým nebo síťovým kartám. Do některých bloků nemusí být povoleno zapisovat (např. proto, že odpovídají nezapisovatelné – read-only – paměti). Bloky, které nepatří RAM, obvykle nelze používat pro pracovní data výpočtů.

- fyzická adresa → fyzická paměťová buňka
- většina adres odpovídá buňkám paměti RAM
  - použití zejména jako paměť programů
- některé adresy patří periferiím

Příklad: Fyzický adresní prostor klasického 32b osobní počítače s procesorem x86 vypadá přibližně takto:

od	do	účel
0000'0000	0000'04ff	RAM (IVT + BIOS)
0000'0500	0007'ffff	RAM (volné použití)
0008'0000	0009'ffff	RAM (BIOS)
000a'0000	000b'ffff	periferie (framebuffer)
000c'0000	000f'ffff	ROM (BIOS)
0010'0000	00ef'ffff	RAM (blok 2)
00f0'0000	00ff'ffff	periferie (sběrnice ISA)
0100'0000	bfff'ffff	RAM (blok 3)
c000'0000	ffff'ffff	periferie (sběrnice PCI, ...)

Toto rozložení není 100% fixní, např. oblast 00f0'0000 – 00ff'ffff může náležet také paměti RAM, čím se bloky 2 a 3 spojí do jednoho většího. Protože první tři bloky jsou všechny uloženy v RAM, lze je také chápat jako jediný blok.
Zajímavější jsou bloky 000a'0000–000b'ffff, 00f0'0000–00ff'ffff a c000'0000–ffff'ffff, které **nejsou** uloženy v RAM: patří různým periferním zařízením.

1.2: Virtualizace

1.2.1 Motivace V principu nic nebrání tomu, aby program používal k práci s pamětí přímo fyzické adresy. V takovém případě bychom program přizpůsobili tomu, které fyzické adresy jsou na našem počítači platné a odpovídají operační paměti.
Jakmile bychom ale chtěli na jednom počítači spustit několik programů najednou, začneme narážet na problémy. Programy by mezi sebou musely použití adres koordinovat: používá-li program A adresu 0x1005 pro svoji proměnnou, nemůže ji zároveň používat program B.
Dalším důležitým problémem je, že i kdybychom programy psali tak, aby si dostupné adresy nějak rozdělily, libovolná chyba (např. překročení mezí pole) by mohla lehce vést k zásahu jednoho programu do dat jiného programu. Taky by se nemuselo nutně jednat o chybu: škodlivý program by mohl poškodit nebo přečíst data jiného programu zcela záměrně.

- výpočty lze provádět i s fyzickými adresami
- více programů → problémy s koordinací
- chyby v programech
- bezpečnost a škodlivé programy

- virtuální adresní prostor
- různé prostory, stejná čísla → různý význam
- lze více virtuálních prostorů

1.2.2 Virtuální a fyzické adresy Bylo by tedy lepší, aby adresní prostor programu byl od toho fyzického nezávislý. Proto procesory určené pro běžné počítače[10] poskytují oddělený **virtuální** adresní prostor, který je viditelný pro programy, a který je od toho fyzického důsledně oddělen. Fyzický adresní prostor je uživatelským programům zcela nepřístupný.[11]

To, že se jedná o oddělené adresní prostory, zejména znamená, že sice mohou mít neprázdný průnik (mohou existovat čísla, která představují jak platnou fyzickou, tak platnou virtuální adresu), tato čísla ale mají obecně **jiný význam**.

Virtuálních adresních prostorů může existovat rovnou několik: v takovém případě o nich platí totéž: jsou to oddělené prostory a adresa z jednoho z nich nemusí v tom druhém vůbec existovat, a pokud ano, může znamenat něco jiného. Zejména může ukazovat na jinou fyzickou paměťovou buňku.

Proto budeme pojmem **virtuální adresa** označovat nejen číslo, které ji reprezentuje, ale implicitně i to, do kterého virtuálního adresního prostoru patří. Je-li tedy

- A = adresa 0x0100 ve virtuálním prostoru P,
- B = adresa 0x0100 ve virtuálním prostoru Q,

budeme o A, B uvažovat jako o **dvou různých** virtuálních adresách.

- virtuální adresa → fyzická buňka
- realizace překladem adres
- 1 buňka = 1 fyzická adresa
  - může mít několik virtuálních adres
  - nebo také žádnou
- lze použít k oddělení programů

1.2.3 Překlad adres Aby měla virtuální adresa nějaký smysl, musí být možno na ní uložit, a později vyzvednout, nějaká data. Virtuální adresa tedy musí ve skutečnosti, podobně jako ta fyzická, představovat nějakou fyzickou paměťovou buňku. A tato buňka má jistě nějakou fyzickou adresu. Tím se nabízí velmi jednoduchý mechanismus, jak virtuální adresní prostor realizovat: virtuální adresy budeme **překládat** na adresy fyzické, které již určí, kde budou příslušná data skutečně uložena.

O fyzické buňce pak můžeme říct, že má právě jednu **fyzickou** adresu (a ta je této buňce přidělena pevně) a nějaké **virtuální** adresy (to jsou ty, které se přeloží na její fyzickou adresu). Buňka může mít virtuálních adres hned několik, ale také třeba žádnou.[12]

Chceme-li pak například spustit víc oddělených programů na jednom počítači, stačí zabezpečit, aby žádná fyzická buňka neměla dvě různé **virtuální** adresy[13] (nebo alespoň aby neměla dvě virtuální adresy, které každá náleží jinému oddělenému programu).

Příklad: Uvažme dva virtuální adresní prostory, např. P a Q a jeden možný odpovídající fyzický adresní prostor F.

P	0	1	2	×	×	5	6	×	8	9	A	B
Q	0	1	×	×	×	5	6	7	×	9		
F	8	9	A	9	?	?	5	6	7	?	?	?
	?	?	?	0	1	?	B	0	1	2	?	?

- překlad adres musí být rychlý
- realizován HW (součást CPU)
- programovatelný překladovými tabulkami
- řízen jádrem OS

1.2.4 Jednotka správy paměti Přístup do paměti (čtení, zápis) je časově kritickou operací – procesor jich provádí miliony za vteřinu. Přitom adresní operand instrukce, která takový přístup realizuje, představuje vždy **virtuální adresu**, která musí být ještě před samotným přístupem přeložena na adresu fyzickou.

Překlad adres proto musí být velmi rychlý, a není nijak překvapivé, že je realizován specializovaným hardwarem, který je přímo součástí procesoru. Zároveň ale potřebujeme zabezpečit, aby o konkrétní podobě překladu mohl rozhodovat operační systém. Proto musí být tato tzv. jednotka správy paměti **programovatelná** – pomocí vhodně sestavených tabulek může operační systém (konkrétně jeho **jádro**) řídit mapování virtuálních adres na fyzické, a tedy i to, jak budou vypadat jednotlivé virtuální adresní prostory.

[10] Prozatím nemáme k dispozici přesnou definici uživatelského programu. Budeme tedy prozatím uvažovat „běžné“ programy, které může spustit libovolný uživatel bez speciálních privilegií.

[11] Netýká se např. výpočetních jader jednočipových počítačů, které často nedisponují jednotkou správy paměti a aplikace pro ně určené používají přímo fyzické adresy.

[12] Pozor, adresa není vlastnost buňky – ani fyzická, a už vůbec ne virtuální – o nic víc, než je číslo popisné vlastnost budovy.

[13] Připomínáme, že virtuální adresy náležící různým adresním prostorům považujeme za různé, i když jsou reprezentované stejným číslem.

1.2.5 Stránky Víme už, že překlad adres je řízen překladovými tabulkami. Pochopitelně tyto tabulky nebudou mít samostatný řádek pro každou virtuální adresu: taková tabulka by byla neprakticky velká. Virtuální adresy proto seskupujeme do tzv. **stránek** pevné velikosti (konkrétní velikost stránky záleží na konkrétním hardwaru, ale je zvykem, že je to mocnina dvou).
Stránka je pak základní jednotkou překladu. Tím se jednak zmenší potřebné tabulky, jednak se tím zjednoduší proces překladu. Budeme-li totiž požadovat, aby:

1. stránka obsahovala 2^n adres (tj. měla velikost 2^n bajtů),
2. stránka začínala adresou, která je beze zbytku dělitelná 2^n,
3. byla mapována na fyzickou adresu, která je také beze zbytku dělitelná 2^n,

můžeme spodních n bitů virtuální adresy přímo použít jako spodních n bitů adresy fyzické. Zbytek fyzické adresy již dopočítáme podle překladové tabulky. To má mimo jiné za důsledek, že stránky se nemohou překrývat.

- blok virtuálních adres pevné velikosti
- 1 stránka = 1 řádek překladové tabulky
- začátek dělitelný velikostí
- stránky se nepřekrývají
- posledních n bitů mapovaných přímo

Příklad: uvažujme stránku velikosti 4 KiB (celkem běžná velikost), která je tvořena $4096 = 2^{12}$ po sobě jdoucími virtuálními adresami. Máme tedy $n = 12$ a při překladu spodních 12 bitů pouze opíšeme z virtuální adresy do fyzické:

```
0xb000'c140 – virtuální adresa
       ↓↓↓
0x????'?140 – odpovídající fyzická adresa
```

Zároveň překladové tabulky budou 4096-krát menší, než proti naivnímu přístupu 1 adresa = 1 řádek tabulky.

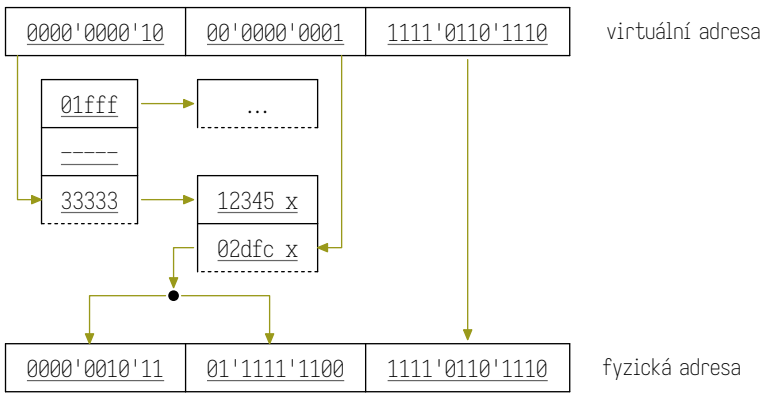
1.2.6 Stránkové tabulky Stránkové tabulky jsou uloženy v operační paměti, ale jejich přesná struktura je opět vlastností konkrétního hardwaru. Protože moderní počítače mají velké adresní prostory (2^{32} nebo 2^{64} virtuálních adres), používají obvykle **řídké**, **víceúrovňové** překladové tabulky. Co to znamená:

- řídká (sparse) tabulka nemusí explicitně mapovat všechny virtuální adresy – nepřítomnost některé stránky v tabulce znamená, že příslušná virtuální adresa není platná,
- zároveň je potřeba, aby vyhledávání v tabulce bylo rychlé – procesor při překladu adresy nemůže zdlouhavě hledat odpovídající řádek.

Virtuální adresa je proto pomyslně rozdělena na několik segmentů: z předchozího už víme, že poslední segment se přímo přepíše do fyzické adresy. Pro každý vyšší segment existuje jedna **úroveň překladu**, řízená tabulkou s odpovídajícím počtem řádků.
Má-li segment například 10 bitů, potřebujeme pro jeho překlad tabulku o $2^{10} = 1024$ řádcích. Na příslušném řádku je odkaz na tabulku o jednu úroveň nižší, ve které se vybere řádek podle dalšího segmentu, atd., až v tabulce poslední úrovně je uvedena fyzická adresa, kterou přímo použijeme (resp. ji doplníme n bity z posledního, přímo mapovaného segmentu virtuální adresy).[14]
Řádky jednotlivých překladových tabulek mohou být také označeny jako nepřítomné – pokusíme-li se takový řádek vybrat, znamená to, že jsme se pokusili přeložit neplatnou virtuální adresu.

- uloženy v operační paměti (RAM)
- velké adresní prostory → řídké tabulky
- realizováno pomocí více úrovní překladu
- segment adresy = úroveň překladu
- hodnota segmentu → řádek tabulky

Příklad: Pro systém s 32b adresami se nabízí rozdělení virtuální adresy na 3 segmenty: 2× 10 bitů = dvě úrovně překladu + 12 bitů mapovaných přímo (bez překladu). Na obrázku je ilustrován překlad adresy 0x0080'1f6e (binárně 0000'0000'1000'0000 0001'1111'0110'1110):



[14] Pozorný čtenář si jistě uvědomil, že takto vedený překlad adresy vyžaduje pro překlad každého segmentu jeden přístup do paměti (načtení příslušného řádku překladové tabulky). Při naivní implementaci by to stále znamenalo režii, která výrazně převyšuje cenu původního přístupu, pro který překlad provádíme. Toto řeší tzv. TLB – translation lookaside buffer – velmi rychlá asociativní paměť, která uchovává nedávno přeložené adresy. Naprostá většina překladů je vyřešena právě vyhledáním v TLB.

Všimněte si, že řádky jednotlivých překladových tabulek obsahují adresy, které mají pouze 20 bitů: každá jednotlivá překladová tabulka má velikost právě jedné stránky (1 řádek po 4 bajtech = 4 KiB), a musí být skutečně uložena v jedné stránce. Její adresa tedy musí končit 12 nulami, které se do tabulky neukládají. Místo toho se těchto 12 bitů využívá na uložení dodatečných informací o příslušné stránce (naznačeno symbolem 'x' na obrázku).

1.3: Procesy

- neformálně „běžící program“
- formálněji virtuální adresní prostor
  - obsahuje kód programu
  - pracovní data
- asociovaný libovolný počet vláken
- abstrakce paměti (von Neumann)

1.3.1 Proces Pojem **běžící program** je poněkud vágní, proto si pro něj zavedeme formálnější alternativu – **proces**. Tímto pojmem budeme označovat entitu, která:

- je spojena s **virtuálním adresním prostorem**, který
  - obsahuje **kód** – instrukce – nějakého programu,
  - veškerá **data**, která tento program potřebuje pro svůj běh,
- operační systém o ni vede záznam,
- může vlastnit krom paměti i další zdroje.

Naopak nepožadujeme, aby byl proces připraven k běhu, nebo aby se v rámci procesu vykonávaly instrukce sekvenčně. Vykonávání instrukcí programu zastřešíme v další kapitole pojmem **vlákno**. Pro tuto chvíli pouze poznamenáme, že k jednomu procesu se může vázat celkem libovolný počet vláken.[15]

V abstrakci počítače, kterou jsme zavedli v kapitole B, odpovídá proces virtuální operační paměti, zatímco vlákno bude odpovídat virtuální výpočetní jednotce. V jednoduchém případě, kdy se k procesu váže právě jedno vlákno, vytvoří dohromady klasický (i když pouze virtuální) von Neumannův počítač s adresovatelnou pamětí a jedním procesorem.

- adresní prostory jsou ± oddělené
- procesy nemají přístup k paměti jiných procesů
- 2 základní výjimky:
  - sdílení pouze pro čtení (zejména kód)
  - komunikační mechanismus

1.3.2 Ochrana paměti Jak jsme již naznačili v části 1.2.3, virtuální adresní prostory různých procesů budou z velké části oddělené. Protože každý adresní prostor je realizován samostatnou sadou stránkových tabulek, operační systém má přímou kontrolu nad tím, které fyzické adresy jsou pro který proces dostupné.

Tím je realizována základní forma ochrany paměti: protože proces je omezen na použití virtuálních adres, k fyzickým adresám, kterým žádná virtuální adresa neodpovídá, přistupovat nemůže. Stránkové tabulky navíc obvykle obsahují dodatečnou možnost omezit přístup i k platným virtuálním adresám: běžně je k dispozici možnost povolit nebo zakázat zápis a nezávisle také spouštění instrukcí.[16] Některé, ale ne všechny, procesory umožňují nezávisle omezit i čtení jednotlivých stránek.

Z tohoto pravidla existují dvě významné kategorie výjimek:

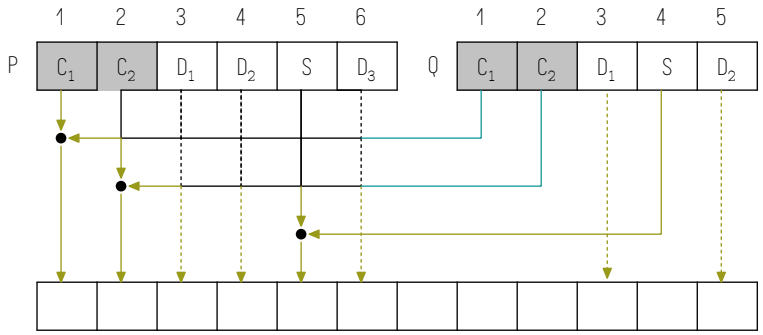
- Kód programu (a případná data, která jsou určena výhradně ke čtení) může být ve fyzické paměti uložen pouze jednou i v případě, kdy tento kód využívá více procesů. Jedná se o užitečnou a zároveň bezpečnou optimalizaci, protože odpovídající virtuální adresy jsou v obou procesech označeny příznakem **pouze pro čtení**, a procesy se tak nemohou skrze tyto adresy vzájemně ovlivňovat.
- Na žádost programu může být dvěma procesům do jejich virtuálních adresních prostorů namapován stejný blok fyzické paměti i v režimu umožňujícím zápis. Smyslem takto namapované paměti je umožnit **komunikaci** mezi dotčenými procesy: mohou si tímto způsobem totiž jednoduše předávat data. Pozor: virtuální adresy takto namapované paměti nebudou obecně v obou procesech stejné.[17]

Příklad: Uvažme dva procesy, které sdílí kód a mají jednu „komunikační“ stránku, která vede na stejné fyzické adresy (proces P tak může do této stránky zapsat nějaká data, a proces Q si je může později přečíst – nebo naopak). Šedě jsou vyznačeny stránky s příznakem pouze pro čtení. Přerušovaně jsou vyznačena nepřekrývající se mapování.

[15] V principu nic nebrání ani tomu, aby byl tento počet nula, i když takový proces by nebyl příliš užitečný.

[16] V úvodu jsme zmiňovali, že nebudeme uvažovat programy, které do paměti zapíšou nové instrukce a ty pak spustí. Toto omezení je pro většinu programů operačními systémy přímo vynuceno, a to tak, že každá stránka má buď právo zápisu, nebo právo spuštění, ale ne obě práva najednou. Jedná se především o bezpečnostní opatření.

[17] S tím jsou spojeny určité obtíže s předáváním ukazatelů, resp. s použitím datových struktur, které ukazatele vnitřně používají.



Všimněte si, že stránka označená S se v obou procesech mapuje na stejnou fyzickou adresu (a tedy stejnou fyzickou paměťovou buňku), v procesu P má virtuální adresu 5, zatímco v procesu Q má adresu 4. Adresy stránek C_1 a C_2 se sice shodují, ale nemuselo by to tak nutně být.

1.3.3 Přepnutí procesu Zatím jsme nezmínili, jak procesor (resp. jednotka správy paměti) najde stránkovou tabulku první úrovně. Odpověď na tuto otázku je klíčem k přepínání procesů:

- adresa stránkové tabulky → registr procesoru
- změna registru aktivuje novou tabulku
- přepnutí procesu = změna tabulky
- souvisí: přepnutí vlákna (později)

1. fyzická adresa stránkové tabulky 1. úrovně je uložena ve speciálním registru procesoru,
2. tuto adresu je možné nastavit privilegovanou instrukcí (privilegovanou v tomto případě znamená, že ji může provést pouze jádro),
3. změnou hodnoty v tomto registru se aktivuje překlad adres podle nově zavedené stránkové tabulky.[18]

Samotné přepnutí procesu je tedy velmi jednoduché: stačí přepnout aktivní stránkovou tabulku na tu, která popisuje virtuální adresní prostor nového procesu.
S přepnutím procesu je spojeno také přepnutí vlákna, které si blíže popíšeme v další kapitole. Stojí zde ovšem za zmínku, že výměna hodnot výpočetních registrů (tj. těch, které jsou dostupné běžnému programu) je spojena právě s přepnutím vlákna. Je to proto, že registry jsou součástí výpočetní jednotky (procesoru), a jak jsme již zmiňovali v sekci 1.3.1, virtuální výpočetní jednotku představuje právě vlákno a nikoliv proces.

1.3.4 Vytvoření procesu Asi nejjednodušší způsob, jak vytvořit nový proces, je duplikace nějakého existujícího (POSIX této operaci říká fork). V mnoha operačních systémech je to dokonce způsob jediný.[19]

- vytvoření procesu duplikací → fork
- copy on write:
  - duplikuje adresní prostor (data se sdílí)
  - označit vše „jen pro čtení“
  - data kopírovat při pokusu o zápis
- iluze oddělených adresních prostorů

Mohlo by se zdát, že je to způsob dost neefektivní: procesy mohou mít virtuální adresní prostor o velikosti mnoha GiB a duplikace takového množství paměti vyžaduje vynaložení značných prostředků. Operační systémy proto používají trik, který zde vede k výrazné úspoře, a se kterým se setkáme ještě v několika kontextech.
Tento trik nese název „copy on write“ a jeho myšlenka je poměrně jednoduchá:

1. vytvoříme kopii **adresního prostoru** (tedy relevantních stránkových tabulek[20], nikoliv samotných dat),
2. v obou kopiích označíme všechny stránky příznakem **jen pro čtení** a poznačíme si také, že se jedná o „copy on write“ stránky,
3. máme nyní zaručeno, že ani jeden proces tomu druhému nemůže jakkoliv narušit data, zároveň ale může data libovolně číst (a data, která přečte, budou stejná, jako by druhý proces vůbec neexistoval),
4. pokusí-li se některý z dotčených procesů data zapsat, vyvolá tím výjimku ochrany paměti,
5. procesor předá řízení jádru operačního systému, které zjistí na které adrese došlo k nepovolené operaci,
6. jedná-li se o adresu označenou jako „copy on write“, jádro vytvoří kopii dat a upraví stránkové tabulky tak, aby každá ukazovala na jednu z kopií (zároveň u nich povolí zápis),
7. procesoru nařídí, aby přerušenou operaci zopakoval a pokračoval ve výpočtu.

Vznikne tak iluze, že procesy jsou zcela nezávislé, přestože většinu dat ve fyzické paměti sdílí. Takto implementovaná operace fork sice není zadarmo, ale je nesrovnatelně rychlejší, než by bylo provedení skutečné kopie procesu.

---

[18] Tato operace není tak levná, jak by se mohlo na první pohled zdát – její cenu ale nezaplatíme okamžitě, projeví se totiž zneplatněním záznamů v TLB. Proto bude těsně po přepnutí procesu překlad adres mnohem pomalejší (dokud se TLB nenaplní novými daty).

[19] Aby takový systém mohl fungovat, musí existovat jakýsi prapůvodní proces, který vznikne „ex nihilo“ při startu systému. Tomuto procesu se obvykle říká init.

[20] Ve skutečnosti nemusíme kopii vytvářet hned – v této chvíli zatím nic nebrání tomu, aby byla celá stránková tabulka sdílená mezi procesy (to se ovšem změní jakmile budeme potřebovat provést jakýkoliv zápis do paměti jednoho z procesů). V systémech s víceúrovňovým překladem navíc nic nebrání tomu, aby byly sdílené pouze tabulky druhé a dalších úrovní.

1.4: Externí stránkování

- neplatná virtuální adresa → výjimka
- situaci řeší jádro
- nepřítomná stránka → „prázdný“ řádek tabulky
- OS může využít pro vlastní potřeby

1.4.1 Neplatné adresy Z diskuse o metodě „copy on write“ již víme, že pokusí-li se program zapsat na virtuální adresu, která je označená příznakem „jen pro čtení“, dojde k výjimce ochrany paměti, a řízení je předáno jádru OS. Stejně tomu tak je i v případě, že je požadovaná virtuální adresa neplatná. V takovém případě navíc nemusí být v příslušném řádku[21] stránkovací tabulky uložena žádná fyzická adresa – operační systém může toto číslo využít pro vlastní účely.

- OS může data přesunout z operační paměti
- v stránkové tabulce označí adresu za neplatnou
- může si tam rovnou poznačit skutečnou lokaci
- lze přidělit více paměti než existuje

1.4.2 Externí stránkování Nabízí se zde možnost tohoto mechanismu využít k dalšímu triku: má-li OS nedostatek použitelných fyzických adres (například proto, že běží hodně programů, které využívají hodně paměti), může některou stránku odstěhovat z operační paměti někam jinam – např. na externí (pevné) úložiště, které je sice obvykle mnohem pomalejší, ale také má obvykle mnohem větší kapacitu. Nebude-li se stávat příliš často, že program potřebuje k takto „odklizené“ stránce přistoupit, nemusí se jednat o zásadní problém.

- vazby virtuálních adres jsou nyní proměnné
- stránka = virtuální adresy, „data“
- rámec = fyzické adresy, paměťové buňky
- stránka je uložena v rámci
- rámec lze přesunem stránky uvolnit

1.4.3 Rámce a stránky Protože až do této chvíle byla každá virtuální adresa relativně pevně svázaná s nějakou adresou fyzickou, vystačili jsme si se samotným pojmem stránka. Externí stránkování ale z povahy věci bude stránky ve fyzické paměti různě přemísťovat. Bylo by proto dobré zavést přesnější terminologii:

- **stránkou** budeme i nadále označovat **rozsah virtuálních adres**,
  - také stále platí, že stránky mají pevnou velikost (počet virtuálních adres, které obsahuje), a že
  - nejnižší adresa stránky musí být beze zbytku dělitelná velikostí stránky,
- **rámcem** budeme označovat rozsah **fyzických** adres,
  - které splňují stejné požadavky na velikost a zarovnání jako stránky,
  - a na jeden rámec mapovat právě jednu stránku.

Lze pak mluvit o tom, že daná stránka je uložena v nějakém rámci, případně že daný rámec je volný. Operační systém rozhoduje, který rámec v případě potřeby uvolnit (stránka, která do té doby rámec obývala říkáme oběť) atp. Algoritmy a dalšími technickými detaily kolem stránkování na disk bychom mohli zabrat celou kapitolu; protože ale tento typ externího stránkování není pro moderní systémy příliš důležitý, nebudeme se jím zabývat hlouběji.

- stejný mechanismus, jiná aplikace
- načítání spustitelného souboru dle potřeby
- jednodušší → není potřeba hledat oběť
- lze kombinovat s předchozím

1.4.4 Líné načítání To, kde se externí stránkování uplatní více, je tzv. líné načítání – zavádíme-li program z externí paměti do paměti operační (například spouštíme program z disku nebo SSD), není potřeba celý obsah spustitelného souboru[22] nahrávat do operační paměti ihned. Podobně jako v případě „copy on write“ stačí sestavit vhodnou stránkovou tabulku, která zaručí načtení správných stránek přímo ze spustitelného souboru ve chvíli, kdy jich bude skutečně zapotřebí.
Mechanismus, který se k tomu využívá, je jinak shodný s předchozí formou externího stránkování, odpadá ale hledání oběti a ukládání stránky na disk – její obsah je již uložen v samotném spustitelném souboru.[23]
Líné načítání lze také výhodně kombinovat s klasickým externím stránkováním – vybereme-li jako oběť stránku s kódem (nebo konstantními daty) programu, není potřeba ji nikam ukládat – lze ji přímo odkázat zpátky na spustitelný soubor.

- lze použít i pro přístup k datovým souborům
- na vyžádání aplikace
- přístup k souboru jako k místu v paměti
- v praxi nejdůležitější

1.4.5 Mapování souborů Posledním, jednoznačně nejdůležitějším, využitím externího stránkování je mapování datových souborů do paměti (na žádost aplikace). V tomto režimu jsou externí stránky uloženy v jinak běžném souboru. Místo komplikovaných vstupně-výstupních operací tak může program s obsahem souboru pracovat stejně, jako by byl uložen v operační paměti. Využijeme-li tohoto mechanismu také k zápisu změněných stránek zpátky do souboru, může program soubor i zcela transparentně upravovat.

Část 2: Virtualizace procesoru

Druhým klíčovým zdrojem, krom adresovatelné paměti, je **procesor**. V této kapitole si popíšeme, jak operační systém zabezpečí, aby měl každý program „svůj vlastní“ virtuální procesor (vlastní paměť už mu zabezpečila předchozí kapitola).

---

[21] Konkrétněji v tom, který rozhodl o neplatnosti adresy.

[22] Intuitivně, spustitelný soubor je takový, který obsahuje zejména kód programu a jeho konstantní data. Blíže se spustitelnými soubory budeme zabývat ve třetí kapitole.

[23] Tato technika je poměrně běžně používaná, má ale jednu nevýhodu: takto běžící program spoléhá, že spustitelný soubor, ze kterého byl spuštěn, nebude modifikován.

Doporučené čtení: A. Tanenbaum, H. Bos – Modern Operating Systems (4th Ed.): § 2.1.1 The Process Model ◇ § 2.1.5 Process States ◇ § 2.2.1 Thread Usage ◇ § 2.2.2 The Classical Thread Model ◇ § 2.4 Scheduling.

2.1: Procesor

Nejprve si ale musíme upřesnit, jak procesor pracuje (samozřejmě na relativně abstraktní úrovni).

2.1.1 Výpočet Procesor **vykonává instrukce**, čím **realizuje výpočet**. Nejjednodušší třídou instrukcí jsou tzv. aritmetické a logické instrukce (tedy ty, které provádí ALU – aritmeticko-logická jednotka). Tím se myslí zejména:

- **aritmetika**: sčítání, odečítání, násobení a dělení,
- **bitové** operace: and, or, xor po bitech, bitové posuvy,
- **srovnání** dvou hodnot (rovnost, nerovnost) – výsledek se uloží do běžného registru nebo do stavového příznaku procesoru.

Další třídy instrukcí provádí složitější akce, které mění stav paměti, nebo řídí průběh výpočtu:

- již zmiňovaný přístup do paměti (load, store),
- řízení toku – rozhodnutí o tom, kterou instrukcí bude výpočet pokračovat (zejména **podmíněný skok**, případně **nepřímý skok**),
- instrukce pro realizaci podprogramů[24] zahrnují jak přístupy do paměti, tak řízení toku: patří sem zejména
  - práce s hardwarovým zásobníkem (přesuny hodnot mezi zásobníkem a registry),
  - instrukce pro aktivaci podprogramu (přímo nebo nepřímo),
  - instrukce pro **návrat** z podprogramu.

Pro operační systém jsou důležité také **privilegované** a jiné speciální instrukce, které mu umožňují procesor efektivně spravovat (a virtualizovat).

- aritmetické a logické (ALU) instrukce
  - sčítání, odečítání, násobení, dělení
  - logické operace po bitech, bitové posuvy
  - relační operátory
- přístup do paměti: load, store
- řízení toku: podmíněné a nepřímé skoky
- (volitelně) realizace podprogramů

2.1.2 Registry Podobně jako paměť, registry slouží k ukládáni čísel – existují dva klíčové rozdíly mezi registry a pamětí:[25]

1. pojmenování registru je pevnou součástí instrukce, kdežto paměťovou adresu lze vypočítat (paměť lze indexovat, registry nikoliv),
2. reprezentace čísla v registru je **monolitická** – registry nejsou složené z bajtů, daný registr obsahuje **celé slovo** (částečně důsledek předchozího bodu: registr lze pojmenovat pouze jako celek).[26]

- ukládají čísla (slova, ne bajty!)
- aritmetické registry (patří ALU)
- programový čítač
  - virtuální adresa prováděné instrukce

Příklad: Budeme-li potřebovat mluvit konkrétně, budeme uvažovat hypotetický procesor, který má 16 aritmetických registrů r0 … r9, rA … rF. Jak velká čísla lze do registrů ukládat je opět vlastností konkrétního procesoru – aby se nám dobře psala a četla, budeme uvažovat 16b registry. Moderní počítače mají často registry o šířce 64 bitů (někdy též 32, méně jen u velmi malých počítačů, které obvykle nedisponují ani jednotkou správy paměti).

Vyhrazený registr (programový čítač, angl. **program counter**, někdy také instruction pointer, budeme jej označovat pc) pak obsahuje **virtuální** adresu právě vykonávané instrukce.[27] Tento registr **rozhoduje** o tom, která instrukce se má vykonat, není do něj ale obvykle možné zapisovat běžnými (aritmetickými, atp.) instrukcemi. K tomu jsou určeny instrukce řízení toku, kterých hlavním efektem je právě změna hodnoty programového čítače.

2.1.3 Instrukce Je **ucelený elementární příkaz** strojového kódu; to znamená:

- instrukce je ucelený elementární příkaz
- procesor zná jen konečný počet instrukcí
  - lze je tedy očíslovat
- instrukce lze sdružovat podle operací
  - add, load, atp. jsou operace
  - schéma → operace + typ a počet operandů

[24] Tyto instrukce nejsou pro rozumnou funkčnost procesoru nutné – lze je vždy „rozepsat“ na jednodušší instrukce. Jejich ustálený tvar a význam nám ale abstrakci ve skutečnosti zjednoduší.

[25] Je také obvyklé, že přístup do registrů je mnohem rychlejší než do paměti. To je pro nás v tuto chvíli celkem nepodstatné: většinou si vystačíme s abstrakcí, kde každá instrukce trvá stejně dlouho, bez ohledu na to, co počítá, nebo jestli přistupuje do paměti. Budeme také většinou zanedbávat mezipaměti atp. (bude-li nějaký efekt tohoto typu důležitý, upozorníme na něj).

[26] Možná z architektury x86 znáte např. registry al, ah, ax, eax, rax, které se „překrývají“. To lze ale chápat tak, že skutečný registr je pouze rax (resp. eax na 32b procesorech, ax na 16b procesorech) a ty ostatní jsou virtuální: „syntaktické zkratky“ pro extrakci příslušných bitů skutečného registru. Vztah mezi „pojmenovaným“ (virtuálním) registrem a fyzickým registrem (klopnými obvody) je ostatně ve skutečnosti dost komplikovaný. Zde si vystačíme s abstrakcí, kde se registry nepřekrývají, a vůbec nebudeme virtuální a fyzické registry rozlišovat.

[27] Jiný speciální registr, který obsahuje (fyzickou) adresu stránkové (překladové) tabulky nejvyšší úrovně, jsme zmiňovali v předchozí kapitole.

- **ucelený** – obsahuje veškeré informace potřebné k provedení konkrétních akcí (zejména udává **operaci**, která se má provést, a **konkrétní registry**, se kterými se bude pracovat),
- **elementární** – je to nejmenší jednotka činnosti, kterou lze procesoru zadat,
- **příkaz** – instrukce řídí činnost procesoru, „přikazují“ mu provedení nějaké akce.

Instrukcí je pouze konečně mnoho, je tedy zejména možné je **očíslovat** (nebo jinak konečně kódovat, např. do sekvencí bajtů). Každé takové číslo (kódování) popisuje konkrétní akci, kterou může procesor provést.

Příklad: Instrukce z pohledu člověka vypadají například takto:

- r0 ← add rA rB; jiná (i když podobná) instrukce je
- r1 ← add rA rB; ještě jiná je
- r0 ← add rA rC; atd.

Takto instrukce zapisujeme pro vlastní potřebu – tomuto zápisu bychom mohli říkat mnemonický zápis strojového kódu, ale v dalším nebudeme striktně zápis a podstatu strojového kódu odlišovat – mezi číselným a mnemonickým zápisem je podobný vztah jako mezi „7“, „VII“ nebo „sedm“. V počítači jsou instrukce reprezentovány čísly:[28]

- r0 ← add rA rB by mohlo být například 0x010a000b,
- r1 ← add rA rB by mohlo být 0x011a000b,
- r0 ← add rA rC by zase mohlo být 0x010a000c.

Pozor, jiný je **jazyk symbolických adres** (angl. **assembly language**), který používá stejný mnemonický zápis, ale nezapisuje jím nutně přímo instrukce.[29] To, co jsme zde nazvali mnemonickým zápisem, je velmi jednoduchou podmnožinou jazyka symbolických adres.

Striktně vzato tedy neexistuje nic jako „instrukce add“ (nebyla by totiž ucelená).[30] Samozřejmě má ale smysl mluvit o takto příbuzných instrukcích nějak souhrnně. Takovou rodinu instrukcí budeme popisovat jako **schéma**, které má společnou **operaci**[31] (např. sčítání) a počet a typ operandů (registrů), ale každá instance pracuje s různými konkrétními registry.

- instrukce mění stav
  - zejména procesoru (vždy)
  - ale také paměti, periferií (někdy)
- výpočet → posloupnost změn stavu
- efekt instrukce = elementární změna stavu

2.1.4 Efekt instrukce Každá instrukce má nějaký efekt na **stav procesoru**, a případně (podle konkrétní instrukce) na další připojená zařízení (zejména paměť). Tento efekt je zároveň **definující** charakteristikou dané instrukce.
Program totiž nedělá nic jiného, než že vhodnou **manipulací stavu** (procesoru, paměti, periferií) postupuje od **vstupů** k požadovaným **výstupům**. Této posloupnosti změn stavu říkáme **výpočet**.
Jak vstupy tak výstupy programu jsou **součástí stavu** – vstupy na začátku, výstupy na konci výpočtu.[32] Vstup může být například hodnota zapsaná v nějaké buňce paměti; výstup může být třeba stav obrazovky, kdy rozsvícené pixely vytváří obrazec, který přečteme jako slova hello world. **Efekt instrukce** tedy není nic jiného, než **elementární změna stavu**, nebo jinak řečeno **elementární výpočet**. Instrukci samotnou tak můžeme chápat jako pokyn k provedení takového elementárního výpočtu.

Příklad: Rodina instrukcí se schématem result ← add op_1 op_2 má obvykle tyto efekty:

1. sečte hodnoty z registrů op_1 a op_2 a výsledek zapíše do registru result,
2. k programovému čítači přičte velikost svého vlastního kódování a tím de facto spustí vykonávání následující instrukce.

Konkrétněji, uvažujeme-li instrukci rA ← add r1 r3:

[28] Nebo ještě přesněji posloupností bajtů. Na naší úrovni abstrakce si ale můžeme dovolit chápat instrukce jako čísla, protože číslo je mnohem jednodušší objekt, než sekvence bajtů uložená na nějaké sekvenci adres (už samotný pojem „adresa“ je značně komplikovaný – viz předchozí kapitola). Od sekvencí bajtů (potenciálně proměnné délky) můžeme k číselnému kódování přejít předřazením vhodného dekodéru. Jedná se tedy o podobné zjednodušení, jako zanedbání složených instrukcí (které můžeme nahradit sekvencí instrukcí jednodušších).

[29] Z jednotlivých příkazů jazyka symbolických adres je nutné konkrétní instrukce nejprve vypočítat. Tento proces je navíc jednosměrný: pojmenovaná návěstí ve strojovém kódu neexistují, a ani není možné ze strojového kódu jednoznačně určit, na kterých místech byla, a kde a jak byla použita, natož jaká nesla jména.

[30] Přesto, že o takové instrukci často slýcháme – je to běžná jazyková zkratka. Protože ale lehce vede k nedorozuměním, budeme se jí radši vyhýbat.

[31] Operace je **obvykle** určena **operačním kódem**: autoři instrukčních sad často volí nějakou pravidelnou strukturu čísel, které jednotlivé instrukce kódují – v takovém případě je operační kód ta část, která určuje **operaci**. Jiné části obvykle určují operandy (registry, se kterými se bude pracovat). Pozor: kódování instrukcí žádnou takovou strukturu dodržovat **nemusí**. A aby nebylo zmatku málo, někdy se operačním kódem myslí i kódování instrukce jako celku.

[32] U interaktivních programů jsou vstupy a výstupy definovatelné mnohem hůře, ale celkem jednoznačně je smyslem takového programu vhodně manipulovat **stavem** tak, aby reakcí na **vnější** změnu stavu (například stisk klávesy) byla nějaká užitečná, **programem řízená** změna stavu (např. vykreslení písmene na obrazovku).

- zakódovanou do čísla 27328515 (= 0x01a10003),
- které může být dále zakódované do 4 bajtů jako sekvence 01 a1 00 03,[33]

její efekt bude:

1. sečte hodnoty registrů r1 a r3 a výsledek zapíše do registru rA,
2. k programovému čítači přičte hodnotu 4 (posune se na adresu, která ukazuje „těsně za“ právě provedenou instrukci).

2.1.5 Program Co bude program počítat (resp. co bude počítat procesor řízený daným programem) je určeno **textem programu**: instrukcemi uloženými v paměti. Instrukce, které nejsou skoky, obvykle posouvají programový čítač za svůj vlastní konec, tedy na instrukci na nejbližší vyšší adrese. Většina programu je tedy prováděna v pořadí od nižších k vyšším (virtuálním) adresám. Výjimku samozřejmě tvoří **instrukce skoku**, které mohou některé adresy přeskočit (typicky podmíněné příkazy – if), nebo se naopak vrátit k některé dřívější, už vykonané instrukci (typicky cykly – while).

- je tvořen textem → instrukce v paměti
- prováděn od nižších adres k vyšším
- výjimkou jsou instrukce skoku
  - podmíněný příkaz → obvykle dopředu
  - cyklus → obvykle dozadu

2.1.6 Zásobník Dalším běžným důvodem pro vykonání skoku je **aktivace podprogramu** – podprogram je, už podle názvu, nějaký ucelený blok instrukcí, který se podobá na program – určuje, jak se bude provádět nějaký výpočet. V případě podprogramu se jedná o pomocný výpočet, který je obvykle v programu jako celku potřebné provádět opakovaně a/nebo v různých kontextech (a/nebo s různými vstupními hodnotami).

Za účelem aktivace podprogramu by se nám hodila jakási odlehčená verze virtualizace procesoru:

1. podprogram je dostatečně uzavřený celek na to, aby mělo smysl jej alespoň částečně oddělit od ostatních výpočtů – často např. používá nějaká data, která mimo tento podprogram nemají žádný význam (ve vyšším programovacím jazyce **lokální proměnné**),
2. protože takový podprogram je navíc často potřeba spouštět z různých míst programu, je žádoucí mít nějaký mechanismus, který po ukončení výpočtu podprogramu vrátí řízení na místo, ze kterého byl původně aktivován.

Obě tyto funkce zastává **zásobník**[34] (hardwarový zásobník, zásobník volání, angl. **call stack**, atp.). Jedná se o spojitou oblast virtuálního adresního prostoru, které rozsah je dán hodnotou **ukazatele vrcholu zásobníku** (jedná se o druhý „speciální“ registr, vedle programového čítače) a pevným **dnem**.

- podprogram → pomocný, vnořený program
  - použitelný opakovaně a/nebo z více kontextů
- aktivace (volání) podprogramu = typ skoku
  - uloží část stavu procesoru
  - vyhradí místo na lokální data (proměnné)
- návrat → skok zpět za místo aktivace
- realizace hardwarovým zásobníkem

2.1.7 Aktivační záznam Zásobník typického programu je složen z **aktivačních záznamů**, známých též jako **rámců**[35] (angl. **call frame**). Každý takový záznam odpovídá jedné aktivaci podprogramu, která dosud neskončila. Jak jistě víte, podprogramy se mohou aktivovat (volat) vzájemně, nebo může dokonce podprogram opakovaně aktivovat sám sebe – jev, který pravděpodobně znáte jako **rekurzi**.

Je také relativně logické, že má-li nějaký podprogram skončit (a tedy má být jeho aktivační záznam odstraněn), musí nejprve skončit všechny podprogramy, které sám aktivoval (přímo či nepřímo). Tato skutečnost je odpovědná za to, že aktivační rámce tvoří právě zásobník[36] (a ne třeba frontu, strom, graf, nebo jinou strukturu).

- zásobník → sekvence aktivačních záznamů
- jeden záznam = jedna aktivace podprogramu
- volaný končí dřív než volající
  - poslední zavolaný končí první
  - LIFO (last in, first out) → zásobník

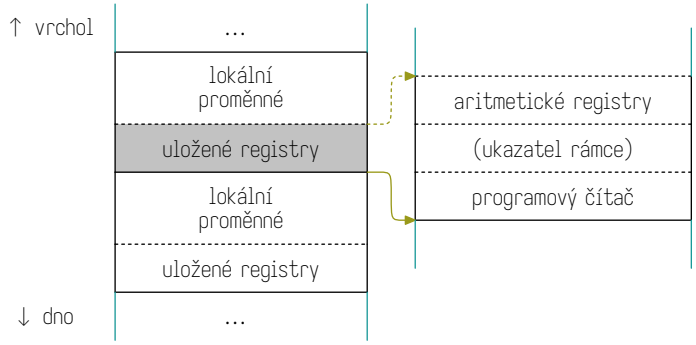
Příklad: Zásobník obvykle kreslíme dnem dolů. Jaký má orientace na obrázku vztah k vyšším nebo nižším adresám je určeno procesorem (je-li vybaven instrukcemi pro práci se zásobníkem). Řada procesorů používá zásobníky, které „rostou“ směrem k **nižším** adresám (později zavolané podprogramy mají rámce na číselně menších adresách).

[33] Uvedený zápis je v pořadí MSB-first (nejvýznamnější bajt na nejnižší adrese). Kdybychom chtěli použít pořadí LSB-first, bylo by totéž číslo zapsáno sekvencí 03 00 A1 01.

[34] V principu nic nebrání tomu tyto dvě funkce oddělit na dva samostatné zásobníky, i když to není obvyklé. Ve světle nespočetných bezpečnostních problémů (zranitelností), které kombinace obou funkcí do jedné struktury způsobila, bylo jejich spojení možná chybou.

[35] Pozor, jedná se v tomto kontextu o úplně jiný druh rámců než ty, které jsme zmiňovali v předchozí kapitole. O jakém druhu rámce je řeč, by mělo být vždy zřejmé z kontextu.

[36] Upuštěním od tohoto požadavku se od podprogramů posouváme k tzv. koprogramům (častěji označovaných pojmem korutiny). Pak už zásobník použít nelze. Koprogramy jsou naštěstí značně mimo rámec tohoto předmětu.



Uschovaný programový čítač je často nazýván **návratovou adresou**. Ukazatel rámce označuje mechanismus, kterým se určuje, jak se má při návratech posunovat ukazatel zásobníku. Má-li rámec daného podprogramu pevnou velikost, nemusí být tato informace vůbec přítomna.[37]

2.2: Vlákna a virtualizace

- výpočet → postupné změny stavu
- omezíme se na stav procesoru:
  a. hodnoty aritmetických registrů
  b. hodnota programového čítače
  c. hodnota ukazatele zásobníku
- izolovaný výpočet je deterministický
  - určen počátečním stavem a programem

2.2.1 Stav procesoru Výpočet jsme si výše definovali jako **posloupnost stavů**, resp. posloupnost změn stavu. Nyní se zaměříme na **stav procesoru** – nedostaneme tím sice úplný pohled (pro výpočet je důležitý také **stav paměti**, a vstupují-li do hry periferie, také celkový „stav světa“), ale momentálně se nám takto omezený pohled hodí lépe.

Příklad: Uvažme velmi jednoduchý program uložený na adrese `0x1000` (přičemž každá instrukce je kódovaná 4 bajty):

1. `r0 ← add r1 r1`
2. `r1 ← mul r1 r0`
3. `stop`

Sekvence provedených instrukcí je v tomto případě zjevná, ale jak budou vypadat odpovídající stavy procesoru záleží na tom, v jakém stavu bude procesor, když program spustíme. Mohlo by to být např.:

pc	r0	r1	instrukce pod pc
0x1000	0x0010	0x0007	r0 ← add r1 r1
0x1004	0x000e	0x0007	r1 ← mul r1 r0
0x1008	0x000e	0x0062	stop

Stav procesoru tedy sestává z:

1. hodnot uložených v aritmetických registrech,
2. hodnoty programového čítače,
3. hodnoty ukazatele zásobníku.[38]

V případě, že:

- program neobsahuje instrukce přístupu do paměti[39],
- **nebo** celý virtuální adresní prostor je přístupný pouze aktivnímu programu,

je celý výpočet **jednoznačně určen** programem samotným a **počátečním stavem** procesoru (a případně paměti). Tato situace se významně změní, může-li být do adresního prostoru zasaženo „zvenčí“ (jiným aktérem, než je samotný aktivní program). Příkladem by mohlo být namapování fyzických adres některé periferie do adresního prostoru programu.

- co když výpočet není izolovaný?
- opakované čtení → různé výsledky
  - pouze hodnoty v paměti
  - externí změna hodnoty → událost
- interakce výpočtů → synchronizace

2.2.2 Synchronizace Uvažme nyní situaci, kdy adresní prostor programu **není** izolovaný, a tedy do něj mohou zasahovat externí entity (jiné programy, operační systém, periferie). Uvažme velmi jednoduchý program:

1. `x = True`
2. `while x: pass`

[37] To by ale vylučovalo konstrukce typu `alloca` nebo tzv. VLA (variable-length array) jazyka C. Architektura x86 má zvláštní registr, `rbp` (`bp` od **base pointer**), který je zde uložen, a který určuje „dno“ aktuálního rámce, a tedy hodnotu ukazatele zásobníku (na x86 se tento jmenuje `rsp`) po návratu z podprogramu.

[38] Krom uvedených lze také uvažovat další „speciální“ registry. Podobně jako složené instrukce nebo bajtové kódování instrukcí je můžeme pro zjednodušení abstrakce (bez újmy na obecnosti) ignorovat.

[39] Samotný text programu považujeme za neměnný.

Je celkem jasné, že takový program **v izolaci** bude navěky zaseknutý ve smyčce, která „nic nedělá“. Podívejme se nejprve blíže na ono nic; velmi podobný program by ve strojovém kódu mohl vypadat třeba takto (v hranatých závorkách uvádíme adresu, na které příslušná instrukce začíná):

```
[0x00] r1 ← 0x0001       # nastavíme registr r1 na „True“
[0x04] store r1 0x1000   # hodnotu uložíme na adresu 0x1000
[0x08] r0 ← load 0x1000  # načteme hodnotu z adresy 0x1000
[0x0c] goto 0x08 if r0   # podmíněný skok na předchozí instrukci
[0x10] stop              # konec programu
```

V této verzi programu má ono „nic“, které se donekonečna opakuje, tyto dva kroky:

1. načti hodnotu z paměti,
2. zjisti, je-li nenulová, a pokud ano, opakuj krok 1.

Je nyní jasné, že ve chvíli, kdy nějaká externí entita zapíše na adresu 0x1000[40] hodnotu 0, program skončí. Takové interakci říkáme **synchronizace** – průběh výpočtu se změnil v návaznosti na nějakou externí **událost**: v tomto případě zápis do paměti.[41]

2.2.3 **Vlákno** Nyní jsme konečně připraveni definovat pojem **vlákno**: je to

- **výpočet** (posloupnost změn stavu), který vznikne
- nepřerušenou činností jednoho **procesoru**, který je
- po celou dobu řízen jedním **programem**.

- výpočet 1 procesoru a 1 programu
- není izolovaný a tedy ani jednoznačný
- povolujeme synchronizaci
  - pasivně čtením, aktivně zápisem paměti
- stav procesoru určen výlučně programem
- zásobník patří vláknu

Všimněte si, že neklademe žádné požadavky na adresní prostor. Takový výpočet tedy **není** jednoznačně určen, protože **není izolován** – součástí takového výpočtu může být **synchronizace**, a tedy průběh výpočtu krom samotného programu a počátečního stavu bude záviset také na **vnějších událostech**, které může vlákno skrze svůj adresní prostor **pozorovat**.
Takové pozorování je (alespoň pro tuto chvíli) omezeno na instrukce, které **čtou paměť** (samozřejmě skrze virtuální adresu; nemusí se ale nutně jednat o **operační** paměť). Vlákno může zároveň synchronizaci **iniciovat**, a to **zápisem do paměti** (za předpokladu, že zapsanou hodnotu přečte nějaké jiné vlákno, periferie, atp.).
Za povšimnutí také stojí, že definice **nepovoluje**, aby se stav procesoru[42] měnil jakkoliv jinak, než řízením programu. Bude také výhodné předpokládat, že **zásobník** je pro externí entity **nepřístupný**[43] a že jej tedy chápat jako součást **stavu vlákna** (který je jinak shodný se stavem procesoru).

2.2.4 **Logický procesor** Definice vlákna se odvolává na nepřerušenou činnost procesoru. Pro účely této definice si ale vystačíme s velmi abstraktním chápáním procesoru:

- má stav: právě hodnoty registrů
- stav se mění právě vykonáním instrukce
- stav lze uložit a obnovit
  - 1 fyzický, více logických procesorů
  - virtualizace procesoru

1. procesor má stav, který sestává právě z hodnot registrů,
2. procesor svůj stav mění vykonáváním instrukcí (a nijak jinak).

Takto popsaný procesor budeme nazývat **logickým procesorem**. Má několik zajímavých vlastností:

1. je zřejmé, že výpočet logického procesoru lze přímočaře realizovat na fyzickém procesoru,
2. stav logického procesoru je velmi jednoduchý a tedy není těžké si představit, že bychom ho mohli například uložit do paměti (to se nakonec částečně děje i při aktivaci podprogramu),
3. lze si také představit, že bychom mohli fyzický procesor do takto uloženého stavu zase vrátit.

Předpokládejme, že operace z bodů 2 a 3 skutečně existují, a že je může operační systém provést bez součinnosti aktuálně prováděného vlákna.[44] Pak už je jednoduše vidět, jak realizovat několik logických procesorů pomocí jednoho fyzického, a tím dosáhnout **virtualizace procesoru**.

---

[40] Z pohledu takové externí entity může mít odpovídající buňka paměti samozřejmě adresu úplně jinou: 0x1000 je virtuální adresa v adresním prostoru programu.

[41] Lze si lehce představit i situaci, kdy adresa 0x1000 odpovídá stavovému příznaku nějaké periferie, třeba klávesnice. Přečtení hodnoty 0 by tak mohlo znamenat třeba stisk klávesy uživatelem.

[42] Myslíme stav, který je pozorovatelný programem, jak jsme si jej definovali výše: hodnoty aritmetických registrů, ukazatel vrcholu zásobníku a programový čítač.

[43] To striktně vzato není pravda, ani na teoretické, ani na praktické úrovni. Zásobník je oblast virtuálního adresního prostoru, která není jinak ničím speciální (vyznačuje se pouze tím jak tuto oblast program používá). Zároveň není v praxi před externím zásahem chráněn: jiná vlákna ve stejném procesu (a tedy stejném virtuálním adresním prostoru) mohou k zásobníkům jiných vláken volně přistupovat. Je ale obecně uznávaným pravidlem, že to dělat nebudou. Jedná se tedy o podobné zjednodušení, jako předpoklad, že program samotný je neměnný.

[44] To není úplně jednoduché, protože stav logického – a dočasně tedy i toho fyzického – procesoru je řízen **výlučně** běžícím vláknem. Aby to vůbec bylo možné, musí do hry vstoupit nějaká vnější událost, která není přímo pozorovatelná vláknem, ale umožní operačnímu systému převzít kontrolu nad skutečným procesorem (aniž by byl logický procesor jakkoliv dotčen). Takovou událostí je **přerušení** a bude předmětem 8. kapitoly.

- uložení a obnova všech registrů
- vlákno → vlastní logický procesor
- nelze zcela bez podpory procesoru
- lze využít mechanismus přerušení

2.2.5 **Přepnutí vlákna** Protože můžeme uložit, a později obnovit, stav logického procesoru, můžeme na jednom fyzickém procesoru provádět střídavě několik různých vláken, a zároveň zabezpečit, že každé vlákno má, ze svého vlastního pohledu, pomyslný **vlastní procesor**.
Co obnáší uložení a obnova registru závisí na jejich typu:

1. **aritmetické registry** – teoreticky nepředstavují problém, ale protože bez aritmetických registrů nelze nic počítat, nelze bez asistence procesoru uložit ani obnovit **všechny**,
2. **ukazatel zásobníku** v principu jednoduché jak uložit tak obnovit, problém ale nastane, používáme-li zásobníkové instrukce k manipulaci s ostatními registry,
3. **programový čítač** představuje největší problém: nelze přímo ani uložit (vyžadovalo by součinnost prováděného programu), ani obnovit (obnovou automaticky ztrácíme kontrolu nad procesorem, musí tedy být provedena v posledním kroku, kdy už ale nemáme k dispozici žádné aritmetické registry).

Situace tedy není příliš slibná, a v podstatě nezbývá, než spoléhat na speciální podporu zabudovanou v procesoru. Naštěstí, jak později uvidíme, obsluha přerušení, bez ohledu na vlákna nebo virtualizaci, musí uložení alespoň části stavu vyvolat. Podobně při návratu z obslužného podprogramu je potřeba takto uložený stav (resp. jeho část) opět obnovit. Potřebná podpora je díky tomu přítomna v prakticky každém procesoru.
Je-li programový čítač a alespoň část aritmetických registrů uložena, není již problém uložit (nebo obnovit) zbytek stavu běžnými instrukcemi. Podobně lze doplnit částečné hardwarově realizované obnovení stavu (pouze v opačném pořadí – aritmetické registry, které budou později obnovené hardwarově, lze použít během obnovy těch ostatních).

2.3: Plánování vláken a procesů

Protože v obecném případě máme vláken více, než dostupných fyzických procesorů, musí operační systém nějak určit, která vlákna budou kdy spuštěna (budou mít přidělený procesor).

- aktivován pravidelně přerušením (časovač)
- na každém fyzickém procesoru
- rozhoduje o přepnutí vlákna
  - zda přerušit aktivní
  - které aktivovat jako další
- preempce: nevyžaduje součinnost programu

2.3.1 **Plánovač** Části jádra operačního systému, která je odpovědná za přidělování procesorů vláknům říkáme **plánovač vláken**, často také z historických důvodů **plánovač procesů**. Plánovač je na každém procesoru aktivován v pravidelných intervalech[45] a je mu tedy umožněno odebrat procesor aktivnímu vláknu i bez součinnosti tohoto vlákna.[46]
Základním rozhodnutím plánovače při každé aktivaci je: „Má aktivní vlákno (logický procesor) pokračovat ve výpočtu, nebo má být přerušeno (preempted). Které vlákno má být na procesoru spuštěno jako další?“

- maximální propustnost (práce/čas)
- minimální latence (reakční doba)
- férovost (rovnoměrné rozdělení zdrojů)
- nelze vše najednou → kompromis

2.3.2 **Cíle** Na plánovací algoritmus (a plánovač jako celek) máme několik základních požadavků:[47]

1. maximalizovat **propustnost**:
   - rozhodnutí nesmí trvat příliš dlouho (chceme co nejvíce času trávit produktivně, nikoliv rozhodováním o tom, co bychom měli dělat první),
   - vlákna by se neměla střídat příliš často (přepnutí vlákna není zadarmo, zejména v případě, kdy vyměňovaná vlákna patří do různých procesů),
   - vlákna by se neměla příliš často přesouvat z jednoho fyzického procesoru na jiný (podobně to je akce, která není zadarmo),
   - všechny procesory (a další zdroje) by měly být neustále maximálně využité,
   - důležité pro **výpočetní** programy,
2. minimalizovat **latenci** (prodlevu):
   - vlákna by neměla na procesor čekat příliš dlouho (často se stává, že uživatel čeká na reakci programu, a nechce čekat déle, než je nutné),
   - důležité pro **interaktivní** programy (jak přímo uživatelské, ale také např. síťové služby),
3. udržovat **férovost**:

[45] Zabezpečuje se programovatelným časovačem, který dokáže vyvolat přerušení. Může být periodický (naprogramuje se jednou a poté v pravidelných intervalech „tiká“) nebo jednorázový (je potřeba ho pokaždé znovu spustit). Přerušeními se budeme blíže zabývat v osmé kapitole.

[46] Plánovače, které tuto možnost nemají existují také, ale jsou v současné době poměrně vzácné. Takovému přístupu k plánování říkáme **kooperativní**.

[47] Na specializované plánovače (např. pro systémy reálného času) můžeme klást další požadavky, které jsou ale mimo záběr tohoto předmětu.

- všechna vlákna by měla dostat v průměru stejný díl procesorového času (resp. poměrný podle nastavené priority),
- totéž platí o procesech, skupinách procesů, uživatelích, atp.,
- důležité pro oba typy programů – zabezpečuje jak rozumně brzké dokončení (pro výpočetní programy), tak rovnou příležitost reakce na požadavky (pro ty interaktivní).

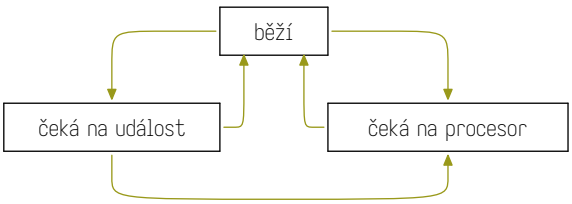
Je na první pohled vidět, že některé požadavky jsou protichůdné: zejména latence a propustnost často stojí proti sobě, a zlepšení jednoho parametru vede ke zhoršení toho druhého.

2.3.3 Stav vlákna Vlákno (které ještě neskončilo) může být, z pohledu plánovače, ve třech základních stavech:

- běží – má přidělen procesor,
- čeká na procesor / připraveno
- čeká na událost / spí
- běží ↔ připraveno: rozhoduje plánovač
- běží → spí: rozhoduje vlákno

1. právě běží – má aktuálně přidělený procesor,
2. připraveno ke běhu, ale **čeká na procesor** – vlákno bylo přerušeno plánovačem (nebo nebylo dosud nikdy spuštěno), ale jinak mu nic nebrání pokračovat ve výpočtu,
3. **čeká na událost** – vlákno je „zaseknuté“ v **synchronizační smyčce** a nemůže pokračovat ve výpočtu, dokud nenastane nějaká externí událost (plánovači musí být tento stav explicitně oznámen, jinak jej nemá jak rozeznat od běžného výpočtu).

Mezi těmito základními stavy jsou povoleny tyto přechody:



1. běží → čeká na procesor: plánovač rozhodl o odebrání procesoru,
2. čeká na procesor → běží: plánovač rozhodl o přidělení procesoru,
3. běží → čeká na událost: vlákno signalizovalo, že čeká na událost, případně to signalizovala jiná část operačního systému, od které si vlákno vyžádalo nějakou službu, a která vyžaduje vnitřní synchronizaci,
4. čeká na událost → čeká na procesor: událost nastala, ale není právě k dispozici žádný fyzický procesor,
5. čeká na událost → běží: událost nastala a byl k dispozici procesor, vlákno bylo tedy probuzeno.

Zejména se tedy nemůže stát, že by vlákno, které čekalo na procesor, přešlo do stavu čekání na událost – tento přechod totiž vyžaduje nějakou akci, kterou musí provést samotné vlákno.

2.3.4 Fronta úloh Základním pracovním nástrojem plánovače jsou běhové **fronty úloh** (angl. **run queue**), do kterých plánovač řadí vlákna, která čekají na procesor. Konkrétní plánovací algoritmy se pak liší zejména v tom, jak se tyto fronty chovají. Fronta může být:

- vlákna čekající na procesor
- prioritní nebo FIFO (round robin)
- lokální (vlastní procesoru) nebo globální
- rozhoduje o chování plánovače

1. podle vztahu front a procesorů:
   - jedna **globální** (společná pro všechny fyzické procesory), nebo
   - mnoho **lokálních** (každý procesor má vlastní frontu),
2. podle pořadí vláken:
   - FIFO (první příchozí je první obsloužen) nebo
   - prioritní (vlákna ve frontě mohou „předbíhat“ podle nějakých kritérií, např. priority),
3. prioritní pak dále podle implementace:
   - monolitická (binární halda, červeně-černý strom),
   - složená (každá prioritní třída má vlastní FIFO).

Konkrétní realizace fronty má významný dopad na chování plánovače, protože rozhoduje jak o tom, které vlákno poběží další, tak o tom, na kterém fyzickém procesoru.

2.3.5 Afinita Protože migrace vlákna na jiný fyzický procesor není zadarmo[48], je žádoucí zbytečným migracím vláken zamezit. Mluvíme v takovém případě o **afinitě vlákna** ke konkrétnímu fyzickému procesoru: plánovač má snahu vlákno opakovaně plánovat na stejný fyzický procesor.
Mezní situace jsou:

- vlákna lze libovolně přesouvat
  - fyzické procesory jsou ± záměnné
  - přesun vlákna je ale drahý
- zamezení přesunů → afinita
- v napětí s dobrým využitím zdrojů
  - lokální fronty + kradení práce

1. plánovač zcela bez afinity: např. proto, že má globální frontu a tedy je prvnímu čekajícímu vláknu vždy přidělen první uvolněný procesor,

[48] Souvisí to zejména s obsahem mezipamětí, a skutečností, že tyto obecně nejsou mezi procesory sdílené.

2. plánovač bez možnosti migrace: např. proto, že má lokální fronty, a neumožňuje přesun vláken mezi nimi.

Kompromisní řešení (vždy s lokálními frontami):

1. srovnávání front: při odebrání procesoru zařazuje plánovač vlákna přednostně do méně zaplněných front,
2. kradení práce: procesor, který má prázdnou frontu (nemá žádné vlákno, které by mohl spustit) „ukradne“ vlákno z nějaké jiné fronty (tento přístup vede na lepší využití zdrojů a méně přesunů → obvykle lepší řešení).

V obou případech je nutné zabezpečit, aby mohly dva procesory (tzn. dvě souběžné aktivace plánovače) bezpečně přistupovat ke stejné frontě (tím se implementace značně komplikuje).[49]

- umožňují preferovat některá vlákna
- staticky → rozhodnutím uživatele
- dynamicky → v závislosti na chování

2.3.6 **Prioritní fronty** Interaktivní plánovače[50] používají téměř výhradně nějakou formu prioritního plánování – vlákna mají přidělenu prioritu (staticky nebo dynamicky), která ovlivňuje jejich schopnost získat procesor – a to jak latenci (jak dlouho musí vlákno čekat ve frontě) tak celkový přidělený výpočetní čas (jaký díl procesorového času je vláknu přidělen).
Klasickým řešením je pevná množina prioritních tříd, kde v rámci každé priority jsou vlákna uspořádaná do klasické (FIFO) fronty. Vlákno, které poběží jako další, se vybere z nejvyšší neprázdné fronty. Toto řešení má několik výhod:

- používá pouze jednoduché datové struktury,
- všechny operace jsou asymptoticky konstantní (vzhledem k počtu čekajících vláken – počet prioritních tříd je pevný).

A také jednu důležitou nevýhodu:

- dlouho běžící výpočetní vlákno s vysokou prioritou zablokuje procesor pro všechna ostatní vlákna.

Tuto nevýhodu lze vyřešit dynamickou úpravou priority: za každou přidělenou jednotku výpočetního času je vlákno „potrestáno“ snížením priority. Existují-li jiná čekající vlákna, dříve nebo později se tak dostanou ke slovu.

- rovnoměrné rozdělení zdrojů
- podobné dynamickým prioritám
- priorita = inverzní využitý čas
- slabina → hladovění starých vláken

2.3.7 **Férové plánování** Cílem férového plánovače je, aby každé vlákno dostalo přiděleno podle možnosti stejné množství výpočetního času (případně váženo prioritou). V tomto se podobá na systém s dynamickou úpravou priority z předchozí sekce, ale místo pevného systému prioritních tříd má priority v libovolném rozsahu. Realizuje se proto klasickou (monolitickou) prioritní frontou – např. binární haldou nebo binárním vyhledávacím stromem.
Férové plánování lze tedy chápat jako formu prioritního plánování, kde priorita je inverzní hodnota dosud použitého výpočetního času. Případná statická priorita se použije jako multiplikativní faktor – vlákno s vyšší prioritou „platí“ za každou jednotku výpočetního času menším snížením priority.
Nevýhodou je, že v situaci, kdy je systém plně vytížen, jsou dlouhodobě běžící interaktivní procesy znevýhodněny – mají velmi nízkou prioritu, protože musely zaplatit za dosavadní výpočetní čas. Lze heuristicky kompenzovat, např. zvýšením priority jako „odměnou“ za interakci.

- jak dlouho běží aktuální vlákno?
- jakou má další vlákno prioritu?
- minimální přidělený čas → kvantum
- FIFO / RR → právě kvantum
- dobrovolné propuštění → spíš ne

2.3.8 **Odebrání procesoru** Rozhodnutí o odebrání procesoru (tzn. přesunu vlákna ze stavu „běží“ do stavu „čeká na procesor“) má dva základní vstupy:

1. jak dlouho již běží právě aktivní vlákno,
2. srovnání aktivního vlákna a následujícího vlákna ve frontě.

Konkrétní rozhodnutí opět závisí na konkrétním plánovacím algoritmu. Je ale obvyklé nechat aktivní vlákno běžet nějaký minimální čas, i v případě, že další naplánované vlákno má vyšší prioritu (tímto se brání příliš častému přepínání vláken) – tento minimální čas je **plánovací kvantum**. Delší kvantum má pozitivní vliv na propustnost, ale negativní vliv na latenci (reakční dobu).
Rovnocenná vlákna, která jsou mezi sebou řazena systémem FIFO, jsou rotována na procesoru po stejně dlouhých časových úsecích (není-li samozřejmě některé vlákno uspáno a tím procesor předčasně uvolněn).

[49] Souběžností se bude zabývat celý druhý blok kapitol.
[50] Jinými se v tomto předmětu ani nezabýváme. Jiné možnosti jsou dávkové a plánovače reálného času.

Dobrovolné propuštění procesoru: v moderních preemptivních[51] systémech prakticky nemá místo; je-li důvodem čekání na událost, je vždy lepší signalizovat přímo toto čekání.[52] Je-li důvodem dobrovolné snížení priority, lze lépe vyřešit přímo nastavením plánovací priority.

2.3.9 **Čekající vlákna** Vlákno je označeno jako **čekající na událost** (říkáme o něm také, že je **uspané**) buď na vlastní žádost (umožňuje-li to operační systém a povaha události), nebo na popud některé jiné části operačního systému:

- vyřízení požadavku
  - vlákno probouzí vyřizující
  - např. čekání na vstup
  - nebo čekání na data ze sítě
- soutěž o zdroj
  - čekací fronta
  - při uvolnění probuzeno první
  - např. kritická sekce

- vyžádá-li například vlákno čtení ze souboru, a operační systém nemá aktuálně data k dispozici, zařádí požadavek na data do fronty a vlákno uspí,
- pokusí-li se vstoupit do kritické sekce, která je aktuálně zamčená jiným vláknem,
- vyžádá vstup z klávesnice (a zrovna není žádný k dispozici),
- čeká na data ze sítě,
- atd.

Tyto události jsou v zásadě dvou typů:

- čekání na vyřízení „soukromého“ požadavku: takto uspané vlákno je poznačeno u příslušného požadavku, a jakmile je tento vyřízen, odpovědná komponenta vlákno probudí (aktivuje plánovač, který vlákno přesune do **běhové fronty**, nebo mu ihned přidělí procesor),
- soutěž o nějaký zdroj (např. zmiňovaná kritická sekce), který může vlastnit v danou chvíli nejvýše jedno vlákno, ale na který jich může zároveň čekat několik: takové zdroje mají **čekací frontu** (angl. **wait queue**) a příslušná událost probudí (přesune do **běhové fronty**) pouze první vlákno z čekací fronty: nemá smysl probouzet všechna, protože by okamžitě všechna krom jednoho musela být zase uspána (vyhrát soutěž může nejvýše jedno).

Část 3: Souborové systémy

Tato kapitola se zabývá ukládáním a organizací dat na pevných úložištích – na úrovni operačního systému k tomu slouží zejména souborové systémy.

Doporučené čtení: A. Tanenbaum, H. Bos – Modern Operating Systems (4th Ed.): § 4.1 Files ◇ § 4.2 Directories ◇ § 4.3.2 Implementing Files ◇ § 4.3.3 Implementing Directories ◇ § 4.3.6 Journaling File Systems ◇ § 4.4.1 Disk-Space Management ◇ § 4.4.3 File System Consistency.

3.1: Bloková zařízení

Tento typ zařízení představuje abstrakci perzistentních úložišť. Operace (abstraktního) blokového zařízení jsou přizpůsobené běžným schopnostem odpovídajících reálných zařízení:

- zápis a čtení je prováděno po blocích pevné velikosti,
- v libovolném pořadí, ale
- s velkou latencí a malou propustností (relativně k operační paměti).

3.1.1 **Trvalé úložiště** je fyzické zařízení, které se podobá na operační paměť (pamatuje si data), s několika klíčovými rozdíly:

- zařízení, které dlouhodobě ukládá data
- je pomalejší (zejména latence)
- operace nad bloky (512+ bajtů)
- ukládá zejména uživatelsky zajímavá data
- např. SSD, HDD, NVMe

1. data zde uložená přetrvávají „dlouhodobě“, tedy zejména i po vypnutí počítače (nebo restartu operačního systému),
2. přístup k datům je pomalejší – zejména má mnohem vyšší latenci (prodlevu) mezi vystavením požadavku a odpovědí zařízení; je proto nepraktické adresovat takové zařízení po jednotlivých bajtech (jako tomu je u operační paměti),
3. proto je obvyklé, že poskytují operace, které pracují s celými bloky dat najednou (velikost 512 bajtů a víc, často 4 KiB) – základní operací je pak přesun takového bloku mezi zařízením a operační pamětí.

Protože se sice jedná o typ paměti, ale s vlastnostmi značně odlišnými od paměti operační, obvykle se tento typ zařízení využívá také k výrazně jinému účelu:

[51] Kooperativní systémy samozřejmě spoléhají na dobrovolné uvolnění procesoru. Kooperativní plánování je běžné i v moderních programech, nikoliv ale na úrovni operačního systému – programy, které jsou vnitřně asynchronní, protože např. odpovídají na mnoho externích požadavků, mají vlastní vnitřní plánovače, které jsou obvykle právě kooperativní.

[52] Výjimečně se může stát, že potřebný typ události není v operačním systému podporován. V takové situaci může dobrovolné propuštění procesoru alespoň částečně redukovat neefektivitu spojenou s aktivním čekáním (tzn. použitím „zamlčené“ synchronizační smyčky).

- smyslem perzistentního úložiště je ukládání dlouhodobě užitečných dat – např. vstupů pro výpočty a jejich výsledků, nebo jiných „uživatelsky zajímavých“ dat – dokumentů, obrázků, audio a video záznamů, atp.,
- naopak se nepoužívají pro ukládání uživatelsky nezajímavých mezivýsledků (k tomuto ve většině případů slouží operační paměť).[53]

Spadají sem zejména magnetické disky (HDD), permanentní polovodičová úložiště (SSD, NVMe), výměnné magnetické a částečně i optické disky,[54] ale nikoliv třeba magnetické pásky.[55]

- souvislý adresní prostor
- adresa náleží celému bloku
- načtení (celého) určeného bloku
- zápis (celého) určeného bloku

- prodleva mezi požadavkem a vyřízením
- nelze skrývat hardwarově
- problém musí řešit OS
- skrytí latence vs spolehlivost

- skrývá latenci (podobné CPU vs RAM)
- nedávno čtené bloky zůstávají v RAM
  - které přesně to budou → politika
- implementuje ji operační systém
- nemá vliv na spolehlivost (jen rychlost)

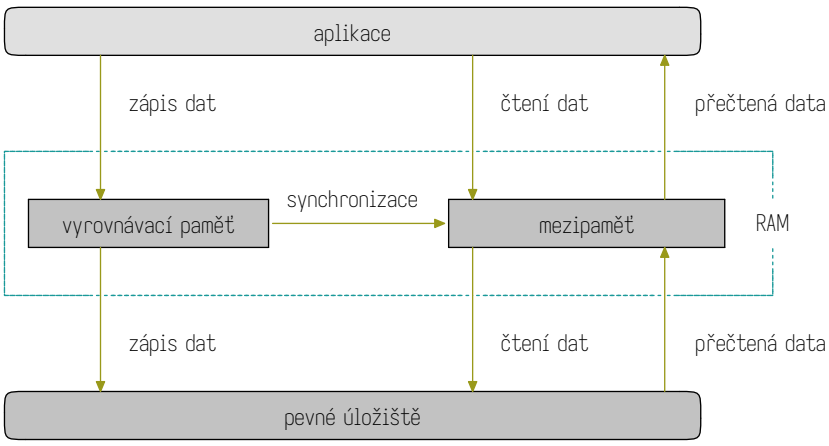
3.1.2 **Blokové operace** (rozdíl proti operační paměti; přenos dat mezi zařízením a operační pamětí po větších celcích; odpovídá fyzické struktuře zařízení; asynchronní komunikace, protože → latence)

3.1.3 **Latence** doba mezi požadavkem a odpovědí; příliš velká, než aby bylo lze ignorovat; rozdíl proti operační paměti: nelze skrýt hardwarově, musí řešit OS; problém zejména pro spolehlivé uložení dat – **durability** v ACID – nutné čekat na potvrzení zápisu

3.1.4 **Mezipaměť** Hlavní problém, který mezipaměť řeší, je opakovaný přístup ke stejným adresám daného datového úložiště. Přístup k datům obvykle není rovnoměrný – některá data jsou potřebná často (např. aktivně využívaná databáze), zatímco jiná (archiv dat z minulého roku) jen velmi zřídka. Proto je výhodné si nedávno čtená data pamatovat v rychlejší paměti: může se totiž lehce stát, že je bude potřeba přečíst v blízké budoucnosti znovu.
Podobně má smysl některá data načíst do mezipaměti **s předstihem** (existuje-li volná přenosová kapacita; této technice se angl. říká **prefetch**). Příklad: při sekvenčním čtení většího souboru (třeba během přehrávání videa) má smysl načítat následující bloky i v případě, že je aplikace ještě nevyžádala (v očekávání, že tak brzo učiní). Lze tak mimo jiné předejít situaci, kdy je po nějakou dobu přenosová kapacita nevyužitá, a následně se sejde více požadavků, na které kapacita nedostačuje.
V případě mezipaměti trvalých úložišť je onou „rychlou“ (a malou) pamětí paměť operační a pomalou (a velkou[56]) pak samotné trvalé úložiště. Samotná správa mezipaměti je v tomto případě realizována softwarově (operačním systémem).

- ukládá data určená k zápisu
  - opačný směr než mezipaměť
  - „vyrovnává“ pomalý zápis dat
- při výpadku jsou data ztracena
- synchronizace s mezipamětí
- také: vnitřní dočasná paměť zařízení

3.1.5 **Vyrovnávací paměť** Podobný problém (vysoká latence, malá propustnost) v opačném směru, tzn. od aplikace k trvalému úložišti, řeší operační systém tzv. vyrovnávací pamětí. Tato uchovává požadavky na zápis, které dosud nebyly plně vyřízeny (aplikace ale obvykle může pokračovat ve své činnosti bez dalšího čekání).



Existují dvě základní možnosti, jak vyrovnávací paměti realizovat:

1. oddělené od mezipaměti: vyrovnávací paměť obsahuje jak samotné požadavky, tak veškerá data s nimi spojená; jsou-li některé dotčené bloky uložené v mezipaměti, tyto jsou buď zneplatněny, nebo (častěji) upraveny na místě,

[53] Podobně jako v případě externího stránkování, lze perzistentní úložiště využít pro dočasná data, ale je to spíše kompromis vynucený nedostatkem operační paměti, nebo vhodných komunikačních primitiv.

[54] Zápis na optické disky je obvykle komplikovaný a neumožňuje přepis už zapsaných částí – takové systémy tedy chápeme jako perzistentní úložiště dostupné pouze pro čtení. Zápis je pak obvykle řešen jiným mechanismem.

[55] Magnetické pásky neumožňují přístup k datům v libovolném pořadí: páska je sekvenční médium a rychlost a přesnost přesouvání hlavy je tak nízká, že je i relativně benevolentní abstrakce blokových zařízení v tomto případě neudržitelná.

[56] Ve skutečnosti ale klasická poučka o velikostech v současném světě úplně neplatí. Není těžké najít systémy, kde je velikost operační paměti srovnatelná s připojenými trvalými úložišti.

2. propojená s mezipamětí: vyrovnávací paměť obsahuje pouze frontu požadavků na zápis – samotné datové bloky se ukládají vždy do mezipaměti, fronta nevyřízených zápisů do ní pouze odkazuje; takto koncipovaný systém se, jako celek, chová jako mezipaměť s opožděným zápisem (angl. write-back cache).

Krom mezipaměti a vyrovnávací paměti implementované operačním systémem existují také paměti obou typů na straně zařízení (tzn. umístěné na stejné straně sběrnice, jako samotné trvalé úložiště), kde plní podobné úlohy – snižovat dopad latence hlavního úložiště. Tento typ dočasných pamětí nemá na operační systém příliš velký dopad[57] a nebudeme se jimi tedy podrobněji zabývat.

- sekvenční přístup je rychlejší
- operace mají vysokou míru souběžnosti
- přeuspořádání → vyšší propustnost
- jednodušší pro zápisy

3.1.6 **Plánování operací** U většiny úložišť platí, že sekvenční přístup (čtení po sobě následujících adres) je mnohem rychlejší, než nahodilý přístup (postupné čtení adres, které spolu nijak nesouvisí). Různé technologie navíc kladou další omezení na rychlost reakce – klasickým příkladem jsou rotační disky, kde prodleva další operace závisí na vzdálenosti hlavy od místa, kde se potřebná data fyzicky nachází. Naopak polovodičové paměti jsou často složené z nezávislých celků, které mohou pracovat souběžně, ale požadavky do stejné části paměti musí vyčkat na dokončení těch předchozích.
Bez ohledu na technologii ukládání dat ale platí, že operace s trvalým úložištěm vykazují vysokou míru souběžnosti: do systému přichází mnoho nezávislých požadavků na diskové operace a existuje tedy jistá volnost odpovídat na tyto požadavky v různém pořadí.[58] Je to způsobeno zejména tím, že je často spuštěno několik (souběžných) vláken a každé z nich provádí vstupně-výstupní operace nezávisle na ostatních.

Příklad: uvažme situaci, kdy 3 souběžná vlákna provedou postupně každé 3 operace zápisu na postupně se zvyšující adresy, které do operačního systému např. dorazí v tomto sledu:

vlákno	1	2	3	4	5	6	7	8	9
A	A_1		A_2			A_3			
B		B_1			B_2			B_3	
C				C_1			C_2		C_3

Bude-li systém provádět operace v pořadí přijetí, výsledkem bude 9 operací o velikosti jednoho bloku: $A_1 - B_1 - A_2 - C_1 - B_2 - A_3 - C_2 - B_3 - C_3$.
Bude-li ale místo toho ukládat požadavky do fronty (vyrovnávací paměti) a provádět je až s určitou prodlevou, není problém je přeuspořádat do pořadí $A_1\ A_2\ A_3 - B_1\ B_2\ B_3 - C_1\ C_2\ C_3$ – jako 3 souvislé zápisy každý o velikosti 3 bloky.

Přeuspořádání operací má ale určitá úskalí, zejména v případech, kdy jsou prohozeny zápisy, které náleží stejné aplikaci, nebo se jedná o operace související s údržbou metadat souborového systému. Vzniká tak napětí mezi výkonem (více přeuspořádání → vyšší propustnost) a spolehlivostí (více přeuspořádání → větší šance nekonzistence[59] v případě výpadku).
Pokud jde o operace čtení, zde je prostor pro přeuspořádání obvykle menší, protože aplikace, která čtení vyžádala, obvykle nemůže pokračovat ve své činnosti dříve, než jsou data k dispozici (a tedy zejména nemůže vyžádat další operace čtení).[60]

- izolace vs uživatelská data
  - vyšší úroveň abstrakce, nebo
  - virtualizace na aplikační úrovni
- jeden program odpovědný za úložiště
  - operace určené tímto programem

3.1.7 **Problémy virtualizace** Náš dosavadní přístup k virtualizaci nebude v případě trvalých úložišť příliš dobře fungovat. Vzpomeňte si, že v případě paměti a procesoru řešení spočívalo ve vytvoření soukromých (virtuálních) instancí příslušného zařízení. Taková instance je pak vždy ve výlučném užívání jednoho programu.
Zde ale narážíme na to, že trvalé úložiště slouží především k ukládání **uživatelsky zajímavých** dat. Je tedy potřebné, aby mohl uživatel s těmito daty nějak interagovat, a zejména tedy nemůžou být skryta v soukromém prostoru jednoho programu.[61]
Nabízí se zde dvě možnosti řešení:

[57] Může hrát roli zejména v případě, kdy ovlivní pořadí trvalého zápisu dat. Přesné důsledky takového chování jdou ale mimo rámec tohoto předmětu.
[58] Souběžnost má v tomto kontextu tentýž význam, jak jsme jej definovali v úvodním přehledu pojmů. Operace považujeme za nezávislé – souběžné – nejsou-li v relaci předcházeni uspořádané (bez ohledu na to, v jakém pořadí do operačního systému dorazí).
[59] Problémy spojené s konzistencí dat detailněji probereme ve druhé části této kapitoly.
[60] Asynchronní čtení z lokálního souborového systému – tzn. přístup, kdy aplikace pokračuje v činnosti a dostupnost dat je jí později oznámena jako vnější událost – není příliš rozšířené a jde mimo rámec tohoto předmětu.
[61] Zejména uvážíme-li, že pro přístup k těmto datům a jejich organizaci uživatel obvykle využívá nějaký specializovaný program (můžeme ho pracovně nazvat **prohlížeč souborů**). Kdybychom zařídili, že každý program bude mít zcela soukromou virtuální instanci trvalého úložiště, neuvidí uživatel v prohlížeči souborů žádná data (musel by je tam uložit samotný prohlížeč).

1. můžeme zcela změnit přístup k virtualizaci: místo toho, abychom virtualizací vytvořili věrný obraz skutečného zařízení, může operační systém poskytovat nějaký jiný druh abstrakce, a trvalé úložiště nebude aplikacím vůbec přístupné,
2. vyčleníme nějakou aplikaci, která bude odpovědná za správu uživatelských dat v trvalém úložišti, a ostatní aplikace budou své požadavky na interakci s těmito daty řešit skrze tuto speciální aplikaci.

Obě řešení jsou ve skutečnosti (s trochou nadsázky) vlastně totéž řešení. V obou případech je přímý přístup k úložišti omezen na jeden nebo několik málo programů (v prvním případě je tento program součástí operačního systému), tento program rozhoduje o tom, jak budou data „fyzicky" organizovaná, jaké operace lze nad daty provádět, které další programy k nim budou mít přístup, atp.

- souborový systém (poskytuje OS)
  - nejběžnější a nejobecnější
- na úrovni aplikace (RDBMS)
- izolovaná virtuální instance
  - zejména virtualizace OS

3.1.8 **Metody virtualizace** Ve výsledku tedy existují tři základní metody virtualizace trvalých úložišť:

1. souborový systém, nebo obecněji abstrakce na vyšší úrovni poskytovaná jako služba operačního systému, která umožňuje řízený přístup k datům mnoha aplikacím současně, a zároveň umožňuje uživateli data organizovat a spravovat,
2. virtualizace na aplikační úrovni, kdy je přímý přístup k úložišti poskytnut některé aplikaci, která spravuje data a ostatním aplikacím poskytuje přístup pomocí vhodného aplikačního protokolu (typickým příkladem jsou databázové systémy),
3. analogicky k operační paměti, vytvořením virtuální soukromé instance – jak již bylo naznačeno, v obvyklých situacích není příliš užitečné, ale nachází uplatnění v oblasti virtualizace operačních systémů a v menší míře jako podpůrný mechanismus strategie z 2. bodu.

- trvalá úložiště jsou nespolehlivá
  - obnovení záloh je časově náročné
- RAID = Redundant Array of Inexpensive Disks
  - živá replikace dat napříč úložišti
  - existuje v několika variantách
- systém může fungovat i po selhání úložiště

3.1.9 **RAID** Zatímco standardním řešením problému s nízkou propustností a/nebo velkou prodlevou paměťových operací jsou mezipaměti, standardním řešením problémů se **spolehlivostí** paměti je **redundance**. Nejjednodušší formou redundance je pořízení kopie (zálohy) – v případě ztráty primárních dat obnovíme data ze záložní kopie. S tím jsou spojeny dva problémy:

1. aby byla užitečná, musíme takovou kopii udržovat aktuální a zároveň konzistentní,
2. při poruše musíme vyměnit dotčené zařízení a data překopírovat ze zálohy, přitom obě operace mohou být poměrně zdlouhavé.

Technologie RAID nabízí řešení obou problémů. Je postavena na systému „obrácené" virtualizace v tom smyslu, že spojuje několik fyzických zařízení (pevných úložišť) do jednoho pomyslného (virtuálního). Hlavní charakteristikou je rovnoměrné rozložení dat mezi všechna fyzická zařízení, ve většině konfigurací s nějakou mírou redundance (výjimkou je RAID úrovně 0). Tato redundance umožňuje zvýšit celkovou spolehlivost systému – ztráta jednoho fyzického zařízení neznamená okamžitou ztrátu dat, protože chybějící data lze nahradit nebo dopočítat z těch zbývajících. Systém tak může zůstat v provozu, a po výměně vadného zařízení lze ztracenou redundanci obnovit průběžně.

RAID pracuje na úrovni blokových zařízení a může být implementován jak hardwarově tak softwarově, přitom softwarové implementace jsou v současných systémech běžnější. Softwarový RAID je součástí blokové vrstvy operačního systému a vyšším vrstvám (zejména souborovému systému) je prezentován jako jediné ucelené virtuální úložiště. Čtení a zápis z takového virtuálního zařízení způsobí, že podsystém RAID rozdělí tyto operace mezi jednotlivá fyzická zařízení.[62] S výjimkou RAID 0 je pak takové zařízení schopno pokračovat bez ztráty dat (a bez přerušení) i v situaci, kdy dojde k výpadku jednoho z fyzických zařízení.[63]

RAID má samozřejmě dopad i na rychlost:[64] čtení je obvykle výrazně rychlejší, protože data jsou skládána z několika nezávislých zdrojů paralelně a každý z nich přispívá k celkové propustnosti. Situace se zápisem je komplikovanější: ten může být v závislosti na konfiguraci a okolnostech rychlejší nebo pomalejší.[65]

[62] Konfigurace nebo též úroveň RAID označuje způsob distribuce dat mezi jednotlivá fyzická zařízení. Běžně používané jsou RAID 0 (prokládání po blocích, bez redundance), RAID 1 (zrcadlení – každé zařízení má kompletní kopii dat), RAID 4 (prokládání po blocích + samostatný paritní disk), RAID 5 (totéž, ale parita je rovnoměrně rozložena mezi disky) a RAID 6 (totéž, ale parita je navíc zdvojená), RAID 1+0 / RAID 10 (prokládání po blocích, ale složky jsou virtuální zařízení typu RAID 1).

[63] Obsahuje-li RAID pole n zařízení, RAID 1 toleruje výpadek $n - 1$ zařízení, RAID 4 a RAID 5 výpadek jednoho zařízení, RAID 6 výpadek dvou zařízení. RAID 10 toleruje násobné výpadky postihnou-li různá vnitřní pole.

[64] Uvažujeme situaci, kdy je pole plně funkční – nechybí žádné fyzické zařízení, ani právě neprobíhá obnova redundance. V degradovaném režimu je rychlost většiny operací výrazně nižší.

[65] Faktor, který zápis zrychluje je rozdělení fixního množství dat mezi několik zařízení (platí pro RAID 0, 4, 5, 6 a 10), zpomalení je naopak způsobeno nutností zapsat více dat, aby byla udržena redundance (platí pro RAID 1, 4, 5, 6 a 10) a nutností počítat paritní bloky (RAID 4, 5 a 6).

3.1.10 Šifrování, integrita Další vlastností běžně přítomnou v moderních operačních systémech je šifrování dat na úrovni blokových zařízení. To chrání systém v situaci, kdy má útočník fyzický přístup k pevnému úložišti (například z odcizeného počítače).

- symetrické, zachovávající délku
- klíč podle hesla nebo z tokenu
- známé také jako „full disk encryption“
- relativně malá režie
- velmi důležité pro bezpečnost a soukromí

Tento typ šifrování dat pracuje na podobném principu virtuálního blokového zařízení jako RAID – fyzické zařízení obsahuje data v zašifrované podobě, přitom každý zápis a čtení z virtuálního zařízení data transparentně šifruje a dešifruje pomocí **symetrické** blokové šifry.[66] Tento typ šifrování zachovává velikost, zašifrovaný blok lze tedy přímo 1:1 uložit do bloku na fyzickém zařízení. Požadujeme-li navíc kontrolu integrity dat, tento předpoklad již neplatí – bloky opatřené kontrolními součty jsou větší než původní.[67]

Podobně jako RAID je šifrování pro zbytek systému transparentní – souborový systém nemusí mít o jeho existenci žádné zvláštní povědomí.

3.2: Soubor

Soubor je základní jednotkou organizace dat v souborovém systému – pro operační systém již nemá další vnitřní strukturu – operační systém obsah souborů neinterpretuje (jedná se pouze o souvislý blok bajtů, který lze číst a přepisovat, případně soubor jako celek zvětšovat a zmenšovat). Souborům se často říká také **i-uzly**, abychom předešli záměně s uživatelským chápáním pojmu „soubor“.

3.2.1 Operace Základními operacemi pro práci se souborem je čtení (v POSIX-u voláním read) a zápis (v POSIX-u write) souvislé posloupnosti bajtů (libovolné velikosti – abstrakce souboru skrývá blokový charakter zařízení, na kterém je soubor uložen). Podobně může být libovolná velikost souboru (samozřejmě ale musí soubor obsahovat celočíselný počet bajtů) – tuto velikost je navíc možné dynamicky měnit. Zápis „za konec“ souboru jej automaticky prodlouží, zkrácení je nutné explicitně vyžádat (v POSIX-u voláním ftruncate).

- soubor je abstrakce
- čtení a zápis po bajtech
  - nebo libovolně velkých blocích
- (automatické) prodloužení
- adresováno od 0 spojitě

Soubor tak představuje velmi jednoduchý adresní prostor (analogický k virtuálnímu adresnímu prostoru) – platné jsou právě adresy počínaje nulou až po velikost souboru. Podobně jako u operační paměti je motivací souboru paměť – abstraktní zařízení, které si pro každou platnou adresu pamatuje jeden bajt informace, kterou lze přečíst a/nebo upravit. Podobně jako u adresních prostorů svázaných s operační pamětí je soubor zobecněním této základní motivace – společným jmenovatelem zůstává práce s daty (vstup a výstup dat), ale většina operací je volitelná:

- soubor může být možné číst, ale nikoliv do něj zapisovat,
- nebo naopak, existují soubory, do kterých lze data zapsat, ale nelze je později vyzvednout (tzn. soubory, které nereprezentují paměť),
- soubory nemusí být možné libovolně adresovat, ale pouze je číst (nebo do nich zapisovat) sekvenčně.

Druhým důležitým rozdílem oproti virtuálnímu adresnímu prostoru je, že soubory jsou obvykle **perzistentní** – existují dlouhodobě, nezávisle na běžících procesech, nebo dokonce na tom, je-li vůbec aktivní operační systém (nebo samotný hardware). Abychom mohli s perzistentními soubory rozumně pracovat, musí být navíc tyto opatřeny **identitou**.

Abychom mohli se souborem pracovat, ve většině systémů je nutné jej **otevřít**[68] – v systémech POSIX k tomu slouží volání open, kterého výsledkem je **popisovač otevřeného souboru**[69] (angl. file descriptor).

3.2.2 Obyčejný soubor je právě oním „motivačním případem“ – obyčejný soubor reprezentuje paměť, tzn. jeho smyslem je uchovávat data (obyčejný soubor si tedy pamatuje posloupnost bajtů). Jaké konkrétní bajty to jsou je irelevantní – souborový systém obsah obyčejných souborů nijak neinterpretuje.

- všechny zmíněné operace
- ukládá data (paměť vs periferie)
- nemusí být zarovnán
- fyzické umístění řeší OS

Krom abstrakce (perzistentní – trvalé) paměti je obyčejný soubor také abstrakcí nad pevným úložištěm v tom smyslu, že skrývá detaily přístupu k tomuto zařízení. Operace nad pevným úložištěm pracují po jednotlivých blocích, přičemž bloky mají pevnou velikost a jejich počáteční adresa musí být dělitelná touto velikostí – pro soubory žádné takové omezení neplatí. Zároveň je aplikace odstíněna od fyzického umístění dat na pevném úložišti (funguje zde opět analogie

[66] Obvykle se používá hardwarově urychlená šifra AES v módu CBC nebo XTS.

[67] Konkrétními řešeními se zde nebudeme blíže zabývat – zájemce odkazujeme např. na dokumentaci modulu dm-integrity Linuxového jádra.

[68] Rozhraním pro práci se soubory se budeme blíže zabývat v kapitole 11.

[69] Tento pojem definovaný normou POSIX budeme používat univerzálně i pro analogické koncepty v jiných operačních systémech.

s virtuálním adresním prostorem a překladem virtuálních adres na fyzické). Analogie překladu adres je v tomto případě ale zcela v režii operačního systému.
Aby mohla tato abstrakce fungovat, musí souborový systém udržovat o každém souboru metadata, která mimo jiné obsahují informaci o tom, ve kterých blocích se nachází potřebná data (a v jakém pořadí). Když aplikace požádá o přečtení o nějaké části souboru, souborový systém příslušné bloky vyhledá a načte podle těchto metadat, podobně je-li nějaká část souboru přepsaná. Je-li zápis proveden na konci (tzn. soubor je prodloužen), souborový systém musí vyhledat volné bloky, tyto alokovat, data do nich zapsat a v metadatech poznačit, že náleží příslušnému souboru. Krom informace o fyzickém rozmístění dat si souborový systém o každém souboru pamatuje další údaje, např. kdo je vlastníkem nebo kdy byl soubor naposledy změněn.

- využití externího stránkování
- přístup k souboru jako k místu v paměti
- lze jak číst tak zapisovat
- menší režie než explicitní operace

3.2.3 Mapováni do paměti Operace read a write nejsou vždy efektivní, protože musí mimo jiné kopírovat data mezi mezipamětí a pamětí, která náleží žádajícímu procesu.[70] Efektivita aplikací, které data především čtou, může být výrazně zlepšena mapováním souborů do paměti za pomoci líného načítání (viz také 1.4.4).
V tomto režimu jsou externí stránky uloženy v jinak běžném souboru, ale místo komplikovaných vstupně-výstupních operací může program s obsahem souboru pracovat stejně, jako by byl uložen v operační paměti. Využijeme-li tohoto mechanismu také k zápisu změněných stránek zpátky do souboru, může program soubor i zcela transparentně upravovat.
Je-li soubor takto mapován do paměti, je navíc možné mít soubor v operační paměti uložen pouze jednou, a fyzické adresy, které používá mezipaměť, zároveň přímo namapovat do adresního prostoru procesu.

- k souboru lze přistupovat z více vláken
- pouze čtení → ok
- hazard souběhu → zamykání
  - čtení vs zápis
  - zápis vs zápis

3.2.4 Souběžný přístup Jmenný prostor souborového systému (blíže jej popíšeme v další sekci) je **sdílený** mezi všemi procesy, může se tedy lehce stát, že více procesů bude pracovat s jedním souborem. Tato situace je analogická tomu, že stejná oblast operační paměti může být namapovaná ve více virtuálních adresních prostorech. Jde-li o souběžný[71] přístup pouze pro čtení, nevznikají žádné významnější problémy. Vstoupí-li ale do hry zápisy, jak čtení tak jiné zápisy mohou způsobovat problémy – zejména různé instance hazardu souběhu.
Aby se těmto problémům předešlo, operační systémy umožňují soubory **zamykat**[72] čím je umožněno programům k souboru přistupovat bezpečně – tzn. bez rizika poškození dat – i v situaci, kdy by se se souborem mohl pokusit souběžně pracovat i jiný program (proces). V systémech POSIX k tomu slouží volání flock (uzamkne celý soubor najednou) a fcntl (umožňuje zamknout konkrétní rozsah bajtů).[2]

- typ obyčejného souboru
- program „v klidu“
- zejména text (instrukce)
- data (konstanty)
- parametry pro zavedení do paměti

3.2.5 Spustitelný soubor Z pohledu souborového systému není na spustitelném souboru nic zvláštního, nicméně na rozdíl od většiny ostatních obyčejných souborů je jeho obsah důležitý pro jiné části operačního systému.
Spustitelné soubory představují programy v klidu (tzn. ve stavu, kdy nejsou spuštěné) a obsahují veškeré informace, které jsou potřeba k tomu, aby bylo možné v nich obsažený program spustit. Hlavní část spustitelného souboru je tvořena počátečním **obrazem paměti**, podle kterého se při spuštění programu (v POSIX-u voláním exec v již existujícím procesu) inicializuje[73] virtuální adresní prostor. Tento obraz obsahuje zejména text programu (instrukce) a počáteční hodnoty globálních proměnných. Obsahuje také počáteční hodnoty speciálních registrů, zejména programového čítače.

- jednoduché komunikační zařízení
- jedna strana data (bajty) zapisuje
- druhá strana (jiné vlákno, proces) čte
- může ale nemusí mít jméno

3.2.6 Roura Roury se podobají na obyčejné soubory v tom, že je možné do nich zapisovat a číst z nich data (bajty). Ve většině případů data jeden program (proces) zapisuje a jiný je čte – na rozdíl od obyčejného souboru nejsou tato data nikde trvale uložena – z roury zmizí jakmile jsou přečtena.
S rourou je samozřejmě svázán buffer, ale je uložen pouze v operační paměti. Díky tomu lze do roury data zapisovat i ve chvíli, kdy je právě druhá strana nečte – operační systém zapsaná data dočasně uchová.
Za normálních okolností je roura anonymní a přístupná pouze skrze popisovače otevřených souborů. Jakmile jsou tyto uzavřeny, roura je zničena. V systémech POSIX existuje i tzv. **pojmenovaná**

[70] Výjimku tvoří případ, kdy je rozsah virtuálních adres, na které aplikace vyžádala přesun dat ze souboru, přesně zarovnaný (na obou koncích) na okraje stránek a zároveň je stejně zarovnaný čtený rozsah souboru. V takovém případě některé operační systémy místo kopírování dat příslušné stránky pouze přemapují tak, aby dotčené virtuální adresy ukazovaly přímo do mezipaměti (s využitím mechanismu copy-on-write).
[71] Souběžnost jsme si již přiblížili v sekci B.3, podrobněji se jí budeme zabývat v kapitolách 5 až 7.
[72] Analogicky k vzájemnému vyloučení a zařízení rwlock, kterými se budeme zabývat v sekci 6.1.
[73] V případě, že program spouštíme v existujícím adresním prostoru, tento je inicializací ze spustitelného souboru zničen.

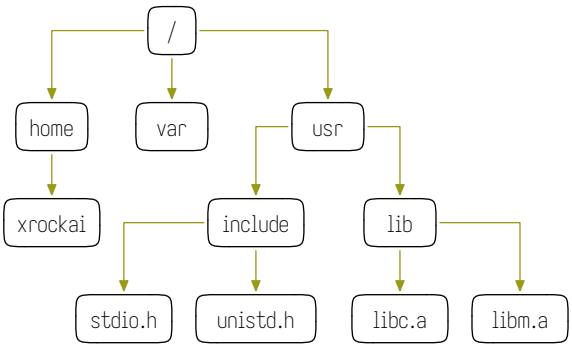
roura, která existuje v souborovém systému trvale a má zde přiřazeno jméno, podobně jako většina obyčejných souborů. To ale neznamená, že by **data** která rourou proudí byla kdekoliv trvale uložena – tento typ roury pracuje stejně, jako ta anonymní, liší se pouze v tom, že ji lze otevřít podobně jako obyčejný soubor.

3.2.7 **Zařízení** Mnoho periferií produkuje nebo konzumuje sekvence bajtů – tyto reprezentujeme typem souboru, kterému se říká **znakové zařízení** (angl. character device), který se chová podobně jako roura – data, která program zapisuje, jsou odeslána periferii, a data od periferie příchozí může program číst.
Uvažme například tiskárnu: zapíšeme-li nějaká data do znakového zařízení, které ji reprezentuje, tato budou tiskárnou vytištěna (jak bude výsledná stránka vypadat samozřejmě závisí na tom, jak tato data tiskárna interpretuje).
Jiným příkladem by mohl být skener – poté, co je mu uloženo naskenovat dokument, pixely které zachytí optický senzor jsou zakódovány a aplikace je může takto zakódované přečíst z příslušného znakového zařízení.
V obou případech se znaková zařízení chovají jako roury, jen místo jiného programu je na jejich opačném konci hardwarové zařízení (resp. firmware tohoto zařízení).

- „souborový“ ekvivalent periferie
  - abstrakce operačního systému
- většina zařízení čte/zapisuje po bajtech
  - nemusí být ale adresovatelné
  - trvalá úložiště po blocích
- terminály, tiskárny, atp. – příště

3.3: Adresářová struktura

Příklad:



3.3.1 **Cesta** Cesta nám umožňuje popsat pozici souboru nebo složky (adresáře) v adresářové struktuře (též adresářovém stromě).[74] Kořenová složka má cestu / (dopředné lomítko).[75] Každá adresářová položka nese nějaké **jméno** a odkaz na samotný soubor nebo podsložku, kterou reprezentuje – toto jméno může být v cestě použito jako označení příslušné odkazované entity (i-uzlu).
Máme-li například cestu /usr/include/…, začneme v kořenové složce (počáteční znak /), v této složce vyhledáme položku se jménem usr a když ji najdeme, ověříme, že odkazuje na další složku. Je-li tomu tak, nalezneme v ní položku s názvem include, atd.

- soubory označujeme cestami
- řetězec, znak / odděluje adresáře
  - může být jiný (např. \ on Windows)
- absolutní vs relativní cesta
- příklad: /usr/include, ./data

3.3.2 **Adresář (složka)** Složka je uzel adresářové struktury, který může mít další potomky. Úkolem složek je tyto potomky (obyčejné soubory, podsložky, atd.) **pojmenovávat** a tím je zpřístupnit za pomoci cest. Podobně jako obyčejné soubory jsou složky datovými objekty, které ale místo neinterpretovaných sekvencí bajtů obsahují strukturovaná data.[76] Složka zobrazuje jména na odkazy na další entity (i-uzly – soubory, složky, atd.) – chová se v tomto ohledu podobně jako slovník (asociativní pole).
Za normálních okolností se interakce uživatelských programů se složkami omezuje na dva typy použití:

1. pomocí cest, kdy např. volání open předáme cestu k souboru, a veškerá práce se složkami proběhne uvnitř operačního systému,

- funguje jako slovník
  - klíč = název
  - hodnota = odkaz na soubor
- operace
  - přidat, odebrat klíč
  - vyhledat odkaz dle klíče
- lze odkazovat i jiné složky

[74] Z historických důvodů mluvíme o stromě, i když ve skutečnosti se je ve většině současných operačních systémů jedná o acyklický orientovaný graf (neuvažujeme-li měkké odkazy, které mohou být i cyklické).
[75] Používáme zde opět konvenci normy POSIX.
[76] V zásadě by bylo možné i pro složky implementovat volání read a write, které aplikaci zpřístupní složku jako sekvenci bajtů, ale s tím jsou spojeny značné obtíže. Zpřístupníme-li takto aplikaci přímo diskovou reprezentaci složky, tato by mohla lehce datovou strukturu poškodit, což je krajně nežádoucí.

2. iterace, kdy si aplikace vyžádá seznam všech položek nějaké složky – pro tyto účely existuje speciální rozhraní.[77]

- stejný soubor může mít více jmen
  - jména jsou určena adresářem
  - označujeme jako tvrdé (pevné) odkazy
  - obvykle nelze více odkazů na složku
- nelze vytvořit mezi souborovými systémy

3.3.3 Tvrdé odkazy Jasným důsledkem výše popsané organizace složek je existence tzv. tvrdých odkazů[78] – situace, kdy několik adresářových položek (v jedné nebo několika složkách) odkazuje tutéž entitu (i-uzel). Všechny odkazy na tentýž soubor jsou zcela rovnocenné a z pohledu uživatele se pouze tentýž soubor objevuje na různých místech adresářové struktury.
I-uzly si navíc udržují počítadlo odkazů – samotný soubor (i-uzel) je zničen pouze v případě, kdy toto počítadlo dojde na nulu. To mimo jiné znamená, že odstraněním adresářové položky (angl. unlinking) může, ale nemusí způsobit smazání souboru. Za odkaz se počítá i popisovač otevřeného souboru – se souborem lze tedy pracovat i v situaci, kdy už na něj z adresářové struktury neexistuje žádný pojmenovaný odkaz.

- problém? přidejte úroveň nepřímosti
- měkký odkaz obsahuje cestu
- lze odkazovat i složky
  - může vzniknout cyklus
- cílová cesta nemusí existovat

3.3.4 Měkké odkazy Občas je užitečné odkazovat soubor nikoliv přímo, ale skrze nějakou cestu která k němu vede. Toho lze dosáhnout tzv. měkkým odkazem: tento je (na rozdíl od tvrdého odkazu) skutečným objektem v souborovém systému, který je reprezentován samostatným i-uzlem.[79] Narazí-li při procházení adresářové struktury (zejména při hledání i-uzlů podle cesty)[80] operační systém přečte cestu obsaženou v tomto měkkém odkazu a ve vyhledávání pokračuje touto cestou.[81] Zbývají-li v původní cestě nějaká nezpracovaná jména, na jejich zpracování dojde ve chvíli, kdy je ukončeno zpracování cesty nalezené v měkkém odkazu. Může se samozřejmě stát, že cesta uložená v měkkém odkazu není platná (nevede k žádnému i-uzlu) – v takovém případě skončí operace chybou.

- adresářová struktura je globální
- problémy: hazard souběhu, bezpečnost
- výhody:
  - předávání souborů mezi programy
  - orientace uživatele

3.3.5 Sdílení Je důležité si uvědomit, že adresářová struktura je **globální** – sdílená mezi všemi procesy.[82] To má určité nevýhody:

- sdílené zdroje jsou obecně náchylné na chyby způsobené hazardem souběhu – různé programy se mohou pokusit vytvořit soubor se stejným názvem, atp.,
- do hry vstupuje otázka zabezpečení dat – jak před neautorizovaným čtením, tak změnou, poškozením nebo zničením.

Zároveň se ale jedná o vlastnost velmi užitečnou:

- soubory vytvořené jedním programem lze přirozeně a přímočaře předat jinému – pro oba programy jsou dostupné pod stejnou cestou,
- uživatel vidí stejnou adresářovou strukturu ve všech programech, co mu zjednodušuje orientaci i organizaci dat,
- uživatelé si mohou jednoduše předávat soubory navzájem.

3.4: Implementace

Posledním podtématem této kapitoly je otázka jak se souborový systém realizuje, zejména jak jsou na trvalém úložišti organizovaná data a metadata souborů a složek.

- podobné jako klasické
- uzpůsobeny práci po blocích
- robustní vůči přerušení operace

3.4.1 Datové struktury Základním stavebním kamenem souborového systému, jako ostatně každého systému pro ukládání nebo správu dat, jsou datové struktury. Od těch klasických se liší v několika ohledech:

1. musí být přizpůsobeny tomu, že čtení a zápis jsou realizované po blocích, a prodleva mezi čtením bloků může být velká – je proto důležité minimalizovat počet bloků, které je potřeba pro realizaci dané operace načíst (nebo zapsat),
2. musí být robustní vůči přerušení operace – datové struktury souborového systému by měly být použitelné (nebo alespoň opravitelné) i v případě, že nějaká operace (zápis) je uprostřed nečekaně ukončená.

---

[77] V systémech POSIX je tvořeno voláními opendir/fdopendir, readdir, seekdir, closedir atd. Tato poskytují jednoduché vysokoúrovňové rozhraní pro čtení složek – přidávání a odebírání složek se provádí použitím cest (vytvořením složky nebo souboru s danou cestou, odstraněním souboru, atp.).

[78] Pozor, nejedná se zde o žádnou speciální entitu, pouze o popis situace.

[79] Ano, znamená to, že lze mít několik jmen pro tentýž měkký odkaz – situaci, kterou bychom popsali jako existenci tvrdého odkazu na měkký odkaz.

[80] Včetně situace, kdy se pokusíme měkký odkaz přímo otevřít.

[81] Tato cesta nemusí vést do téhož souborového systému, ve kterém existuje příslušný měkký odkaz. Taková situace při použití tvrdých odkazů nastat nemůže.

[82] V některých operačních systémech lze do jisté míry sdílení adresářové struktury omezit. Protože je ale sdílení jednou ze základních funkcí souborového systému, taková omezení jsou obvykle jen částečná (např. každý uživatel má vlastní složku na dočasné soubory, neviditelnou pro ostatní uživatele).

Nevhodné budou tedy struktury, které vyžadují mnoho navazujících operací s malým objemem dat (např. klasický zřetězený seznam), nebo vyžadují složité operace, které mohou strukturu zanechat v silně nekonzistentním stavu (např. klasické vyvažování stromu rotacemi, kdy je část stromu dočasně odpojena – takto odpojený podstrom by se lehce mohl navždy ztratit).

- velmi jednoduchá
- 1 bit o každém bloku
- kompaktní → rychlé hledání
- robustní zápis (bez odkazů)
- použití: hledání volného místa

3.4.2 **Bitmapa** V souborových systémech se využívá několik velmi jednoduchých datových struktur, které jsou díky své jednoduchosti zároveň relativně robustní.[83] Asi nejjednodušší strukturou tohoto typu je **bitmapa**, která se používá k mapování využitých resp. volných bloků nebo řádků v tabulkách (přiblížíme si za chvíli).

Bitmapa udržuje informaci o lineárním sledu bloků a informace o jednotlivém bloku se omezuje na jediný bit (obvykle právě ona využitost). Je-li velikost bloku 4 KiB, jeden blok bitmapy obnáší 32768 bitů a pokrývá 128 MiB úložiště (každý bit shrnuje informace o celém datovém bloku). Díky tomu dokážeme velmi rychle prohledat velké oblasti, a to i přesto, že asymptoticky je hledání v bitmapě lineární.

Zároveň je bitmapa velmi robustní – jednotlivé bity jsou pevně mapované na konkrétní datové bloky, při zápisu se tak v podstatě nemá co porouchat. Jediný potenciální problém nastane, kdybychom potřebovali atomicky změnit bity, které spadají do různých bloků bitmapy.

- pole fixně velkých struktur
- alokace řádků často bitmapou
- odkaz číslem (indexem) řádku
- přirozeně robustní zápis

3.4.3 **Tabulka** Bitmapa je sice jednoduchá a rychlá, neumí ale uchovat příliš mnoho zajímavých informací. O něco málo složitější datovou strukturou je **tabulka**, která v podstatě odpovídá klasickému poli:

1. je to souvislá oblast fixně velkých struktur (řádků, položek), zvolených tak, aby se jich do jednoho bloku vešel celočíselný počet (vyhovuje například velikost jednoho řádku 128 nebo 256 bajtů),
2. alokace položek se provede např. bitmapou (určuje které řádky jsou resp. nejsou použité, aby bylo v případě potřeby možné rychle najít řádek, do kterého můžeme zapsat novou položku),
3. chceme-li se do takové tabulky odkázat (např. proto, že se jedná o tabulku záznamů o souborech), stačí nám znát číslo řádku (index).

Tabulky se využívají např. pro ukládání informací (metadat) o jednotlivých souborech – v takovém případě se jejím jednotlivým řádkům obvykle říká i-uzly. Je také obvyklé, že tabulek stejného typu je na pevném úložišti rozmístěno několik, aby metadata (řádky tabulky) a data (datové bloky, které tabulka případně odkazuje) nebyly fyzicky v datovém úložišti příliš daleko od sebe.

Pro jednoduchost jsou tabulky obvykle alokované pevně, tzn. při vzniku souborového systému se určí na kterých adresách budou umístěny které tabulky a kolik budou mít položek. Toto zjednodušení je vykoupeno menší efektivitou využití místa, a také rizikem, že v pevně alokovaných tabulkách dojdou volné řádky dříve, než dojdou volné datové bloky.

- vyhledávací strom
- samovyvažovací (log. hloubka)
- vysoká arita → malá hloubka
- složitá implementace, není robustní
- univerzální

3.4.4 **B-strom** Tato datová struktura představuje proti dvěma předchozím zásadní zlom – jedná se totiž o strukturu značně sofistikovanou. Přesný popis jejího fungování jde nad rámec tohoto předmětu, proto si ji popíšeme jen v hrubých rysech. B-strom je

1. n-ární vyhledávací strom, tzn. každý uzel má až n potomků, kde n je relativně velké číslo (desítky až stovky),
2. je samovyvažovací, tzn. bez ohledu na to, v jakém pořadí přidáváme nebo ubíráme klíče, má hloubku logaritmickou vůči celkovému počtu klíčů (a zároveň i přidání i ubrání klíče má logaritmickou složitost),
3. je optimalizovaný pro vysokou latenci blokových operací – zejména tím, že velikost uzlu je stejná jako velikost bloku, z čeho také plyne vysoká arita a tedy i malá výška stromu (v poměru k počtu klíčů).

Protože se jedná o relativně složitou strukturu, která obsahuje vnitřní odkazy, může být těžké udržet ji při zápisu konzistentní – to se často řeší tím, že se uzly nepřepisují na místě, ale vytvoří se nové verze na jiném místě a dotčené rodičovské uzly se rekurzivně opraví stejným způsobem.

Využití B-stromů je široké – mohou nahradit bitmapy i tabulky, včetně dalších pomocných struktur (tzn. je možné v nich uchovávat informace o souborech nebo o volném místě), zároveň mohou reprezentovat adresáře.[84]

[83] To neznamená, že souborový systém vystavěný z takových struktur je imunní vůči poškození – metadata uložená v různých datových strukturách musí být také vzájemně konzistentní.

[84] Je možné implementovat plnohodnotný souborový systém, který nepoužívá žádné jiné datové struktury než B-stromy.

- tři základní typy porušení
  a. jednotlivá datová struktura
  b. mezi strukturami vzájemně
  c. mezi daty a metadaty
- přerušení nebo přeuspořádání
- možnost detekce a dodatečné opravy

3.4.5 Konzistence Důležitým problémem souborového systému je udržení konzistence metadat (jak různých typů nebo složek metadat mezi sebou, tak se samotnými daty). Porušení konzistence může být trojího typu:

1. narušení jednotlivé datové struktury, např.
   - B-strom obsahuje uzel, který ve skutečnosti není platným uzlem, např. proto, že nový odkaz byl zapsán dříve, než odkazovaný uzel,
2. konfliktní informace v různých datových strukturách, např.:
   - bitmapa označuje řádek tabulky za volný, ale tento je zároveň vyplněn smysluplnými metadaty, nebo
   - datový blok je označený jako volný (v bitmapě nebo B-stromě), ale zároveň je odkazován jako součást nějakého souboru,
3. nesoulad mezi metadaty a datovými bloky – např.
   - podle metadat je vlastníkem souboru uživatel B, ale odkazované datové bloky obsahují data uživatele A.[85]

Existují dva hlavní důvody, proč by k podobným situacím mohlo dojít:

1. přerušením kritické operace, která provádí více souvisejících změn, např. výpadkem napájení, nebo kritickou chybou („pádem“) celého systému,
2. i v případě, kdy je souborový systém vůči takovému přerušení robustní, tato jeho vlastnost může být narušena přeuspořádáním zápisů (plánovačem nebo samotným zařízením).

Existuje několik metod, jak se s problémem vypořádat. Jednou je detekce problému (např. příznakem, který se zapíše při korektním ukončení operačního systému) a následná křížová kontrola všech metadat v situaci, kdy mohlo k poškození teoreticky dojít. Tato kontrola může a nemusí být schopna souborový systém vrátit do konzistentního stavu (v závislosti na rozsahu resp. povaze poškození).

- sekvence záznamů o plánovaných akcích
- jednoduchá robustní struktura na disku
- transakční zpracování
- snižuje riziko poškození metadat
- rychlejší zotavení z havárií

3.4.6 Žurnál Sofistikovanější možností jak se s problémem vypořádat je tzv. žurnál, který ve své nejběžnější podobě obsahuje sekvenci záznamů o akcích, které se mají provést (angl. write-ahead log případně intent log).
Oproti klasickým metadatům souborového systému má žurnál jednoduchou strukturu – záznamy jsou na disku uloženy sekvenčně (obvykle „do kruhu“ – nový záznam přepíše nejstarší, už neplatný, záznam). Tato struktura je velmi robustní, jak vůči nahodilému přerušení, tak proti typickým vzorům přeuspořádání.[86]
Záznamy v žurnálu jsou obvykle seskupeny do **transakcí** (podobných těm, které znáte z relačních databázových systémů), které mohou sestávat z několika provázaných operací. Změny v metadatech se začnou na pevné úložiště posílat až ve chvíli, kdy je transakce ukončena a je potvrzen zápis[87] příslušných položek v žurnálu. Transakce, která není ukončená, se při obnově přeskočí, čím je zabezpečeno, že se ve výsledku provede buď celá, nebo vůbec.
Za cenu komplikovanější implementace (a určitého snížení výkonu) získáme:

1. silnější záruky konzistence při havárii nebo výpadku systému,
2. rychlejší zotavení z takové situace.

Přerušená operace na datové struktuře může vést k nejednoznačnému stavu, kdy lze datovou strukturu opravit více než jedním způsobem. Žurnál tento problém řeší tím, že existuje záznam o tom, jaká operace probíhala a tedy je možné ji na základě této informace dokončit. Zároveň není potřeba kontrolovat resp. opravovat konzistenci všech datových struktur (to může ve velkém souborovém systému trvat dlouhou dobu), ale pouze těch (resp. těch jejich částí), kterých se dotýkají operace zanesené v žurnálu.
Protože při obnově žurnálu může systém opět havarovat, nebo může být transakce do metadat před běžnou havárií zanesena částečně, musí být záznamy v žurnálu tzv. **idempotentní** – jejich druhé a další provedení nezpůsobí v souborovém systému žádnou další změnu.

---

[85] To by se mohlo stát například tak, že uživatel A smaže soubor X, uživatel B vzápětí vytvoří nový soubor Y, souborový systém pro něj recykluje datové bloky souboru X, a příslušné úpravy metadat jsou zapsány dříve, než samotné datové bloky.

[86] Nabízí se otázka, proč nepoužívat přímo žurnál jako jedinou formu metadat pro souborový systém. Pro většinu použití má takový systém zásadní omezení: vyhledávání v takto strukturovaných metadatech je velmi neefektivní. Existují ale specializované souborové systémy, které takovou strukturu používají (angl. log-structured file systems, česky sekvenční nebo záznamově orientované). Jsou-li metadata dostatečně malá, lze jejich stínovou kopii (uzpůsobenou vyhledávání) udržovat výhradně v operační paměti.

[87] Tím je vynucena určitá míra synchronizace navíc – žurnál tím omezuje schopnost vstupně-výstupního plánovače přeuspořádat zápisy. Zároveň každá operace vyžaduje minimálně jeden zápis navíc, čím se propustnost souborového systému dále sníží. Tento problém lze částečně řešit tím, že se žurnál zapisuje na jiné fyzické zařízení, než zbytek souborového systému. Takové uspořádání ale není příliš časté.

Příklad: Jedním z jednodušších souborových systému se žurnálem je ext4, používaný v OS Linux. Používá dvouúrovňový žurnál: základní vrstvou je JBD2 (z angl. journalled block device) a pracuje na velmi jednoduchém principu: každý blok, který má být zapsán do souborového systému se nejprve uloží do žurnálu jako celek. Tento žurnál tedy obsahuje záznamy o nízkoúrovňových operacích – zápisech na jiná místa souborového systému (v tomto smyslu se jedná o **fyzický** žurnál). Tento typ operace (zapiš data x do bloku y) je triviálně idempotentní.
V obvyklé konfiguraci se takto do žurnálu zapisují pouze metadatové bloky. Platí pravidlo, že do samotného souborového systému se bloky začnou zapisovat až ve chvíli, kdy je transakce uzavřená (commited).
Druhá úroveň žurnálu je tzv. fast commit – nad poslední uzavřenou „plnou" transakcí prvního druhu si ext4 uchovává ještě **logický** žurnál, kde uchovává záznamy o operacích jako „odstraň položku x ze složky y" (kde x je jméno a y je číslo i-uzlu) nebo „do i-uzlu x vlož rozsah datových bloků y". U tohoto typu záznamů je naopak nutné návrh přizpůsobit tomu, aby takové záznamy idempotentní byly (rozmyslete si, že uvedené příklady tento požadavek splňují).
V obou případech je velkým faktorem v robustnosti struktury žurnálu proti poškození použití sekvenčních čísel pro identifikátor transakce; tyto identifikátory se recyklují až po velmi dlouhé době (2^{32} transakcí) a identifikátor transakce se zapisuje do každého bloku, který jí náleží: je tedy jednoduché detekovat situaci, kdy transakce nebyla zapsána kompletně (např. vinou přeuspořádání operací „uvnitř" pevného úložiště). Další kontrola je zajištěna kontrolními součty (v tomto případě CRC32).

3.4.7 Funkcionální metadata Jinou možností jak předejít nekonzistenci je uspořádat metadata tak, že použité datové struktury nebudeme na místě upravovat vůbec. Podobně jako ve funkcionálním programování můžeme místo úpravy existující struktury vytvořit její novou verzi. Přitom využijeme toho, že nezměněné části můžeme z nové verze odkázat – nemusíme tedy kopírovat celou datovou strukturu.

- neměnit datové struktury na místě
- vytvoření upravené kopie
- kopie odkazujících záznamů → rekurze
- vhodné pro stromové struktury
- synchronizace → vždy konzistentní

Takovému přístupu musí být ale příslušná datová struktura uzpůsobena – vhodné jsou zejména stromové struktury (v souborových systémech tedy především B-stromy), kde změna v libovolném uzlu znamená vytvoření nové verze tohoto uzlu a jeho předků (v předcích totiž nemůžeme upravit ukazatel na potomka – to by narušilo princip neměnnosti staré verze),[88] přitom ve vyváženém stromě je takových nejvýše logaritmický počet. Všechny ostatní uzly ale zůstávají nezměněné.
Tento přístup lze srovnat také s principem „copy on write", který známe z první kapitoly, kde nám posloužil k optimalizaci vytváření kopií procesů (virtuálních adresních prostorů). Chápeme-li datovou strukturu jako neměnnou, pokus o zápis do ní vyvolá vytvoření kopie dotčeného datového bloku – důležitým rozdílem v tomto případě je, že kopie má jinou adresu, proto je nutné opravit i všechna místa, která tuto adresu odkazují, atd. (rekurzivně).
Je-li onou datovou strukturou strom, stačí při každé takovéto úpravě jedna synchronizace zápisu – nový kořen je zapsán až ve chvíli, kdy skončil zápis celého nového podstromu. Tím je zaručena nepřetržitá konzistence metadat.

3.4.8 Prázdné místo Jsme tedy konečně vyzbrojeni datovými strukturami vhodnými pro použití v souborovém systému a můžeme se blíže podívat na jejich konkrétní využití.

- cíl: (rychle) najít oblast volného místa
  - také udržet data souboru blízko u sebe
  - také minimalizovat externí fragmentaci
- při vytvoření nebo zvětšení souboru
- často bitmapy nebo B-stromy

Prvním úkolem bude organizace **volných bloků**, do kterých lze uložit nově příchozí data (např. proto, že uživatel vytvořil nový soubor, přidal data do existujícího, ale třeba i proto, že vytvořil novou složku). Vyhledání vhodného datového bloku je častá operace, musí být proto efektivní – jak samotné nalezení, tak poznačení informace o tom, že vybraný blok (resp. bloky) již nejsou volné.
Situaci trochu komplikuje skutečnost, že soubory často zabírají více než jeden datový blok, a je žádoucí, aby byly bloky jednoho souboru podle možnosti blízko u sebe (tzn. aby měly sousední, nebo alespoň nepříliš vzdálené adresy).[2] Žel často není jasné, jak velký výsledný soubor bude[3] – je tedy potřeba udělat nějaký heuristický odhad. Máme-li k dispozici odhad, je dále potřeba nalézt spojitou oblast volného místa – nejlépe v blízkosti již alokovaných bloků. Není-li vhodně velká oblast k dispozici, můžeme se pokusit nalézt alespoň spojitou oblast pro již vyžádané zápisy, nicméně i zde musíme být připraveni degradovat na nespojitou alokaci. Při tom všem se ideálně vyhneme oblastem, o kterých lze předpokládat, že budou využity pro rozšíření jiných souborů. Konečně snahu udržet soubory spojité je potřeba vyvážit s protichůdným požadavkem – nevytvářet zbytečně velké mezery mezi malými soubory. Heuristiky pro alokaci volného místa jsou tedy ve výsledku relativně složité.

[88] Analogicky k funkcionálnímu seznamu – změna hodnoty v uzlu seznamu znamená vytvoření jeho nové verze a nové verze všech jeho předchůdců. Srovnejte např. složitost operace zřetězení a ++ b vůči prvnímu vs vůči druhému operandu.

Na úrovni datových struktur existují dvě obvyklá řešení:

1. souborový systém je rozčleněn do alokačních skupin, přitom každá skupina má vlastní bitmapu volných bloků, vlastní tabulku i-uzlů a souhrnnou informaci o své obsazenosti (používají Unixové souborové systémy klasického návrhu – ufs, ffs, ext2 atd.),
2. informace o volném místě jsou udržovány v B-stromě kde:
   a. klíčem je adresa prvního bloku volné souvislé skupiny[89] – takový strom realizuje tzv. intervalovou mapu, která umožňuje efektivně hledat nejbližší volnou oblast od zadané adresy, ale nikoliv hledat podle velikosti (používá např. btrfs),
   b. hlavním klíčem je velikost volné oblasti, vedlejším pak její adresa – umožňuje efektivně nalézt nejmenší volnou oblast potřebné velikosti (nejmenší, která je alespoň tak velká jako požadavek), a mezi těmito nalézt nejbližší k zadané adrese,
   c. kombinace předchozích dvou – vyžaduje dva samostatné B-stromy a vytváří tak prostor pro nekonzistence, ale umožňuje oba typy hledání (používá např. xfs).

- externí: volné místo (obvykle) není spojité
  - např. zvětšení více souborů najednou
  - smazání staršího souboru
- datová: soubory jsou nespojité
  - často důsledek externí fragmentace
  - zhoršuje výkon (zejména) čtení

3.4.9 Externí a datová fragmentace Ukládání strukturovaných dat do nestrukturovaného pole bajtů vyžaduje vždy určité kompromisy. Jedním z nich je efektivita využití kapacity – ukládání dat více natěsno většinou vede k pomalejším operacím a složitějším metadatům.
V případě souborů se musíme vypořádat jednak se situací, kdy se postupným vytvářením a mazáním souborů prázdné místo rozptýlí mezi alokované bloky. Při vytváření nových souborů to znamená pracnější hledání volných bloků, protože je nutné potřebné místo „slepit“ z několika nesouvislých oblastí (fragmentů).
Tím se jednak zvětšují potřebná metadata (průměrná délka spojitého rozsahu klesá) a zároveň dochází k roztroušení – fragmentaci – samotných dat, která jsou do takto alokovaného souboru uložena. Přístup k takovým souborům je pak méně efektivní, protože s každým skokem z jedné spojité oblasti do jiné je spojena prodleva (daná povahou blokových zařízení).

- odkazy na datové bloky
  - tabulka: přímé + nepřímé odkazy
  - B-strom: klíč je offset v souboru
- rozsah (extent) – spojitá datová oblast
  - úspornější a efektivnější
  - lze kombinovat s tabulkou i stromem

3.4.10 Obyčejné soubory Tradiční reprezentace obyčejného souboru na disku odkazuje každý datový blok samostatně, pomocí krátké tabulky uvnitř i-uzlu (u větších souborů rozšířené pomocnými tabulkami v tzv. nepřímých blocích). Běžným zlepšením je místo odkazu na jeden blok odkazovat celý spojitý rozsah datových bloků (angl. extent).[90] Hlavní nevýhodou tohoto přístupu je, že vyhledání bloku podle adresy bajtu (offsetu) je v takto organizovaných metadatech **lineární** vzhledem k délce takové tabulky.[91]

- soubory mají proměnnou délku
- musí být uloženy v pevně velkých blocích
- nevyužité volné místo
- částečná alokace → slučování konců

3.4.11 Vnitřní fragmentace Vnitřní fragmentace je způsobena zarovnáním – některé operace jsou mnohem efektivnější, když každý soubor začíná na hranici bloku,[92] a tedy je pro něj alokován bloků celočíselný počet. Protože mají ale soubory libovolnou velikost, často je na konci souboru nějaké nevyužité místo. Toto nevyužité místo představuje režii – neobsahuje žádná užitečná data. Jinými slovy je na většiny souborů malý fragment paměti který nelze využít (protože je menší než nejmenší možná velikost souboru – jeden blok).

- důležitá rychlost vyhledání
- lze ukládat do datových bloků
  - obyčejné seznamy (pomalé!)
  - hašovací tabulka (nahodilé pořadí)
- nebo do metadat (B-stromy)
  - klíč = jméno

3.4.12 Adresáře Existují tři základní možnosti, jak v souborovém systému reprezentovat složky:

1. Klasické → hledání podle jména a odstranění položky jsou lineární operace, vkládání je naopak konstantní – takový přístup funguje dobře pro malé složky, ale protože souborový systém nemůže obecně předvídat, kolik položek bude mít daný adresář, tato organizace se spíše nepoužívá.
2. Hašované → položky jsou uloženy v hašovací tabulce a jsou tedy „pravděpodobně“ konstantní – přesné chování záleží na hašovací funkci a výběru jmen. Iterace vrací položky ve zdánlivě náhodném pořadí.
3. Stromové → položky jsou uloženy jako B-strom, jména položek jsou klíče – všechny operace jsou zaručeně logaritmické a iterace vrací položky seřazené podle jména.

První dva typy obvykle pro uložení adresáře používají datové bloky stejně jako pro obyčejné soubory. Stromové složky je naopak přirozenější chápat jako součást metadat (je obvyklé, že takové souborové systémy používají B-stromy v metadatech ve více rolích).

[89] Ekvivalentně může strom udržovat informace o alokovaných blocích, resp. alokovaných oblastech. Tyto reprezentace jsou duální – umožňují stejné operace se stejnou efektivitou.
[90] Lze chápat jako jednoduchou formu RLE – run-length encoding.
[91] Tato je ale obvykle mnohem menší, než odpovídající tabulka jednotlivých adres a potřebujeme-li adresy všech bloků – což je situace, která nastane kdykoliv čteme soubor celý – je celková složitost asymptoticky stejná a prakticky mnohem lepší.
[92] Krom rychlejšího přístupu k souborům tento kompromis umožňuje i použití jednodušších metadat.

Část 4: Virtualizace periferií

Periferie umožňují počítači komunikovat s okolním světem – ať už lidmi (vstup a výstup), jinými počítači (počítačové sítě) nebo i jinými typy zařízení (čtení senzorů, řízení motorů, atp.). Za periferie také považujeme paměťová média (jiná než operační paměť), jak zabudovaná (pevné disky, SSD) tak výměnná (paměťové karty, optické disky).

Doporučené čtení: A. Tanenbaum, H. Bos – Modern Operating Systems (4th Ed.): § 5.1.1 I/O Devices ◇ § 5.1.2 Device Controllers ◇ § 5.1.3 Memory-Mapped I/O ◇ § 5.1.4 Direct Memory Access ◇ § 5.2 Principles of I/O Software ◇ § 5.3.2 Device Drivers.

4.1: Periferie obecně

Nejprve si charakterizujeme periferie jako obecnou abstrakci, na propojení periferií k výpočetní části počítače pomocí sběrnic a jejich řadičů a na mechanismy, které se používají ke komunikaci s periferiemi.

4.1.1 Typy periferií Tři základní kategorie:

1. pomocná vnitřní zařízení – zejména řadiče sběrnic,
2. perzistentní úložiště – dlouhodobé ukládání dat (předchozí kapitola),
3. síťová rozhraní – realizují komunikaci mezi počítači,
4. terminál – komplex periferií pro komunikaci s uživatelem:
   - obrazovka a přidružená zařízení,
   - vstupní zařízení – klávesnice, myš / ukazovací zařízení,
   - audio,
   - tiskárny, skenery, čtečky čárových kódů, atp.

1. pomocná zařízení (řadiče)
2. trvalá úložiště
3. síťová rozhraní
4. terminál (obrazovka, klávesnice, …)

4.1.2 Co je periferie? Zařízení, které produkuje a konzumuje data, případně události. Komunikace v malém objemu se obvykle realizuje mapováním registrů[93] zařízení na fyzické adresy.[94] Pozor, je zde důležitý rozdíl proti operační paměti: hodnoty se „samovolně“ (bez účasti hlavního procesoru[95], a tedy i operačního systému) **mění v čase** – takovou změnu lze považovat za **událost**.
Dostupnost dat tak může být časově závislá: data jsou zpřístupněna počínaje nějakou vnější událostí (stisk klávesy atp.), ale zároveň je jejich životnost omezena, obvykle velikostí paměti zařízení. Jakmile místo v této paměti dojde, nejstarší data jsou obvykle přepsána novými (princip FIFO). Je proto důležité, aby operační systém data přečetl včas a uložil je v případě potřeby na trvalejší místo (např. do operační paměti).
Podobně je omezená schopnost zařízení data přijímat – zařízení zpracovává data nějakou rychlostí (obvykle mnohem menší, než je rychlost procesoru, často i mnohem menší, než je rychlost sběrnice). Data je možné zapisovat, dokud je v paměti zařízení volné místo, pak je potřeba vyčkat, než jsou všechna data zpracována a poté ta nejstarší přepsat.

- produkuje a konzumuje data
- mapování registrů na fyzické adresy
- časová závislost → události
- důležitá doba reakce OS

4.1.3 Programovaný vstup/výstup (PIO) Nejjednodušší metodou komunikace se zařízením je přenos dat postupným čtením z paměti (registrů) zařízení (již zmiňovaným mapováním paměti zařízení na fyzické adresy procesoru). Je-li tato činnost prováděna hlavním procesorem, mluvíme o tzv. „programovaném“ vstupu resp. výstupu (angl. programmed IO, PIO).
Tento způsob komunikace tedy vyžaduje aktivní účast procesoru v stanovených časových intervalech (podle přenosové rychlosti, velikosti vyrovnávací paměti, atp.). Tato metoda je použitelná pro občasné přenosy a/nebo přenosy s velmi malou šířkou pásma (počtem bajtů přenesených za sekundu). Při větších objemech dat vede tento způsob komunikace k příliš vysoké režii.

- přenos čtením/zápisem registrů
- periodicky dle přenosové rychlosti
- velká rychlost → velká režie
- pouze občasné nebo velmi krátké přenosy

4.1.4 Přímý přístup do paměti (DMA) Je proto často žádoucí tuto činnost provádět mimo hlavní procesor. Zdaleka nejčastějším cílem přenosu dat z periferie je jejich uložení do operační paměti – další zpracování zpravidla musí vyčkat do doby, než je v operační paměti nějaký kompletní celek

- periferie ↔ operační paměť
- asynchronně vůči CPU
- řídí periferie (nebo sběrnice)
- časté a/nebo velké přenosy (síť, SSD, GPU)

[93] Pozor, registr zařízení není totéž co registr procesoru. V případě periferií za registr považujeme pevnou paměťovou buňku s pevnou funkcí, ale z pohledu procesoru (a tedy operačního systému) se jedná o adresovatelnou entitu.

[94] Existují i jiné možnosti komunikace, zejména tzv. port-mapped IO. Rozdíl oproti komunikaci čtením fyzických adres (resp. zápisem na ně) je pouze v tom, že porty mají vlastní adresní prostor a vyžadují speciální instrukce procesoru. Tento typ komunikace je spíše zastaralý a používá se jen vzácně.

[95] Procesor není jediné výpočetní zařízení v běžném počítači – periferie mají často nějakou schopnost provádět výpočty a podobají se tak na procesor (resp. relativně často nějaký procesor i obsahují). Abychom jasně odlišili výpočty, které provádí podobné „podružné“ procesory, budeme v této kapitole explicitně mluvit o **hlavním procesoru** – myslíme tím ten procesor, resp. ty procesory, které provádí kód operačního systému.

(blok, rámec, atp.). Tato metoda se používá zejména pro velké a/nebo časté přenosy dat: síť, SSD, GPU.

Asynchronní přenos dat (bez účasti procesoru, resp. bez účasti softwaru)[96] může být realizován dvěma základními metodami:

1. dedikovaným pomocným procesorem, který od hlavního procesoru přijímá pokyny na provedení přenosu (z jaké periferie, na jakou adresu v operační paměti, kolik bajtů, případně opačně, z jaké adresy v operační paměti a jaké periferii), a který je de-facto součástí sběrnice,
2. přenos je řízen přímo periferií (na základě podobného pokynu od hlavního procesoru).[97]

Výhodou první metody je, že periferie nemusí nijak rozlišovat DMA a PIO režimy přenosu dat a může být tedy jednodušší. Druhá metoda je obvykle efektivnější a v moderních systémech mnohem běžnější.

Pozor! Je důležité rozlišovat přímý přístup do paměti (komunikuje periferie a operační paměť, bez účasti procesoru) a mapování paměti zařízení do fyzického adresního prostoru (komunikuje procesor s periferií, bez účasti operační paměti).

- neomezený přístup do RAM → nebezpečné
- obdoba MMU, ale pro periferie
  - virtuální adresní prostor pro zařízení
- programuje jádro OS
- umožňuje neprivilegované ovladače

4.1.5 IO-MMU Přestože pro zařízení, které přenáší velké objemy dat (pevná úložiště, síťová rozhraní, atp.), je DMA nepostradatelné, jsou s ním spojeny určitá bezpečnostní rizika. V klasické implementaci DMA má totiž každá periferie neomezený přístup do fyzické paměti – operační systém sice dává periferii pokyn, které fyzické adresy má použít, ale nemá jak vynutit, aby periferie takový pokyn dodržela.

V takovém systému tedy periferiím tedy nic tedy nebrání v tom libovolně upravovat obsah paměti – například i přepsat kód jádra a získat tak plnou kontrolu nad systémem. Toto se týká nejen případných podvratných periferií, ale i ovladačů, které mohou jinak nevinnou periferii naprogramovat tak, aby narušila bezpečnost systému.

Toto je jistě nežádoucí, zejména chceme-li izolovat ovladače od zbytku jádra, nebo v případech, kdy samotná periferie není nutně důvěryhodná.[98] IO-MMU je zařízení, které tento problém řeší, a to tím, že realizuje překlad adres pro periferie (podobně jako MMU realizuje překlad adres pro software). IO-MMU je programovatelná operačním systémem (a pouze operačním systémem) a umožňuje tedy izolovat periferie jak vzájemně, tak od operačního systému a softwaru obecně. Je-li IO-MMU správně naprogramovaná, je DMA bezpečné.

- fyzická + logická vrstva
- spojení sběrnic → řadič (typ periferie)
- přenáší data (včetně adresace)
- signalizuje události

4.1.6 Sběrnice Má dvě základní vrstvy:

1. fyzickou, která odpovídá za signalizaci a časování a
2. logickou (protokolovou), která popisuje chování zařízení na sběrnici na vyšší úrovni – adresaci, konfiguraci zařízení, přenosy dat atp.

Sběrnic existuje v počítači celá řada a jsou vzájemně propojené. Ta strana sběrnice, která je blíže hlavnímu procesoru, se obvykle nazývá hostitelská. Na této straně je spojena s další sběrnicí tzv. řadičem (až na případ, kdy se už nacházíme uvnitř procesoru). Protože řadič sběrnice je entita, se kterou je možné (a často nutné) komunikovat, jedná se také o typ periferie.

Příklad: Procesory řady Intel Skylake jsou vnitřně propojené pomocí sběrnice Ring Bus, která spojuje výpočetní jádra, integrované GPU, L3 cache a tzv. system agent („uncore“).
Součástí bloku system agent je řadič paměti a řadič PCIe, na kterých začínají tyto dvě sběrnice. Na paměťové sběrnici jsou připojené pouze moduly RAM.
Na sběrnici PCIe je připojen téměř celý zbytek systému – přímo jsou to síťová rozhraní a pevná úložiště typu NVMe, případně externí GPU. Dále jsou to řadiče dalších sběrnic – zejména USB, SATA. Síťová karta má vlastní vnitřní sběrnici – MII – která propojuje komponenty MAC a PHY (z našeho pohledu je tedy MAC řadičem sběrnice MII). Další sběrnice mohou být připojeny skrze USB, atd.

Hlavním úkolem sběrnice je přenos dat (a s tím související adresace) a signalizace událostí,[99] vedlejším pak konfigurace a enumerace připojených periferií. Má-li řadič sběrnice přidělen rozsah

[96] Procesor, ve smyslu diskrétní komponenty počítače, se často takového přenosu ve skutečnosti účastní (a to i v systémech, které nejsou celé integrované na jediném čipu), protože fyzicky obsahuje jak paměťový řadič, tak řadič sběrnice PCIe. Tyto přenosy se ale nedotýkají výpočetních jader, na kterých běží operační systém (to, čemu zde říkáme hlavní procesor).

[97] Přenosy dat na **sdílené** sběrnici (např. PCI, ale ne PCIe) řídí jedno z připojených zařízení (angl. bus master). Obvykle je touto řídící komponentou procesor, ale přenosů DMA (mezi periferií a pamětí) se procesor neúčastní. Proto v takové situaci dočasně sběrnici řídí periferie, která přenos provádí. Protože sběrnici může v danou chvíli řídit pouze jedno zařízení, musí existovat arbitrážní protokol, který řeší konfliktní situace.

[98] Tento problém se dotýká zejména externích portů, které připojeným periferiím umožňují provádět DMA (jedním takovým je firewire, starší vysokorychlostní sběrnice pro externí zařízení). Tím je umožněno vytvořit fyzické zařízení, které stačí připojit do vnějšího portu laptopu a pořídit tím kopii celé fyzické paměti (bez ohledu na zámky obrazovky a podobná softwarová bezpečnostní opatření).

[99] Signál o události se obvykle projeví na úrovni procesoru jako **přerušení**. Detailněji budeme přerušení zkoumat v osmé kapitole.

fyzických adres, je také jeho úkolem tento dále rozdělit mezi připojené periferie (včetně podružných řadičů dalších sběrnic).

4.1.7 Enumerace Enumerací sběrnice rozumíme vyjmenování všech připojených periferií a jejich základních parametrů. Všechny moderní sběrnice umožňují enumeraci – operační systém tak může zjistit, jaké periferie jsou v systému přítomny, a to bez zásahu uživatele.[100]
Enumerace zároveň umožňuje každé periferii poskytnout operačnímu systému unikátní identifikátor, podle kterého operační systém zjistí, o jaké konkrétní zařízení se jedná (dodavatel, typ) a může tak aktivovat nebo i nainstalovat potřebné softwarové vybavení (ovladače).

- vyjmenování připojených periferií
- důležité pro automatickou konfiguraci
- unikátní identifikátory zařízení

4.1.8 Ovladač Jakmile je známý konkrétní typ zařízení, komunikaci s ním převezme **ovladač** – program, který poskytuje softwarovou abstrakci dané třídy zařízení zbytku operačního systému (a nepřímo i aplikacím).
Na jedné straně komunikuje ovladač se zbytkem operačního systému (pomocí vhodného rozhraní, které je ale obvykle specifické pro daný operační systém), na straně druhé komunikuje s konkrétním fyzickým zařízením (které je zase často specifické pro daný model, nebo alespoň modelovou řadu). Připomínáme zde, že komunikace probíhá čtením a zápisem dat, a zpracování takových dat není nic jiného než **výpočet**. Hlavním úkolem ovladače je:

1. zpracování příchozích dat do formy, která je přijatelná pro zbytek systému (tzn. do formy nezávislé na konkrétním modelu zařízení),
2. převod dat příchozích ze systému (které jsou ve formě nezávislé na konkrétním zařízení) do formy, kterou umí daná periferie zpracovat,
3. to vše v reakci na události – buď požadavky ze systému na periferii, nebo naopak.

Data v tomto případě nemusí být pouze užitná data (bloky uložené na disku, rámce přijímané nebo odesílané síťovým rozhraním, atp.), ale také řídící data, která ovlivňují jak se bude zařízení chovat, nebo přímo aktivují další funkce zařízení (jiné, než je samotný přenos užitných dat).

- program, který abstrahuje zařízení
- prostředník mezi OS a periferií
  - převod mezi formami dat → výpočet
  - řídící i užitná data
- rozhraní k OS – třída (disk, síťové rozhraní, ...)
- k periferii → používá ovladač sběrnice

Příklad: Operační systém potřebuje zapsat blok dat na pevné úložiště. Vstupem této operace je fyzická adresa dat, identifikátor cílového zařízení a adresa cílového bloku daného zařízení. Ovladač disku vytvoří požadavek (datovou strukturu), kterou zapíše (za pomoci ovladače sběrnice) do fronty disku. Disk pak vnitřně provede potřebné operace a za pomoci sběrnice data přenese na ovladačem určené místo v operační paměti. Nakonec vyvolá událost, kterou ovladači oznámí, že byl přenos ukončen. Ovladač přečte potvrzení operace (opět datovou strukturu) a informace předá do operačního systému.

Je-li periferie, kterou daný ovladač obhospodařuje, připojená nějakou sběrnicí (jinou než vnitřní sběrnicí procesoru), využívá ovladač periferie ke komunikaci se svým zařízením ovladač příslušné sběrnice.

Příklad: Ovladač pro SATA disky komunikuje s diskem vkládáním příkazů pro čtení a zápis dat do fronty, která je uložena v paměti samotného disku.
Jak tento zápis do fronty proběhne je ale dané standardem SATA, proto tyto zápisy nerealizuje ovladač disku přímo zápisem na příslušnou fyzickou adresu (tuto často vůbec nezná), ale požádá o provedení zápisu ovladač sběrnice SATA.
Řadič sběrnice SATA je připojen ke sběrnici PCIe. Proto ovladač sběrnice SATA nepočítá fyzické adresy potřebných registrů přímo, ale využije k tomu služeb ovladače PCIe.

4.2: Terminál

Pojmem **terminál** budeme označovat sadu periferií, která slouží ke komunikaci s uživatelem. Současné počítače mají obvykle jeden nebo žádný fyzický terminál, ale historicky nejsou výjimečné ani systémy s mnoha fyzickými terminály.

4.2.1 Textový terminál Fyzický terminál sestává z výstupní části (zejména obrazovky) a vstupní části (zejména klávesnice, případně ukazovacího zařízení – myši a podobně). Umožňuje tak oboustrannou ke komunikaci s uživatelem – uživatel zadává vstupy na klávesnici a na obrazovce čte výstupy.
Chceme-li textový terminál virtualizovat, tzn. vytvořit virtuální terminály, kterých může být víc než těch fyzických, musíme si zejména zapamatovat obsah obrazovky. Aktivace virtuálního

- výstupní část = obrazovka
- vstupní část = klávesnice (+ myš)
- virtualizace → pomyslné terminály
  - zapamatovat obsah obrazovky
  - přepojování vstupů

[100] Historicky existovaly sběrnice, které enumeraci neumožňovaly, např. klasická ISA. V takovém případě se buď daný typ zařízení vždy nastavoval na stejnou adresu (např. sériové porty), nebo vyžadoval konfiguraci uživatelem.

terminálu přepíše obsah fyzické obrazovky tím zapamatovaným (uloženým v operační paměti). K virtuálnímu terminálu lze připojit nebo od něj odpojit fyzickou klávesnici, aniž by se to nějak dotklo program, který terminál používá. Přepojování klávesnice a obrazovky je samozřejmě synchronizované – fyzická obrazovka zobrazuje tentýž virtuální terminál, který má právě připojenu fyzickou klávesnici.

- aktivní vs přerušené programy
- výstup do textových buněk
  - řádkový vs obrazovkový režim
- nemusí být připojen k obrazovce
  - např. do grafického okna (emulátor)
  - do sítě (ssh)

4.2.2 Výstup na obrazovku Jeden (virtuální) terminál může používat více programů najednou. Jeden z nich je aktivní (ovládá terminál), ostatní čekají. Tím je umožněno spustit interaktivní program z jiného (interaktivního) programu („volající“ program přepustí terminál „volanému“; terminál je navrácen ukončením nebo přerušením volaného programu).
Obsah obrazovky typicky ovládá pouze aktivní program (často není vynuceno, ale cokoliv jiného vede spíš k nepoužitelným výsledkům).
Obrazovka pomyslně sestává z obdélníkové mřížky, v každém políčku je jeden znak. V řádkovém režimu lze text psát pouze do jednoho řádku, který řádek to je ovládá terminál; dojde-li na konec obrazovky, začnou se nejstarší řádky odsunovat mimo (scrollování). V obrazovkovém režimu může program cíleně měnit obsah (písmeno) libovolného políčka.
Virtuální terminál nemusí být realizován fyzickým terminálem: může být místo toho např. vykreslen do okna grafické aplikace – takové aplikaci pak říkáme terminálový emulátor. Virtuální terminál může být připojen také k síťovému spojení a vykreslen na jiném počítači (např. program ssh).

- virtuální klávesnice
  - přeposílá data/události z fyzické
  - nebo nulová → nedělá nic
- k fyzické připojená obvykle 1 virtuální
- lze připojit i k programu → ssh, emulátor

4.2.3 Vstup z klávesnice U klávesnice (a vstupních zařízeních obecně) nelze úplně mluvit o virtualizaci v klasickém smyslu: aplikace nijak klávesnici neovládá, ani ji nemůže jakkoliv přímo využívat. Jinak řečeno, aplikace je vůči klávesnici pasivní. „Nulová“ klávesnice, která nedělá vůbec nic, tedy splňuje vše, co může aplikace od klávesnice vyžadovat – to zjevně neplatí např. o procesoru nebo o operační paměti.
Fyzickou klávesnici navíc nelze rozumně virtualizovat, protože nelze virtualizovat uživatele. Virtuální klávesnice je tedy zařízení, které má dva režimy:

- transparentně přeposílá data z fyzické klávesnice,
- nedělá nic („nulová“ klávesnice).

Celkově tedy situace vypadá takto:

- vstup z fyzické klávesnice je směřován pouze aktivnímu programu v aktivním virtuálním terminálu,
- ostatní programy jsou připojeny k „nulové“ klávesnici (čekají-li na vstup, budou čekat až do chvíle, než jsou přepojeny na fyzickou klávesnici).

Jiný typ virtuální klávesnice může být řízen programově – na druhé straně není uživatel, ale program, který události (stlačení kláves) generuje. Konkrétní využití takovéhoto uspořádání je například již zmiňovaný vzdálený terminál (uživatel používá jiný počítač, který virtuální terminál pouze zobrazuje). V takové situaci virtuální klávesnici řídí program, který dostává pokyny (které obvykle pochází od vzdáleného uživatele) po síti.

- buňky s písmeny nahradíme pixely
- pixel = malý čtvereček nějaké barvy
- vstup → podobná omezení virtualizace
- výstup → flexibilnější

4.2.4 Grafický režim Textové terminály jsou relativně omezené a nevhodné pro řadu aplikací (lze si jen těžko představit editor fotografií, který by komunikoval výhradně textově). Buňky s písmeny, které jsou základním prvkem textového terminálu, jsou v grafickém režimu nahrazeny **pixely** – drobnými obdélníky v mřížce. Aplikace určuje barvu jednotlivých pixelů, a tím také obraz, který tvoří.
Z pohledu virtualizace vstupu platí podobná omezení, jako v textovém režimu – klávesnici, případně ukazovací zařízení (myš) lze virtualizovat jen v relativně omezené míře. Grafická obrazovka je ale výrazně flexibilnější než ta textová, a nabízí tak i novou metodu virtualizace.

- každá aplikace má virtuální obrazovku
- jsou zobrazeny na fyzické jako okna
- uživatel ovládá zobrazení a rozmístění
- zobrazení a ovládání oken → podle systému

4.2.5 Okenní systémy Okenní systém představuje nejběžnější metodu virtualizace grafické obrazovky: každý program dostane vlastní „virtuální obrazovku“, která je pak vykreslena na vyhrazenou část skutečné obrazovky.[101] Uživateli obvykle je umožněno manipulovat s takto vytvořenými oblastmi (okny), případně je úplně skrýt nebo naopak nechat skryté zobrazit.
Přesná sémantika chování oken závisí na konkrétním okenním systému, důležitý je koncept oken jako virtuálních obrazovek. Nebudeme ani požadovat, aby vůbec bylo možné zobrazit více než jedno okno zároveň, umožní-li systém okna alespoň přepínat (můžeme tak do tohoto pojmu zahrnout

[101] Na první pohled se jedná o celkem jednoduchou záležitost, situaci ale značně komplikují další aspekty okenních systémů, které přímo nesouvisí s virtualizací: schopnosti jako kopírování dat mezi okny (schránka, „přetažení“ myší - drag&drop), modální prvky (dialogová okna, kontextové nabídky, hlavní nabídka), atp.

i rozhraní mobilních telefonů, která fungují na velmi podobných principech, i když z pohledu uživatele vypadají jinak než klasické systémy s „plovoucími“ okny).
Krom vykreslování virtualizuje okenní systém také vstupní zařízení:

- klávesnici již dobře známým postupem, kdy je tato připojena k aktivnímu oknu,
- události ukazovacích zařízení (kliknutí myši nebo touchpadu, dotek nebo gesto na dotekové obrazovce) navíc vyžadují převod mezi souřadnými systémy obrazovky a okna.

4.2.6 GPU Výpočet barev pro jednotlivé pixely (rasterizace) je proces náročný na zdroje[102] a jeho nároky rostou s počtem pixelů a složitostí vykreslovaných útvarů. Obvyklým řešením je specializovaný hardware, který umožňuje potřebné výpočty provádět velmi rychle. Současné generace tohoto typu hardwaru nesou označení GPU.
Moderní GPU mají zabudovanou podporu virtualizace podobnou té z procesorů: mají hardwarové kontexty, které lze efektivně přepínat, a tak jedno GPU sdílet mezi několika aplikacemi (každá aplikace má „vlastní“ virtuální GPU, stejně jako má vlastní virtuální paměť nebo procesor). Pixely, které GPU pro aplikaci vypočte, se obvykle nejprve uloží do paměti.

- výpočet pixelů (rasterizace) je drahý
- speciální hardware → GPU
- virtualizace přepínáním kontextů
- vypočtené pixely → paměť

4.2.7 Kompozitor Dále je potřeba je vykreslit na obrazovku (to platí i pro pixely vypočtené softwarově). Výstupní část okenního systém, ve kterém je každá aplikace odpovědná za „vlastní“ pixely, nazýváme **kompozitor** – skládá „obrázky“ jednotlivých aplikací do jednoho snímku, který se pak vykreslí na obrazovce. Je obvyklé, že kompozitor pro výpočet tohoto snímku opět využívá GPU (vypočtené obrázky jednotlivých aplikací se tak nemusí vracet do operační paměti, aby je kompozitor mohl zpracovat softwarově).
Kompozitor musí spolupracovat se vstupní částí systému, zejména musí odpovídat převody souřadnic událostí s rozložením oken na obrazovce.

- výstupní část okenního systému
- skládá „obrázky“ aplikací
- výpočet snímku obvykle provádí GPU
- spolupráce se vstupem → souřadné systémy

4.2.8 Grafický server Alternativní metodou virtualizace grafického podsystému (včetně hardwarového urychlování rasterizace) je tzv. grafický server, který přijímá příkazy vyšší úrovně (vykreslení 2D nebo i 3D objektů, nikoliv jednotlivých pixelů). V takovém systému má grafický server plnou kontrolu nad obsahem obrazovky a může pro rasterizaci využívat hardware, který nemá hardwarové kontexty a nelze jej tedy virtualizovat přímo. Potenciální výhodou takového systému je, že objem dat přenášený mezi aplikací a grafickým serverem může být až o několik řádů menší, než odpovídající rastrová reprezentace, čím se usnadňuje např. vzdálené vykreslování.

- alternativa kompozitoru
- poskytuje kreslící příkazy
- spravuje celý obsah obrazovky
- nevyžaduje přepínání kontextu

4.2.9 Audio Logicky jsou zvuková rozhraní součástí terminálu. Platí zde podobná omezení jako u obrazovky a klávesnice: výstup lze do jisté míry virtualizovat (přepínáním proudů, nebo mixováním několika proudů do jednoho), vstup (mikrofon, MIDI zařízení) lze pouze přepínat.
Podobně jako v případě grafického zobrazení je zvukový podsystém citlivý na latenci a na nepřerušenost proudu dat. Vysoká latence způsobí viditelné opoždění mezi akcí uživatele a její zvukovou odezvou (např. virtuální klavír – je nežádoucí, aby mezi stlačením klávesy a začátkem tónu vznikla prodleva). Nedostatek dat zase způsobí (velmi slyšitelný) výpadek zvuku. Latence a nepřerušenost datového proudu jsou v přímém protikladu: čím větší vyrovnávací paměť, tím menší šance, že nebude včas doplněna, ale tím větší latence.

- výstupní proudy → mixování
  - reproduktory, sluchátka
  - lze i přeposílat vzdáleně
- vstupy → přepínání
  - mikrofon, MIDI

4.2.10 Tiskárny Tiskárny se od ostatních „terminálových“ periferií liší v jednom zásadním ohledu: jsou spíše dávkové než interaktivní. Jednotlivé požadavky na tisk představují ucelené úlohy, a tiskárna sama o sobě není schopná pružně reagovat na požadavky operačního systému na změnu úlohy – systém virtualizace, který používáme pro obrazovku, tedy nepřipadá v úvahu.
Zpřístupnění tiskárny aplikacím má dva aspekty:

1. virtualizace – jak tiskárnu sdílet mezi programy – tento problém je analogický k problému plánováni úloh v dávkových systémech, a má i analogické řešení: frontu úloh ke zpracování,
2. abstrakce – jak zahladit rozdíly mezi jednotlivými tiskárnami tak, aby byly z pohledu programu podle možnosti záměnné.

Řešení druhého bodu je trochu složitější, a některé aspekty tiskáren nelze úplně v aplikacích ignorovat (černobílý vs barevný tisk, jednostranný vs oboustranný, atp.). Částečným řešením je použití společného formátu pro popis dokumentů k tisku, přičemž operační systém již zařídí konverzi do formátu, který tiskárna akceptuje. Starším standardem tohoto typu je PostScript,

- pouze dávkové zpracování
- virtualizace frontou úloh
- abstrakce: skrývá rozdíly
  - společný formát popisu dokumentu
  - PDF (historicky PostScript)

[102] Prakticky každý pixel (nebo malá skupina pixelů) vyžaduje alespoň jeden zápis do paměti a práce s pamětí je pro CPU obzvláště drahá díky čekání na mnohem pomalejší paměť a/nebo sběrnici. Situace je ještě horší, je-li potřeba pro výpočet barvy pixelu nějaká data z paměti načíst (textury, sprity, atp.).

novějším PDF.[103] Některé tiskárny podporují tisk dokumentů v těchto formátech přímo, bez potřeby dalšího zpracování v operačním systému.
Z pohledu aplikace může mít tedy virtuální tiskárna jako svou hlavní operaci vložení dokumentu ve formátu PDF do tiskové fronty.
Krom lokálních tiskáren, které jsou periferiemi v klasickém smyslu, existují i tiskárny síťové – v jednodušším případě je rozdíl pouze v metodě připojení (virtualizaci stále řeší operační systém), ale některé síťové tiskárny mají interní frontu úloh (lze na nich tedy zároveň virtuálně tisknout nejen z několika aplikací, ale i několika různých počítačů).

4.3: Síťová rozhraní

Umožňují vzájemnou komunikaci mezi počítači, resp. programy běžícími na různých počítačích.

- spojují počítače → umožňují přenos dat
  - soubory, proudy, zprávy, atp.
- mohou být drátové a bezdrátové
- prozatím se soustředíme na nižší vrstvy

4.3.1 Počítačová síť Základní funkcí počítačové sítě je přenášet data a obecně umožnit komunikaci mezi jednotlivými počítači. K tomu síť využívá nějaké propojovací médium (může být drátové i bezdrátové), ke kterému jsou jednotlivé počítače – **síťové uzly** – připojeny **síťovými rozhraními**. Síťové rozhraní je zařízení, které je na jedné straně (ke zbytku počítače) připojeno sběrnicí (např. PCIe) a na straně druhé k přenosovému médiu sítě, např. konektorem RJ-45 nebo anténou. Síť jako celek se pak chová podobně jako sběrnice[104] uvnitř počítače, přitom síťová rozhraní plní úlohu podobnou řadiči sběrnice. Sítě nicméně nemají význačný centrální prvek, který by měl ke zbytku uzlů podobný vztah, jako má procesor k periferiím.

- ISO/OSI model síťové komunikace
  - 1 + 2 → především hardware
  - 3 + 4 → především OS
  - 5 - 7 → především aplikace
- virtualizace zejména mezi 4. a 5. vrstvou

4.3.2 Vrstvy Architektura sítě je rozložena do řady **vrstev**. Situace je podobná perzistentnímu úložišti: nelze rozumně virtualizovat přímo hardwarem poskytované služby (např. posílání rámců do sítě). Místo toho buduje operační systém řadu abstrakcí, rozložených do několika vrstev:

1. fyzická – záležitost hardwaru, má ale dopad na vyšší vrstvy,
2. linková – tvoří faktické rozhraní mezi hardwarem a softwarem (operačním systémem),
3. síťová – na koncových stanicích řeší převážně software (operační systém), jejím úkolem je zabezpečit komunikaci mezi koncovými uzly (resp. mezi operačními systémy na nich provozovanými),
4. transportní doručuje data mezi **aplikacemi**: tvoří rozhraní mezi operačním systémem a aplikací, a je tak přirozeným místem pro virtualizaci,
5. a vyšší: záležitost aplikací, staví na virtualizaci poskytované 4. vrstvou.

- abstrakce pro potřeby OS
  - mezi 2. a 3. vrstvou
- abstrakce a virtualizace pro aplikace
  - mezi 4. a 5. vrstvou
  - adresa aplikace = adresa uzlu + port

4.3.3 Abstrakce a virtualizace Dva důležité přechody:

1. pro interní potřeby OS: mezi 2. a 3. vrstvou, zejména abstrakce (operační systém je pouze jeden, nepotřebuje tedy obvykle hardware pro vnitřní potřeby virtualizovat),
2. pro potřeby aplikací: mezi 4. a 5. vrstvou, jak virtualizace tak abstrakce (3. i 4. vrstva skrývají mnoho detailů nižších vrstev).[105]

Hlavním stavebním prvkem virtualizace síťové komunikace je multiplexing (znásobování). Pakety jsou na třetí vrstvě doručovány mezi uzly. Čtvrtá vrstva přidává virtuální koncové body jednotlivých aplikací, realizované pomocí portů. Každý port reprezentuje nezávislý proud dat (nebo diskrétních paketů, v případě paketově orientovaných protokolů).

- fyzická (PHY) + linková vrstva (MAC)
- moderní ethernet je point-to-point
  - používá aktivní přepínače
- základní jednotka přenosu = rámec

4.3.4 Ethernet (IEEE 802.3) Ethernet je zdaleka nejrozšířenější technologie pro realizaci počítačových sítí. Je složen ze 2 vrstev:

1. fyzická vrstva, PHY, která:
   - odpovídá za signalizaci a kabeláž,
   - základní přenosová jednotka je jeden bit,
   - mapuje se poměrně přesně na 1. vrstvu OSI modelu,
   - např. 802.3ab / 1000BASE-T („gigabitový“ ethernet po osmižilovém kabelu složeného z kroucených dvojic),
   - nebo optické varianty, např. 802.3ae / 10GBASE-SR (10Gb/s ethernet pro lokální sítě),
2. vrstva MAC (media access control):
   - odpovídá OSI vrstvě 2,

[103] Nejsou to jediné možnosti – OS Windows historicky poskytuje vlastní způsob popisu stránek nezávislý na tiskárně, založený na GDI (rozhraní, které se používalo hlavně na vykreslování na obrazovku).
[104] V obecném smyslu, nikoliv ve smyslu konkrétní topologie. Ani moderní sběrnice, ani moderní sítě obvykle sběrnicovou topologii nepoužívají.
[105] Rozhraním, které umožňuje aplikacím využívat a poskytovat síťové služby, se budeme blíže zabývat v 10. kapitole.

- základní přenosovou jednotkou je **rámec** (max. 1500 bajtů).

Toto rozdělení odpovídá také stavbě hardwaru – ethernetové síťové rozhraní má části PHY a MAC spojené sběrnicí MII (medium-independent interface) resp. některou její novější verzí, podle potřebné přenosové rychlosti. V principu jsou PHY a MAC části (při daném limitu přenosové rychlosti) zcela nezávislé a vzájemně záměnné. Operační systém přímo komunikuje pouze s částí MAC.

Základní přenosovou jednotkou je rámec, který má tyto složky:

1. preambule (synchronizační kód, pro všechny rámce stejný),
2. cílová MAC adresa,
3. zdrojová MAC adresa,
4. pole EtherType
   - menší než 0x600 = velikost,
   - větší = rozlišení vnitřního protokolu,
5. užitná data (angl. payload, max 1500 bajtů).

4.3.5 Adresace Na úrovni ethernetu funguje adresace pouze lokálně, v rámci jednoho ethernetového segmentu.[106] Rozhraní připojené do nějakého segmentu je schopno adresovat pouze jiná rozhraní připojená do stejného segmentu. Všechna rozhraní mají přidělenu z výroby tzv. MAC adresu[107], která rozhraní identifikuje. V obvyklé moderní topologii jsou jednotlivá rozhraní připojena k přepínačům,[108] které si udržují informaci o tom, které adresy jsou dostupné skrz který port a rámce příslušným způsobem přeposílají. Cílové rozhraní dekóduje všechny rámce, které k němu fyzicky dorazí; to, jestli je právě toto rozhraní zamýšleným příjemcem, pak zjistí tím, ze přečte cílovou MAC adresu uvedenou v hlavičce rámce.

- pouze v rámci segmentu (lokální)
- každé rozhraní má tovární MAC adresu
- překlad adres mezi 2. a 3. vrstvou
- přepínače mapují MAC adresy na porty

Aplikace (a uživatelé) ale používají k identifikaci uzlů adresy třetí vrstvy[109] – aby bylo možné paket třetí vrstvy fyzicky doručit, je tedy potřeba získat adresu druhé vrstvy, která odpovídá cílové adrese třetí vrstvy. Obecně k tomu slouží překladové tabulky, které operační systém udržuje. V sítích IP/Ethernet jsou tyto tabulky sestaveny za pomoci protokolu ARP (Address Resolution Protocol).

4.3.6 Odchozí fronta Potřebuje-li operační systém odeslat paket (rámec) do sítě, přidá je na konec tzv. odchozí fronty (angl. transmit queue, Tx queue). Z této fronty je vyzvedne hardware a jakmile je to možné, provede fyzický přenos. Odchozí fronta funguje přibližně takto:

- rozhraní vyzvedává rámce z paměti
- OS je nachystá do odchozí fronty
- rozhraní je asynchronně čte (DMA)
- a autonomně je posílá do sítě

1. každá odchozí fronta (může jich existovat několik) má přiřazenu dvojici registrů mapovaných do fyzického adresního prostoru: jeden reprezentuje hlavový ukazatel a ten druhý koncový (angl. head a tail),
2. tyto ukazatele popisují **kruhovou frontu** pevné velikosti, uloženu v operační paměti, ke které síťové rozhraní přistupuje za pomoci DMA; každá položka (buňka) této kruhové fronty reprezentuje jeden rámec,
3. ukazatele dělí frontu na dvě části – jedna patří rozhraní a jedna operačnímu systému,
4. operační systém (resp. ovladač síťového rozhraní) upravuje koncový ukazatel:
   a. pro odeslání rámce pro něj operační systém nejprve vyhradí paměť a uloží do ni obsah rámce (data),
   b. zapíše příslušnou adresu a velikost do své části kruhové fronty,
   c. posune koncový ukazatel, čím předá odpovědnost za nově vyplněné buňky síťovému rozhraní,
5. síťové rozhraní ovládá hlavový ukazatel: kdykoliv zpracuje odchozí rámec, posune hlavový ukazatel tak, že paměť asociovaná s odeslaným rámcem se přesune do části fronty, která patří operačnímu systému.

Události, které se zpracováním kruhové fronty souvisí, signalizuje síťové rozhraní pomocí přerušení.[110]

4.3.7 Příjmová fronta (angl. receive queue nebo Rx queue) pracuje analogicky. Síťové rozhraní po přidání prvků tuto změnu signalizuje přerušením. Alokace paměti pro rámce je v kompetenci operačního systému – přesune-li operační systém nějakou položku (buňku) do části kruhové fronty,

- data v opačném směru mají také frontu
- rozhraní kopíruje data do příjmové fronty
- nová data ve frontě → událost
  - události lze sdružovat (jedna událost pro více rámců)
- příliš pomalé odebírání z fronty → ztráta dat

[106] Segment může znamenat jak kolizní tak broadcast doménu. Moderní ethernetové sítě ale žádné netriviální kolizní domény neobsahují, proto budeme uvažovat segment = broadcast doména.

[107] Adresu lze softwarově změnit.

[108] Přepínač ~ víceportový most – zařízení, které přeposílá rámce přijatém na jednom portu na jiný svůj port.

[109] Resp. čtvrté, ale tato v sítích IP deleguje většinu odpovědnosti za adresaci na vrstvu třetí. Jména uzlů (hostname) jsou podobně překládána na adresy třetí vrstvy.

[110] Detailněji se přerušeními budeme zabývat v osmé kapitole.

která náleží síťovému rozhraní, dává tím najevo, že paměť touto položkou odkázaná může být přepsána novými daty. Jakmile tak síťové rozhraní učiní, příslušnou buňku příjmové fronty přesune do části patřící operačnímu systému.
Je obvyklé, že každý blok paměti, který operační systém rozhraní předá, má velikost největšího možného rámce (MTU z angl. maximal transfer unit, obvykle 1500 bajtů), i když některá rozhraní umí příchozí rámce rozdělit do více buněk, je-li to potřeba.
Je-li fronta plná a rámce nadále přichází, dojde k jejich ztrátě (proto musí operační systém tuto frontu vyklidit dostatečně rychle). Rámce nemusí být operačním systémem zpracovány okamžitě – není nutné, aby pro ně alokovanou paměť předal zpět síťovému rozhraní, může je stejně dobře nahradit nově alokovanými bloky. Plné bloky jsou uvolněny (nebo znovu použity) jakmile je jejich obsah zpracován.

- rychlý adaptér může saturovat CPU
  - např. 10GbE nebo víceportový GbE
- takové adaptéry mohou spravovat více front
  - každá fronta má vlastní signalizaci události
  - různé fronty můžou zpracovávat různé procesory

4.3.8 Vícefrontové adaptéry Současná síťová rozhraní jsou schopna odesílat a přijímat rámce tak rychle, že jediné výpočetní (procesorové) jádro je nedokáže dostatečně rychle chystat (resp. zpracovávat) – s každým rámcem, resp. paketem, je totiž během jeho cesty mezi aplikací a síťovým rozhraním spojeno značné množství práce.
Tato rozhraní proto umožňují zpracování většího počtu Tx a Rx front, každá s vlastní dvojicí registrů (hlavového a koncového ukazatele) a vlastním přerušením. Je na operačním systému, aby tyto fronty aktivoval – obvyklé uspořádání je nastavit pro každé procesorové jádro jednu Tx a jednu Rx frontu.
Při odesílání síťové rozhraní prolíná rámce ze všech odchozích front, protože o tom, kterou frontu použít rozhoduje operační systém (zpravidla je to ta, která přísluší procesoru, který rámec odesílá).
Příjem je poněkud složitější, protože zde musí o výběru fronty rozhodnout síťové rozhraní. Toto je schopno rámce, resp. jejich vybrané části, filtrovat nebo hashovat a rozřazovat je podle výsledku. Cílem je udržet příbuzné rámce (pakety) pohromadě (ve stejné frontě), protože to vede na vyšší lokalitu zpracování, ale zároveň zaplňovat fronty co nejrovnoměrněji (zlepšuje se tím rozložení zátěže v systému).

- bezdrátové síťové rozhraní rodiny IEEE 802
- sdílené médium – elektromagnetické vlny ve vzduchu
- (prakticky) povinné šifrování
  - jinak velmi snadné odposlouchat nebo nabourat
- na poměry hardwaru velmi složitý protokol
  - částečně realizovaný ve firmware (běží na adaptéru)

4.3.9 WiFi Ve srovnání s relativní jednoduchostí drátových sítí je WiFi extrémně komplikovaná – je to způsobeno povahou komunikačního média, které je sdílené, náchylné na šum, lehce odposlouchatelné a obecně nespolehlivé. Zařízení, která se k bezdrátovým sítím připojují, jsou obvykle navíc mobilní a musí se přepínat mezi různými přístupovými body nebo i sítěmi.
Protože komunikace musí být chráněna šifrováním, je nutné, aby se klienti a přístupové body vzájemně autentizovaly a ustavily společné klíče. Autentizace je nutná proto, že jinak by aktivní útočník mohl předstírat, že je přístupovým bodem a nabídnout klientovi připojení. Komunikaci pak přeposílá skutečnému přístupovému bodu, ale stal se prostředníkem (angl. man in the middle), který může veškerou komunikaci číst, přesto že je zašifrovaná. Protože je autentizace stejně nevyhnutná, lze ji zároveň využít k řízení přístupu.
Protokoly z rodiny WiFi jsou částečně implementované v hardwaru, částečně ve firmwaru (softwaru, který je spuštěn na pomocném procesoru uvnitř bezdrátového síťového rozhraní) a softwaru (součást operačního systému, běží na hlavním procesoru).

Část 5: Souběžnost a synchronizace

Tato kapitola otevírá nový tematický blok – v této a následujících třech kapitolách se budeme zabývat vztahy mezi výpočty, které probíhají zároveň a vzájemně se při tom ovlivňují.

Doporučené čtení: A. Tanenbaum, H. Bos – Modern Operating Systems (4th Ed.): § 2.3.1 Race Conditions ◇ § 2.3.2 Critical Regions ◇ § 2.5 Classical IPC Problems.

5.1: Souběžnost

Prvním cílem bude vytvořit formální aparát pro práci s událostmi, které se odehrávají v čase, a které mezi sebou mají (nebo nemají) kauzální vztahy. Tento aparát nám pak umožní zkoumat různé jevy, které mohou nastat při interakci několika výpočtů.

- muselo se stát před
- acyklický orientovaný graf (DAG)
- vrchol = událost
- hrana = akce

5.1.1 Relace předcházení

Připomenutí: **Událost** je jev, který nastane v čase (ne nutně pevně určeném nebo známém), který můžeme pozorovat, a o kterém můžeme říct, že nastal před nebo po nějaké jiné události, případně že

s ní nastal souběžně. Relaci uspořádání, která tuto chronologii popisuje, budeme říkat **předcházení** (anglicky „happens before“).

Takto definovaná relace určuje vztah „muselo se stát před“ neboli kauzální návaznost. **Grafem** předcházení nazveme **tranzitivní redukci** této relace. Protože grafem uspořádání je acyklický orientovaný graf (angl. DAG = directed acyclic graph), jeho tranzitivní redukce je určena jednoznačně (znáte nejspíš jako Hasseův diagram).
Vrcholy grafu předcházení jsou **události** (něco se stalo), jeho hrany budeme nazývat **akce** (něco se děje). Akce tedy vedou od jedné události k nějaké další, ale nemohou je „přeskakovat“ (formálněji: existuje-li mezi událostmi A a B cesta délky alespoň 2, neexistuje hrana/akce která vede z A do B přímo).

5.1.2 **Souběžnost** je vztah mezi událostmi, které mezi sebou nemají přímou kauzální souvislost.

- souběžné = neuspořádané předcházením
- v grafu: nevede mezi nimi cesta
- události můžou nastat v libovolném pořadí
- nebo i zároveň (např. na různých CPU)

Připomenutí: O dvou různých událostech, nazvaných třeba A a B musí platit právě jedna z těchto možností (plyne z vlastností uspořádání):

1. A předchází B (tzn. A musí nastat první),
2. B předchází A (tzn. B musí nastat první),
3. A nepředchází B ani B nepředchází A – události jsou souběžné.

5.1.3 **Časový sled** Je-li nějaká relace předcházení lineární, mluvíme o **časovém sledu**.

- lineární uspořádání událostí
- lze i: přidělení časových razítek
- kompatibilita s relací předcházení

Připomenutí: **Časovým sledem** událostí rozumíme **lineární** uspořádání událostí, tedy takové, že pro **každou** dvojici A, B událostí platí buď:

- A předchází B nebo
- B předchází A.

Časový sled si můžeme představit i jako přiřazení **časového razítka** každé události takové, že žádné dvě události nenastanou ve stejné chvíli.

5.1.4 **Hazard souběhu**

- relace předcházení = abstrakce
- popisujeme jí vnější chování
- porušení abstrakce = hazard souběhu
- nemusí být nutně chybou

Připomenutí: Předcházení je **abstrakce**, která skrývá vnitřní detaily procesů (dějů odehrávajících se v čase), které se mohou stát v různém pořadí díky náhodným vlivům, a snažíme se jejich **vnější chování** popsat pomocí této relace. Vnější chování nějakého systému závisí pouze na jeho relaci předcházení, nikoliv už na tom, jak přesně budou v čase rozloženy konkrétní události.

Je-li takto zavedená abstrakce porušena, mluvíme o **hazardu souběhu**. Jinými slovy, hazard souběhu nastává kdykoliv vedou dva různé časové sledy, které jsou oba konzistentní s relací předcházení pro daný systém, k různému vnějšímu chování.
Existence hazardu souběhu nemusí nutně být chybou (i když často je), ale vždy se jedná o porušení abstrakce a tedy je nežádoucí i když v danou chvíli přímo nevede k nějakému problému. Někdy lze ale situaci řešit rozšířením abstrakce tak, aby rozdíl v chování nebyl pozorovatelný.

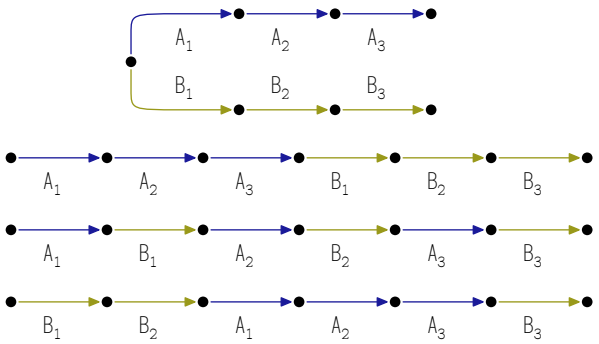
5.1.5 **Hybatel** Akce (které reprezentujeme hranami v grafu předcházení) jsou prováděny vždy nějakým **hybatelem** (v našem kontextu obvykle vláknem nebo periferií). Hrany (akce) tedy můžeme tomuto hybateli přisoudit (např. můžeme říct, že vlákno T má modré hrany a periferie P má žluté hrany). Zároveň hybatele můžeme považovat za „peška“ který se v grafu předcházení pohybuje po hranách své vlastní barvy.

- akce někdo provádí = hybatel
- typicky vlákno nebo periferie
- pešek který se pohybuje grafem
- „barva“ hrany = „barva“ peška

Příklad: Uvažme dvě vlákna, které obě provádí de facto stejný program (pro názornost ale v každém použijeme jiný registr):

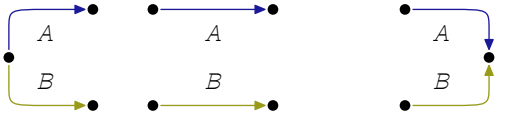
vlákno A	vlákno B
A_1: load r1 from V	B_1: load r2 from V
A_2: r1 ← r1 + 1	B_2: r2 ← r2 + 1
A_3: store r1 to V	B_3: store r2 to V

Jsou-li obě vlákna spuštěna naráz, **relace předcházení** a několik **konzistentních časových sledů** pro ně vypadá takto:



- souběžnost akcí (vs událostí)
- mezi souběžnými událostmi
  - nezávislé: vedou do souběžných
  - synchronizace: vedou do společné
- lze i souběžnost sledů

5.1.6 Synchronizace Akce jsou souběžné právě tehdy, když jsou souběžné libovolné dvě události, kterých se tyto akce týkají. Událost zejména nemůže být souběžná sama se sebou, a nemohou být souběžné ani události spojené nějakou akcí. Zbývají tedy 3 situace (souběžné akce jsou zde označeny A, B):



nezávislé synchronizace

Akce v prvních dvou případech jsou nezávislé a mohou proběhnout v libovolném pořadí (protože vedou do souběžných událostí).
Zajímavá je tedy zejména poslední situace, kdy se akce „sejdou“ ve společné události. Takové dvojici akcí budeme říkat **synchronizace** – tyto akce musí proběhnout „najednou“.
Definici souběhu akcí můžeme snadno rozšířit na sledy – souběžné sledy jsou takové, které nesdílí žádnou událost, s možnou výjimkou první a/nebo poslední.

- akce spojují události a stavy
- výpočet = posloupnost stavů + akcí
- stavový prostor: graf stavů + akcí
- stav ~ registry + paměť

5.1.7 Stavový prostor Pojem akce nám umožní dát do souvislosti **události** a **stavy**. O stavech jsme mluvili ve druhé kapitole v kontextu výpočtů, které jsme chápali jako **lineární** posloupnost změn stavu, kde přechod z jednoho stavu do dalšího byl **efektem** nějaké **instrukce**. Nyní si tento koncept zobecníme.

Příklad: Uvažme velmi jednoduchý program:

1. `r0 ← add r1 r1`
2. `r1 ← mul r1 r0`
3. `r0 ← mul r1 r1`
4. `stop`

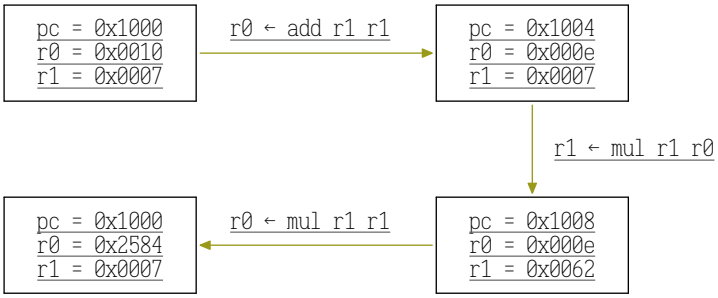
Tento by mohl vést např. k takovýmto změnám stavu:

pc	r0	r1	instrukce pod pc
0x1000	0x0010	0x0007	`r0 ← add r1 r1`
0x1004	0x000e	0x0007	`r1 ← mul r1 r0`
0x1008	0x000e	0x0062	`r0 ← mul r1 r1`
0x1008	0x2584	0x0062	`stop`

Stavový prostor je **orientovaný graf** kde vrcholy jsou **stavy** a hrany jsou **akce** (ve stejném smyslu jako v grafu předcházení). Pojem „stav“ je v tomto kontextu velmi abstraktní, nicméně můžeme si bez velké újmy nadále představovat stav jako:

- hodnoty procesorových registrů,
- buněk paměti (jak operační, tak paměti a registrů periferií).

Příklad: Stavový prostor předchozího programu můžeme reprezentovat tímto jednoduchým grafem:



Stav vnějšího světa nás bude zajímat jen nepřímo – z našeho pohledu „zevnitř počítače“ ho můžeme pozorovat výhradně skrze stav periferií, resp. ještě konkrétněji skrze stav jejich registrů. Vstupní periferie (ty, které reagují na změny vnějšího stavu světa) svůj stav mění nezávisle na operačním systému.[111]

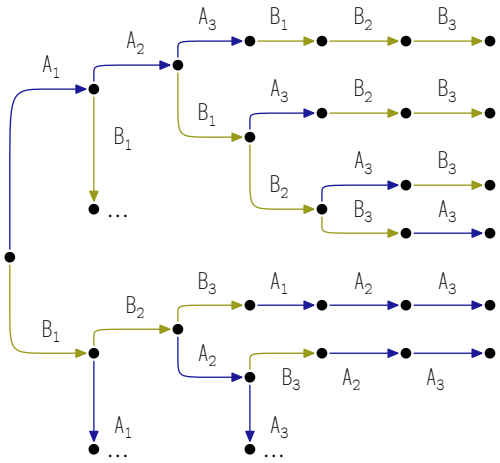
5.1.8 Běh Orientovanou cestu ve **stavovém prostoru** označíme za **běh**. Je zde jasná korespondence mezi během (posloupnost stavů propojených akcemi) a časovým sledem (posloupnost událostí propojených akcemi). Sekvenci akcí můžeme chápat i jako běh i jako sled – v obou případech stačí určit počáteční stav (událost) a zbytek běhu (sledu) je již určen akcemi jednoznačně.

- cesta ve stavovém prostoru
- lineární, příbuzný se sledem
- počáteční stav/událost + akce
- zobecnění výpočtu

Příklad: Podívejme se nyní opět na systém ze sekce 5.1.5:

vlákno A	vlákno B
A_1: load r1 from V	B_1: load r2 from V
A_2: r1 ← r1 + 1	B_2: r2 ← r2 + 1
A_3: store r1 to V	B_3: store r2 to V

Jeho **stavový prostor** je poněkud složitější:



Uvažme např. „nejsevernější“ cestu $A_1\ A_2\ A_3\ B_1\ B_2\ B_3$:

r1	r2	A	instrukce pod pc
0x0000	0x0000	0x0001	A_1: load r1 from V
0x0001	0x0000	0x0001	A_2: r1 ← add r1 1
0x0002	0x0000	0x0001	A_3: store r1 to V
0x0002	0x0000	0x0002	B_1: load r2 from V
0x0002	0x0002	0x0002	B_2: r2 ← add r2 1
0x0002	0x0003	0x0002	B_3: store r2 to V
0x0002	0x0003	0x0003	–

Srovnejme cestu $A_1\ A_2\ B_1\ A_3\ B_2\ B_3$:

[111] Vstupní periferií jsou tedy například i hodiny – reagují na změnu fyzického času.

r1	r2	A	instrukce pod pc
0x0000	0x0000	0x0001	A_1: load r1 from V
0x0001	0x0000	0x0001	A_2: r1 ← add r1 1
0x0002	0x0000	0x0001	B_1: load r2 from V
0x0002	0x0001	0x0001	A_3: store r1 to V
0x0002	0x0001	0x0002	B_2: r2 ← add r2 1
0x0002	0x0002	0x0002	B_3: store r2 to V
0x0002	0x0002	0x0002	–

Je vidět, že výsledný **stav** je pro tyto dva běhy **odlišný**. V situaci, kdy je hodnota uložená na adrese A součástí **vnějšího chování** – např. proto, že je to adresa výstupní periferie – jedná se o **hazard souběhu**.

5.2: Problémy souběžnosti

Se souběžností je spojena celá řada problémů. Většina z nich jsou instance hazardu souběhu, ale protože pod tuto hlavičku spadá mnoho různých případů, má smysl studovat některé kategorie blíže.

- běh, který nesmí být přerušen
- relativní vůči jinému běhu (resp. běhům)
- porušení je z definice chybou
- speciální případ hazardu souběhu

5.2.1 Kritická sekce Uvažme běh $R \equiv (r_1, r_2, ..., r_n)$ nějakého vlákna (nebo jiného hybatele) T_1 a **souběžný** běh $S \equiv (s_1, ..., s_n)$.[112] . Říkáme, že R je kritickou sekcí vůči S, vede-li $(r_1, ..., S, ..., r_n)$ k události „chyba“, a to přesto, že samotné běhy R ani S k chybě nevedou. Proto se tomuto typu problému také říká **chyba atomicity**.

Co přesně taková chybová událost obnáší je závislé na doméně: může to být zdvojení (nebo ztracení) peněz – objevuje se v klasickém příkladu na kritickou sekci „převod prostředků mezi účty“. Může to ale být havárie systému nebo celkem jakákoliv jiná nežádoucí situace.

Pro účely naší definice je důležité, že časové sledy $(S, r_1, ..., r_n)$ ani $(r_1, ..., r_n, S)$ k chybě **nevedou**. Chyba je způsobená konzistentním ale přesto nežádoucím uspořádáním **souběžných akcí**: jedná se tedy zejména o instanci **hazardu souběhu**.

Příklad: Vraťme se k vláknům z předchozí sekce:

vlákno A	vlákno B
A_1: load r1 from V	B_1: load r2 from V
A_2: r1 ← r1 + 1	B_2: r2 ← r2 + 1
A_3: store r1 to V	B_3: store r2 to V

Předpokládejme, že na začátku V = 0. Považujeme-li v takové situaci výsledek V = 1 za chybu, je běh $A \equiv (A_1, A_2, A_3)$ kritickou sekcí vůči běhu $B \equiv (B_1, B_2, B_3)$ a naopak B je kritickou sekcí vůči A.

Pozor! Taková symetrie je sice častá, ale není nutná. Uvažme programy:

vlákno A	vlákno B
A_1: r_1 ← 7	B_1: load r2 from V
A_2: store r_1 to V	B_2: r2 ← r2 + 1
A_3: store r1 to W	B_3: store r2 to V

Pak je za podobných podmínek B kritickou sekcí vůči A – za přípustné zde uvažujeme výsledky V = 7 nebo V = 8:

- sekvence $A_1\ A_2\ A_3\ B_1\ B_2\ B_3$ vede na výsledek 8,
- sekvence $A_1\ A_2\ B_1\ B_2\ B_3\ A_3$ vede na výsledek 8,
- sekvence $A_1\ B_1\ B_2\ B_3\ A_2\ A_3$ vede na výsledek 7,
- sekvence $B_1\ B_2\ B_3\ A_1\ A_2\ A_3$ vede na výsledek 7.

To jsou všechny možnosti jak by B mohlo nevhodně přerušit běh A, a tedy A není kritickou sekcí vůči B. Naopak to ale neplatí:

- sekvence $B_1\ B_2\ B_3\ A_1\ A_2\ A_3$ vede na výsledek 7,
- sekvence $B_1\ B_2\ A_1\ A_2\ A_3\ B_3$ vede na výsledek 1,
- sekvence $B_1\ A_1\ A_2\ A_3\ B_2\ B_3$ vede na výsledek 1,
- sekvence $A_1\ A_2\ A_3\ B_1\ B_2\ B_3$ vede na výsledek 8.

Protože výsledek A = 1 považujeme za nežádoucí, je B kritickou sekcí vůči A.

[112] Protože je souběžná, musí patřit nějakému jinému hybateli. Rozmyslete si proč.

5.2.2 Čtenáři a písaři Představme si situaci, kdy máme běhy R_1, R_2, ... R_n a běhy W_1, W_2, ..., W_m, pro které platí:

- asymetrická kritická sekce
- sdílený „zdroj“ (např. datová struktura)
- mnoho vláken může bezpečně číst
- modifikace ohrožuje čtení

1. $\forall i, j$ platí R_i **není** kritickou sekcí vůči R_j,
2. $\forall i, j$ platí R_i **je** kritickou sekcí vůči W_j,
3. $\forall i, j$ platí W_i **je** kritickou sekcí vůči W_j.

Takovou situaci nazýváme „čtenáři a písaři“ – máme tedy n čtenářů R_i a m písařů W_i. Čtenáři si vzájemně nepřekáží – mohou číst zároveň v libovolném pořadí. Písaři se chovají jinak: sdílená data nejen čtou, ale i **modifikují** – proto čtenář, který by byl přerušen písařem, by mohl přečíst nekonzistentní data. Podobně si překáží dva různí písaři – protože data jak čtou tak modifikují, může být výsledek opět nekonzistentní.
Nejjednodušší řešení je chovat se k problému stejně, jako by čtenáři byli vůči sobě vzájemně kritickou sekcí, i přesto že nejsou – to by znamenalo zabránit situaci, kdy čtenář přeruší jiného čtenáře. Takové řešení není příliš efektivní, proto budeme hledat lepší řešení (takové, které umožní čtenářům pracovat nezávisle na sobě – jinými slovy, nebudeme bránit souběhu čtenářů).

5.2.3 Hladovění Připustíme-li souběh čtenářů, vznikne nový problém: může nastat situace, kdy je neustále nějaký čtenář aktivní (další vlákno do sekce čtenáře vstoupí dřív, než ji opustí všechny předchozí). Při naivním řešení tohoto problému tak můžou být písaři trvale zablokováni (čekají, až všechna vlákna opustí příslušnou sekci R).

- co když je čtenář vždy aktivní?
- nemožnost pokračovat ve výpočtu → hladovění
- uváznutí je častý důvod hladovění

Hladovění je situace, kdy se vlákno natrvalo zasekne, aniž by svůj výpočet ukončilo. Zaseknutím v tomto případě myslíme, že nemůže vykonat žádnou další akci, resp. obecněji žádnou **užitečnou** akci. Tento typ problému nelze řešit synchronizací (naopak, synchronizace je jeho častým důvodem). Hladovění může mít různé důvody, tím nejznámějším a nejvíce prozkoumaným je **uváznutí** – situace, kdy vlákna čekají „v kruhu“ a tedy nemůže ani jedno z nich pokračovat.[113]

5.2.4 Souběžná datová závislost je vztah mezi akcemi, který je nejlépe vidět ve stavovém prostoru: je-li vstupem nějaké akce Z paměťová buňka nebo registr, do které naposled zapisovala akce X, říkáme, že Z má **datovou závislost** na X.

- výpočet a použití jsou souběžné
- pokus o použití příliš brzo → chyba
- často skryté časovou náročností
- méně časté než kritická sekce

Uvažme zároveň situaci, kdy máme dva běhy, $A \equiv (a_1, ..., X, ...a_n)$, $B \equiv (b_1, ..., Z, ..., b_m)$, které jsou zcela souběžné. To se v programu může lehce stát, a často je těžké takovou chybu odhalit, zejména je-li výpočet $(a_1, ..., X)$ krátký, zatímco $(b_1, ..., Z)$ dlouhý (časově náročný).
Všimněte si, že se opět jedná o hazard souběhu (akce Z může při nešikovném seřazení souběžných akcí přečíst nesprávnou vstupní hodnotu a spočítat tak nesprávný výsledek). Zároveň je ale vidět, že se jedná o kvalitativně odlišný problém, než kritická sekce. Existují-li souběžné datové závislosti, mluvíme také o **chybě pořadí**.

5.2.5 Producenti a konzumenti Uvažme situaci, kdy účelem několika vláken je vytvářet mezivýsledky určené k dalšímu zpracování (tato vlákna – producenty – označíme $P_1 ... P_n$), a několik dalších vláken tyto mezivýsledky dále zpracovává (tato nazveme konzumenty a označíme $K_1 ... K_m$). Všechna vlákna P_i jsou souběžná jak vzájemně tak s vlákny K_i. Tato souběžnost je důležitá a užitečná: umožňuje nám práci distribuovat na různá procesorová jádra a tím celý proces značně urychlit. Protože jsou ale všechna vlákna souběžná, můžou lehce nastat dvě problémové situace:

- souběžně vypočtené mezivýsledky
- může je zpracovat jedno z mnoha vláken
- co když nemá kdo zpracovat?
- co když není co zpracovat?

1. producenti mohou generovat výsledky rychleji, než je konzumenti dokáží zpracovávat – mezivýsledky se budou hromadit a postupně zaberou veškerou vyhrazenou paměť, nebo se začnou ztrácet (protože je producenti začnou přepisovat dříve, než jsou zpracovány),
2. konzumenti mohou zpracovávat výsledky rychleji, než je producenti dokáží vytvářet – bez synchronizace by taková situace vedla k opakovanému zpracování stejného mezivýsledku, případně k pokusu o zpracování nějaké nesmyslné informace (je-li mezivýsledek očekáván na adrese, kam ještě nebyl producentem zapsán).

V těchto dvou situacích se tedy musí konzumenti a producenti synchronizovat – je-li mezivýsledků nedostatek, upřednostníme práci producentů, naopak je-li jich přebytek, upřednostníme práci konzumentů. V ideálním případě tak, aby existovala nějaká pevná mez na počet nezpracovaných mezivýsledků (a tedy i na množství paměti potřebné pro jejich uložení).

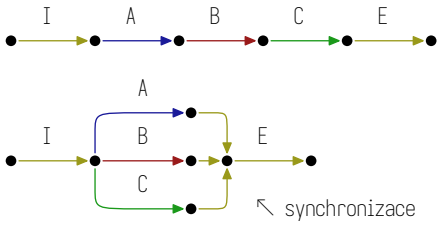
5.2.6 Rozvětvení a setkání Dosud jsme se zabývali problémy, kdy je v systému **příliš mnoho** souběžnosti, a některé konzistentní časové sledy jsou díky tomu chybné. Možná překvapivě může ale nastat i opačná situace, kdy je v systému souběhu **nedostatek**. Vzpomeňme si, že vlákno ze své

- dosud: příliš mnoho souběžnosti
- příliš málo je také špatné
- možné řešení: rozvětvit vlákno
- nutné vyčkat na všechny větve

[113] Detailněji se budeme hladověním (a zejména uváznutím) zabývat v přespříští kapitole.

podstaty vynucuje uspořádání všech svých akcí (výpočet je lineární a vlákno je výpočet, tedy musí mít i lineární relaci předcházení).
Akce, které nejsou souběžné, **nelze** provádět paralelně. Máme-li tedy více výpočetních jader, než máme vláken, nevyužíváme hardwarové zdroje efektivně. Relativně častým jevem je, že výpočet má dva (nebo několik) bloků, které lze provést v libovolném pořadí, aniž by se změnil výsledek. V takovém případě bychom chtěli výpočet **rozvětvit** tak, aby byl každý takový blok (běh) souběžný s těmi ostatními. Máme-li procesorů málo, vykonají se v libovolném pořadí (to nám nevadí), ale máme-ji jich dostatek (nebo přebytek), mohou se tyto běhy provést najednou (každý na jiném procesoru).
Tím ale vzniká nový problém – výpočet nemůže pokračovat, než skončí **všechny** takto vyčleněné bloky – jinak bychom se dostali do situace „výpočetní závislost“ (a tím do známého terénu „příliš mnoho souběžnosti“). Proto je potřeba, aby se běhy i setkaly.

Příklad: Uvažme výpočet $X \equiv (I, A, B, C, E)$ kde nezáleží na pořadí běhů A, B, C. Zároveň ale všechny tyto běhy vyžadují výsledek běhu I a běh E naopak vyžaduje výsledky všech běhů A, B, C. Graficky:



Část 6: Synchronizace

Z předchozí kapitoly známe základní pojmy a definice: událost, relace předcházení, souběžnost, akce, stav, stavový prostor. Také jsme se seznámili ze základními problémy souběžnosti. V této kapitole se podíváme na jejich standardní způsoby řešení a také na to, jaké nové problémy tato řešení přinesou.

Doporučené čtení: A. Tanenbaum, H. Bos – Modern Operating Systems (4th Ed.): § 2.3.3 Mutual Exclusion with Busy Waiting ◇ § 2.3.4 Sleep and Wakeup ◇ § 2.3.5 Semaphores ◇ § 2.3.6 Mutexes ◇ § 2.3.7 Monitors ◇ § 2.3.9 Barriers ◇ § 2.3.10 Avoiding Locks: Read-Copy-Update.

6.1: Synchronizační zařízení

Jak jsme viděli, většina problémů je způsobena tím, že je v systému příliš mnoho souběžnosti – akce, které by měly být uspořádány předcházením nejsou a vznikají tak různé instance hazardu souběhu (angl. race condition).

- smysl: omezit souběžnost
- různé problémy → různá zařízení
- datová struktura
- různé instance neinteragují

6.1.1 Společné vlastnosti Základní způsob, jak omezit souběžnost (aniž bychom ji úplně odstranili) je použitím **synchronizace**. K tomu nám budou sloužit různá **synchronizační zařízení**, která jsou uzpůsobena k řešení jednotlivých kategorií problémů. Budeme se zabývat jak jejich rozhraním (operace, sémantika) tak jejich možnou implementací.
Důležité: synchronizační zařízení je (de facto) datová struktura – v programu může existovat v mnoha nezávislých instancích. Mezi různými instancemi se **žádná synchronizace** neděje. Každé synchronizační zařízení má nějaký **stav**, který musí být někde uložen: instanci synchronizačního zařízení tedy ztotožníme s adresou,[114] na které je uložen jeho stav.

- řeší problém kritické sekce
- mutex = mutual exclusion device
- abstraktní stav: zamčen, odemčen
- 2 operace: lock + unlock
- lock může čekat

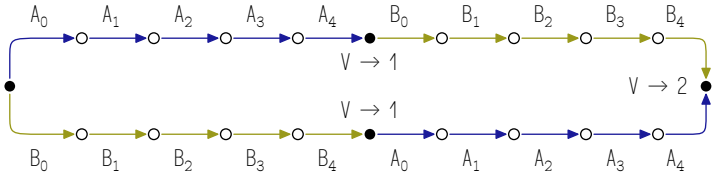
6.1.2 Vzájemné vyloučení (mutex) Zřejmě úplně nejjednodušším synchronizačním zařízením je **mutex**, určený k ochraně kritické sekce. Aby byla kritická sekce ochráněna, musí být chráněná jak kritická sekce samotná, tak i všechny běhy, vůči kterým je kritická, a to **tím stejným mutexem**.

[114] Nastává zde drobná komplikace v situaci, kdy dané zařízení synchronizuje vlákna v různých procesech a tedy je z každého vlákna obecně viditelné pod jinou virtuální adresou. Naopak fyzická adresa díky externímu stránkování v danou chvíli nemusí vůbec existovat, případně se může v čase měnit. Rigorózní definice tedy vyžaduje zavést ekvivalenci na virtuálních adresách, a synchronizační zařízení ztotožnit nikoliv s jednou specifickou adresou ale s třídou příslušné ekvivalence.

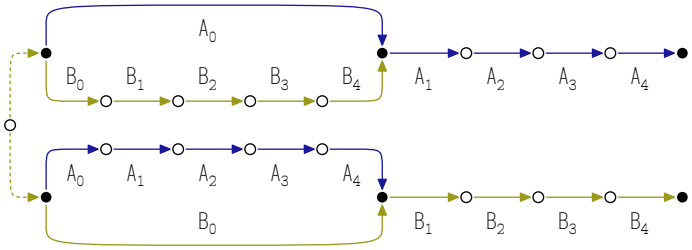
Příklad: Uvažme symetrickou dvojici kritických sekcí a instanci mutexu, kterou označíme adresou X. Vzpomeneme si na příklad s počítadlem z předchozí kapitoly, ale přidáme ochranu mutexem:[115]

vlákno A	vlákno B
A_0: lock X	B_0: lock X
A_1: load r1 from V	B_1: load r2 from V
A_2: r1 ← r1 + 1	B_2: r2 ← r2 + 1
A_3: store r1 to V	B_3: store r2 to V
A_4: unlock X	B_4: unlock X

Stavový prostor se přidáním synchronizace značně zjednoduší:



Relace předcházení se nám naopak trochu zkomplikuje. Je zde na místě poznamenat, že to, že nějaká událost se v relaci objeví, ještě nezaručuje, že může skutečně nastat. Abychom se ale v obrázku lépe vyznali, situaci, kdy může nastat nejvýše jedna z nějaké množiny událostí, naznačíme čárkovanými šipkami. Nemůže-li nastat nějaká událost, jistě pak nemůžou nastat ani žádné události, kterým tato „nemožná“ událost předchází.



Pozor: je velmi důležité, aby vlákno za žádných okolností neprovedlo operaci unlock na mutexu, který nevlastní – klasický mutex, tak jak je tu popsaný, nemůže tuto situaci detekovat a odemčení tak uspěje – přirozeně s katastrofálními důsledky.

- nejjednodušší implementace mutexu
- stav: 1 bit
- lock se opakovaně snaží získat zámek
  - aktivní čekání (angl. busy waiting)
- soutěž o spinlock na 1 CPU je špatná
  - mezi CPU často efektivní

6.1.3 Spinlock je synchronizační smyčka bez interakce s plánovačem (komunikace s plánovačem je potenciálně drahá operace – neplánujeme-li čekat dlouho, může být nevýhodné ji provést).
Implementace operace lock X je sémanticky ekvivalentní tomuto fragmentu kódu:

```
[0x100] r1 := 1             # chceme nastavit X na 1
[0x104] load r0 from X      # přečteme X            ←┐
[0x108] goto 0x104 if r0    # je-li nenulové, opakuj ┘
[0x10c] store r1 to X       # nastav X = 1
```

Problém této implementace spočívá v tom, že nebude fungovat – mezi operacemi load a store můžou dvě různá vlákna přečíst hodnotu 0 a obě tak dospět tak k závěru, že jejich operace lock uspěla.
V moderních systémech[116] se spinlock implementuje pomocí **atomické instrukce**, která umožní provést operaci load, úpravu hodnoty a store jako jediný krok, který je pro všechny procesory v systému pozorovatelný pouze jako celek. Atomických operací existuje celá řada, my použijeme operaci cmpxchg (z angl. „compare and exchange“), která má 3 operandy: adresu, očekávanou hodnotu a požadovanou hodnotu. Instrukce cmpxchg X r0 r1 odpovídá v jazyce C tomuto složenému příkazu:

```
if ( *X == r0 )
    *X = r1;
else
```

[115] Synchronizační operace zde zapisujeme jako **abstraktní instrukce** – v skutečném programu se příslušné operace řeší buď sekvencí konkrétních instrukcí, nebo voláním podprogramu.

[116] Nemáme-li k dispozici atomické instrukce, existují složitější algoritmy, které vzájemné vyloučení realizují, nejznámější z nich je algoritmus Petersonův. Protože se v současné praxi nepoužívá, nebudeme se jím blíže zabývat – zájemce odkazujeme na sekci 2.3 knihy Modern Operating Systems (A. Tanenbaum).

```
r0 = *X;
```

nebo těmto strojovým krokům:

```
[0x100] load r2 from X    # přečti hodnotu z adresy X
[0x104] r3 := r2 == r0    # srovnej ji s r0
[0x108] r0 := r2          # r0 nastav na přečtenou hodnotu
[0x10c] goto 0x114 if r3  # jsou-li odlišné, konec
[0x110] store r1 to X     # jsou-li stejné, přepiš X hodnotou r1
```

S instrukcí cmpxchg vypadá operace lock pro spinlock takto:[117]

```
[0x100] r1 := 1             # chceme nastavit X na 1
[0x104] r0 := 0             # očekáváme X = 0          <┐
[0x108] cmpxchg X r0 r1     # srovnej a přepiš          │
[0x10c] goto 0x104 if r0    # selže-li srovnání, opakuj ┘
```

Operace unlock je jednoduchá: na adresu X zapíše hodnotu 0.
Spinlock má několik výhod – krom jednoduchosti implementace je také velmi úsporný (celý stav se vejde do jediného bajtu paměti), má minimální prodlevu při čekání (čekající vlákno zámek získá okamžitě), a v situaci, kdy o zámek není soutěž, je v podstatě optimální.
Spinlock (resp. aktivní čekání obecně) má ale jednu důležitou nevýhodu: soutěží-li o stejný zámek dvě vlákna, která sdílí procesorové jádro, čekajícímu vláknu se nemůže podařit zámek získat, dokud nebude vlastník zámku probuzen.[118] Proto je použití spinlocku v uživatelských programech obvykle chybou (nelze zaručit, že vlákna budou naplánována na různá procesorová jádra).
Druhou potenciální nevýhodou je, že chrání-li spinlock delší kritickou sekci, doba čekání se prodlužuje, a tím i čas „spálený“ aktivním čekáním (čas, který by jiné vlákno mohlo využít produktivně). Proto je spinlock vhodný jen pro ochranu kritických sekcí, které trvají v průměru jen krátkou dobu.

- vyžaduje spolupráci s plánovačem
- stav: 1 bit + fronta vláken
- lock selže → „čekám na událost“
- unlock → „událost nastala“

6.1.4 Uspávající mutex Opačný extrém představuje „uspávající“ mutex, kterého operace lock je realizována jako služba operačního systému (systémové volání); jádro nejprve ověří, je-li zámek odemčen:

1. pokud ano, poznačí ho jako zamčený a vrátí kontrolu vláknu, které zámek vyžádalo,
2. v opačném případě zařadí vlákno do fronty, která danému mutexu náleží, a nechá plánovač probudit nějaké jiné vlákno.

Odemčení zámku pak ověří, je-li nějaké další vlákno ve frontě, a pokud ano, mutex ihned zase zamkne a předá ho prvnímu vláknu z této fronty.
Problém takto implementovaného mutexu je efektivita: systémové volání má oproti atomické instrukci velmi velkou režii (řádově stovky instrukcí). Rádi bychom našli řešení, které kombinuje silné stránky (alespoň z pohledu efektivity) obou přístupů.

- rozdělení stavu mezi systém a vlákno
- spinlock + počítadlo + uspávání
- systémová část = futex
- probuzení → počítadlo pokusů o zamčení

6.1.5 Rychlý mutex (spinlock + futex) Řešení spočívá v rozdělení odpovědnosti za dvě části stavu mezi uživatelské vlákno a operační systém. Rozhodnutí o tom, je-li zámek zamčený nebo nikoliv, vyřeší vlákno ve vlastní režii (stejně, jako by se jednalo o spinlock – pomocí atomické instrukce). Systémové volání se provede pouze v případě, kdy pokus o zamčení selže.
Situace s odemykáním je poněkud složitější – mohlo by se zdát, že zde se systémovému volání nevyhneme, protože nemůžeme vědět, jestli nějaké vlákno na zámek čeká nebo nikoliv. Ale i tento problém má efektivní řešení: uživatelskou část stavu rozšíříme z jediného bitu (odemčen/zamčen) na počítadlo vláken, která se pokusila o zamčení (bez ohledu na úspěch). Při odemykání pak požádáme systém o probuzení nějakého čekajícího vlákna pouze v případě, kdy je počítadlo v této chvíli větší než jedna.[119]
Systémová část stavu se jmenuje futex (z angl. „fast mutex“), a sestává pouze z fronty uspaných vláken. Fronty jsou s jednotlivými uživatelskými mutexy svázány adresou (tzn. operační systém

[117] Operace cmpxchg není sice nejjednodušší možná, má ale jinou výhodu – umožňuje atomicky provést libovolný výpočet nad načtenou hodnotou. Zejména ničemu nepřekáží, že takový výpočet nějakou dobu trvá a sám o sobě není atomický: změní-li se během výpočtu hodnota na adrese X, operace cmpxchg selže, a výpočet provedeme znovu (s nově načtenou hodnotou).

[118] Tento problém může být výrazně zhoršen prioritním plánováním vláken – má-li čekající vlákno vysokou prioritu, může se stát, že potrvá velmi dlouho, než bude vlákno, které zámek vlastní, probuzeno, aby mohlo zámek uvolnit. Nebo se případně nemusí probudit nikdy.

[119] Je zde potřeba jisté opatrnosti, aby nemohla nastat situace, kdy vlákno A přečte při odemykání hodnotu 1, jiné vlákno B mezitím selže při pokusu o zamčení mutexu a zařadí se do fronty, a vlákno A systém nepožádá o probuzení čekajícího vlákna – pak by totiž vlákno B zůstalo ve frontě potenciálně navždy.

má pomyslný slovník front, kde klíčem je adresa příslušného mutexu).[120] Systémové volání, které s futexem pracuje, má rozhraní podobné, jako již zmiňovaná instrukce cmpxchg. Díky tomu je možné uživatelskou část implementovat bezpečně (bez hazardu souběhu).

6.1.6 Rekurzivní mutex Uvažme situaci, kdy máme dva podprogramy, f a g a oba jsou chráněny stejným mutexem. Zároveň bychom ale chtěli v implementaci f využít podprogramu g. Naivní řešení nebude fungovat – pokus o opětovné zamčení mutexu ve stejném vlákně toto vlákno zablokuje.

- podprogramy se stejným mutexem
- nelze přímo zavolat (blokuje)
- někdy lze refaktorovat program
- jinak rekurzivní mutex
- id vlastníka + počítadlo zanoření

V jednodušších případech lze problém vyřešit přeuspořádáním programu, např. vyčlenit tělo funkce g do pomocné funkce g' a položit g ≡ lock; g'; unlock. Ne vždy je ale takové řešení praktické – kritická sekce nemusí pokrývat celé tělo funkce, a museli bychom alespoň část kódu duplikovat mezi „zamčenou“ a „odemčenou“ verzí téže funkce.

Jiným možným řešením je rozšíření mutexu o možnost opakovaného zamčení ve stejném vlákně. K tomu potřebujeme přidat dva atributy:

1. identifikátor aktuálního vlastníka, abychom v operaci lock rozeznali, je-li mutex zamčený aktuálním vláknem (operace lock uspěje) nebo nějakým jiným (musíme čekat),
2. počítadlo zamčení – každá operace lock musí mít odpovídající operaci unlock, ale provedeme-li dvě operace lock, nesmíme zámek uvolnit dříve, než proběhne unlock odpovídající **první** operaci lock (která zámek pro vlákno získala); unlock tedy sníží počítadlo a zámek odemkne pouze dojde-li toto na nulu.

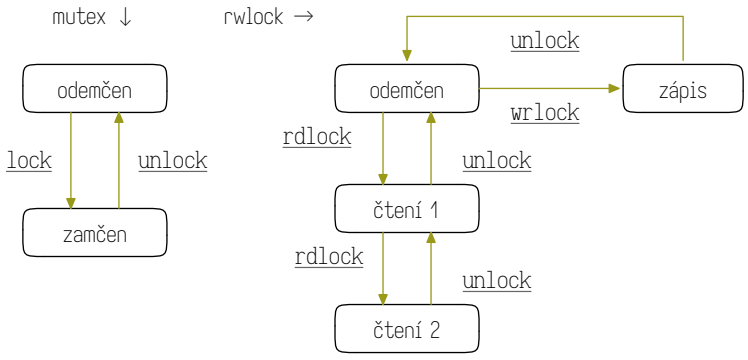
Takto upravený mutex má oproti tomu klasickému ještě jednu výhodu – dokáže spolehlivě poznat, že došlo k pokusu odemčení zámku nesprávným vláknem. Cena za tyto výhody spočívá zejména ve větší reprezentaci stavu – obvykle jeden až dva ukazatele (4–16 bajtů), v závislosti na hardwarové architektuře a velikosti identifikátoru vlákna.

6.1.7 Zařízení rwlock Zařízení rwlock můžeme považovat za rozšíření mutexu o nový stav (resp. sérii stavů) – zamčeno pro čtení. Toto zařízení má 3 operace:

- řeší situaci „čtenáři a písaři“
- stavy odemčeno, čtení 1, 2, …, zápis
- operace rdlock, wrlock, unlock
- stav zápis → rdlock blokuje
- stav čtení v zápis → wrlock blokuje
- unlock → poslední odemkne

1. rdlock – operace zamčení pro čtenáře (sdílené zamčení), uspěje buď ve stavu **odemčeno** nebo ve stavu **zamčeno pro čtení**, ve stavu **zamčeno pro zápis** blokuje (čeká),
2. wrlock – operace zamčení pro písaře (výlučné/exkluzivní zamčení), uspěje pouze ve stavu **odemčeno**, jinak blokuje,
3. unlock – ve stavu **zamčeno pro zápis** zámek odemkne (umístí do stavu **odemčeno**), ve stavu **zamčeno pro čtení** sníží počítadlo čtenářů o jedna – byl-li odemykající čtenář poslední, umístí zařízení do stavu **odemčeno**.

Srovnejme stavové diagramy klasického dvoustavového mutexu a zařízení rwlock:



Tento typ zámku lze dále zobecňovat, např. přidáním stavu „sdílený (souběžný) zápis“, kdy některé operace zápisu lze provádět souběžně s jinými.

6.1.8 Read-Copy-Update V některých případech lze problém čtenářů a písařů řešit bez použití jakýchkoliv zámků[121] – mechanismem, kterému se říká RCU (z angl. read-copy-update; volně přeloženo čti–kopíruj–uprav). Takto koncipovaná synchronizace se zcela vyhýbá kritickým sekcím (to je důvod, proč není potřeba použít žádné zámky).

- alternativní řešení čtenářů a písařů
- umožňuje čtenářům pracovat i během zápisu
  - písař vytvoří kopii kterou upraví
  - pozdější čtenáři vidí novou verzi
- kdy je bezpečné uvolnit předchozí verzi?

Princip fungování je jednoduchý: místo aby písař měnil datovou strukturu na místě, vytvoří její kopii (skutečnou nebo pomyslnou resp. částečnou) a tuto kopii upraví – obě operace jsou

[120] Připomínáme zde problém s virtuálními vs fyzickými adresami, je-li synchronizace prováděna mezi různými procesy. Jádro může např. použít nějaký vnitřní klíč podsystému správy paměti, který zůstává v platnosti i pro stránky, které právě nejsou v operační paměti.

[121] Tedy bez toho, aby muselo kterékoliv vlákno na synchronizaci čekat.

bezpečné, protože kopírování je vzhledem k původní datové struktuře forma čtení, a nová kopie je prozatím ve výlučném vlastnictví písaře (jiná vlákna k ní nemají vůbec přístup). Jakmile je úprava hotová, přesměruje všechny budoucí čtenáře na tuto novou verzi, obvykle tím, že upraví sdílený ukazatel.[122]

Starší kopie je možné uvolnit až ve chvíli, kdy je jisté, že k nim nepřistupuje žádný čtenář – protože čtenáři jsou se zapisujícím písařem souběžní, potřebujeme další synchronizační mechanismus – častou volbou je **počítadlo odkazů**, které čtenáři udržují; poslední čtenář pak již nepotřebnou starší verzi dat uvolní.

Srovnejte: copy-on-write řešení konzistence souborového systému.

- skrytá data + viditelné procedury (jako OOP)
- jazykový prostředek / vzor (nikoliv OS)
- vnitřně postavený na vzájemném vyloučení
- vstoupit smí pouze jedno vlákno najednou

6.1.9 **Monitor** Programování s jednotlivými mutexy je dost náročné na koordinaci – každá třída kritických sekcí musí mít vlastní zámek, který je konzistentně uzamykaný a odemykaný, a jakákoliv chyba v zamykací logice znamená hazardy souběhu, které je těžké odhalit a odladit. Proto je velmi žádoucí mít synchronizační primitiva vyšší úrovně, resp. taktiky (návrhové vzory), jak zamykání uspořádat tak, abychom v programech podobné chyby synchronizace nevytvářeli.

Monitor je jednoduchou, ale velmi užitečnou abstrakcí – uvažme datovou strukturu, která sestává z nějaké datové reprezentace a operací nad ní. Realizace jednotlivé operace je často kritickou sekcí vůči ostatním operacím (to platí zejména pro ty operace, které strukturu mění).

Organizujeme-li program tak, že jediný přístup přímo k datům takové struktury je skrze operace definované v podprogramech, lze datové struktuře jako celku přidružit jeden mutex, který se na začátku každé operace zamkne a na konci odemkne. Lze si zde představit dvě implementační strategie:

1. v ideálním případě zamykání a odemykání v operacích řeší překladač (týká se některých OOP jazyků); pak lze navíc v situaci, kdy jedna operace volá jinou operaci jako podprogram, zamykání přeskočit („staticky“ víme, že mutex je již zamčený),
2. v méně ideálním případě, kdy je „monitor“ pouze ručně udržovaný invariant, můžeme využít rekurzivního mutexu (vnořené zamykání se tak vyřeší „dynamicky“ – za běhu).

Je také potřeba si uvědomit, že monitor přidává do programu více synchronizace, než je striktně potřeba (a tedy výsledek vykazuje méně souběžnosti, než by mohl). Jedná se zde o kompromis: méně souběžnosti sice znamená méně příležitostí pro paralelizaci, ale také méně příležitostí pro chyby. V mnoha programech je monitor kompromisem velmi rozumným.

Důležité: monitor nemůže zaručit, že program **jako celek** nebude obsahovat chyby synchronizace – není těžké představit si invariant, který přesahuje hranice jedné datové struktury, a kterého dočasné porušení je kritickou sekcí.

Příklad: Máme-li dvě fronty, každou jednotlivě chráněnou monitorem, a vlákno A přesouvá prvek P z jedné fronty do druhé, může vlákno B pozorovat situaci, kdy P není ani v jedné z front (nebo je naopak v obou frontách).

V takové situaci nám monitor, který chrání pouze jednotlivou datovou strukturu, nepomůže (resp. ne vždy je možné takto provázané datové struktury sloučit do společného monitoru).

- řeší problém uspořádání
- stav: 1 bit nebo množina vláken
- operace: wait, signal
- wait blokuje do příštího signal
- lze i s aktivním čekáním

6.1.10 **Podmínková proměnná** Pro řešení problému souběžné datové závislosti lze použít synchronizační zařízení, kterému se říká podmínková proměnná (angl. condition variable). Operace jsou jednoduché:

- wait – čekej – zablokuje volající vlákno až do chvíle, kdy nějaké jiné vlákno provede operaci signal,
- signal odblokuje vlákno (tzn. umožní mu pokračovat ve výpočtu) zablokované operací wait.

Narazíme zde ale na drobný problém s popisem stavu – lze si totiž představit dvě verze, které vedou na trochu odlišné chování (sémantiku):

1. stav je jediný bit který reprezentuje, zda na proměnnou nějaké vlákno čeká – čekat pak může nejvýše jedno vlákno (ve stavu „čeká“ není operace wait přípustná),
2. (abstraktním) stavem je množina čekajících vláken, operace wait aktuální vlákno do této množiny přidá a zablokuje – operace signal v takovém případě může probudit jedno nebo všechna nebo nějakou podmnožinu čekajících vláken.

[122] Existuje-li nejvýše jeden písař, tento systém funguje bez dalších problémů. V situaci, kdy může být písařů několik, přímočará implementace vede k situacím, kdy mezi písaři vzniká hazard souběhu – projeví se pouze jedna úprava z potenciálně několika souběžných – ta, která jako poslední upraví sdílený ukazatel. Problém lze samozřejmě řešit návratem k vzájemnému vyloučení (výhradně mezi písaři) a tedy zámky, ale v některých případech existují i jiné možnosti.

Obvyklým způsobem implementace je interakce s plánovačem, podobně jako třeba u uspávajícího mutexu – vlákno, které na podmínkovou proměnnou čeká, tak uvolní procesor pro nějakou jinou práci. Podmínkovou proměnnou lze implementovat i pomocí aktivního čekání – taková implementace se podobá na spinlock, ale nepoužívá se příliš často – u problému datové závislosti je obecně mnohem těžší předvídat, jak dlouho bude vlákno čekat.

6.1.11 Semafor Klasický semafor přímo neřeší žádný zajímavý problém (který by mutex nebo některá jeho varianta neřešila lépe), nicméně ze dvou semaforů lze sestavit řešení problému producentů a konzumentů.[123]

- motivace: „producenti a konzumenti“
- stav: počítadlo v rozsahu 0–n
- down = wait (zamkni), blokuje na 0
- up = post (odemkni), neblokuje

Semafor lze také mírně upravit tak, aby problém producentů a konzumentů řešil přímo – stačí změnit operaci post tak, aby při pokusu o překročení hodnoty n blokovala. Takovému semaforu bychom mohli říkat třeba symetrický (díky nově získané symetrii operací wait a post). Klasické řešení problému producentů a konzumentů pak není nic jiného, než kombinace dvou klasických, asymetrických semaforů do jednoho symetrického.

Klasické řešení funguje takto:

1. vytvoříme dva semafory, E, F, první s počáteční hodnotou n a druhý s počáteční hodnotou 0,
2. „produkuj výsledek“ odpovídá operaci wait E ; post F,
3. zatímco „použij výsledek“ provede operaci wait F ; post E.

Význam semaforu E je počet volných míst ve frontě mezivýsledků, zatímco význam semaforu F je počet zabraných míst ve frontě mezivýsledků. Operace udržují invariant E + F = n.

6.1.12 Bariéra Inverzní semafor – umožňuje pokračovat ve výpočtu, až když na bariéru čeká dostatek vláken. Smyslem bariéry je řešit problém rozvětvení a setkání (zejména jeho druhou část – setkání většího počtu vláken, které řeší nezávislé podproblémy nějakého většího výpočetního celku).

- řeší problém rozvětvení a setkání
- stav: počítadlo
- operace init a wait
- wait blokuje než ho zavolají všichni
- čekající vlákna obvykle spí

Bariéru lze implementovat například jako počítadlo + podmínkovou proměnnou:

1. init nastaví počítadlo na počet vláken,
2. wait atomicky sníží počítadlo o jedna:
   - je-li výsledek > 0, čeká na podmínku,
   - je-li výsledek 0, signalizuje podmínku.

Lze teoreticky implementovat i přímo, s použitím aktivního čekání, taková implementace ale neplní obvyklý účel bariéry – lze ji použít pouze v situaci, kdy je zaručeno, že všechna vlákna dorazí do místa synchronizace v přibližně stejnou dobu.

Část 7: Komunikace, uváznutí

Doporučené čtení: A. Tanenbaum, H. Bos – Modern Operating Systems (4th Ed.): § 2.3.8 Message Passing ◇ Chapter 6: Deadlocks.

7.1: Sdílená paměť

Přirozenou formou komunikace je sdílená paměť, kdy více vláken přistupuje ke společné datové struktuře v této paměti uložené. Existuje celá dlouhá řada specializovaných „komunikačních“ datových struktur – ukážeme si pouze některé z nich.

7.1.1 Komunikace a synchronizace Každá komunikace je zároveň formou synchronizace: každé **čtení** nějaké informace musí být **předcházeno** odpovídajícím **zápisem**. Nelze dost dobře obdržet zprávu, která dosud nebyla odeslána. Proto je každé komunikační zařízení také zařízením synchronizačním. Opačný vztah ale neplatí: komunikace přidává možnost předat nějakou informaci.

- komunikace → synchronizace
- zápis předchází čtení
- producenti a konzumenti
- komunikace → produkce zpráv

Komunikační zařízení mají často i další synchronizační funkci, než jen tu, která přímo plyne z kauzální souvislosti zápis → čtení. Nejčastější je chování, které odpovídá symetrickému semaforu, protože většina komunikace je zároveň zobecněnou verzí problému producentů a konzumentů (zobecnění spočívá v tom, že účastníci komunikace jsou obvykle zároveň producenty i konzumenty).

7.1.2 Datové struktury Ke komunikaci lze použít celkem libovolnou datovou strukturu, je-li chráněna mutexem nebo jiným vhodným synchronizačním zařízením (např. je realizována jako monitor).

- mutex → schránka: čtu × zapisuji
- soutěž o zámek brzdí komunikaci
- specializované struktury bez zámků
- důležité zejména fronty

[123] Semafor lze teoreticky využít také k přidělování nějakého omezeného zdroje, který je k dispozici v n instancích, ale takové použití je spíše umělé.

Problém, který zde vzniká, je **soutěž** (angl. contention) o příslušný zámek.[124] Takové kombinované komunikační zařízení si lze představit jako klasickou poštovní schránku – vkládá-li právě pošťák psaní do schránky, nemůžete si z ní dost dobře ve stejnou chvíli vyzvedávat nějaké jiné – jak přesně datová struktura funguje vnitřně z tohoto pohledu není důležité.
Zajímavější budou struktury, které jsou ke komunikaci přímo navržené, a umožňují souběžné použití dvěma nebo více vlákny. Nejdůležitější jsou komunikační **fronty**, které umožňují předávat data v pevném pořadí, ale nevyžadují synchronizaci při každém jednotlivém předání a umožňují tak větší celkovou míru souběžnosti.

- vložení/zápis → odeslání
- odebrání/čtení → příjem
- 1 producent, 1 konzument
- efektivní kruhová fronta

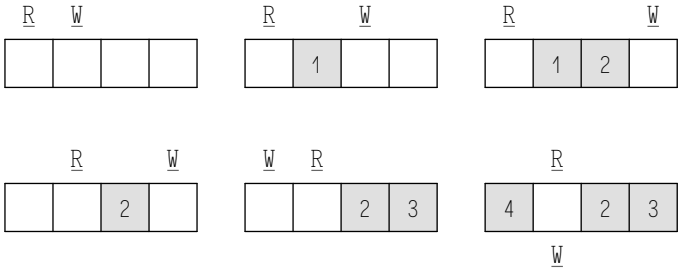
7.1.3 Komunikační fronta Možnosti implementace: libovolná fronta chráněná mutexem, kruhová fronta (pevná velikost), jednostranně zřetězený seznam (použitelné pouze pro velké položky).
Použití: 1 producent + 1 konzument. Je-li potřeba obousměrné komunikace, použijí se dvě fronty (každým směrem jedna).
Obzvláště efektivní je **kruhová fronta**, která má pevnou velikost a dva „ukazatele" – jeden pro čtení (odebrání) a jeden pro zápis (vložení). Pokusí-li se „čtecí" ukazatel předběhnout ten zapisovací, fronta je prázdná (čtení musí vyčkat, než dojde k nějakému zápisu). V opačné situaci – zapisovací ukazatel se pokusí předběhnout ten čtecí – je fronta plná (zápis musí vyčkat, než se čtením nějaká buňka uvolní).
Zásadní implementační výhodou je, že každá strana mění pouze jeden z ukazatelů – díky tomu není ani vybírání ani vkládání hodnot kritickou sekcí, je-li vhodně uspořádané. Označme ukazatele R (čtecí) a W (zapisovací), S je počet buněk a operace % získá zbytek po dělení.
Algoritmus vložení (zápisu):

1. srovnej W a R; je-li W rovno R, opakuj srovnání[125] (fronta je plná), jinak
2. zapiš hodnotu na pozici W (tato jistě není obsazená, ani ji nemůže čtoucí vlákno přečíst),
3. změň W ← (W + 1) % S – je důležité, aby byl zápis z bodu 2 již ukončen.

Algoritmus odebrání (čtení):

1. srovnej W a R; je-li (R + 1) % S rovno W, opakuj srovnání (fronta je prázdná), jinak
2. přečti hodnotu z pozice (R + 1) % S (tato jistě obsahuje platná data),
3. změň R ← (R + 1) % S (je důležité, aby bylo čtení z bodu 2 již ukončeno).

Příklad: Uvažme frontu o 4 buňkách a sekvenci operací push 1, push 2, pop, push 3, push 4. Všimněte si, že použitelná kapacita fronty je o jedna menší, než počet buněk.



- specializace komunikační fronty
- pracuje s bajty
- efektivní dávkové operace
- obvykle poskytuje OS

7.1.4 Roura Specializace fronty, která pracuje s bajty. Vyznačuje se zejména efektivními dávkovými operacemi – zápisem resp. čtením většího počtu položek (bajtů) najednou. Často je poskytována operačním systémem. Prakticky vždy implementovaná jako omezená – existuje nějaký maximální objem dat, který lze do roury zapsat, než musí další zápis vyčkat na čtení. Stejně jako omezená fronta má tedy synchronizační chování stejné jako symetrický semafor.

- zobecnění komunikační fronty
- N producentů, M konzumentů
- bez zámků: zřetězený seznam
- použití: společný seznam úloh

7.1.5 Sdílená fronta Zobecnění komunikační fronty – vkládat i vybírat může více než jedno vlákno. Příklad implementace: zřetězený seznam bez zámků, postavený na operaci cmpxchg. Použití: producenti a konzumenti, společná fronta úloh, paralelní prohledávání stromů a grafů.

- operace: dotaz, vložení
- použití: ukončené úlohy
- implementace read-copy-update
- nebo cmpxchg, nebo spinlock / buňka

7.1.6 Sdílená množina Implementace: pomocí zámku, read-copy-update strom, hashovací tabulka + spinlock v každé buňce, hashovací tabulka + cmpxchg.

[124] Pro občasnou komunikaci takové uspořádání nepředstavuje zásadnější problém. Je-li ale komunikace intenzivní, prostoje kvůli zamykání mohou tvořit výraznou část celkového výpočetního času.
[125] Naivní implementace vede na aktivní čekání. Frontu lze doplnit vhodným synchronizačním zařízením, které čekající vlákno místo toho uspí.

Použití: uzavřená množina (prohledávání grafu), obecněji uzavřené podúkoly, množina stránek, které je potřeba přepsat do souborů („dirty“ stránky), atp.

7.2: Předávání zpráv

Alternativou ke sdílené paměti je předávání zpráv – má dvě výhody:

1. je bezpečnější na použití – méně problémů s hazardy souběhu,
2. lze použít i mezi různými počítači (po síti),

a dvě nevýhody:

1. méně pohodlné, protože nelze odkazovat do jiných datových struktur – všechna relevantní data je potřeba „přibalit“ do samotné zprávy,
2. méně efektivní, jednak kvůli samotné konstrukci zpráv, jednak kvůli režii s kopírováním a předáváním zprávy.

7.2.1 Zpráva Z pohledu systému pro předávání zpráv je obvykle obsah zprávy neprůhledný (jsou to pouze bajty), důležitá jsou připojená metadata: zejména adresát. Způsob adresace je zároveň hlavní rozdíl mezi předáváním zpráv a frontami – mezi pevnou dvojicí komunikujících vláken je jinak předávání zpráv ekvivalentní dvojici komunikačních front.
Srovnejte: pakety.

- blok dat → interpretuje příjemce
- metadata → zejména adresát
- doručeno jako celek
- spolehlivé × nespolehlivé

7.2.2 Základní operace Nejjednodušší systém předávání zpráv má dvě operace: odešli zprávu a přijmi zprávu. Operace „přijmi“ může v situaci, kdy žádná doručená zpráva není k dispozici buď blokovat, nebo oznámit selhání. Operace odeslání může obvykle blokovat také, protože schopnost systému pamatovat si odeslané (ale dosud nepřijaté) zprávy je omezená (stejná situace jako v obecném problému konzumentů a producentů, resp. u fronty ve sdílené paměti).
Je-li maximální počet nedoručených zpráv, které je systém ochoten uložit, nulový, mluvíme o tzv. „setkání“ (rendezvous) – odesílající a přijímající vlákno se musí „sejít“ aby si mohly zprávu předat. Synchronizační aspekt komunikace je zde obzvláště výrazný – dokonce mnohem výraznější, než je obvyklé u **synchronizačních** zařízení.
Je-li důležitá propustnost a/nebo latence, mohou být obě operace realizované **asynchronně** – v takovém případě lze často ušetřit jedno kopírováni zprávy, za cenu komplikovanější správy paměti.

- odešli → zprávu převezme systém
- přijmi → doručí jednu zprávu
- volitelně může blokovat (obojí)
- volitelně asynchronní (bez kopie)

7.2.3 Zprostředkovatel Systém předávání zpráv má často jednoho nebo více **zprostředkovatelů** (angl. **broker**), kteří zprávy přijímají od jednotlivých komunikujících entit (klientů) a realizují jejich doručení. Mezi klientem a zprostředkovatelem musí existovat nějaký komunikační kanál, který umožňuje zprávy fyzicky předat (logicky nelze využít samotné předávání zprávy klientem k předání zprávy zprostředkovateli). Tímto komunikačním kanálem může být sdílená paměť, speciální systémové volání, síťové spojení, atp.
Úkolem zprostředkovatele je zprávu od odesílatele přebrat a doručit ji adresátovi. Předáni zprávy může probíhat ve 3 režimech:

- Je-li operace odeslání synchronní a systém doručení spolehlivý, za zprávu je již dále odpovědný zprostředkovatel – klient může věc považovat za vyřízenou.
- Je-li operace asynchronní, klient musí paměť se zprávou zachovat až do doby, než zprostředkovatel potvrdí, že ji převzal.
- Je-li doručení zpráv nespolehlivé, a klient potřebuje zajištěné doručení, musí zprávu zachovat až do doby, než mu ji příjemce potvrdí (opět odesláním zprávy).

Příjem zprávy obvykle funguje v jednom ze dvou režimů:

1. Synchronní, kdy se příjemce dotáže na další zprávu (případně na ni vyčká, není-li právě žádná k dispozici), tuto zpracuje a poté se zeptá na další.
2. Asynchronní – klient zpracovává větší počet zpráv souběžně, a zpracováni příchozí zprávy začne na výzvu zprostředkovatele. Lze realizovat buď pomocí klasických vláken, nebo tzv. fibrů.[126]

Zprostředkovatel může také poskytovat složitější (odvozené) operace (příklady uvedeme později).

- může být žádný, jeden nebo několik
- realizuje předávání zpráv
- komunikace s klientem na nižší úrovni
- lze i složitější operace

[126] Fibry jsou forma „uživatelských“ vláken, kdy je samotný program vnitřně organizovaný jako souběžný systém s kooperativním plánovačem. Fibrů lze vytvořit mnohem více, než skutečných vláken na úrovni operačního systému, a jejich přepínání je často mnohem levnější.

- nejjednodušší → broadcast
- scatter/gather, all-to-all
- synchronizace: bariéra
- remote procedure call
- subscribe / publish

- mikrojádrové operační systémy
- obecněji izolace komponent
- distribuované výpočty (MPI)
- jiné distribuované systémy

7.2.4 Odvozené operace Tyto operace lze vždy v principu realizovat pomocí jednotlivých point-to-point zpráv. Umožňuje-li ale centrální zprostředkovatel přímo vyžádat složitější operace, jsou často mnohem efektivnější.

…

7.2.5 Využití Existuje typ operačních systémů, kde je předávání zpráv základním mechanismem meziprocesové komunikace. Tento návrh silně koreluje s mikrojádrovou architekturou. V takovém systému jsou služby operačního systému z velké části realizovány „běžnými" procesy, a proto je komunikace mezi procesy v takových systémech velmi důležitá. Zprostředkovatelem je v takových systémech často samotné mikrojádro.

Zároveň je v těchto systémech často kladen důraz na spolehlivost a bezpečnost, je tedy důležité mít základní IPC mechanismus co nejrobustnější. Předávání zpráv je v tomto kontextu vhodné – takto komunikující entity jsou jen velmi slabě provázané a většina rizik spojená se souběžností se redukuje. Celý systém pracuje jako systém producentů a konzumentů zpráv, kde synchronizaci zabezpečuje operační systém centrálně.

Samozřejmě techniku předávání zpráv (ať už s centrálním zprostředkovatelem, nebo bez něj) lze aplikovat v mnoha dalších kontextech. Často se k nějaké formě předávání zpráv přiklání systémy, které sestávají z většího počtu relativně nezávislých komponent, kde jejich vzájemná izolace je důležitá pro bezpečnost systému.

V **distribuovaných systémech** je předávání zpráv zcela standardní metodou komunikace – protože nemají k dispozici sdílenou paměť, většina ostatních možností vůbec nepřichází v úvahu.[127] Významnou podmnožinu zde tvoří distribuované vědecké výpočty, zejména různé simulace (superpočítače mají často architekturu postavenou na velmi efektivním předávání zpráv).

Samozřejmě lze také uvažovat řadu dalších distribuovaných systémů: distribuované databáze (ať už z návrhu distribuované nosql systémy, nebo tradiční replikace v kontextu relačních systémů), „obláčkové" aplikace, které mají často stovky nebo tisíce aplikačních serverů, atp.

7.3: Zdroje

- hardwaru je konečné množství
- virtualizace není všelék
  - některé periferie nelze virtualizovat
  - neumí „množit" zdroj, pouze sdílet
- alternativa: rezervace a uvolnění zdroje

7.3.1 Hardwarové zdroje Nejpřirozenějším typem zdroje je hardware: jak výpočetní (procesor, paměť) tak periferie. V první části kurzu jsme se zabývali virtualizací, která nám umožňuje předstírat, že zdroje vlastníme ve více instancích, než jich fyzicky existuje. To je sice velmi užitečná a úspěšná taktika, ale není bez limitů.

Virtualizaci zdroje lze uplatnit jen v situacích, kdy každý uživatel daného zdroje využívá jen část jeho skutečné kapacity – část obrazovky (okno), část procesorového jádra (vlákno), část operační paměti (proces), část šířky přenosového pásma (síťové spojení) atp.

Virtualizace také přináší situace, kdy jakákoliv operace s daným zdrojem může selhat, protože byla vyčerpána jeho fyzická kapacita – týká se zejména operační paměti, ale také například volného místa v souborovém systému.

- alternativa k virtualizaci
- může být vynuceno povahou zdroje
- nebo situací (zdroj je klíčový)
- uvolnění → vrácení do oběhu

7.3.2 Rezervace Alternativou je **rezervace**, kterou využijeme v situacích, kdy virtualizaci použít nelze:

- neumožňuje to povaha zdroje (např. pásková jednotka nebo optická zapisovací mechanika) nebo
- nečekané selhání zdroje je nepřípustné (např. by došlo k ohrožení zdraví, života nebo majetku).

Rezervace a virtualizace **není** vzájemně výlučná – daný zdroj může být virtualizován, ale zároveň může systém poskytovat možnost rezervovat zaručenou kapacitu – v takovém případě je rezervovaný zdroj (dočasně) nepřístupný pro systém virtualizace. Operační systémy například běžně poskytují možnost rezervovat pro daný proces nějaké množství fyzické paměti. Takto rezervovanou fyzickou paměť pak systém nebude používat pro stránky jiných procesů, ani nebude stránky vlastníka z takto rezervované paměti přesouvat do trvalého úložiště.

Rezervace sestává ze dvou operací:

1. samotná rezervace, která od systému vyžádá nějaký zdroj (resp. nějaký počet jednotek zdroje), pro **výlučné** použití daným programem (procesem, vláknem, atp.), – po úspěšném provedení rezervace je zdroj ve **vlastnictví** příslušného programu,
2. **uvolnění zdroje** kdy vlastník zdroje oznámí systému, že již zdroj nebude využívat, a tento může být vrácen „do oběhu" (může být předán jinému programu, procesu, vláknu, …).

[127] Distribuovaná sdílená paměť je předmětem výzkumu, pro většinu aplikací má ale zásadní problémy s výkonem. Tyto problémy jsou často největší při použití ve spojení s běžnými vzory komunikace skrze sdílenou paměť.

Důležitou vlastností rezervace je, že v době požadavku na zdroj nemusí být tento zdroj dostupný. S tím se lze vypořádat dvojím způsobem:

1. ten méně zajímavý je **selhání** – požadavek je zamítnut a dotčený program se musí s nedostupností zdroje vypořádat,
2. zajímavější řešení je **čekání** – nelze-li požadavek splnit, vyčkáme do doby, než nastane situace, kdy jej splnit lze (například proto, že současný vlastník zdroje jej uvolnil).

7.3.3 **Abstraktní zdroje** Rezervaci lze chápat nejen jako mechanismus pro správu zdrojů, ale i jako **abstrakci**. Jakoukoliv entitu, kterou lze rezervovat a pak vrátit, lze tedy považovat za **abstraktní zdroj**.

- cokoliv co lze rezervovat
- příklad: mutex
- zamčení = rezervace
- společný jazyk pro teorii uváznutí

Pro příklad nemusíme chodit daleko: uvažme kritickou sekci chráněnou mutexem. Roli **abstraktního zdroje** zde bude hrát mutex[128] – neposkytuje sice prostředky (výpočetní ani žádné jiné), přesto je ale možné jej:

1. **rezervovat** – odpovídá **zamčení** mutexu, resp. vstupu do kritické sekce, zatímco situace, kdy zdroj-mutex „vlastní“ jiné vlákno, odpovídá čekání na tento zdroj,
2. **uvolnit** – odpovídá **odemčení** mutexu, resp. opuštění kritické sekce.

Další synchronizační zařízení lze chápat podobně (např. semafor je abstraktní zdroj, který existuje v několika instancích a tyto instance lze přidělovat nezávisle).

7.3.4 **Instance** Zdroje (ať už abstraktní nebo hardwarové) mohou existovat v několika nezávislých ale záměnných instancích. Záměnnost se projevuje **v době rezervace** – již rezervovanou instanci nelze vyměnit za jinou.

- musí být volně záměnné
- rezervace zabere libovolnou
- dodatečně nelze vyměnit

7.3.5 **Komunikace** Protože komunikace je zároveň synchronizací, jedná se (možná ne zcela intuitivně) také o abstraktní zdroj. Novým aspektem je zde skutečnost, že **rezervaci** a **uvolnění** neprovádí nutně tentýž aktér – předání zprávy (resp. obecně informace) může zároveň předat vlastnictví tohoto typu „zdroje“.

- komunikace → synchronizace → zdroj
- rezervace – odeslání zprávy
- uvolnění – přijetí zprávy
- provádí různá vlákna (procesy)

Tento jev je vázán k problému producentů a konzumentů (který, jak jsme zmiňovali výše, s komunikací úzce souvisí): vytvoření a (zejména) uložení mezivýsledku je formou rezervace – počet mezivýsledků, které jsme si ochotni pamatovat, je omezené, a jedná se tedy o konečný zdroj, který musí být producentovi přidělen. Takto zablokovaná instance se ale uvolní tím, že **konzument** mezivýsledek použije.

Podobný princip lze aplikovat aj na jiné typy zdrojů: lze si představit situace, kdy program předá nějaký zdroj jinému programu, který ho pak později uvolní (samozřejmě to vyžaduje součinnost operačního systému).

7.3.6 **Odnímatelné zdroje** Implicitně považujeme zdroje za neodnímatelné, ve smyslu, že jediný způsob, jak může dojít k uvolnění zdroje, je dobrovolné vrácení vlastníkem. U řady zdrojů by mělo odebrání pravděpodobně fatální následky minimálně pro dotčený program, a někdy také pro dotčený zdroj (3D tiskárna, optická zapisovací mechanika, fotografická tiskárna, atp.). Uvážíme-li abstraktní zdroje, násilné odebrání např. zmiňovaného mutex-u jistě nemůže mít pozitivní dopad na další fungování vlákna, resp. programu jako celku.

- uvolnění zdroje pouze akcí vlastníka
- některé zdroje lze odebrat násilně
- odnímatelnost může záviset na kontextu
- příklad: zahození paketu

Jiná je situace u zdrojů virtualizovaných, kde odebrání fyzického prostředku obvykle nepředstavuje zásadní problém – plánovač vláken zcela běžně odebírá procesor běžícímu vláknu bez jeho souhlasu, správce virtuální paměti může stránku přestěhovat z operační paměti do externí, atp. Na dotčený proces (vlákno) to má jistě dopad, ale důsledky obvykle nejsou zdaleka tak závažné, jako v případě zdrojů neodnímatelných.

Ne vždy musí být odnímatelnost vlastností zdroje, ale může záviset i na kontextu: stejné zařízení může být v nějaké situaci odnímatelné (třeba díky virtualizaci), ale v jiné nikoliv (zaručený zdroj).

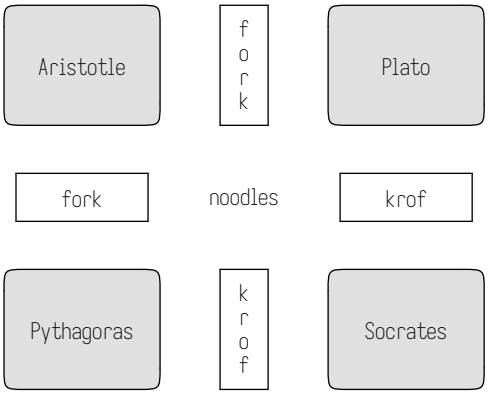
Příklad: V počítačové síti jsou obvykle prvky typu „ulož a přepošli“ (store and forward), které musí příchozí paket nejprve uložit do vyrovnávací paměti a později (i když jen o zlomek vteřiny) jej přeposlat na jiném rozhraní.

[128] Ještě o něco abstraktněji bychom za zdroj mohli považovat i samotnou kritickou sekci – narážíme tady ale na problémy s definicemi. V literatuře se často s pojmem kritická sekce operuje dost volně – obvykle ve smyslu třídy ekvivalence, kterou získáme jako RST (reflexivní, symetrický a tranzitivní) uzávěr vztahu (relace) „kritický vůči“. Takovou třídu ekvivalence pak lze chápat jako abstraktní zdroj.

Paměť je samozřejmě omezená a přichází-li pakety rychleji, než je lze odesílat, může se zaplnit. Jednotlivé buňky pro jednotlivé pakety představují instance zdroje – není-li možné další rezervovat, některý paket je ztracen. Příjem paketu nemůže vyčkat na blokující rezervaci – abstraktně tedy situaci odpovídá odebrání zdroje.

7.4: Hladovění

K ilustraci hladovění a uváznutí (v kontextu souběžných systémů) se často používá problém tzv. „večeřících filozofů“:



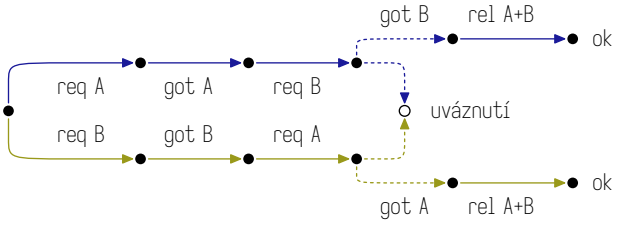
Filozofové jsou usazeni kolem stolu, uprostřed kterého je umístěna mísa s jídlem. Mezi každou dvojicí filozofů je jedna vidlička. Filozofové sedí a uvažují, a když některý dostane hlad, vezme si vidličku z každé strany a začne jíst (aby se mohl najíst, potřebuje nutně dvě vidličky).
Zkuste, jestli dokážete vymyslet instrukce takové, že když je dáte každému filozofovi zvlášť takové, žádný z nich neumře hlady (tzn. bude mu umožněno se najíst v konečném čase). Zvažte, proč jednoduché algoritmy nefungují.

- uvažme dva zdroje, A & B
- a dvě vlákna P & Q
- P rezervuje A, Q rezervuje B
- P požádá o B, ale musí vyčkat na Q
- Q požádá o A, ale musí vyčkat na P

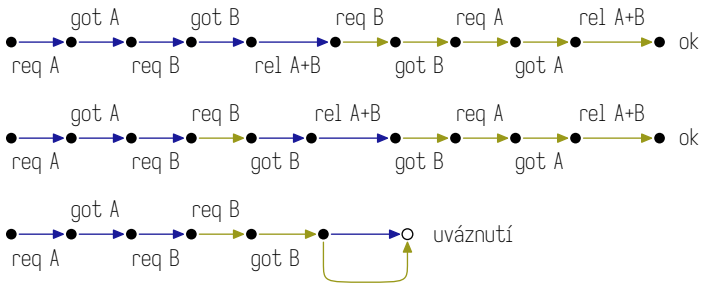
7.4.1 Uváznutí Nyní se můžeme konečně blíže podívat na problém uváznutí. Uvažme situaci, kdy máme 2 vlákna P a Q, a dva zdroje, A a B. Nejjednodušší možné uváznutí může nastat například takto:

čas	P má	akce P	Q má	akce Q
1	ø	žádá o A	ø	
2	A	–	ø	žádá o B
3	A	žádá o B	B	–
4	A	čeká na B	B	žádá o A
5	A	čeká na B	B	čeká na A
…	A	čeká na B	B	čeká na A
666	A	čeká na B	B	čeká na A
…	A	čeká na B	B	čeká na A

Bez vnějšího zásahu nemůže dvojice P, Q učinit žádný pokrok – obě vlákna budou navěky zablokovaná čekáním na příslušný zdroj. Je obvyklé, ale nikoliv nutné, že obě žádosti vlákna P jsou souběžné s oběma žádostmi vlákna Q. V takovém případě se navíc jedná o hazard souběhu, a uváznutí je chybovým chováním, které nastane jen v některých případech.
Situaci lze ilustrovat například takto (req značí požadavek, got značí přidělení zdroje, rel značí uvolnění zdroje):



Některé kompatibilní časové sledy:



Samozřejmě tato situace může být zobecněna na libovolný počet vláken a zdrojů (dokud je každého alespoň 2).

7.4.2 Podmínky uváznutí Uváznutí může nastat, jsou-li splněny všechny 4 následující podmínky – přitom požadovat umožnění každé z nich je samo o sobě přirozené a smysluplné:

1. vzájemné vyloučení
2. čekající vlastník
3. neodnímatelnost
4. kruhové čekání

1. **vzájemné vyloučení**[129] (angl. mutual exclusion) je přímým důsledkem **rezervace** – zdroj může být přisouzen v dané chvíli nejvýše jednomu vláknu a ostatní musí vyčkat,
2. **čekající vlastník** (angl. hold and wait) může nastat, je-li povoleno, aby vlákno rezervovalo zdroj, i když je mu již nějaký jiný zdroj přisouzen – zde je podobně těžké cokoliv namítat, protože takové chování odpovídá povaze většiny programů (nelze jednoduše předvídat, jaké přesně zdroje bude program v různých situacích vyžadovat a pesimisticky rezervovat všechny možná potřebné zdroje je značně neefektivní),
3. **neodnímatelnost** (angl. non-preemptability) značí, že přisouzený zdroj nelze vlastníkovi odebrat bez relativně závažných důsledků a často plyne z povahy zdroje[130], a konečně
4. **kruhové čekání** (angl. circular wait) lze popsat pomocí tzv. **statického** grafu závislostí zdrojů – uzly tohoto grafu jsou zdroje a hrana $R \to S$ existuje právě když existuje vlákno (program), který může vyžádat rezervaci S v době, kdy mu je přisouzen zdroj R, a podmínka kruhového čekání je splněna existuje-li v tomto grafu cyklus.

Tyto podmínky (ani společně) **nejsou postačující** – systém může splňovat všechny a zároveň být vůči uváznutí imunní. Plyne to mimo jiné z jejich statické povahy: protože popisuji chování pouze v hrubých rysech, může uváznutí bránit nějaká dynamická vlastnost, kterou tyto podmínky nedokáží popsat.

- uváznutí je těžké odladit
- může být velmi vzácné
- vzácnost lze ale využít
- uváznutí? restartujte systém

7.4.3 Pštrosí algoritmus Mnoho (pravděpodobně naprostá většina) případů uváznutí je důsledkem nějakého hazardu souběhu a mohou být proto velmi vzácné. Pštrosí algoritmus toho využívá – nastane-li uváznutí, násilně ukončíme všechny dotčené procesy, nebo dokonce restartujeme celý systém. Rozhodnout, nastalo-li uváznutí, ovšem může být problematické, vždy ale existuje možnost toto rozhodnutí delegovat na uživatele (systém nereaguje → zmáčkněte reset).

- uváznutí lze rozeznat
- obvykle kontrolou kruhového čekání
- udržujeme graf vlastnictví vs čekání
- cyklus → systém uvázl

7.4.4 Detekce uváznutí Jednou možností jak uváznutí detekovat je použít **dynamickou** variantu grafu závislostí zdrojů, který jsme použili k definici kruhového čekání. V tomto případě obsahuje graf dva různé typy uzlů: vlákna, nebo jiné potenciální **vlastníky** a **zdroje**. Hrana vede vždy mezi uzly různých typů, řekněme vláknem T a zdrojem R:

- hrana $R \to T$ existuje kdykoliv je vlákno T současným vlastníkem zdroje R,
- hrana $T \to R$ existuje kdykoliv vlákno T vyžádalo rezervaci zdroje R, ale této dosud nebylo vyhověno – jinými slovy, vlákno T čeká na zdroj R.

Existuje-li cyklus[131] v tomto grafu, systém uvázl (konkrétně uvázla všechna vlákna, která na takovém cyklu leží).

Příklad: Vraťme se k dřívějšímu příkladu s vlákny P a Q a zdroji A a B:

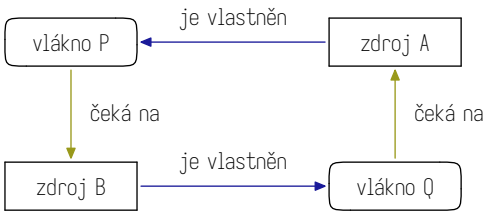
1. P úspěšně **rezervuje** A, tedy existuje hrana $A \to P$,
2. podobně Q **rezervuje** B, tedy existuje $B \to Q$,
3. P se pokusí rezervovat B ale musí **čekat**, tedy $P \to B$,
4. Q se pokusí rezervovat A ale opět **čeká**, tedy $Q \to A$.

Výsledný graf vypadá takto (obsahuje celkem zjevný cyklus):

[129] V obecném významu, tzn. nemusí být nutně způsobeno použitím synchronizačního zařízení stejného jména.

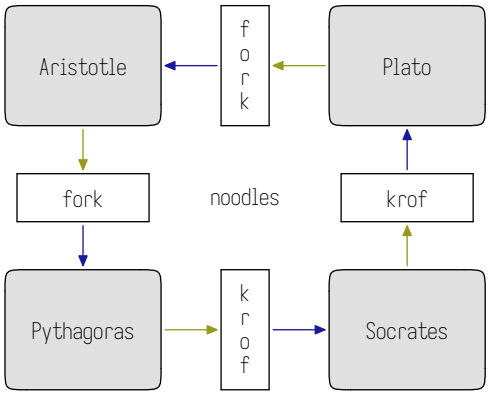
[130] Je-li dotčeným zdrojem například zařízení vzájemného vyloučení (mutex), existuje zde určitý manévrovací prostor. Umožní-li to povaha chráněná kritická sekce, lze její pozorovatelné efekty vrátit, podobně jako při zrušení databázové transakce (angl. rollback), mutex odemknout a pokusit se ji provést od začátku. Implementovat takový mechanismus ale není vůbec jednoduché.

[131] Rozmyslete si, že každý cyklus v takovém grafu musí být sudé délky a obsahovat alespoň 4 hrany.



Tento přístup funguje pouze pro zdroje, které existují v jediné instanci. Nejdůležitějším zdrojem z pohledu uváznutí je mutexem chráněná kritická sekce, která samozřejmě nemůže mít více než jednu „instanci“ (pozor, různé mutexy nejsou záměnnými instancemi téhož zdroje).

Příklad: V problému večeřících filozofů existuje n vidliček, nicméně tyto **nejsou** záměnné, protože filozof potřebuje **konkrétní** dvě vidličky, aby mohl jíst. Algoritmus založený na detekci cyklu je tedy v tomto problému bez problémů aplikovatelný:



Pro situace, kdy je nějaký zdroj dostupný ve více instancích, existuje složitější algoritmus, který se ale v praxi příliš často nepoužívá, a kterým se ani nebudeme blíže zabývat.[132]

- odeber z cyklu odnímatelný zdroj
- nelze-li, jiná možnost je rollback
  - např. databázové systémy
- jinak násilně ukončit proces na cyklu
  - může být nutné reinicializovat zdroje

7.4.5 Zotavení z uváznutí Účastní-li se cyklického čekání alespoň jeden odnímatelný zdroj, může k uváznutí[133] sice dojít, ale na rozdíl od standardní situace se lze z tohoto typu uváznutí relativně lehce zotavit (zejména se lze zotavit bez násilného ukončování vláken). Dočasným odebráním odnímatelného zdroje[134] se totiž cyklus čekajících vláken přeruší, a systém může pokračovat.
V opačném případě – všechny zdroje na cyklu jsou neodnímatelné – nám zbývá ještě jedna možnost jak se ze situace zotavit, aniž bychom museli restartovat systém jako celek – vybrat na cyklu jedno vlákno (nejlépe to, které považujeme za nejméně důležité) a násilně jej ukončit (obvykle včetně celého procesu, kterému toto vlákno náleželo). Tím se uvolní některý dotčený zdroj a zbytek systému může pokračovat.

- můžeme zamítnout rezervace
- musíme znát maximální požadavky
- vyhýbání využívá bezpečné stavy
- počítáme s nejhorším případem

7.4.6 Vyhýbání uváznutí Jinou možností jak se s uváznutím vypořádat je technika **vyhýbání** (angl. avoidance). Klíčovým prvkem je zamítnout některé požadavky na rezervaci, i když je příslušný zdroj dostupný, může-li taková rezervace vést k pozdějšímu uváznutí. Stavy, ze kterých může vždy alespoň jedno vlákno vyváznout a úspěšně skončit (resp. uvolnit zdroje) označíme jako **bezpečné**.
Asi nejznámějším algoritmem pro vyhýbání uváznutí je tzv. bankéřův algoritmus, navržený E. Dijkstrou. Tento algoritmus je určen pro situace, kdy má každý zdroj více (záměnných) instancí než kolik jich může jednotlivé vlákno vyžádat.[135] Vstupem pro algoritmus je maximální počet instancí všech zdrojů, které může každé vlákno v systému vyžádat. Invariant, který algoritmus udržuje, je tento:

1. existuje vlákno T takové, že volné zdroje jsou dostatečné pro vyhovění jeho maximálním požadavkům,

[132] Popis tohoto algoritmu najdete například v knize Modern Operating Systems, sekce 6.4.2.
[133] Nastává zde drobný terminologický konflikt – striktně vzato uvádíme neodnímatelnost jako nutnou podmínku uváznutí, a tedy situace, kdy se cyklického čekání účastní odnímatelný zdroj není v tomto smyslu uváznutím, na straně druhé, nebudeme-li takové situace detekovat a řešit, systém zůstane zablokovaný.
[134] Zrušení a restart transakční kritické sekce chápeme jako speciální případ odebrání odnímatelného zdroje.
[135] Striktně vzato je podmínkou použití „zdroj má alespoň tolik instancí jako maximum, které si může nějaké vlákno vyžádat“ – v takové situaci se ale bankéřův algoritmus redukuje na přístup „zamezení uváznutí předrezervací“.

2. po ukončení T a navrácení všech jeho zdrojů tento invariant nadále platí díky nějakému dalšímu vláknu, atd.

Jakákoliv rezervace, která by tento invariant porušila, je zamítnuta. Vstupní podmínka zaručuje, že systém v bezpečném stavu začne a protože přechod z bezpečného do nebezpečného stavu nepřipustíme, uváznutí nemůže nastat (algoritmus je tedy korektní).[136]

Příklad: Následující tabulka ilustruje bankéřův algoritmus s jediným typem zdroje, který existuje v 5 instancích – P, Q a R jsou vlákna, čísla v jejich sloupcích označují „počet alokovaných / maximální počet“ instancí, sloupec „dostupné“ označuje počet instancí, které má systém k dispozici:

krok	P	Q	R	dostupné	akce
1	0/3	0/2	0/4	5/5	P rezervuje 1 inst.
2	1/3	0/2	0/4	4/5	R rezervuje 2 inst.
3	1/3	0/2	2/4	2/5	Q rezervuje 1
4	1/3	1/2	2/4	1/5	P je zamítnuta 1
5	1/3	1/2	2/4	1/5	Q rezervuje 1
6	1/3	2/2	2/4	0/5	Q je ukončeno
7	1/3	–	2/4	2/5	P rezervuje 1
8	2/3	–	2/4	1/5	

Všimněte si, že v kroku 4 je požadavek vlákna P zamítnut, protože jinak by už žádné vlákno nemohlo zaručeně získat maximální zdroje a tedy pokračovat dost dlouho na to, aby zdroje opět uvolnilo (tzn. byl by porušen invariant bezpečnosti).

7.4.7 Zamezení uváznutí Vyhýbání, tak jak jsme jej popsali v předchozí sekci, žel ve většině situací není prakticky použitelné – mnoho zdrojů má právě jednu instanci a bankéřův algoritmus pak způsobí, že zdroje smí v jednu chvíli vlastnit pouze jedno vlákno a všechna ostatní (resp. ta, která mají neprázdný průnik používaných zdrojů) musí čekat, než je všechny uvolní.

- vyhýbání je obvykle nepraktické
- uváznutí má 4 nutné podmínky
- zaútočit lze na každou z nich
- odstraněním libovolné zamezíme uváznutí

Posledním mechanismem, který lze pro vypořádání se s problémem uváznutí použít, je tzv. **zamezení** (angl. prevention). Vzpomeňme si, že existují 4 **nutné** podmínky uváznutí – dokážeme-li zamezit kterékoliv z nich, můžeme zamezit i uváznutí:

- **čekající vlastník** – tuto podmínku lze vyřadit **předrezervací**,
- **vzájemné vyloučení** – neguje již dobře známá **virtualizace**,
- **kruhové čekání** – lze vyřadit globálním **uspořádáním** zdrojů.

Jsou-li zdroje **odnímatelné**, můžeme využít již popsanou techniku **zotavení**.

7.4.8 Zamezení předrezervací Myšlenka předrezervace je jednoduchá – nesmíme připustit, aby vlákno, které nějaký zdroj již vlastní, čekalo na nějaký zdroj (je lehce vidět, že systém pak nemůže uváznout). Rezervace proto připustíme pouze v situaci, kdy žádající vlákno žádné zdroje nevlastní – aby byl systém prakticky použitelný, musíme ovšem povolit rezervaci několika zdrojů najednou (dávkově).

- pokusíme se zamezit čekání vlastníka
- dávková rezervace (vše najednou)
- často nepraktické
- alternativa: uvolnit a vyžádat znovu

Potřebuje-li tedy vlákno využít více zdrojů, musí je vyžádat všechny jedinou atomickou akcí (uvolňovat je ovšem může postupně). Je-li potřeba rezervovat nějaký zdroj v situaci, kdy už dané vlákno nějaký jiný zdroj vlastní, musí nejprve všechny zdroje uvolnit a opět je rezervovat společně s tím novým.

7.4.9 Zamezení virtualizací Virtualizací jsme se podrobně zabývali v prvních 4 kapitolách – zdroje, které jsou virtualizované, vůbec nepoužívají systém výlučné rezervace, nesplňují tedy podmínku vzájemného vyloučení a nemohou přímo způsobit uváznutí.[137] Mezi zdroje, které lze virtualizovat, patří samozřejmě paměť, procesor a pevné úložiště (kapitoly 1–3) a některé periferie – terminál (obrazovka, tiskárna, atp.) nebo síťová rozhraní (kapitola 4).

- útočíme na vzájemné vyloučení
- umožníme více programům využívat zdroj
- např. fronta: sbíráme požadavky
- odesíláme je zařízení po jednom

Příklad: Virtualizace tiskáren je obvykle založená na frontě úloh – aplikaci je přístupná virtuální tiskárna, která „tiskne“ do souboru, přitom takto získané jednotlivé soubory se posílají tiskárně jeden po druhém.

Tento přístup je teoreticky náchylný na vyčerpání místa na pevném úložišti, ale v tomto kontextu se jedná o relativně jednoduše odnímatelný zdroj (zejména ve srovnání s tiskárnou samotnou

[136] Pozor, algoritmus předpokládá, že jediný možný důvod pro zablokování vlákna je čekání na rezervaci jednoho ze sledovaných zdrojů. Může-li se nějaké vlákno, které vlastní zdroje, zaseknout z jiného důvodu, k uváznutí dojít může.

[137] Používá-li ale virtualizace nějakého zdroje jiný zdroj (např. virtualizace paměti využívá skrze externí stránkování pevné úložiště), může se stále stát, že dojde tento jiný zdroj, a systém nebude schopen požadavkům na takto virtualizovaný zdroj dostát.

– zrušíme-li tisk zablokovaný pro nedostatek místa, nemusíme se zabývat částečně vytištěnou úlohou). Soubor s nedokončenou úlohou odstraníme a později (když se uvolní místo) se o tisk pokusíme znovu.

- útočíme na kruhové čekání
- zavedeme globální uspořádání zdrojů
- povolíme rezervaci pouze v tomto pořadí
- jinak nutno zdroje nejprve uvolnit

7.4.10 **Zamezení uspořádáním** Zbývá nám podmínka kruhového čekání – tomu lze zamezit například tím, že se na zdrojích ustaví globální lineární uspořádání, které musí každé vlákno při rezervacích dodržet: rezervovat lze pouze zdroj, který je v uspořádání větší než dosud největší zdroj vláknu přisouzený. V takovém systému je statický graf závislostí zdrojů acyklický, k uváznutí tak nemůže dojít.

Podobně jako v případě předrezervace lze řešit i situace, kdy potřebujeme globální pořadí porušit – vlákno se musí nejprve vzdát některých zdrojů a pak je znovu rezervovat ve správném pořadí. Tento přístup je poměrně praktický na úrovni jednotlivého programu, nebo jiného uzavřeného systému,[138] nicméně pro operační systém jako celek se nehodí.

Část 8: Přerušení a periferie

Periferie mohou reagovat na vnější události (ve fyzickém světě, např. stisk klávesy nebo příjem rámce na ethernetovém rozhraní, atp.), případně signalizovat ukončení nějaké úlohy, kterou jim operační systém zadal (přenos dat z/do paměti).

Doporučené čtení: A. Tanenbaum, H. Bos – Modern Operating Systems (4th Ed.): § 5.1.5 Interrupts Revisited ◇ § 5.3.1 Interrupt Handlers ◇ § 5.5 Clocks.

8.1: Přerušení

Operační systém potřebuje být o podobných událostech informován – mechanismus, který se k tomu používá se obecně nazývá **přerušení**.

- přerušení = synchronizační zařízení
- periferie vs operační systém
- podobá se na podmínkovou proměnnou
- asymetrie → signalizuje vždy periferie

8.1.1 **Synchronizace** Přerušení je mechanismus, který umožňuje **synchronizovat** periferii a software – v abstraktní rovině lze tedy o přerušení uvažovat jako o synchronizačním zařízení. Od těch, se kterými jsme se dosud setkali se liší tím, že synchronizace neprobíhá mezi dvěma softwarovými entitami (vlákny), ale mezi hardwarem (periferií) a softwarem (operačním systémem). S tím souvisí druhý důležitý rozdíl – přerušení je **asymetrické** v tom smyslu, že ho může vyvolat pouze periferie, nikoliv operační systém nebo software obecně.[139] Máme tedy zařízení, které je vždy aktivováno periferií – operační systém je vždy pasivním účastníkem. Až na tyto rozdíly se přerušení podobá na **podmínkovou proměnnou** – slouží k signalizaci nějaké události.

- přerušení je hardwarový mechanizmus
- operační systém běží na CPU
- přerušení musí realizovat CPU
- preemptivní → pozastaví aktuální program

8.1.2 **Procesor** Operační systém je samozřejmě program, který musí být vykonáván procesorem – proto krom periferie a operačního systému musí do hry vstoupit i procesor.

Zejména se může stát, že ve chvíli, kdy přerušení nastane (je periferií vyvoláno), procesor nějaký program právě vykonává. Navíc je často důležité, aby byla reakce na přerušení dostatečně rychlá – velká prodleva obsluhy může mít řadu důsledků, od uživatelského nepohodlí až ke ztrátě dat.

Aby se prodleva minimalizovala, přerušení je realizováno **preemptivně** – cokoliv[140] procesor v danou chvíli prováděl je pozastaveno, aktuální stav vlákna (hodnoty registrů) je uložen do paměti a je spuštěna obsluha přerušení.

- přerušení existuje v instancích
- počet omezen hardwarem
- identifikována číslem
- každá instance má vlastní obsluhu

8.1.3 **Instance** Obsluha přerušení je realizována **podprogramem**, kterého adresa je uvedena ve speciální tabulce obsluhy přerušení. Chápeme-li přerušení jako synchronizační zařízení (podobné podmínkové proměnné), je přirozené, že může existovat ve vícero instancích. Protože je ale realizováno hardwarově na relativně nízké úrovni, těchto instancí je pevný počet (často do 256). Tabulka obsluhy přerušení má pak pro každou instanci jednu položku, která určí který podprogram je daným přerušením aktivován. Různé periferie pak typicky používají různé instance přerušení (instance jsou identifikovány číslem).[141]

[138] Často se používá např. pro zámky v monolitických jádrech, resp. nějakou jejich podmnožinu. Tento systém je prakticky užitečný i v situacích, kdy nepokrývá všechny zdroje.

[139] Neuvažujeme zde tzv. softwarové přerušení, které vůbec není synchronizačním zařízením – je to efektivně forma volání podprogramu. Asymetrie přerušení spočívá v tom, že přerušení nemůže směrovat od procesoru k periferii.

[140] Přerušení lze dočasně zablokovat, tzn. „cokoliv“ není zcela přesné. Více si o blokování přerušení povíme za chvíli.

[141] Podle počtu periferií a dostupnosti přerušení může být nutné, aby různé periferie využívaly společnou instanci. Operační systém pak musí zpětně zjistit, která periferie přerušení vyvolala zpětným dotazem. Detaily takové situace jdou ale mimo záběr tohoto předmětu – budeme předpokládat, že každá periferie má alokované vlastní přerušení.

8.1.4 Stav procesoru Obslužný podprogram přerušení je (až na speciální prolog a epilog) stejný jako libovolný jiný – může být třeba zapsaný v jazyce C, a zejména může volat další podprogramy. Proto musí mít k dispozici jak registry (zabezpečeno uložením stavu procesoru před jeho aktivací), tak zásobník. Protože na zásobníku, který byl ve chvíli kdy k přerušení došlo, nemusí být volné místo,[142] je obvyklé, že dojde také k přepnutí zásobníku.[143] Konečně procesor se přepne do režimu jádra (privilegovaného režimu) – obsluha přerušení má tedy stejný privilegovaný přístup k výpočetním zdrojům jako jakákoliv jiná součást jádra.[144]
Obsluha přerušení se nápadně podobá na aktivaci vlákna. Zároveň má ale vlastnosti aktivace podprogramu, protože na rozdíl od vlákna se při **ukončení** obsluhy stav procesoru neukládá, a v obsluze tedy po jejím ukončení nelze pokračovat. Můžeme tak obsluhu přerušení chápat jako vlákno s velmi krátkým životem.

- obsluha je ± normální podprogram
- stav při vstupu je uložen do RAM
- přepnutí na vyhrazený zásobník
- podobné krátkodobému vláknu

8.1.5 Souběžnost Přerušení (a jeho obsluha) má vzhledem k relaci předcházení speciální charakter – na jedné straně je souběžné prakticky s čímkoliv, co se v systému děje, na straně druhé nemůže být obslužný podprogram uspán plánovačem.[145]
To nicméně neznamená, že by byla obsluha automaticky synchronizována – ostatní procesorová jádra pokračují v běžném provozu. Zejména mohou vykonávat kód jádra, včetně obsluhy přerušení (třeba i stejným obslužným podprogramem). Navíc může periferie během obsluhy přerušení vyvolat libovolné další (i stejné) přerušení – periferie nemá informaci o stavu procesoru, která by jí umožnila přerušení synchronizovat.
Jakékoliv kritické sekce tedy musí být chráněny, navíc k tomu nelze použít žádné synchronizační zařízení, které interaguje s plánovačem. Obsluha přerušení musí řešit 3 synchronizační scénáře:

1. synchronizace se zbytkem systému, který běží paralelně na **jiných** procesorových jádrech (včetně případné obsluhy přerušení tam probíhající) – zde je v nějakém smyslu největší volnost, obvykle lze použít spinlocky, případně některé nezamykající komunikační zařízení,
2. synchronizace se zbytkem systému, který běžel na **stejném** procesorovém jádře a byl obsluhou přerušen – zde spinlock nepřichází v úvahu, protože přerušený podprogram nemůže být spuštěn před ukončením obsluhy,
3. synchronizace s obsluhou souběžného přerušení vyvolané na **stejném** procesorovém jádře – spinlock opět nepřichází v úvahu (ze stejného důvodu), pomůže zde ale speciální jednoúčelové synchronizační zařízení – zákaz přerušení.

Je-li v platnosti zákaz daného přerušení, obsluha případného příchozího přerušení je odložena až do chvíle, kdy je přerušení opět povoleno. Jedná se tak o formu vzájemného vyloučení specifickou pro obslužné podprogramy.

- přerušení je vysoce souběžné
- synchronizace mezi CPU
  - spinlock, komunikace bez zámků
- synchronizace na stejném CPU
  - spinlock nelze (!)
  - zákaz přerušení, komunikace bez zámků

8.1.6 Reentrance Přesto, že je zákaz přerušení primárně určen k vypořádání se scénářem č. 3, může být užitečný i pro synchronizaci v situaci č. 2: zákaz přerušení může provést jádro kdykoliv, nikoliv pouze v obslužném podprogramu. Taková synchronizace je ale nutně asymetrická – obslužný podprogram nemůže zákazem přerušení ovlivnit žádnou jinou součást jádra.
Je-li důvodem synchronizace mezi obsluhou přerušení a zbytkem jádra kritická sekce, musíme rozlišit 2 varianty:

1. obsluha přerušení obsahuje sekci kritickou vůči zbytku jádra – týká se pouze scénáře 1, protože obsluha přerušení nemůže být na stejném procesoru zbytkem jádra přerušena,
2. jiná část jádra je kritická vůči akcím, které se provádí v obsluze přerušení – týká se i scénáře 2, a hraje zde speciální roli.

- zákaz přerušení je asymetrický
  - pouze směrem k obsluze
  - obsluha je na jednom CPU atomická
- kritické sekce je nutné chránit
- zákaz přerušení nesmí trvat dlouho

8.2: Obsluha druhé úrovně

Abychom mohli popsat jak se řeší souběžné akce navázané na přerušení, které ale potřebují provést větší množství práce, nebo se běžnými mechanismy synchronizovat se zbytkem systému, musíme si nejprve přiblížit jak jádro operačního systému organizuje svoji vlastní práci.

8.2.1 Kontext Vzpomeňme si na definici vlákna: je to

- vlákno = výpočet jednoho programu
- bez návaznosti na adresní prostor
- podobně přerušení → bez přepnutí procesu
- paměť obsluhy mapovaná ve všech procesech

[142] Připomínáme, že „zásobník“ je oblast virtuální paměti, do které ukazuje speciální registr (ukazatel zásobníku). Tato oblast má v libovolné chvíli nějakou pevnou velikost, a může se tedy stát, že mezi aktuálním ukazatelem zásobníku a koncem oblasti platných virtuálních adres není dostatek místa (použitelných adres).
[143] Obsluha přerušení může mít svůj vlastní speciální zásobník (opět tím myslíme blok virtuálních adres), nebo může využívat tzv. zásobník jádra – více v 11. kapitole.
[144] To, že je obsluha přerušení součástí jádra, plyne de facto z toho, že ji procesor automaticky spouští v privilegovaném režimu.
[145] Ne, že by to technicky nešlo realizovat, ale jednalo by se o značnou komplikaci návrhu. Musí-li obsluha přerušení počkat na nějakou událost nebo se jinak synchronizovat se zbytkem systému, používá se k tomu tzv. obsluha druhé úrovně.

- **výpočet** (posloupnost změn stavu), který vznikne
- nepřerušenou činností jednoho **procesoru**, který je
- po celou dobu řízen jedním **programem**.

Platí také, že vlákno není navázáno na adresní prostor: jeden proces může obsahovat více než jedno vlákno. V předchozí sekci jsme zmínili, že aktivace obsluhy přerušení[146] se podobá na aktivaci vlákna. Také si vzpomeňme, že aktivace **procesu** je relativně drahá operace: musí se změnit mapování paměti.

Proto se při aktivaci obsluhy přerušení proces **nepřepíná**: běží v adresním prostoru, který byl zrovna aktivní. To mimo jiné znamená, že struktury, které obsluha využívá, musí být dostupné ve **všech** virtuálních adresních prostorech, a navíc musí být dostupné na **stejných adresách**.[147] Obsluha přerušení tak vytvoří pomyslné vlákno v procesu, který je právě aktivní.

- provizorní podmínky (proces)
- obsluha blokuje přerušené vlákno
- nelze běžně synchronizovat

8.2.2 Prodleva To je také jeden z důvodů, proč je nežádoucí obsluhu přerušení „protahovat" – běží v provizorních podmínkách, blokuje tím přerušené vlákno (takto přerušené vlákno obvykle nelze probudit na jiném procesoru – návrat z obsluhy přerušení by ho efektivně duplikoval) a nemůže se běžnými prostředky synchronizovat s ničím, co by mohlo mít svůj protějšek v takto přerušeném vlákně.[148]

1. zakázat přerušení
2. minimální nutná akce
3. zbytek naplánován na později

8.2.3 Struktura obsluhy Obvyklá strategie je tedy takováto:

1. při vstupu do obsluhy přerušení se zakážou další přerušení (může vykonat přímo procesor jako součást aktivace obsluhy) –
   - minimálně toho typu, které bylo právě aktivováno, mají-li přerušení priority tak také všechna přerušení nižších priorit, a v některých systémech úplně všechna přerušení,[149]
   - tím jsou ochráněny libovolné kritické sekce obsluhy přerušení vůči sobě samé, a také je tím omezen počet aktivačních záznamů na zásobníku (jinak by hrozilo, že příliš mnoho rychle příchozích přerušení zásobník vyčerpá),
2. obsluha vykoná minimální nutnou akci, která uvede systém do provozuschopného stavu:
   - např. vyprázdní mezipaměti, které by jinak přetekly, a provede další akce, které nelze odložit,
   - minimálně část plánovače je tohoto charakteru (je aktivován obsluhou přerušení časovače – více později),
   - libovolné datové struktury, které zde využívá, a které využívá i jiná část jádra, musí být buď použitelné zcela bez zamykání (např. některé komunikační zařízení probrané v předchozí kapitole, které nepoužívá zámky), nebo musí být **na straně zbytku jádra** chráněny zákazem přerušení,
3. naplánuje zbývající akce (obsluha druhé úrovně) na pozdější vykonání – tento krok vyžaduje komunikaci se zbytkem jádra (někde musí informaci o potřebné návazné akci převzít jiná část jádra, která je s obsluhou přerušení jinak souběžná).

8.3: Základní typy přerušení

Každý obslužný podprogram je v nějakém smyslu unikátní, ale zároveň lze pozorovat určité společné rysy, které souvisí s účelem daného přerušení.

- oznámení nějaké události
- např. vsync/vblank
- ukončení souběžné operace
- výzva k přečtení HW vyrovnávací paměti

8.3.1 Notifikace Nejzákladnější formou přerušení je notifikace – upozornění na nějakou obecnou událost. Tento typ přerušení je obvykle relativně nezávazný v tom smyslu, že zdržení nebo i úplný výpadek obsluhy systém ani zpracovávaná data nijak neohrozí. Může se například jednat o informaci, že se vyměnil obraz na displeji (tzv. vsync) a je tedy potřeba jej překreslit – obvykle se nic zásadního nestane, když se třeba vykreslení o jednu periodu opozdí (typicky 1/60 vteřiny).

O něco zásadnější může být informace, že periferie má ve své vlastní paměti data připravena k vyzvednutí operačním systémem. Předávání dat tímto mechanismem (kdy je v reakci na přerušení operační systém přečte z registru periferie) se ale obvykle používá jen pro periferie s malou šířkou pásma a tedy opět není obsluha příliš časově kritická.

---

[146] Není-li explicitně uvedeno jinak, obsluhou přerušení se vždy myslí obsluha první úrovně.

[147] To neznamená, že je tato část adresního prostoru běžnému uživatelskému programu přístupná: tyto adresy jsou přístupné výhradně v privilegovaném režimu procesoru.

[148] Přerušené vlákno nemusí být nutně součást uživatelského procesu: může to být vlákno jádra (běží mimo jakýkoliv proces), nebo pokud je uživatelské, může být uprostřed systémového volání – hlavní problém není synchronizace jádra s uživatelským kódem, ale jádra samého se sebou.

[149] S výjimkou tzv. nemaskovatelných přerušení, která zakázat není možné. Tato jsou vyhrazena pro obzvláště závažné události.

Příklad: Typické zařízení v této kategorii je UART (sériová linka) – typické přenosové rychlosti se pohybují v rozmezí 9600 až 115200 bitů/s, přitom vyrovnávací paměť má obvykle kapacitu 128 a více bitů, což se promítne při plném vytížení do frekvence přerušení mezi 75–900Hz.

- přenos dat souběžný se systémem
- synchronizace:
  - k periferii zápisem registru
  - k systému pomocí přerušení

8.3.2 DMA Zařízení s větší šířkou pásma obvykle pro přenos dat používají DMA, tzn. režim, kdy periferie přesouvá data do operační paměti souběžně s běžným provozem zbytku systému. Tyto přenosy je ale nutné synchronizovat s operačním systémem (resp. s ovladačem zařízení, který je součástí operačního systému).
Směrem k periferii je synchronizace realizována zápisem do registru, opačným směrem ale podobný mechanismus použít nelze – periferie místo toho signalizuje dokončení přenosu přerušením. Protože přenosové rychlosti jsou u tohoto typu periferií výrazně vyšší (řádově gigabity za vteřinu), je zde i větší riziko přetečení dostupné paměti – a to i přesto, že její velikost je mnohem větší než v předchozím případě.

Příklad: Při přenosové rychlosti 1Gb/s a frekvenci přerušení 1kHz (srovnatelné s předchozím příkladem) je potřeba při každém přerušení zpracovat 1Mb (125KiB) dat, což je cca 700 plných ethernetových rámců (ve srovnání s ±16 bajty při přenosu po sériové lince).
Konkrétněji například síťová rozhraní Intel E1000 podporují příjmové fronty o délce maximálně 256 rámců, tzn. při plném vytížení a standardní velikosti rámce 1500 bajtů je nutná obsluha přerušení s frekvencí ~2,8kHz (aby se předešlo ztrátě dat), což odpovídá maximální prodlevě ~350μs.

- každé CPU má vlastní časovač
- zejména preemptivní plánovač
- komunikace mezi procesory
- synchronizace na frontách

8.3.3 Časovač V nějakém smyslu nejsložitějším typem přerušení z pohledu obsluhy je přerušení časovače, které řídí pravidelné činnosti v operačním systému. Jeho složitost spočívá zejména v interakci s plánovačem vláken[150] – u většiny ostatních typů přerušení se s výhodou používá rozdělení na obsluhu první a druhé úrovně.[2]
Obsluha druhé úrovně je spouštěna plánovačem, který je ale přerušením časovače řízen a obsluha přerušení časovače tedy tento způsob rozdělení práce použít nemůže – zejména nemůže odložit činnosti související s výběrem vlákna, kterému máme odevzdat procesor, a samotným přepnutím kontextu.
Samotné přepnutí kontextu je při návratu z přerušení relativně přímočaré – návrat musí vždy obnovit kontext přerušeného vlákna, chceme-li přepnout do jiného, stačí obnovit uložený kontext cílového vlákna. Situace se komplikuje v situaci, kdy původní a nové vlákno patří jinému procesu – pak je nutné přepnout i stránkové tabulky.
Největší problém ale spočívá v manipulaci s frontami vláken – jak výběr vhodného vlákna ke spuštění, tak jeho odstranění z fronty a případné související aktivity jako úpravy dynamických priorit a podobně. Jak již bylo zmíněno, tuto práci není možné odložit do obsluhy druhé úrovně, musí proto proběhnout v obsluze první úrovně, se všemi problémy souběžnosti s tím spojenými. Manipulace s frontami (resp. datovými strukturami plánovače obecně) může zároveň probíhat na libovolných procesorových jádrech, a to uvnitř i vně obsluhy přerušení. Tyto operace tedy musí být:

1. chráněny proti zásahům jiných procesorových jader (obvykle spinlockem, alternativně by mohl být celý plánovač postaven na komunikačních zařízeních bez zámků; uspání zde nepřichází v úvahu),
2. chráněny proti přerušení (zákazem) a tedy musí být zároveň
3. efektivní (konstantní nebo nejvýše logaritmické v počtu vláken),
4. synchronizace v bodech 1 a 2 musí být velmi důsledně koordinovaná: plánovač nesmí zamknout spinlock, aniž by byla zároveň zakázána přerušení (jinak by mohlo dojít k uváznutí), ale zároveň nesmí na spinlock příliš dlouho čekat (protože musí být při čekání zakázaná přerušení, tzn. jedná se o časově kritickou operaci).

[150] Ve skutečnosti je obvykle plánování navázáno na obsluhu **každého** přerušení (samozřejmě výhradně obsluhu první úrovně), nebo ještě přesněji na každé přepnutí z režimu jádra do uživatelského režimu (tedy včetně návratu ze systémového volání nebo obsluhy výjimky procesoru). Tuto skutečnost ale můžeme považovat za implementační detail a o časovači uvažovat abstraktně jako o speciálním typu přerušení.

Část 9: Interacting with the World

9.1: Command Interpreters

Historically, shells play a dual role in most operating systems. Command-driven interaction is probably the easiest to implement, and was hence what computers and operating systems initially used. As soon as interactive terminals became available, that is (we will skip batch-mode systems today).

Any command interpreter has one interesting property though: you can make a transcript of a sequence of commands to achieve more complex tasks than any individual command can perform. This works without any involvement from the command interpreter itself: you can simply write them down on a piece of paper, and type them back later.

Now of course it would be much more convenient, if the computer could read the commands one by one from a file and execute them, as if you were typing them. This is the origin of shell scripts.

- programming language
- centered on OS interaction
- rudimentary control flow
- untyped, text-centered variables
- dubious error handling

9.1.1 Shell Of course, in your hardcopy or handwritten notes, you could include additional remarks and instructions, like only run this command if the previous one succeeded, or repeat this command 3 times, or repeat this command until such and such thing happens. Wouldn't it be wonderful, though, if you could include such annotations in the transcript that the computer reads and performs?

But of course, we have just invented control flow. And quite obviously this is exactly what shells came to implement. The other 'obvious' invention is placeholders in commands, to be replaced by appropriate values at execution time. For instance, you write down, in your paper notebook, a sequence of commands to update a list of users stored in a text file: you would presumably use a placeholder for the name of the file you are currently working with. And when you type the commands back, replace every occurrence of this placeholder with the real filename in question. But why, you have just invented variables!

Another thing that sort of carries over from these paper-based scripts into the executable sort is error handling... or rather the lack thereof. It so happens that you wouldn't bother instructing yourself that you should stop and investigate if one of the commands from your notebook fails unexpectedly.

- almost all shells have an interactive mode
- the user inputs a single statement on keyboard
- when confirmed, it is immediately executed
- this forms the basis of command-line interfaces

9.1.2 Interactive Shells Before we go on about control flow and variables, let us remind ourselves that most shells are interactive in nature. In this interactive mode, the user enters a single 'statement' (a single line) and confirms it, after which it is immediately executed. Most often this is a single command, but it can be a sequence, a loop or any other construct allowed by the language: there is no distinction in the kinds of syntax available in shell scripts and the interactive command line. This makes it possible to write short scripts (so-called 'one-liners') directly on the command line, to automate simple tasks, without bothering to write the program down. Learning to do it is well worth the investment, as it can save considerable time in day-to-day work.

- a shell script is an (executable) file
- in simplest form, it is a sequence of commands
  - each command goes on a separate line
  - executing a script ~ typing the commands
- but can use structured programming constructs

9.1.3 Shell Scripts In contrast to interactive command execution, a **shell script** is a file with a list of statements in it, executed sequentially. Of course, as discussed above, basic control flow is available to alter this sequential execution, if needed. Variables can be used to substitute in parts of commands that change from one invocation of the script to the next.

- very easy to write simple scripts
- first choice for simple automation
- often useful to save repetitive typing
- definitely not good for big programs

9.1.4 Shell Upsides So how does shell compare as a programming language? First of all, it can be very productive and very easy to use, especially in scenarios where you are not programming as such, but really just automating simple tasks that you would otherwise do manually, by typing in commands.

However, try to create anything bigger and the limitations become significant: larger programs cannot just drop dead whenever something fails, nor can they ignore errors left and right (the two basic strategies available in scripts). The lack of structured data, a type system, and general 'programming hygiene' makes larger scripts fragile and hard to maintain. The next logical step is then a dedicated 'scripting' language, like Perl or Python, which make a compromise between the naivety (and simplicity) of shell and the structure and rigour of heavyweight programming languages like C++ or Java.

9.1.5 Bourne Shell The Bourne shell was created in 1976 and essentially codified the dual nature of shells as both interactive and programmable. We still use its basic model (and syntax) today. There are many Bourne-compatible shells, many of them descended from the Korn shell (ksh, which we will discuss shortly).
You may have heard of bash: the name stands for Bourne Again Shell[152] and it is probably the most famous shell that there is (to the extent that some people believe it is the only shell).

- a specific language in the 'shell' family
- brings consistent programming support
  - available since 1976
- compatible shells are still widely used today
  - best known implementation is bash
  - /bin/sh is mandated by POSIX[151]

9.1.6 C Shell Historically, the second well-known UNIX shell was the C shell[153] – it made improvements in interactive use, many of which were adopted into other shells, among others:

- command history (ability to recall already executed commands),
- aliases (user-defined shortcuts for often-used commands),
- command and filename completion (via tcsh),
- interactive job control.

The tcsh branch is a variant of csh with additional features, maintained alongside the original csh since early 80's. It is still distributed with many operating systems (and is, for instance, the default root shell on FreeBSD).

- also known as csh, first released in 1978
- more C-like syntax than sh (Bourne Shell)
  - but not really very C-like at all
- improved interactive mode (over sh from '76)
- also still used today (mainly via tcsh)

9.1.7 Korn Shell In essence a fusion of sh (the Bourne shell) and csh/tcsh (mainly as a source of improved user interaction; the scripting syntax remained faithful to sh). The original was based on sh source code, with many features added. This is the shell that POSIX uses as a model for /bin/sh.

- also known as ksh, released in 1983
- middle ground between sh and csh
- basis of the POSIX.2 requirements
- a number of implementations exist

9.1.8 Commands The most typical command is simply a name of a program, perhaps followed by arguments (which are not interpreted by the shell, they are simply passed to the program as a form of input).
Commands in this form are performed, conceptually, as follows (details may differ in actual implementations, e.g. point 2 may be done as part of point 4):

1. check that the program given is not the name of a builtin command or a construct (if so, it is processed differently)
2. the name of the program is taken to be a name of an executable file – a list of directories (given by PATH, which will be explained later) is searched to see if an executable file with a given name exists,
3. the shell performs a fork system call to create a new process (see lecture 3),
4. the child process uses exec to start executing the executable located in 2, passing in any command line arguments,
5. the main shell process does a wait (i.e. it suspends until the executed program terminates).

This means that each command involves quite a lot of work, which is not a problem for interactive use, or reasonably-sized shell scripts. Executing many thousands of commands, especially if the commands themselves run quickly, may get a little slow though.

- typically a name of an executable
  - may also be control flow or a built-in
- the executable is looked up in the filesystem
- the shell does a fork + exec
  - this means new process for each command
  - process creation is fairly expensive

9.1.9 Built-in Commands Some commands of the form program [arguments] are interpreted specially by the shell (i.e. they will not use the above fork + exec process). There are basically two separate reasons why this is done:

1. efficiency – some commands are used often, especially in scripts, and creating new processes all the time is expensive – this is purely an optimisation, and is the case of built-in commands like echo or test,
2. functionality – some effects cannot be (easily, reasonably) done by a child process, mainly because changes need to be done to the main shell process: usually, only the main process can do such changes – this is the case of cd (changes the 'current working directory' of the main process), export (changes the environment of the main process, see also below) or exec (performs an exec without a fork, destroying the shell).

- cd change the working directory
- export for setting up environment
- echo print a message
- exec replace the shell process (no fork)

9.1.10 Parameters

```
variable="some text"
echo $variable
```

- variable names are made of letters and digits
- using variables is indicated with $
- setting variables does not use the $
- all variables are global (except subshells)

Earlier, we have mentioned an idea of 'placeholders', in the context of scripts written down in notepads. Shells take that idea, quite literally, and turn it into what we call variables (at least

[152] Because bad puns should be a human right.
[153] What is it with computer people and bad puns?

in most programming languages; the 'official' terminology in shell is **parameters** – rather in line with the idea of a placeholder).
Essentially, the shell maintains a mapping of names to values, where names are strings made of letters and digits and the values are arbitrary strings. To create or update a mapping, the following command is used:

```
variable="some text"
```

The quotes are not required unless there are spaces in the value. Whitespace around `=` is not allowed (writing `variable = value` is interpreted as the command `variable` with arguments `=` and `value`).

- variables are substituted as text
- `$foo` is simply replaced with the content of `foo`
- arithmetic is not well supported in most shells
  - or expressions, e.g. relational operators
  - consider the POSIX syntax `z=$((x + y))`

9.1.11 Parameter Expansion Variables (parameters) can be used more or less anywhere in any command, including as the command name. This is achieved by writing a dollar sign, `$`, followed by the name of the variable (parameter), like this:

```
echo $variable
```

The command will print `some text`. The substitution is done in a purely textual manner (the process is also known as **parameter expansion**). After substitution, everything about variables is 'forgotten' in the sense that whether any part of the text came from a substitution or was present from the start makes no difference and cannot be detected. This may lead to surprises if the value of a variable contains whitespace (we will discuss this later).
Coming from normal programming languages, a user may be tempted to write something like `$a + $b` in a shell. This will not work: if `a=7` and `b=3`, the above 'expression' will be interpreted as the command `7` with arguments `+` and `3`. To perform arithmetic in a shell script, the expression must be enclosed in `$(( … ))` – to make it a little less painful, variables inside `$(( … ))` do not need to be prefixed with `$`.
They are still substituted as text though:

```
a=3+1; echo $a = $((a))
```

will print `3+1 = 4` (and not e.g. an error because `a` is not a number).[154] However, substitutions within `$(( … ))` **without** dollar signs are **bracketed** – in particular,

```
a=3+1; b=7; echo $((a * b))
```

will print `28`, since it is expanded as `$(((3+1) * 7))`. This is **not** the case for `$` substitutions:

```
a=3+1; b=7; echo $(($a * $b))
```

will print `10`.

- basically like parameter substitution
- written as `` `command` `` or `$(command)`
  - first executes the command
  - and captures its standard output
  - then replaces `$(command)` with the output

9.1.12 Command Substitution Sometimes, it is desirable to **compute** a piece of a command, most commonly by running another program. This can be done using `$( … )`, e.g. `cat $(ls)`:

1. first, `ls` is executed as a shell command (since it is a name of a program, it will be `fork`'d and `exec`'d as normal),
2. the output of `ls` (the list of files in current directory, e.g. `foo.txt bar.txt`) is captured into a buffer,
3. the content of the buffer is substituted into the original command, i.e. `cat foo.txt bar.txt`,
4. the command is executed as normal.

Like with parameter substitution, there are whitespace related caveats (see below).

- whitespace is an argument separator in shell
- multi-word arguments must be quoted
- quotes can be double quotes `"x"` or single `'x'`
  - double quotes allow variable substitution
- whitespace from substitution must be quoted
  - `foo="hello world"`
  - `ls $foo` is different than `ls "$foo"`
- bad quoting is a very common source of bugs
- consider also filenames with spaces in them

9.1.13 Quoting

9.1.14 Quoting and Substitution An important feature of parameter (variable) substitution is that it is done before argument splitting. Hence, values which contain whitespace may be interpreted, after substitution, as multiple arguments. Sometimes, this is desirable, but quite often it is not. Consider `cat $file`, clearly the author expects `$file` to be substituted for a single filename. However, if the value is `foo bar`, the command will be expanded to `cat foo bar` and execute the program `cat` with arguments `foo` and `bar`. Quoting can be used to prevent this from happening.

[154] Depending on the implementation, `a=3+b; b=7; echo $((a))` may or may not work (in the sense that it'll print `10`).

Consider the example on the slide above: the first command, `ls $foo` will expand into `ls hello world` and execute with:

```
argv[ 0 ] = "ls"
argv[ 1 ] = "hello"
argv[ 2 ] = "world"
```

In effect, like with the `cat` example, it will be looking for two separate files. The latter, `ls "$foo"`, will be executed as:
argv[0] = "ls" argv[1] = "hello world"

9.1.15 Special Variables Besides variables (parameters) that the users set themselves, the shell provides a few 'special' variables which contain useful information. Positional parameters refer to the command-line arguments given to the currently executing shell script.
Here are a few more variables:

- `$@` expands to all positional parameters, with special behaviour when double-quoted (each parameter is quoted separately),
- `$*` same, but without the special quoting behaviour,
- `$!` the PID of the last 'background process' (created with the `&` operator which will be discussed later),
- `$-` shell options.

- `$?` is the result of last command
- `$$` is the PID of the current shell
- `$1` through `$9` are positional parameters
  - `$#` is the number of parameters
- `$0` is the name of the shell – `argv[0]`

9.1.16 Environment POSIX has a concept of **environment variables**, which are independent of any shell: they are passed around from process to process, both across `fork` and across `exec`. However, since `fork` makes a new copy of the entire environment, changes in those variables can only be passed down (to **new** child processes), never up (to parent processes), nor to already-running processes in general.
Despite being formally independent of shell, environment variables have similar semantics: their names are alphanumeric strings and their content is arbitrary text. To further add to the confusion, shells treat environment variables in the same way they treat their 'internal' variables (parameters). If `FOO` is an environment variable, a shell will replace `$FOO` by its value, and executing `FOO=bar` as a shell command will change its value in the main shell process (and hence all of its future child processes).

- is like shell variables but not the same
- it is passed to all executed programs
- a child cannot modify environment of its parent
- variables → environment via `export`
- environment variables often act as settings

9.1.17 Important Environment Variables By convention, environment variables are named in all-uppercase. There are a few 'well-known' variables which affect the behaviour of various programs: the `PATH` variable gives a list of directories in which to look for executables (when executing commands in a shell, but also when invoking programs by name from other programs). The `HOME` variable tells programs where to store per-user files (both data and configuration), and so on. Some are set by the system when creating the user session (`HOME`, `LOGNAME`), others are set by the shell (`PWD`), some are normally configured by the system administrator (but can be changed by users), like `PATH`, yet others are configured by the user (`EDITOR`, `EMAIL`).

- `$PATH` tells the system where to find programs
- `$HOME` is the home directory of the current user
- `$EDITOR` and `$VISUAL` – which text editor to use
- `$EMAIL` is the email address of the current user
- `$PWD` is the current working directory

9.1.18 Globbing Let us get back to shell and its syntax. Since files are ubiquitous and many commands expect file names as arguments, shells provide special constructs for working with them. One of those is **globbing**, where a single **pattern** can replace a possibly long list of file names (and hence saves a lot of tedious typing).
Glob expansion is done by the shell itself, i.e. the program receives individual file names as arguments, not the glob. Quotes (both single and double) prevent glob expansion (useful to pass strings which contain `*` or `?` as arguments). Unquoted strings with any of the glob 'meta-characters' is treated (and expanded) as a glob, including in results of parameter expansion (substitution).

- patterns for quickly listing multiple files
- e.g. `ls *.c` shows all files ending in `.c`
- `*` matches any number of characters
- `?` matches one arbitrary character
- works on entire paths – `ls src/*/*.c`

9.1.19 Conditionals The most basic of all control flow constructs is **conditional execution**, where a command is executed or skipped based on the outcome of a previous command. Shells use the traditional `if` keyword, optionally followed by `elif` and `else` clauses.
Unlike most programming languages, `cond` is not an expression, but a regular command. If the command 'succeeds' (terminates with exit code 0), this is interpreted as 'true' and the `then` branch is taken. Otherwise, the `elif` branches are evaluated in turn (if present) and if none succeed, the `else` branch (again, if present) is executed.

- allows conditional execution of commands
- `if cond; then cmd1; else cmd2; fi`
- also `elif cond2; then cmd3; fi`
- `cond` is also a command (the exit code is used)

- originally an external program, also known as `[`
  - nowadays built-in in most shells
  - works around lack of expressions in shell
- returns `true` or `false`
  - can be used with `if` and `while` constructs

9.1.20 `test` (evaluating boolean expressions) While the condition of an `if` statement (command) is a command, it would be often convenient to be able to specify expressions which relate variables to each other, or which check for presence of files. To this end, POSIX specifies a special program called `test` (actually built into most shells).
The `test` command receives arguments like any other command, evaluates them to obtain a boolean value and sets its exit code based on this value, so that `if test …; then …` behaves as expected.

- `test file1 -nt file2` → 'nt' = newer than
- `test 32 -gt 14` → 'gt' = greater than
- `test foo = bar` → string equality
- combines with variable substitution (`test $y = x`)

9.1.21 `test` Examples There are 3 classes of predicates provided by `test`:

1. existence and properties of files,
2. integer comparisons, and
3. string comparisons.

The latter two mimic what 'normal' programming languages provide (albeit with odd syntax). The first makes it easy and convenient to write commands that execute only if a particular file exists (or is missing), a very common task in shell programming.

- `while cond; do cmd; done`
  - `cond` is a command, like in `if`
- `for i in 1 2 3 4; do cmd; done`
  - allows globs: `for f in *.c; do cmd; done`
  - also command substitution
  - `for f in $(seq 1 10); do cmd; done`

9.1.22 Loops After conditional execution, loops are the next most fundamental construct. Again, like in general-purpose programming languages, loops allow shell scripts to repeat a sequence of commands, either:

1. until a particular command fails (a `while` loop, the command in question often being `test`, though of course it can be any command),
2. once for each value in a list, often of file names (which can be in turn constructed by using globs).

Another common form of the `for` loop uses **command substitution** (command expansion) to generate the list. An oft-used helper in this context is (sadly, non-standard) `seq` utility, which generates sequences of numbers. A similar (and likewise non-standard) utility called `jot` is available on BSD systems.

- selects a command based on pattern matching
- `case $x in *.c) cc $x;; *) ls $x;; esac`
  - yes, `case` really uses unbalanced parens
  - the `;;` indicates end of a case

9.1.23 Case Analysis A slightly more advanced control flow construct is **case analysis**, which allows the use of glob-like pattern matching on arbitrary strings (i.e. not just filenames). The string to match against is given after `case`, and is usually a result of parameter or command expansion. Note that the patterns after the `in` clause of the `case` statement are not glob-expanded into a list of filenames.

- `a ; b` (semicolon): run `a` and `b` in sequence
- `a && b` run `b` if `a` succeeded
- `a || b` run `b` if `a` failed
- e.g. compile and run: `cc file.c && ./a.out`

9.1.24 Command Chaining While the straightforward command chaining operator `;` (semicolon) is perhaps too banal to call control flow, there are a few similar operators that are more interesting. The first set is the boolean combinators `&&` and `||` which essentially function like a short-hand syntax for `if` statements. Since commands combined with `&&` and `||` are again commands, these can appear in the condition clause of an `if` or a `while` statement.
However, they are also useful standalone, and also in interactive mode. Especially `&&` can be used to type a sequence of commands that stops on the first failure, significantly cutting down on interaction latency (where the user waits for each command to complete, and after each command, the computer waits for the user to type in the next command).

- shells can run pipelines of commands
- `cmd1 | cmd2 | cmd3`
  - all commands are run in parallel
  - output of `cmd1` becomes input of `cmd2`
  - output of `cmd2` is processed by `cmd3`
- `echo hello world | sed -e s,hello,goodbye,`

9.1.25 Pipes Perhaps the most powerful feature of shells are **pipes**, which offer a very flexible and powerful (even if very simple) way to combine multiple commands. The pipe operator causes both commands to be executed in parallel, and anything that the first program writes to its standard output is sent to the second program on its standard input. POSIX specifies a considerable number of utility programs specifically designed to work well in such pipelines, and many more are available as vendor-specific extensions or in 3rd-party software packages.

- you can also define functions in shell
- mostly a light-weight alternative to scripts
  - no need to `export` variables
  - but cannot be invoked by non-shell programs
- functions can also set variables

9.1.26 Functions Recall that the environment is only passed down, never back up. This means that a shell script setting a variable will not affect the parent shell. However, in functions (and when scripts are invoked using `.`), variables can be set and the effect of such changes is visible in the script that invoked the function.

9.2: Networking Intro

In this section, we will mostly deal with familiar network-related concepts, so that we have sufficient context down the line, when we delve into a bit more detail and into OS-level specifics.

9.2.1 Host and Domain Names The first thing we need to understand is how to identify computers within a network. The primary means to do this is via **hostnames**: human-readable names, which come in two flavours: the name of the computer itself, and a fully-qualified name, which includes the name of the network to which the computer is connected, so to speak.

- hostname = human readable computer name
- hierarchical, little endian: www.fi.muni.cz
- FQDN = fully-qualified domain name
- the local suffix may be omitted (ping aisa)

9.2.2 Network Addresses While humans prefer to refer to computers using human-readable names, those are not suitable for actual communication. Instead, when computers need to refer to other computers, they use numeric addresses (just like with memory locations or disk sectors). Depending on the protocol, the size and structure of the address may be different: traditional IPv4 uses 4 octets, while the addresses in the newer IPv6 use up 16 (128 bits). One other type of address that you can commonly encounter is MAC (from media access control), which is best known from the Ethernet protocol.

- address = machine-friendly and numeric
- IPv4 address: 4 octets (bytes): 192.168.1.1
  - the octets are ordered MSB-first (big endian)
- IPv6 address: 16 octets
- Ethernet (MAC): 6 octets, c8:5b:76:bd:6e:0b

9.2.3 Network Types Networks are broadly categorized into two types: local area, spanning an office, a household, maybe a building. LAN is usually a single **broadcast domain**, which means, roughly speaking, that each computer can directly reach any other computer attached to the same LAN. The most common technologies (layers 1 and 2) used in LANs are the wired **ethernet** (the most common variety running at 1Gb/s, less common but still mainstream versions at 10Gb/s) and the wireless **WiFi** (formally known as IEEE 802.11).
Wide-area networks, on the other hand, span large distances and connect a large number of computers. The canonic WAN is the internet, or the network of an ISP (internet service provider). Wide area networks often use a different set of low-level technologies.

- LAN = Local Area Network
  - Ethernet: wired, 1Gb/s, 10Gb/s, ...
  - WiFi (802.11): wireless, up to ~1Gb/s
- WAN = Wide Area Network (the internet)
  - PSTN, xDSL, PPPoE
  - GSM, 2G (GPRS, ...), 3G (UMTS), 4G (LTE)
  - also LAN technologies – Ethernet, WiFi

9.2.4 Networking Layers The standard model of networking (known as Open Systems Interconnection, or OSI for short) splits the stack into 7 layers, but TCP/IP-centric view of networking often only distinguishes 4, as outlined above. The link layer roughly corresponds to OSI layers 1 (physical) and 2 (data), the internet layer is OSI layer 3, the transport layer is OSI layer 4 and the rest (OSI layers 5 through 7) is lumped under the application layer.
We will follow the simplified TCP/IP model, **but** whenever we refer to layers by number, those are the OSI numbers, as is customary (specifically, IP is layer 3 and TCP is layer 4).

1. Link (Ethernet, WiFi)
2. Internet / Network (IP)
3. Transport (TCP, UDP, ...)
4. Application (HTTP, SMTP, ...)

9.2.5 Networking and Operating Systems For the last two decades or so, networking has been a standard service provided by general-purpose operating systems. In systems with a monolithic kernel, a significant part of the network stack (everything up to and including the transport layer) is part of the kernel and is exposed to user programs via the sockets API.
Additional application-layer functionality is usually available in system libraries: most importantly domain name resolution (DNS) and encryption (TLS, short for transport-layer security, which is confusingly enough an application-layer technology).

- a network stack is a standard part of an OS
- a large part can be inside the kernel
- microkernels → user-space networking
- system libraries & utilities

9.2.6 Kernel-Side Networking The link layer is generally covered by device drivers and the client and server sides of TCP/IP are exposed via the socket API. There are additional components in TCP/IP networks, though: some of them, like routing and packet filtering can be often done in software, and if this is the case, they are usually implemented in the kernel. Bridging and switching (which belong to the link layer) can be done in software too, but is rarely practical. However, many operating systems implement one or both to better support virtualisation.
A few application-layer network services may be implemented in the kernel too, most notably network file systems, but sometimes also other protocols (e.g. kernel-level HTTP acceleration).

- device drivers for networking hardware
- network and transport protocol layers
- routing and packet filtering (firewalls)
- networking-related system calls (sockets)
- network file systems (SMB, NFS)

9.2.7 System Libraries Strictly speaking, the socket API is the domain of system libraries (though in most monolithic kernels, the C functions will map 1:1 to system calls; however, in microkernels, the networking stack is split differently and system libraries are likely to pick up a bigger share of the work).
Since nearly all network-related programs need to be able to resolve hostnames (translate the human-readable name to an IP address), this service is usually provided by system libraries. Likewise, encryption is ubiquitous in the modern internet, and most operating systems provide an SSL/TLS stack, including certificate management.

- the socket and related APIs
- host name resolution (a DNS client)
- encryption and data authentication (SSL, TLS)
- certificate handling and validation

9.2.8 System Utilities & Services The last component of the network stack is located in system utilities and services (daemons). Those are concerned with configuration (including assigning addresses to interfaces and autoconfiguration, e.g. DHCP or SLAAC) and route management (especially important for software-based routers and multi-homed systems).

- configuration (ifconfig, dhclient, dhcpd)
- route management (route, bgpd)
- diagnostics (ping, traceroute)
- packet logging and inspection (tcpdump)
- other network services (ntpd, sshd, inetd)

A suite of diagnostic tools is also usually present, at very least the ping and traceroute programs which are useful for checking connectivity, perhaps tools like tcpdump which allow the operator to inspect packets arriving at an interface.

- packets – units of communication
- addressing – identifying recipients
- delivery – moving packets between nodes

9.2.9 **Networking Aspects** When looking at a network protocol, there are three main aspects to consider: the first is, what constitutes the unit of communication, i.e. how the packets look, what information they carry and so on. The second is addressing: how are target computers and/or programs designated. Finally, packet delivery is concerned with how messages are delivered from one address to another: this could involve routing and/or address translation (e.g. between link addresses and IP addresses).

- protocols run on top of each other
- this is why it is called a network stack
- higher levels make use of the lower levels
  - HTTP uses abstractions provided by TCP
  - TCP uses abstractions provided by IP

9.2.10 **Protocol Nesting** Since we are talking about a **protocol stack**, it is important to understand how the individual layers of the stack interact with each other. Each of the above aspects cuts through the stack slightly differently – we will discuss each in a bit more detail in the following few sections.

- higher-level packets are data to the lower level
- an Ethernet frame can carry an IP packet in it
- the IP packet can carry a TCP packet
- the TCP stream can carry an HTTP request

9.2.11 **Packet Nesting** When we consider packet structure, it is most natural to start with the bottom layers: the packets of the higher layers are simply data for the lower layer. The overall packet structure looks like a matryoshka: an ethernet frame is wrapped around an IP packet is wrapped around an UDP packet and so on.
From the point of view of the upper layers, packet size is an important consideration: when packet-oriented protocols are nested in other packet-oriented protocols, it is useful if they can match their packet sizes (most protocols have a limit on packet size). With the size limitations in mind, in the view 'from top', a packet is handed down to the lower layer as data, the upper layer being oblivious to the additional framing (headers) that the lower layer adds.

- abstract delivery is point-to-point
- routing is mostly hidden from upper layers
- the upper layer requests delivery to an address
- lower layers are usually packet-oriented
- a packet can cross between low-level domains

9.2.12 **Stacked Delivery** When it comes to delivery, the relationships between layers are perhaps the most complicated. In this case, the view from top to bottom is the most appropriate, since lower layers provide delivery as a service to the upper layer.
Since the delivery on the internet layer (OSI layers 3 and up) is usually much wider in scope than that of the link layer, it is quite common that a single IP packet will traverse a number of link-layer domains.

- not as straightforward as packet nesting
  - address relationships are tricky
- special protocols exist to translate addresses
  - DNS for hostname vs IP address mapping
  - ARP for IP vs MAC address mapping

9.2.13 **Layers vs Addressing** Finally, since (packet, data) delivery is a service provided by the lower layers to the upper layers, the upper layer must understand and provide correct lower-level addresses. The easiest way to look at this aspect is pairwise: the link layer and the internet layer obviously need to interact, usually through a special protocol which executes on the link layer, but logically belongs to the internet layer, since it deals with IP addresses.
Situation between the internet and transport layers is much simpler: the address at the transport layer simply contains the internet layer address as a field (e.g. a TCP address is an IP address + a port number).
Finally, the relationship between the application layer and the transport layer is analogous (but not entirely the same) to the internet/link situation. The application layer primarily uses host names to identify computers, and uses a special protocol, known as DNS, which operates using transport-layer addresses, but otherwise belongs to the application layer.

- finds the MAC that corresponds to an IP
- required to allow packet delivery
  - IP uses the link layer to deliver its packets
  - the link layer must be given a MAC address
- the OS builds a map of IP → MAC translations

9.2.14 **ARP (Address Resolution Protocol)** The address resolution protocol, which straddles the link/internet boundary, enables the internet layer to deliver its packets using the services of the link layer. Of course, to request link-layer delivery of a packet, a link address is required, but the IP packet only contains an IP address. The ARP protocol is used to find link addresses of IP nodes which exist in the local network (this includes routers, which operate on the internet layer – in other words, packets destined to leave the local network are sent to a router, using the router's IP address, which is translated into a link-layer address using ARP as usual).

- link-level communication protocol
- largely implemented in hardware
- the OS uses a well-defined interface
  - packet receive and submit
  - using MAC addresses

9.2.15 **Ethernet** Perhaps the most common link layer protocol is ethernet. Most of the protocol is implemented directly in hardware and the operating system simply uses an unified interface exposed by device drivers to send and receive ethernet frames.

- **shared media** are inefficient due to **collisions**
- ethernet is typically **packet switched**
  - a **switch** is usually a **hardware device**
  - but also in software (mainly for virtualisation)
  - physical connections form a **star topology**

High-speed networks are almost exclusively **packet switched**, that is, a node sends packets (frames) to a **switch**, which has a number of physical ports and keeps track of which MAC addresses are reachable on which physical ports. When a frame arrives to a switch, the recipient MAC address is extracted, and the packet is forwarded to the physical port(s) which are associated to that MAC address.

- bridges operate at the **link layer** (layer 2)
- a bridge is a two-port device
  - each port is connected to a **different LAN**
  - the bridge joins the LANs by **forwarding** frames
- can be done in hardware or software
  - brctl on Linux, ifconfig on OpenBSD

Bridges are analogous to switches, with one major difference: the expectation for a switch is that there are many physical ports, but each has only one MAC address attached to it (with perhaps the exception of a special 'uplink' port). A bridge, on the other hand, is optimized for the case of two physical ports, but each side will have many MAC addresses associated with it.

9.2.16 **Link-Layer Protocols** Besides ethernet, two link-layer protocols stand out: PPP (Point-to-Point Protocol) and WiFi.

- many protocols exist beyond ethernet
- PPP = Point-to-Point Protocol
- WiFi ~ wireless ethernet

- a **link-layer** protocol for **2-node networks**
- available over many **physical connections**
  - phone lines, cellular, DSL, Ethernet
  - often used to connect endpoints to the ISP
- supported by most operating systems
  - split between the **kernel** and **system utilities**

The point-to-point protocol is another somewhat important and ubiquitous example of a link-layer protocol and is usually found on connections between LANs, or between a LAN and a WAN.

- WiFi is mostly like (slow, unreliable) Ethernet
- needs **encryption** since anyone can listen
- **authentication** to prevent **rogue connections**
  - PSK (pre-shared key), EAP / 802.11x
- encryption needs **key management**

Finally, WiFi is, from the point of view of the rest of the stack, essentially a slow, unreliable version of ethernet, though internally, the protocol is much more complicated.

9.2.17 **Tunneling** Tunnelling is a technique which allows lower-layer traffic to be nested in the application layer of an existing network. The typical use case is to tie physically distant computers into a single broadcast (link layer) or routing (internet layer) domain.
In this case, there are two instances of the network stack: the VPN software implements an application layer protocol running in the outer stack, while also acting as a link-layer interface (or an internet-layer subnet) that is bridged (routed) as if it was just another physical interface.

- tunnels are virtual layer 2 or 3 devices
- encapsulate traffic in a higher-level protocol
- used in Virtual Private Networks
  - e.g. a software bridge over an UDP tunnel
  - the tunnel is usually encrypted

9.3: The TCP/IP Stack

In this section, we will look at the TCP/IP stack proper, and we will also discuss DNS in a bit more detail.

9.3.1 **IP (Internet Protocol)** IP is a low-overhead, packet-oriented protocol in wide use across the internet and most local area networks (whether they are attached to the internet or not). Quite importantly, its low-overhead nature means that it does not guarantee delivery, nor the integrity of the data it transports.

- uses 4 byte (v4) or 16 byte (v6) addresses
  - split into network and host parts
- it is a packet-based protocol
- is a best-effort protocol
  - packets may get lost, reordered or corrupted

- IP networks roughly correspond to LANs
  - hosts on the **same network** are located with ARP
  - **remote** networks are reached via **routers**
- a **netmask** splits the address into network/host parts
- IP typically runs on top of Ethernet or PPP

Within a single IP network, delivery is handled by the link layer – the local network being identified by a common address prefix (the length of this prefix is part of the network configuration, and is known as the netmask).

- routers **forward** packets **between networks**
- somewhat like **bridges** but **layer 3**
- routers act as normal **LAN endpoints**
  - but represent entire remote IP networks
  - or even the entire internet

Packets for recipients outside the local network (i.e. those which do not share the network part of the address with the local host) are **routed**: a layer 3 device, analogous to a layer 2 switch, forwards the packet to one of its interfaces (into another link-layer domain). The **routing tables** are, however, much more complex than the information maintained by a switch, and their maintenance across the internet is outside the scope of this subject.

- networks are generally used to **provide services**
  - each computer can host multiple
- different **services** can run on different **ports**
- port is a 16-bit number and some are given names
  - port 25 is SMTP, port 80 is HTTP, ...

As we have briefly mentioned earlier, transport-layer addresses have two components: the IP address of the destination computer and a **port number**, which designates a particular service or application running on the destination node.

- control messages
  - destination host/network unreachable
  - time to live exceeded
  - fragmentation required
- diagnostics, e.g. the ping command

9.3.2 ICMP: Control Messages ICMP is the 'service protocol' used for diagnostics, error reporting and network management. The role of ICMP was substantially extended with the introduction of IPv6 (e.g. to include automatic network configuration, via router advertisements and router solicitation packet types). ICMP does not directly provide any services to the application layer.

- ping → echo request and echo reply
- combine with TTL for traceroute

- a stream-oriented protocol on top of IP
- works like a pipe (transfers a byte sequence)
  - must respect delivery order
  - and also re-transmit lost packets
- must establish connections

9.3.3 TCP: Transmission Control Protocol The two main transport protocols in the TCP/IP protocol family are TCP and UDP, with the former being more common and also considerably more complicated. Since TCP is stream-oriented and reliable, it needs to implement the logic to slice a byte stream into individual packets (for delivery using IP, which is packet-oriented), consistency checks (packet checksums) and retransmission logic (in case IP packets carrying TCP data are lost).

- the endpoints must establish a **connection** first
- each connection serves as a separate **data stream**
- a connection is **bidirectional**
- TCP uses a 3-way handshake: SYN, SYN/ACK, ACK

To provide stream semantics to the user, TCP must implement a mechanism which creates the illusion of a byte stream on top of a packet-based foundation. This mechanism is known as a **connection**, and essentially consists of some state shared by the two endpoints. To establish this shared state, TCP uses a 3-way handshake.

- IP packets can arrive out of order
- TCP packets carry sequence numbers
- used to re-assemble the stream
- and to acknowledge reception
  - and subsequently to manage re-transmission

9.3.4 Sequence Numbers Sequence numbers are part of the connection state, and allow the byte stream to be reassembled in the correct order, even if IP packets carrying the stream get reordered during delivery.

- packets can get **lost** for a variety of reasons
  - a **link goes down** for an extended period of time
  - **buffer overruns** on routing equipment
- TCP sends **acknowledgments** for received packets
  - the ACKs use **sequence numbers** to identify packets

Besides packet reordering, TCP also needs to deal with **packet loss**: an event where an IP packet is sent, but vanishes without trace en-route to its destination. A lost packet is detected as a gap in sequence numbers. However, it is the **sender** which must learn about a lost packet, so that it can be retransmitted: for this reason, the recipient of the packet must **acknowledge** its receipt, by sending a packet back (or more often, by piggybacking the acknowledgement on a data packet that it would send anyway), carrying the sequence numbers of packets that have been received.
If an acknowledgement is not received within certain time (dynamically adjusted) from the sending of the original packet, the packet is sent again (retransmitted).

9.3.5 UDP: User Datagram Protocol Not all applications need the comparatively strong guarantees that TCP provides, or conversely, cannot tolerate the additional latency introduced by the algorithms that TCP employs to ensure reliable, in-order delivery. For those cases, UDP presents a very light-weight layer on top of IP, essentially only adding the port number to the addresses, and a 16-bit checksum to the packet header (which is, in its entirety, only 64 bits long).

- TCP comes with non-trivial overhead
  - and its guarantees are not always required
- UDP is a much simpler protocol
  - a very thin wrapper around IP
  - with minimal overhead on top of IP

9.3.6 Firewalls Firewall is a device which separates two networks from each other, typically by acting as the (only) router between them, but also examining the packets and dropping or rejecting them if they appear malicious, or attempt to use services that are not supposed to be visible externally. Often, one of these networks is the internet. Sometimes, the other network is just a single computer.

- the name comes from building construction
- the idea is to separate networks
  - making attacks harder from the outside
  - limiting damage in case of compromise

- packet filtering is an **implementation** of a **firewall**
- can be done on a **router** or at an **endpoint**
- **dedicated** routers + packet filters are **more secure**
  - a **single** such **firewall** protects the **entire network**
  - less opportunity for mis-configuration

Like with other services, it usually pays off to centralize (within a single network) the responsibility for packet filtering, reducing the administrative burden and the space for misconfigured nodes to endanger the entire network. Of course, it is reasonable to run local firewalls on each node, as a second line of defence.

- packet filters operate on a set of **rules**
  - the rules are generally **operator**-provided
- each incoming packet is **classified** using the rules
- and then **dispatched** accordingly
  - may be **forwarded**, dropped, **rejected** or edited

A packet filter is, essentially, a finite state machine (perhaps with a bit of memory for connection tracking, in which case it is a **stateful** packet filter) which examines each packet and decides what action to take on it. The specific classification rules are usually provided by the network administrator; in simple cases, they match on source and destination IP addresses and port numbers, and on the connection status (which is remembered by the packet filter), for TCP packets.
After they are classified, the packets can be forwarded to their destination (as a standard router would), quietly dropped, rejected (sending an ICMP notification to the sender) or adjusted before being sent along (most commonly for network address translation, or NAT, the details of which are out of scope of this subject).

- packet filters are often part of the **kernel**
- the rule parser is a system utility
  - it loads rules from a **configuration file**
  - and sets up the kernel-side filter
- there are multiple **implementations**
  - iptables, nftables in Linux
  - pf in OpenBSD, ipfw in FreeBSD

There are usually two components to a packet filter: one is a system utility which reads a human-readable description of the rules, and based on those, compiles an efficient matcher for use in the kernel component which does the actual classification.

9.3.7 Name Resolution In the last part of this section, let's have a look at hostname resolution and the DNS protocol. What we need is a directory (a yellow pages sort of thing), but one that can be efficiently updated (many updates are done every hour) and also efficiently queried by computers on the network. The system must be scalable enough to handle many millions of names.

- numeric addresses are hard to remember
- host names are used instead
- can be stored in a file, e.g. /etc/hosts
  - impractical for more than ~3 computers
  - internet = millions of computers

9.3.8 DNS: Domain Name System Essentially, at the internet scale, we need some sort of a distributed system (i.e. a distributed database). Unlike relational databases though, delays in update propagation are acceptable, making the design simpler.
The name space of host names is organized hierarchically, and the structure of DNS follows this organisation: going from right to left, starting with the top-level domain (a single dot, often left out), one of the DNS servers for that domain is consulted about the name immediately to the left, usually resulting in the address of another DNS server which can get us more information. The process is repeated until the entire name is resolved, usually resulting in an IP address of the host.

- hierarchical protocol for name resolution
  - runs on top of TCP or UDP
- domain names are split into parts using dots
  - domains know whom to ask for the next bit
  - the name database is effectively distributed

Example: DNS Recursion

- take www.fi.muni.cz. as an example domain
- resolution starts from the right at **root servers**
  - the root servers refer us to the cz. servers
  - the cz. servers refer us to muni.cz
  - finally muni.cz. tells us about fi.muni.cz

The process described above is called **recursion** and is usually performed by a special type of DNS server, which performs the recursion on behalf of its clients and caches the results for subsequent queries. This also means that it can, most of the time, start from the middle, since the name servers of the one or two topmost domains are most likely in the cache.

```
$ dig www.fi.muni.cz. A +trace
.               IN NS j.root-servers.net.
cz.             IN NS b.ns.nic.cz.
muni.cz.        IN NS ns.muni.cz.
fi.muni.cz.     IN NS aisa.fi.muni.cz.
www.fi.muni.cz. IN A  147.251.48.1
```

To observe recursion in practice (and perform other diagnostics on DNS), we can use the dig tool, which is part of the ISC (Internet Software Consortium) suite of DNS-related tools.
Record Types

- A is for (IP) Address
- AAAA is for an IPv6 Address
- CNAME is for an alias
- MX is for mail servers
- and many more

Besides NS records, which tell the system whom to ask for further information, there are many types of DNS records, each carrying different type of information about the name in question. Besides IPv4 and IPv6 addresses, there are free-form TXT records (which are used, for instance, by spam filtering systems to learn about authorized mail servers for a domain), SRV records for service discovery in local networks, and so on.

9.4: Multi-User Systems

Multi-user systems had been the norm until the rise of personal computers circa mid-80s: earlier computers were too expensive and too bulky to be allocated to a single person. Instead, earlier systems used some form of multi-tenancy, whether implemented administratively (batch systems) or by the operating system (interactive, terminal-based computers).

- originally a proxy for people
- currently a more general abstraction
- user is the unit of ownership
- many permissions are user-centered

9.4.1 Users The concept of a **user** has evolved from the need to keep separate accounts for distinct people (the eponymous users of the system). In modern systems, a **user** continues to be an abstraction that includes accounts for individual humans, but also covers other needs. Essentially, **user** is a unit of ownership, and of access control.

- computer is a (often costly) resource
- efficiency of use is a concern
  - a single user rarely exploits a computer fully
- data sharing makes access control a necessity

9.4.2 Computer Sharing While efficient resource usage is what drove multi-tenancy of computer systems, it is the global shared file system that drove the requirement for access control: users do not necessarily wish to trust all other users of the system with access to their files.

- various objects in an OS can be owned
  - primarily files and processes
- the owner → whoever created the object
  - though ownership can be transferred
  - restrictions usually apply

9.4.3 Ownership The standard model of access control in operating systems revolves around **ownership** of **objects**. Generally speaking, ownership of an object confers both rights (to manipulate the object) and obligations (owned objects count towards quotas). Depending on circumstances, object ownership may be transferred, either by the original owner, or by system administrators.

- each **process** belongs to some user
- the process acts **on behalf** of the user
  - same privilege as its owner
  - both **constrains** and **empowers** the process
- processes are **active** participants

The perhaps most important ownership relationship is between users and their processes. This is because processes execute code on behalf of the user, and all actions a user takes on a system

are mediated by some process or another. In this sense, processes act on behalf of their owner and the actions they perform are subject to any restrictions which apply to the user in question.

- each **file** also belongs to some user
- this gives **rights** to the **user**
  - they can **read** and **write** the file
  - they can **change permissions** or ownership
- files are **passive** participants

Like processes, files are objects which are subject to ownership. However, unlike processes, files are passive: they do not perform any actions. Hence in this case, ownership simply gives the owner certain rights to perform actions on the file (most importantly change access control rights pertaining to that file).

9.4.4 Access Control Models There are two main approaches to access control: the common **discretionary** model, where owners decide who can interact with their files (or other objects, as applicable) and **mandatory**, in which users are not trusted with matters of security, and decisions about access control are placed in the hands of a central authority.
In both cases, the operating system grants (or denies) access to object based on an **access control policy**: however, only in the latter case this policy can be thought of as a coherent, self-contained document (as opposed to a collection of rules decided by a number of uncoordinated users).

- owners decide who can access their objects
  - discretionary access control
- disallowed in high-security environments
  - known as mandatory access control
  - a central authority decides the policy

9.4.5 (Virtual) System Users Users have turned out to be a really useful abstraction. It is common practice that services (whether system- or application-level) run under special users of their own. This means that these service can own files and other resources, and run processes under their own identity. Additionally, it means that those services can be restricted using the same mechanisms that apply to 'normal' users.

- users are a useful ownership abstraction
- system services get their own 'fake' users
- allows them to own files and processes
- limits their access to the rest of the OS

9.4.6 Principle of Least Privilege The **principle of least privilege** is an important maxim for designing secure systems: it tells us that, regardless of the subject and object combination, permissions should only be granted where there is genuine need for the subject to manipulate the particular object. The rationale is that mistakes happen, and when they do, we would rather limit their scope (and hence damage): mistakes cannot endanger objects which are inaccessible to the culprit.

- give minimum privilege required
  - applies to software components
  - but also to human users of the system
- this limits the scope of mistakes
  - and also of security compromises

9.4.7 Privilege Separation An important corollary of the principle of least privilege is the design pattern known as **privilege separation**. Systems which follow it are split into a number of independent components, each serving a small, well-defined and security-wise self-contained function. Each of these modules can be then isolated in their own little sandbox and communicate with the rest of the system through narrowly defined interfaces (usually built on some form of inter-process communication).

- different components need different privilege
- least privilege → split the system
  - components are isolated from each other
  - they are given only the rights they need
- components communicate using simple IPC

9.4.8 Process Separation There is not much need for access control of memory: each process has their own and cannot see the memory of any other process (with small, controlled exceptions created through mutual consent of the two processes).
The file system is, however, very different: there is a global, shared namespace that is visible to all users and all processes. Moreover, many of the objects (files) are **meant** to be shared, in a rather ad-hoc fashion, either through 'well-known' paths (this being the case with many system files) or through passing paths around. Importantly, paths are **not** any sort of access token and in almost all circumstances, withholding a path does not prevent access to the object (paths can be easily discovered).

- one process → one address space
  - shared memory only upon request
- each user has a view of the filesystem
  - more sharing by default
  - shared namespace (directory hierarchy)

9.4.9 Access Control Policy We have mentioned earlier, that the totality of the rules that decide which actions are allowed, and which disallowed, is known as an **access control policy**. In the abstract, it is a rulebook which answers questions of the form 'Is (subject) allowed to perform (action) on (object)?' There are clearly many different ways in which this rulebook can be encoded: we will look at some of the most common strategies later.

- there are 3 pieces of information
  - the subject (user)
  - the action/verb (what is to be done)
  - the object (the file or other resource)
- there are many ways to encode this information

- in a typical OS, subjects are (possibly virtual) **users**
  - sub-user units are possible (e.g. programs)
  - **roles** and **groups** could also be subjects
- the subject must be **named** (names, identifiers)
  - easy on a single system, **hard** in a **network**

The most common access control **subject** (at least when it comes to access policy **specification**), are, as was already hinted at, **users**, whether 'real' (those that stand in for people) or virtual (which stand for services).
In most circumstances, it must be possible to **name** the subjects, so that it's possible to refer to them in rules. Sometimes, however, rules can be directly attached to subjects, in which case there is no need for these subjects to have stable identifiers attached.

- available 'verbs' depend on **object** type
- a typical object would be a **file**
  - file: **read**, **written**, **executed**
  - directory: **search**, **list**, **change**
- network connections can be established &c.

The particular choice of actions depends on the object type: each such type has a fixed list of actions, which correspond to operations, or variants of operations, that the operating system offers through its interfaces.
The actions may be affected by the policy directly or indirectly – for instance, the **read** permission on a file is not enforced at the time a read call is performed: instead, it is checked at the time of open, with the provision that read can be only used on file descriptors that are **open for reading**. That is, the program is required to indicate, at the time of open, whether it wishes to read from the file.

- objects can be **manipulated** by **programs**
  - not everything is subject to access control
- **files**, **directories**, **sockets**, shared **memory**, ...
- object **names** depend on their type
  - file paths, i-node numbers, IP addresses, ...

Like subjects, objects need to have names unless the pieces of policy relevant to them are directly attached to the objects themselves. However, in case of objects, this direct attachment is much more common: it is rather typical that an i-node embeds permission information.

- always begins with the hardware
- privileged CPU mode for the kernel
- DMA and IO are restricted
- the MMU provides process isolation

9.4.10 **Enforcement** Now that we have an access control policy and we have established the identity of the user, there is one last thing that needs to be addressed, and that is **enforcement** of the policy. Of course, an access control policy is useless if it can be circumvented.
The ability of an operating system to enforce security stems from hardware facilities: software alone cannot sufficiently constrain other software running on the same computer. The main tool that allows the kernel to enforce its security policy is the MMU (and the fact that only the kernel can program it) and its control over interrupt handlers.

- kernel uses **hardware facilities** to implement security
  - it stands between **resources** and **processes**
  - access is mediated through **system calls**
- **file systems** are part of the kernel
- **user** and **group abstractions** are part of the kernel

Hardware resources are controlled by the kernel: memory via the MMU, processors via the timer interrupt, memory-mapped peripherals again through the MMU and through the interrupt handler table. Since user programs cannot directly access physical resources, any interaction with them must go through the kernel (via system calls), presenting an opportunity for the kernel to check the requested actions against the policy.

- the kernel acts as an **arbitrator**
- a process is trapped in its own **address space**
- processes use system calls to access resources
  - kernel can decide what to allow
  - based on its **access control model** and **policy**

When a system call is executed, the kernel knows the owner of that process, and also any objects involved in the system call. Armed with this knowledge, it can easily consult the access control policy to decide whether the requested action is allowed, and if it is not, return an error to the process, instead of performing the action.

- userland processes can enforce access control
- usually system services which provide IPC API
- e.g. via the getpeereid() system call
- user-level enforcement relies on the kernel

9.4.11 **User-space Enforcement** Just as the kernel sits on resources that user programs cannot directly access, the same principle can be applied in userspace programs, especially services.
Probably the most illustrative example is a relational database: the database engine runs under a dedicated (virtual) user and stores its data in a collection of files. The permissions on those

files are set such that only the owner can read or write them – hence, the kernel will disallow any other process from interacting with those files directly.
Nonetheless, the database system can selectively allow other programs to **indirectly** interact with the data it stores: the programs connect to a database server using a UNIX socket. At this point, the database can ask the operating system to provide the user identifier under which the client is running (using getpeereid).
Since the server can directly access the files which store the data, it can, on the behalf of the client, execute queries and return the results. It can, however, also disallow certain queries based on its own access control policy and the user id of the client.

9.4.12 **Subjects in POSIX** In POSIX systems, there are two basic types of subjects that can appear in the access control policy: users and groups. Since POSIX only covers access control for the file system, objects do not need to be named: their permissions are attached to the i-node.
A special user, known as root, represents the system administrator (also known as the super-user). This account is not subject to permission checking. Additionally, there is a number of actions (usually not attached to particular objects) which only the root user can perform (e.g. reboot the computer).

- 2 types of subjects: users and groups
- each user can belong to multiple groups
- users are split into normal users and root
  - root is also known as the super-user

9.4.13 **User and Group Identifiers** In the access control policy, users and groups are identified by numbers (each user and each group getting a small, locally unique integer). Since these identifiers have a fixed size, they can be stored very compactly in i-nodes, and can be also very efficiently compared, both of which have been historically important considerations. Besides efficiency, the numeric identifiers also make the layout of data structures which carry them simpler, reducing scope for bugs.

- users and groups are represented as numbers
  - this improves efficiency of many operations
  - the numbers are called uid and gid
- those numbers are valid on a single computer
  - or at most, a local network

- the system needs a **database** of **users**
- user **identities** can be **shared** in a network
- could be as simple as a **text file**
  - /etc/passwd and /etc/group
- or as complex as a distributed database

The user database serves two basic roles: it tells the system which users are authorized to access the system (more on this later), and it maps between human-readable user names and the numeric identifiers that the system uses internally.
In local networks, it is often desirable that all computers have the same idea about who the users are, and that they use the same mapping between their names and id's. LDAP and Active Directory are popular choices for centralised network-level user databases.

9.4.14 **Changing Identities** Recall that all processes are created using the fork system call, with the exception of init. When a process forks, the child process inherits the ownership of the parent, that is, it belongs to the same user as the parent does (whose ownership is not affected by fork).
However, if a process is owned by the super-user, it can change its owner by using the setuid system call. Additionally, exec can sometimes change the owner of the process, via the so-called setuid bit (not to be confused with the system call of the same name). The init process is owned by the super-user.

- each process belongs to a particular user
- ownership is inherited across fork()
- super-user processes can use setuid()
- exec() can sometimes change a process owner

9.4.15 **Login** You may recall that at the end of the boot process, a login process is executed to allow users to authenticate themselves and start a session. The traditional implementation of login first asks the user for their user name and password, which it checks against the user database. If the credentials match, the login program sets up the basic environment, changes the owner of the process to the user who just authenticated themselves and executes their preferred shell (as configured in the user database).

- a super-user process manages user logins
- the user types in their name and password
  - the login program authenticates the user
  - setuid() – change the process owner
  - exec() – start a shell for the user

9.4.16 **User Authentication** By far, the most common method of authenticating users (that is, ascertaining that they are who they claim they are) is by asking for a secret – a password or a passphrase. The idea is that only the legitimate owner of the account in question knows this secret.
In an ideal case, the system does not store the password itself (in case the password database is compromised), but stores instead information that can be used to check that a password that the user typed in is correct. The usual way this is done is via (salted) cryptographic hash functions. Besides passwords, other authentication methods exist, most notably cryptographic tokens and biometrics.

- the user needs to authenticate themselves
- passwords are most common
- the system must recognize correct passwords
- user should be able to change them
- biometric methods are also quite popular

- authentication over network is harder
- passwords are easiest, but not easy
  - needs encryption
  - along with computer authentication
- also: 2-factor authentication

9.4.17 Remote Login While password is simply short string that can be quite easily sent across a network, there are caveats. First, the network itself is often insecure, and the password could be snooped by an attacker. This means we need to use cryptography to transmit the password, or otherwise prove its knowledge.
The other problem is, in case we send an encrypted password, that the computer at the other end may not be the one we expect (i.e. it could belong to an attacker).
Since the user is not required to be physically present to attempt authenticating, this significantly increases the risk of attacks, making strong passwords much more important. Besides strong passwords, security can be improved by 2-factor authentication.

- 2 different types of authentication
  - harder to spoof both at the same time
- there are a few factors to pick from
  - something the user knows (password)
  - something the user has (keys, tokens)
  - what the user is (biometric)

9.4.18 2-factor Authentication Two-factor (or multi-factor) authentication is popular for remote authentication (as outlined earlier), since networks make attacks much cheaper and more frequent. In this case, the first factor is usually a password, and the second factor is a cryptographic **token** – a small device (often in the form of a keychain) which generates a unique sequence of codes, one of which the user transcribes to prove ownership of the token. Remote biometric authentication is somewhat less practical (though not impossible).
Of course, two-factor authentication can be used locally too, in which case biometrics become considerably more attractive. Cryptographic tokens or smart cards are also common, though in the local case, they usually communicate with the computer directly, instead of relying on the user to copy a code.

- ensure the password is sent to the right party
- an attacker can impersonate a remote computer
- usually via asymmetric cryptography
  - a private key can be used to sign messages
  - sign a challenge to establish an identity

9.4.19 Computer Authentication When interacting with a remote computer (via a network), it is rather important to ensure that we communicate with the computer that we intended to. While the most immediate concern is sending passwords, of course this is not the only concern: accidentally uploading secret data to the wrong computer would be as bad, if not worse.
A common approach, then, is that each computer gets a unique private key, while its public counterpart (or at least its fingerprint) is distributed to other computers. When connecting, the client can generate a random challenge, and ask the remote computer to sign it using the secret key associated to the computer that we intended to contact, in order to prove its identity. Unless the target computer itself has been compromised, an attacker will be unable to produce a valid signature and will be foiled.

Část 10: POSIX a jazyk C

V této kapitole se zaměříme na rozhraní mezi uživatelskými programy a operačním systémem. Zároveň tak otevřeme poslední tematický blok, který se zabývá konkrétními otázkami implementace operačních systémů.

10.1: The C Language

The C programming language is one of the most commonly used languages in operating system implementations. It is also the subject of PB071, and at this point, you should be already familiar with its basic syntax. Likewise, you are expected to understand the concept of a **function** and other basic building blocks of programs. Even if you don't know the specific C syntax, the idea is very similar to any other programming language you might know.

- there are many different languages
  - C, C++, Java, C#, …
  - Python, Perl, Ruby, …
  - ML, Haskell, Agda, …
- but C has a special place in most OSes

10.1.1 Programming Languages Different programming languages have different use-cases in mind, and exist at different levels of abstraction. Most languages other than C that you will meet, both at the university and in practice, are so-called high-level languages. There are quite a few language families, and there is a number of higher-level languages derived from C, like C++, Java or C#.
For the purposes of this course, we will mostly deal with plain C, and with POSIX (Bourne-style) **shell**, which can also be thought of as a programming language.

- except for assembly, C is the "bare minimum"
- you can almost think of C as portable assembly
- it is very easy to call C functions
- and to use C data structures
- C libraries can be used in most languages

10.1.2 C: The Least Common Denominator You could think of C as a 'portable assembler', with a few minor bells and whistles in form of the standard library. Apart from this library of basic and widely useful subroutines, C provides: abstraction from machine opcodes (with human-friendly infix operator syntax), structured control flow, and automatic local variables as its main advantages over assembly.

In particular the abstraction over the target processor and its instruction set proved to be instrumental in early operating systems, and helped establish the idea that an operating system is an entity separate from the hardware.
On top of that, C is also popular as a systems programming language because almost any program, regardless of what language it is written in, can quite easily call C functions and use C data structures.

10.1.3 The Language of Operating Systems Consequently, C has essentially become a 'language of operating systems': most kernels and even the bulk of most operating systems is written in C. Each operating system (apart from perhaps a few exceptions) provides a C standard library in some form and can execute programs written in C (and more importantly, provide them with essential services).

- many (most) kernels are written in C
- this usually extends to system libraries
- and sometimes to almost the entire OS
- non-C operating systems provide C APIs

10.1.4 (System) Libraries Let us have a closer look at libraries: what they contain, how are they stored in the file system, how are they combined with programs. We will also briefly talk about system call wrappers (which mediate low-level access to kernel services – we will discuss this topic in more detail in the next lecture). Finally, we will look at a few examples of system libraries which appear in popular operating systems.
In this course, when we talk about libraries, we will mean C libraries specifically. Not Python or Haskell modules, which are quite different. That said, a typical C library has basically two parts, one is header files which provide a description of the interface (the API) and the compiled library code (an archive or a shared library).
The interface (as described in header files) consists of functions (for which, the types of arguments and the type of return value are given in a header file) and of data structures. The bodies of the functions (their implementation) is what makes up the compiled library code. To illustrate:

- mainly C functions and data types
- interfaces defined in header files
- definitions provided in libraries
  - static libraries (archives): libc.a
  - shared (dynamic) libraries: libc.so
- on Windows: msvcrt.lib and msvcrt.dll
- there are (many) more besides libc / msvcrt

```
// declaration: «what» but not «how»

int sum( int a, int b );

// definition: «how» is it done?

int sum( int a, int b )
{
    return a + b;
}
```

The first example is a declaration: it tells us the name of a function, its inputs and its output. The second example is called a **definition** (or sometimes a **body**) of the function and contains the operations to be performed when the function is called.

10.1.5 The POSIX C Library As we have already mentioned previously, it is a tradition of UNIX systems that libc combines the basic C library and the basic POSIX library. For the following, a particular subset of the POSIX library is going to be rather important, namely the **system call wrappers**. Those are C functions whose only purpose is to invoke their matching **system calls**.

- libc – the C runtime library
- contains ISO C functions
  - printf, fopen, fread
- and a number of POSIX functions
  - open, read, gethostbyname, …
  - C wrappers for system calls

10.1.6 Additional System Libraries While libc is quite central, there are many other libraries that are part of a UNIX system. You would find most of these on most UNIX systems in some form:

- the math library libm provides implementations of math (floating point) functions like sin, cos, exp, etc.,
- functions for writing multi-threaded programs reside in the library called libpthread – including pthread_create to start a new thread, and a host of synchronisation primitives, like pthread_mutex_lock &c,
- the C++ standard library often comes bundled with the system (notably, this is not a library that can be used from C) – libstdc++ or libc++ are the most common implementations,
- cryptographic primitives are provided in the libcrypto library, including symmetric ciphers (like AES), asymmetric crypto (DH, ECDH, DSA, RSA, …), X.509 certificates, authentication codes and hash functions (SHA2, HMAC, CMAC, …),
- the terminal access library libcurses, which allows programs to portably work with a bewildering array of different hardware and software terminals.

- the math library libm
- thread library libpthread
- the C++ standard library
- cryptography: libcrypto (OpenSSL)
- terminal access: libcurses

- /usr/lib on most Unices
  - may be mixed with application libraries
  - especially on Linux-derived systems
  - also /usr/local/lib for user/app libraries
- on Windows: C:\Windows\System32
  - user libraries often bundled with programs

10.1.7 Library Files The machine code that makes up the library (i.e. the code that was generated from function definitions) resides in files. Those files are what we usually call 'libraries' and they usually live in a specific filesystem location. On most UNIX system, those locations are /usr/lib and possibly /lib for system libraries and /usr/local/lib for user or application libraries. On certain systems (especially Linux-based), user libraries are mixed with system libraries and they are all stored in /usr/lib.
On Windows, the situation is similar in that both system and application libraries are installed in a common location. Additionally, on Windows (and on macOS), shared libraries are often installed alongside the application.

- stored in libfile.a, or file.lib (Windows)
- only needed for compiling (linking) programs
- the code is copied into the executable
- the resulting executable is also called static
  - and is easier to work with for the OS
  - but also more wasteful

10.1.8 Static Libraries Static libraries are only used when building executables and are not required for normal operation of the system. Therefore, many operating systems do not install them by default – they have to be installed separately as part of the developer kit. When a static library is linked into a program, this basically entails copying the machine code from the library into the final executable.
In this scenario, after linking is performed, the library is no longer needed since the executable contains all the code required for its execution. For system libraries, this means that the code that comes from the library is present on the system in many copies, once in each program that uses the library. This is somewhat alleviated by linkers only copying the parts of the library that are actually needed by the program, but there is still substantial duplication.
The duplication arising this way does not only affect the file system, but also memory (RAM) when those programs are loaded – multiple copies of the same function will be loaded into memory when such programs are executed.

- required for running programs
- linking is done at execution time
- less code duplication
- can be upgraded separately
- but: dependency problems

10.1.9 Shared (Dynamic) Libraries The other approach to libraries is **dynamic**, or **shared** libraries. In this case, the library is required to actually run the program: the linker does not copy the machine code from the library into the executable. Instead, it only notes that the library must be loaded alongside with the program when the latter is executed.
This reduces code duplication, both on disk and in memory. It also means that the library can be updated separately from the application. This often makes updates easier, especially in case a library is used by many programs and is, for example, found to contain a security problem. In a static library, this would mean that each program that uses the library needs to be updated. A shared library can be replaced and the fixed code will be loaded alongside programs as usual.
The downside is that it is difficult to maintain binary compatibility – to ensure that programs that were built against one version of the library also work with a later version. When this is violated, as often happens, people run into dependency problems (also known as DLL hell on Windows).

- on UNIX: /usr/include
- contains prototypes of C functions
- and definitions of C data structures
- required to compile C and C++ programs

10.1.10 Header Files Like static libraries, header files are only required when building programs, but not when using them. Header files are fragments of C source code, and on UNIX systems are traditionally stored in /usr/include. User-installed header files (i.e. not those provided by system libraries) live under /usr/local/include (though again, on Linux-based systems user and system headers are often intermixed in /usr/include).

```
// Header Example 1 (from ‹unistd.h›)

int      execv(char *, char **);
pid_t    fork(void);
int      pipe(int *);
ssize_t  read(int, void *, size_t);
```

This is an excerpt from an actual system header file, and declares a few of the functions that comprise the POSIX C API.

```
// Header Example 2 (from ‹sys/time.h›)

struct timeval
{
    time_t   tv_sec;
    long     tv_usec;
};

int gettimeofday(timeval *, timezone *);
```

```
int settimeofday(timeval *, timezone *);
```

This is another excerpt from an actual header – this time the snippet contains a definition of a **data type**. The layout (order of fields and their types, along with hidden **padding**) of such structures is quite important, since that becomes part of the ABI. In other words, the definition above describes not just the high-level interface but also how bytes are laid out in memory.

10.1.11 Documentation Most OS vendors provide extensive documentation of their programmer's interfaces. On UNIX, this is typically part of the OS installation itself (manual pages, command man), while on Windows, this is a separate resource (these days accessible online, previously distributed in print or on optical media).

- manual pages on UNIX
  - try e.g. man 2 write on aisa.fi.muni.cz
  - section 2: system calls
  - section 3: library functions (man 3 printf)
- MSDN for Windows
  - https://msdn.microsoft.com

10.2: The C Compiler & Linker

While compiling (and linking) programs is not core functionality of an operating system, it is quite useful to understand how these components work. Moreover, in earlier systems, a C compiler was considered a rather essential component and this tradition continues in many modern UNIX systems to this day. We will discuss different artefacts of compilation – object files, libraries and executables, as well as the process of linking object code and libraries to produce executables. We will also highlight the differences between static and shared (dynamic) libraries.

10.2.1 C Compiler Compilers transform human-readable programs into machine-executable programs. Of course, both those forms of the program need to be stored in memory: the first is usually in the form of **plain text** (usually encoded as UTF-8, or in older systems as ASCII). In this form, bytes stored in the file encode human-readable letters.
On the output side, the file is **binary** (which is really just a catch-all term for files that are not plain text), and stores machine-friendly **instructions** – primitive operations that the CPU can execute. Only the compiler output cannot be directly executed yet, even though most of the instructions are in their final form.
The missing piece are addresses: numbers which describe memory locations within the program itself (they may point at instructions or at data embedded in the program). At this stage, though, neither code nor data has been assigned to particular addresses, and hence the program cannot be executed (it will need to be **linked** first, more on that later).

- many POSIX systems ship with a C compiler
- the compiler takes a C source file as input
  - a text file with a .c suffix
- and produces an object file as its output
  - binary file with machine code in it
  - but cannot be directly executed

10.2.2 Object Files The purpose of object files is to store this semi-finished machine code, along with any static data (like string literals or numeric constants) that appear in the program. All this is sorted into **sections** – usually one section for machine code (also called text and called .text in the object file), another for read-only data (e.g. string literals), called .rodata, another for mutable but statically-initialized variables – .data. Bundled with all this is **metadata**, which describes the content of the file (again in a machine-readable form).
One example of such metadata is a **symbol table**, which gives file-relative addresses of high-level functions that have been compiled into the object file. That is, the compiler will take a definition of a function that we wrote in C and emit machine code for this function. The .text section of an object file will consist of a number of such functions, one after another: the symbol table then tells us where each of the functions begins.

- contain native machine (executable) code
- along with static data
  - e.g. string literals used in the program
- possibly split into a number of sections
  - .text, .rodata, .data and so on
- and metadata
  - list of symbols and their addresses

10.2.3 Object File Formats There is a number of different physical layouts of object files, and each of those also carries slightly different semantics. By far the most common format used in POSIX systems is **ELF**. The other common formats in contemporary use are **PE** (used by MS operating systems) and **Mach-O** (used by Apple operating systems).

- a.out – earliest UNIX object format
- COFF – Common Object File Format
  - adds support for sections over a.out
- PE – Portable Executable (MS Windows)
- Mach-O – Mach Microkernel (macOS)
- ELF – Executable and Linkable Format (UNIX)

10.2.4 Archives (Static Libraries) An archive is the simplest way to bundle multiple object files. As the name implies, it is essentially just a collection of object files stored as a single file. Each object file retains its identity and its content does not change in any way when it is bundled into an archive.
The only difference from a typical data archive (a tar or a zip archive, say) is that besides the object files themselves, the archive contains an additional metadata section – a symbol table, or rather a symbol index. If someone (typically the linker) needs to find the definition of a particular function (symbol), it can first consult this archive-wide index to find which object file provides that symbol. This makes linking more efficient, since the linker does not need to sequentially scan each object file in the archive to find the definition.

- static libraries on UNIX are called archives
- this is why they get the .a suffix
- they are like a zip file full of object files
- plus a table of symbols (function names)

- object files are incomplete
- they can use symbols defined elsewhere
  - the definitions can be in libraries
  - or in other object files
- a linker puts multiple object files together
  - to produce a single executable
  - or maybe a shared library

10.2.5 Linker As pointed out earlier, it is the job of a **linker** to combine object files (and libraries) into executables. The process is fairly involved, so we will describe it across the next few slides. The **input** to the linker is a bunch of **object files** and the output is a single **executable** or sometimes a single **shared library**.
Even though archives are handled specially by the linker, object files which are given to the linker directly will always become part of the final executable. Object files provided in archives are only used if they provide symbols which are required to complete the executable.

- we use symbolic names to call functions &c.
- but machine instructions need an address
- the executable will eventually live in memory
- linker assigns those addresses

10.2.6 Symbols vs Addresses The main entities that come up during linking are **symbols** and **addresses**. In a program, the machine code and the data is loaded in memory, and as we know, each memory location has an **address**. The program in its compiled form can use addresses to refer to parts of itself. For instance, to call a subroutine, we provide its starting address to a special `call` instruction, which tells the CPU to start executing code from that address.
However, when humans write programs, they do not assign addresses to pieces of data, to functions or to individual instructions. Instead, if the program needs to refer to a part of itself, we give those parts names: those names are known as **symbols**. It is the shared responsibility of the compiler and the linker to assign addresses to the individual symbols, in such a way that the objects stored in memory do not conflict (overlap).
If you think about it, it would be very difficult to do by hand: we usually don't know how long the machine code will be for any given function, and we would need to guess and then add gaps in case we need to add more code to a function, and so on. And we would need to remember which code lives at which address and so on. It is all very uncomfortable, and even assembly programmers usually avoid assigning addresses by hand. In fact, one of the primary roles of an assembler is to translate from symbolic to numeric addresses. But I digress.

- the linker processes one object file at a time
- it maintains a symbol table
  - mapping symbols (names) to addresses
  - updated as more objects are processed
- relocations are processed at the end
- resolving symbols = finding their addresses

10.2.7 Resolving Symbols The linker works by maintaining an 'incomplete executable' and makes progress by merging each of the input object files into this work-in-progress file. The strategy for assigning final addresses is simple enough: there's a single output `.text` section, a single output `.data` section and so on. When an input file is processed, its own `.text` section is simply appended to the `.text` produced so far. The same process is repeated for every section.
The symbol tables of the input object files are likewise merged one by one, and the addresses adjusted as symbols are added. In addition to symbol **definitions**, object files contain symbol **uses** – those are known as relocations, and are stored in a relocation table. Relocations contain the **address of the instruction** that needs to be patched and the **symbol** the address of which is to be patched in. Like the sections themselves and the symbol table, the relocation table is built up.
The relocations are also processed by the linker: usually, this means writing the final address of a particular symbol into an as-of-yet incomplete instruction or into a variable in the data section. This is usually done once the output symbol table is complete.
The relocation and symbol tables are often discarded at the end (but may be retained in the output file in some cases – the symbol table more often than the relocation table).

- finished image of a program to be executed
- usually in the same format as object files
- but already complete, with symbols resolved
- except if using shared libraries

10.2.8 Executable The output of the linker is, in the usual case, an **executable**. This is a file that is based on the same format as object files of the given operating system, but is **complete** in some sense. In static executables (those which don't use shared libraries), all references and relocations are already resolved and the program can be loaded into memory and directly executed by the CPU, without further adjustments.
It is also worth noting that the addresses that the executable uses when referring to parts of itself are **virtual addresses** (this is also the case with shared libraries below). We will talk more about those in a later lecture, but right now we can at least say that this means that different programs on the same operating system can use overlapping addresses for their instructions and data. This is not a problem, because virtual addresses are private to each process, and hence each copy of each executing program.

- a shared library is only loaded once
- they use symbolic names (like object files)
- mini-linker in the OS to resolve those names
- shared libraries use other shared libraries
- dependencies form a DAG

10.2.9 Shared Libraries The downside of static libraries is that they need to be loaded separately (often in slightly different versions) along with each program that uses them: in fact, since the linker embedded them into the program, they are quite inseparable from it.
As we have already mentioned, this is not very efficient. Instead, we can store the library code in separate executable-like files that get loaded into the address space of programs that need it. Of course, relocations in the main program that refer to symbols from shared libraries (and vice versa), and obviously also relocations in shared libraries that refer to other shared libraries,

those need to be resolved. This is usually done either when the program is loaded into memory, or lazily, right before the relocation is first used.
In either case, there needs to be a program which will resolve those relocations: this is the **runtime linker** – it is superficially similar to the normal, compile-time linker, but in reality is quite different.
The dependencies of shared libraries usually form a DAG → Directed Acyclic Graph.

10.2.10 Addresses Revisited We mentioned that executables and libraries use virtual addresses to refer to their own parts. However, this does not help in shared libraries as much as it helps in executables. The letdown is that we want to load the same library along with multiple programs: but if the addresses used by the library are fixed, this means that the library needs to be loaded at the same start address into each program that uses that library. This quickly becomes impractical as we add more libraries into the system – no two libraries would be allowed to overlap and none of them would be allowed to overlap with any of the executables.
In practice, what we instead do is that we compile the libraries in such a way that they don't use absolute addresses to refer to parts of themselves. This often adds a little execution overhead, but makes it possible to load the library at any address range that is available in the current process. This makes the job of the runtime linker much easier.

- a program must be loaded into memory
- some of its parts refer to other parts
- naively requires fixed addresses
- shared libraries use position-independent code
- no need to know the base address

10.2.11 Compiler, Linker &c. On many UNIXes, the compiler and the linker are available as part of the system itself. The command names are standardized.

- the C compiler is usually called `cc`
- the linker is known as `ld`
- the archive (static library) manager is `ar`
- the runtime linker is often known as `ld.so`

10.3: File-Based APIs

On POSIX systems, the API for using the filesystem is a very important one, because it in fact provides access to a number of additional resources, which appear as 'abstract' (special) files in the system.

10.3.1 Everything is a File File is an abstraction: it is an object from which we can read bytes and into which we can write bytes (not all files will let us do both). In regular files, we can read and write at any offset, and if we write something we can later read that same thing (unless it was rewritten in the meantime).
Directories are somewhat like this: we can read bytes from them to find out what files are present in that directory and how to find them in the file system. We can create new entries by writing into the directory. Incidentally, this is not how things are usually done, but it's not hard to imagine it could be.
Quite a few **devices** (peripherals) behave this way: all kinds of hard drives (just a big bunch of bytes), printers (write some bytes to have them printed), scanners (write bytes to send commands, read bytes with the image data), audio devices (read bytes from microphones, write bytes into speakers), and so on.
Pipes are like that too: one program writes bytes, and another reads them. And network connections are more or less just pipes that work across the network.

- part of the UNIX design philosophy
- directories are files
- devices are files
- pipes are files
- network connections are (almost) files

10.3.2 Why is Everything a File Since we already have an API to work with **abstract files** (because we need to work with real files anyway), it becomes reasonable to ask why not use this existing API to work with other objects that look like files. It makes sense not just at the level of C functions, but at the level of command-line programs too. In general, re-using existing mechanisms makes things more flexible, and often also simpler. Of course, there are caveats (devices often need to support operations that don't map well to reading or writing bytes, sockets are also somewhat problematic).

- re-use the comprehensive file system API
- re-use existing file-based command-line tools
- bugs are bad → simplicity is good
- want to print? `cat file.txt > /dev/ulpt0`
  - (reality is a little more complex)

10.3.3 Reminder: File Paths Paths are how we refer to files and directories within the tree. The top-level (**root**) directory is named `/`. Each directory **entry** carries a name (and a link to the actual file or directory it represents) – this name can be used in the path to refer to the given entity. So with a path like `/usr/include`, we start at the **root directory** (the initial slash), then in that directory, we look for an entity called `usr` and when we find it, we check that it is a directory again. If that is so, we then look at its direct descendants again and look for an entity labelled `include`.

- filesystems use paths to point at files
- a string with `/` as a directory delimiter
  - the delimiter is `\` on Windows
- a leading `/` indicates the filesystem root
- e.g. `/usr/include`

- very central in Plan9
- central in most UNIX systems
  - cf. Linux pseudo-filesystems
  - /proc provides info about all processes
  - /sys gives info about the kernel and devices
- somewhat reduced in Windows
- quite suppressed in Android (and more on iOS)

10.3.4 The Role of Files and Filesystems Different operating systems put different emphasis on the file system. We will take the way POSIX positions the file system as the baseline – in this case, the file system is quite central: in addition to regular files and directories, all sorts of special files appear in the file system and provide access to various OS facilities. However, there are also many services and APIs that are not based on the file system, including e.g. process management, memory management and so on. In many UNIX-like systems, the reliance on FS-based APIs is notched up a bit: e.g. process management is done via a virtual /proc filesystem (many different systems), or device discovery and configuration via /sys (Linux). Another level above that is Plan9, where essentially everything that can be made into a file system is made into one. Another experimental system, GNU/Hurd, has a similar ambition.
If we go the other way from POSIX, we have the native Windows APIs, which emphasise the file system much less than would be typical in POSIX. Most objects have dedicated APIs, even if they are rather file-like. However, the file system is still prominently present both in the APIs and in the user interface. Both are further suppressed by modern 'scaled-down' operating systems like Android and iOS (even if both are POSIX-compatible under the hood, 'normal' applications are not allowed to access the POSIX API, or the file system, and it is usually also hidden from users).

- you open a file (using the open() syscall)
- you can read() and write() data
- you close() the file when you are done
- you can rename() and unlink() files
- you can use mkdir() to create directories

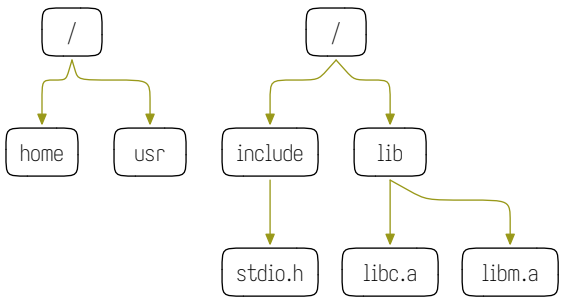
10.3.5 The Filesystem API So how does the file system API look on POSIX systems? To work with a file, you usually need to open it first: you supply a **path** and some flags to tell the OS what you intend to do with the file. More on that in a short while. When you have a file open, you can read data from it and write data into it. When you are done, you use close to free up the associated resources. To work with directories, you usually don't need to open them (though you can). You can rename files (this is a directory operation) using rename, remove them from the file system hierarchy using unlink (this erases the corresponding directory entry), and you can create new directories using mkdir.

- the kernel keeps a table of open files
- the file descriptor is an index into this table
- you do everything using file descriptors
- non-Unix systems have similar concepts
  - descriptors are called handles on Windows

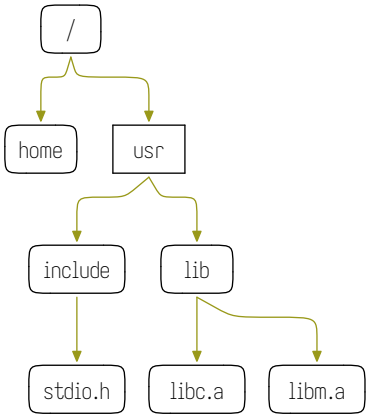
10.3.6 File Descriptors Remember open? When we want to work with a file, we need a way to identify that file, and paths are not super convenient in this respect: someone could rename the file we were working with, and suddenly it is gone, or worse, the file could be replaced by a different file or even a directory. Additionally, looking up a file by its path is a comparatively expensive operation: the OS has to read every directory mentioned in the **path** and run a lookup on it. While this information is often cached in RAM, it still takes valuable time.
When we open a file, we get back a **file descriptor** – this is a small integer, and using this descriptor as an index into a table, the kernel can look up all the metadata it needs (to carry out reads and writes) in constant time. The descriptor is also associated with the file directly, so if the file is moved around or even unlinked from the directory tree, the descriptor still points to the same file.
Most non-POSIX file system APIs have a similar notion (sometimes open does not return a number but a different data type, e.g. a pointer, and sometimes this value is called a **handle** instead of a descriptor… but the concept is more or less the same).

- UNIX joins all file systems into a single hierarchy
- root of one FS becomes a directory in another
  - this is called a mount point
- Windows uses drive letters instead (C:, D: &c.)

10.3.7 Mounts A single computer (and hence, a single operating system) may have more than one hard drive available to it. In this case, it is customary that each such device contains its own file system: the question arises, how to present such multiple file systems to the user. The UNIX strategy is to present all the file systems within a single directory tree: to this end, one of the file systems is picked as a **root** file system: in this (and only in this) file system, the FS root directory / is the same as the system root directory. All other file systems are joined existing directories of other file systems at their root. Consider two file systems:



If we now **mount** the second file system onto the /usr directory of the first, we get the following unified hierarchy:



Usually, file systems are mounted onto **empty** directories: if /usr was not empty on the left (root) file system, its content would be hidden by the mount.
The other strategy is to present multiple file systems using multiple separate trees. This is the strategy implemented by the MS Windows family of operating systems: each file system is assigned a single letter, and each becomes its own, separate tree.

10.3.8 Sockets Sockets are, in some sense, a generalization of pipes. There are essentially 3 types of sockets:

1. a **listening** socket, which allows many **clients** to connect to a single **server** – strictly speaking, these sockets do not transport data, instead, they allow processes to establish **connections**,
2. a **connected** socket, one of which is created for each connection and which behaves essentially like a bidirectional pipe (standard pipes being unidirectional),
3. a **datagram** socket, which can be used to send data without establishing connections, using special send/receive API.

While the third is rather special and un-pipe-like, the first two are usually used together as a point-to-multipoint means of communication: the server listens on an **address**, and any client which has this address can establish communication with the server. This is quite unlike pipes, which usually need to be pre-arranged (i.e. the programs must already be aware of each other).

- the socket API comes from early BSD Unix
- sockets represent network connections
- more complicated than normal files
  - establishing connections is hard
  - packet loss is common
- you get a file descriptor for an open socket
- you can read() and write() to sockets

10.3.9 Socket Types There are two basic address types: internet sockets, which are used for inter-machine communication (using TCP/IP), and **unix domain sockets**, which are used for local communication. A unix socket is like a named pipe: it has a path in the file system, and client programs can use this path to establish a connection to the server.

- sockets can be internet or unix domain
  - internet sockets connect to other computers
  - Unix sockets live in the filesystem
- sockets can be stream or datagram
  - stream sockets are like pipes
  - you can write a continuous stream of data
  - datagram sockets send individual messages

Část 11: The Kernel

This lecture is about the kernel, the lowest layer of an operating system.
First, we will look at processor modes and how they mesh with the layering of the operating system. We will move on to the boot process, because it somewhat illustrates the relationship between the kernel and other components of the operating system, and also between the firmware and the kernel. We will look in more detail at kernel architecture: things that we already hinted at in previous lectures, and will also look at exokernels and unikernels, in addition to the architectures we already know (micro and monolithic kernels).
The fourth part will focus on system calls and their binary interface – i.e. how system calls are actually implemented at the machine level. This is closely related to the first part of the lecture about processor modes, and builds on the knowledge we gained last week about how system calls look at the C level.
Finally, we will look at what are the services that kernels provide to the rest of the operating system, what are their responsibilities and we will also look more closely at how microkernel and hybrid operating systems work.

11.1: Privileged Mode

11.1.1 CPU Modes There is a number of operations that only programs running in supervisor mode can perform. This allows kernels to enforce boundaries between user programs. Sometimes, there are intermediate privilege levels, which allow finer-grained layering of the operating system. For instance, drivers can run in a less privileged level than the 'core' of the kernel, providing a

- privileged (supervisor) and a user mode
- all modern general-purpose CPUs
  - not necessarily with micro-controllers
- x86 provides 4 distinct privilege levels
  - most systems only use ring 0 and ring 3
  - Xen can use ring 1 for guest kernels

level of protection for the kernel from its own device drivers. You might remember that device drivers are the most problematic part of any kernel.
In addition to device drivers, multi-layer privilege systems in CPUs can be used in certain virtualisation systems. More about this towards the end of the semester.

- user mode is restricted
  - this is how user programs are executed
  - also most of the operating system
- privileged mode → can do ~anything
  - most importantly it can program the MMU
  - the kernel runs in this mode

11.1.2 **Privileged Mode** The kernel executes in privileged mode of the processor. In this mode, the software is allowed to do anything that's possible. In particular, it can (re)program the memory management unit (MMU, see next slide). Since MMU is how program separation is implemented, code executing in privileged mode is allowed to change the memory of any program running on the computer. This explains why we want to reduce the amount of code running in supervisor (privileged) mode to a minimum.
The way most operating systems operate, the kernel is the only piece of software that is allowed to run in this mode. The code in system libraries, daemons and so on, including application software, is restricted to the user mode of the processor. In this mode, the MMU cannot be programmed, and the software can only do what the MMU allows based on the instructions it got from the kernel.

- is a subsystem of the processor
- takes care of address translation
  - programs use virtual addresses
  - MMU translates them to physical addresses
- the mappings is managed by the OS (kernel)

11.1.3 **Reminder: Memory Management Unit** Let's have a closer look at the MMU. Its primary role is **address translation**. Addresses that programs refer to are **virtual** – they do not correspond to fixed physical locations in memory chips. Whenever you look at, say, a pointer in C code, that pointer's numeric value is an address in some virtual address space. The job of the MMU is to translate that virtual address into a physical one – which has a fixed relationship with some physical capacitor or other electronic device that remembers information.
How those addresses are mapped is programmable: the kernel can tell the MMU how the translation goes, by providing it with translation tables. We will discuss how page tables work in a short while; what is important now is that it is the job of the kernel to build them and send them to the MMU.

- kernel can be mapped into all processes
  - this improves performance on many CPUs
  - (until meltdown hit us, anyway)
- kernel pages have a special supervisor flag set
  - inaccessible in user mode
  - protects kernel from tampering

11.1.4 **Kernel Protection** Replacing the page tables is usually a rather expensive operation and we want to avoid doing it as much as possible. We especially want to avoid it in the **system call** path (you probably remember system calls from last week, and we will talk about system calls in more detail later today). For this reason, it is a commonly employed trick to map the kernel into **each process**, but make the memory inaccessible to user-space code. Unfortunately, there have been some CPU bugs which make this less secure than we would like.

11.2: Booting

The boot process is a sequence of steps which starts with the computer powered off and ends when the computer is ready to interact with the user (via the operating system).

- upon power on, the system is in a default state
  - mainly because RAM is volatile
- the entire platform needs to be initialised
  - this is first and foremost the CPU
  - console hardware (keyboard, display, ...)
  - then the rest of the devices

11.2.1 **Starting the OS** Computers can be turned off and on (clearly). When they are turned off, power is no longer available and dynamic RAM will, without active refresh, quickly forget everything it held. Hence when we turn the computer on, there is nothing in RAM, the CPU is in some sort of default state and variations of the same are true of pretty much every sub-device in the computer. Except for the content of persistent storage, the computer is in the state it was when it left the factory door. The computer in this state is, to put it bluntly, not very useful.

- the process starts with a built-in hardware init
- followed by firmware init
  - BIOS on 16 and 32 bit systems
  - EFI on current amd64 platforms
- the firmware then loads a bootloader
- the bootloader loads the kernel

11.2.2 **Boot Process** We will not get into the hardware part of the sequence. The switch is flipped, the hardware powers up and does its thing. At some point, firmware takes over and does some more things. The hardware and firmware is finally put into a state, where it can begin loading the operating system. There is usually a piece of software that the firmware loads from persistent storage, called a **bootloader**. This bootloader is, more or less, a part of the operating system: its purpose is to find and load the kernel (from persistent storage, usually by using firmware services to identify said storage and load data from it). It may or may not understand file systems and similar high-level things. In the simplest case, the bootloader has a list of disk blocks in which the kernel is stored, and requests those from the firmware. In modern systems, both the firmware and the bootloader are quite sophisticated, and understand complicated, high-level things (including e.g. encrypted drives).

11.2.3 CPU Init Let's go back to start and fill in some additional details. First of all, what is the state of the CPU at boot, and why does the operating system need to do anything? This has to do with backward compatibility: a CPU usually starts up in the most-compatible mode – in case of 32b x86 processors, this is 16b mode with the MMU disabled. Since the entire platform keeps backward compatibility, the firmware keeps the CPU in this mode and it is the job of either the bootloader or the kernel itself to fix this. This is not always the case (modern 64b x86 processors still start up in 16b mode, but the firmware puts them into **long mode** – that is the 64b one – before handing off to the bootloader).

- depends on both architecture and platform
- on x86, the CPU starts in 16-bit mode
- BIOS & bootloader may stay in this mode
- the kernel switches to protected mode

11.2.4 Bootloader A bootloader is a short, platform-specific program which loads the kernel from persistent storage (usually a file system on a disk) and hands off execution to the kernel. The bootloader might do some very basic hardware initialization, but most of that is done by the kernel itself in a later stage.

- historically tens of kilobytes of code
- the bootloader locates the kernel on disk
  - may allow the operator to choose a kernel
  - limited understanding of file systems
- then it loads the kernel image into RAM
- and hands off control to the kernel

Example: Modern Booting on x86

- the bootloader runs in **protected mode**
  - or even the long mode on 64-bit CPUs
- the firmware understands the FAT filesystem
  - it can **load files** from there into memory
  - this vastly **simplifies** the boot process

The boot process has been considerably simplified[155] on x86 computers in the last decade or so. Much higher-level APIs have been added to the standardized firmware interface, making the boot code considerably simpler.

Example: Booting ARM

- no unified firmware interface
- U-boot is common
- the bootloader needs **low-level** hardware knowledge
- this makes writing bootloaders for ARM quite **tedious**
- current U-boot can use the **EFI protocol** from PCs

Unlike the x86 world, the ARM ecosystem is far less standardized and each system on a chip needs a slightly different boot process. This is extremely impractical, since there are dozens of SoC models from many different vendors, and new ones come out regularly. Fortunately, U-boot has become a de-facto standard, and while U-boot itself still needs to be adapted to each new SoC or even each board, the operating system is, nowadays, mostly insulated from the complexity.

11.2.5 Kernel Boot We are finally getting to familiar ground. The bootloader has loaded the kernel into RAM and jumped at a pre-arranged address inside the kernel image. The instructions stored at that address kickstart the kernel initialization sequence. The first part is usually still rather low-level: it puts the CPU and some basic peripherals (console, timers and so on) into a state in which the operating system can use them. Then it hands off control into C code, which then sets up basic data structures used by the kernel. Then the kernel starts initializing individual peripheral devices – this task is performed by individual device drivers. When peripherals are initialized, the kernel can start looking for the **root filesystem** – it is usually stored on one of the attached persistent storage devices (which should now be operational and available to the kernel via their device drivers).
After mounting the root filesystem, the kernel can set up an empty process and load the `init` program into that process, and hand over control. At this point, kernel stops behaving like a sequential program with `main` in it and fades into background: all action is driven by user-space processes from now on (or by hardware interrupts, but we will talk about those much later in the course).

- the kernel then initialises device drivers
- and the root filesystem
- then it hands off to the `init` process
- at this point, the user space takes over

11.2.6 User-mode Initialisation We are far from done. The `init` process now needs to hunt down all the other file systems and mount them, start a whole bunch of **system services** and perhaps some **application services** (daemons which are not part of the operating system – things like web servers).
Once all the essential services are ready, `init` starts the `login` process, which then presents the familiar login screen, asking the user to type in their name and password. At this point, the boot process is complete, but we will have a quick look at one more step.

- performed by `init`
- mounts file systems
- starts up user-mode system services
- then it starts application services
- and finally the `login` process

[155] For some value of 'simplified'.

- the login process initiates the user session
- loads desktop modules and application software
- drops the user in a (text or graphical) shell
- now you can start using the computer

11.2.7 **After Log-In** When the user logs in, another initialization sequence starts: the system needs to set up a **session** for the user. Again, this involves some steps, but at the end, it's finally possible to interact with the computer.

11.3: Kernel Architecture

In this section, we will look at different architectures (designs) of kernels: the main distinction we will talk about is which services and components are part of the kernel proper, and which are outside of the kernel.

- monolithic kernels (Linux, *BSD)
- microkernels (Mach, L4, QNX, NT, ...)
- hybrid kernels (macOS)
- type 1 hypervisors (Xen)
- exokernels, rump kernels

11.3.1 **Architecture Types** We have already mentioned the main two kernel types earlier in the course. Those types represent the extremes of mainstream kernel design: microkernels are the smallest (most exclusive) mainstream design, while monolithic kernels are the biggest (most inclusive). Systems with **hybrid** kernels are a natural compromise between those two extremal designs: they have 2 components, a microkernel and a so-called **superserver**, which is essentially a gutted monolithic kernel – that is, the functionality covered by the microkernel is removed.
Besides 'mainstream' kernel designs, there are a few more exotic choices. We could consider type 1 (bare metal) hypervisors to be a special type of an operating system kernel, where **applications** are simply virtual machines – i.e. 'normal' operating systems (more on this later in the course). Then there are **exokernel** operating systems, which drastically cut down on services provided to applications and **unikernels** which are basically libraries for running entire applications in kernel mode.

- handles memory protection
- (hardware) interrupts
- task / process scheduling
- message passing
- everything else is separate

11.3.2 **Microkernel** A microkernel handles only the essential services – those that cannot be reasonably done outside of the kernel (that is, outside of the privileged mode of the CPU). This obviously includes programming the MMU (i.e. management of address spaces and memory protection), handling interrupts (those switch the CPU into privileged mode, so at least the initial interrupt routine needs to be part of the kernel), thread and process switching (and typically also scheduling) and finally some form of inter-process communication mechanism (typically message passing). With those components in the kernel, almost everything else can be realized outside the kernel proper (though device drivers do need some additional low-level services from the kernel not listed here, like DMA programming and delegation of hardware interrupts).

- all that a microkernel does
- plus device drivers
- file systems, volume management
- a network stack
- data encryption, ...

11.3.3 **Monolithic Kernels** A monolithic kernel needs to include everything that a microkernel does (even though some of the bits have a slightly different form, at least typically: inter-process communication is present, but may be of different type, driver integration looks different). However, there are many additional responsibilities: many device drivers (those that need interrupts or DMA, or are otherwise performance-critical) are integrated into the kernel, as are file systems and volume (disk) management. A complete TCP/IP stack is almost a given. A number of additional bits and pieces might be part of the kernel, like cryptographic services (key management, disk encryption, etc.), packet filtering, a kitchen sink and so on. Of course, all that code runs in privileged mode, and as such has complete power over the operating system and the computer as a whole.

- we need a lot more than a microkernel provides
- a true microkernel OS has many modules
- each device driver runs in a separate process
- the same for file systems and networking
- those modules / processes are called servers

11.3.4 **Microkernel Redux** The question that now arises is who is responsible for all the services listed on the previous slide (those that are part of a monolithic kernel, but are missing from a microkernel). In a 'true' microkernel operating system, those services are individually covered, each by a separate process (also known as a **server** in this context).

- based around a microkernel
- most services are provided by a superserver
- easier to implement than true microkernel OS
- strikes a middle ground on performance

11.3.5 **Hybrid Kernels** In a hybrid kernel, most of the services are provided by a single large server, which is somewhat isolated from the hardware. It is often the case that the server is based on a monolithic OS kernel, with the lowest-level layers removed, and replaced with calls to the microkernel as appropriate.
Hybrid kernels are both cheaper to design and theoretically perform better than 'true' (multi-server) microkernel systems.

- microkernels are more robust
- monolithic kernels are more efficient
- what is easier to implement is debatable
- hybrid kernels are a compromise

11.3.6 **Micro vs Mono** The main advantage of microkernels is their robustness in face of software bugs. Since the kernel itself is small, chances of a bug in the kernel proper are much diminished compared to the relatively huge code base of a monolithic kernel. The impact of bugs outside the kernel (in servers) is considerably smaller, since those are isolated from the rest of the

system and even if they provide vital services, the system can often recover from a failure by restarting the failed server.
On the other hand, monolithic kernels offer better performance, mainly through reduced context switching, which is still fairly expensive even on modern, virtualisation-capable processors. However, as monolithic kernels adopt technologies such as kernel page table isolation to improve their security properties, the performance difference becomes smaller.
Implementation-wise, monolithic kernels offer two advantages: in many cases, code can be written in direct, synchronous style, and different parts of the kernel can share data structures without additional effort. In contrast, a proper multi-server system often has to use asynchronous communication (message passing) to achieve the same goals, making the code harder to write and harder to understand. Long-term, improved modularity and isolation of components could outweigh the short-term gains in programming efficiency due to more direct programming style.

- exokernel: very reduced services
- hypervisor: runs operating systems as apps
- unikernel: only runs a single application

11.3.7 Other Kernel Types Operating systems based on microkernels still provide the full suite of services to their applications, including file systems, network stacks and so on. The difference lies in where this functionality is implemented, whether the kernel proper, or in a user-mode server.

- smaller than a microkernel
- much **fewer abstractions**
  - applications only get **block** storage
  - networking is much reduced
- only **research systems** exist

With exokernels, this is no longer true: the services provided by the operating system are severely cut down. The resulting system is somewhere between a paravirtualized computer (we will discuss this concept in more detail near the end of the course) and a 'standard' operating system. Unlike virtual machines (and unikernels), process-based application isolation is still available, and plays an important role. No production systems based on this architecture currently exist.
A **bare metal hypervisor** is similar to an exokernel or a microkernel operating system (depending on a particular hypervisor and on our point of view). Typically, a hypervisor provides interfaces and resources that are traditionally implemented in hardware: block devices, network interfaces, a virtual CPU, including a virtual MMU that allows the 'applications' (i.e. the guest operating systems) to take advantage of paging.

- hypervisor can use **coarser abstractions** than an OS
- e.g. entire storage devices instead of a filesystem

Unikernels constitute a different strand (compared to exokernels) of minimalist operating system design. In this case, process-level multitasking and address space isolation are not part of the kernel: instead, the kernel exists to support a single application by providing (a subset of) traditional OS abstractions like a networking stack, a hierarchical file system and so on. When an application is bundled with a compatible unikernel, the result can be executed directly on a hypervisor (or an exokernel).

- kernels for running a **single application**
  - makes little sense on real hardware
  - but can be very useful on a **hypervisor**
- bundle applications as **virtual machines**
  - without the overhead of a general-purpose OS

Comparison of unikernels vs exokernels:

- an exokernel runs **multiple applications**
  - includes process-based isolation
  - but **abstractions** are very **bare-bones**
- unikernel only runs a **single application**
  - provides more-or-less **standard services**
  - e.g. standard hierarchical file system
  - socket-based network stack / API

11.4: System Calls

In the remainder of this lecture, we will focus on monolithic kernels, since the more progressive designs do not use the traditional system call mechanism. In those systems, most 'system calls'

are implemented through message passing, and only services provided directly by the microkernel use a mechanism that resembles system calls as described in this section.

- kernel executes in privileged mode of the CPU
- kernel memory is protected from user code
- but: user code needs to ask kernel for services
- needs to switch the CPU into privileged mode
- cannot be done arbitrarily (security)

11.4.1 Reminder: Kernel Protection The main purpose of the system call interface is to allow secure transfer of control between a user-space application and the kernel. Recall that each executes with different level of privileges (at the CPU level). A viable system call mechanism must allow the application to switch the CPU into privileged mode (so that the CPU can execute kernel code), but in a way that does not allow the application to execute its own code in this mode.

- hand off execution to a kernel routine
- pass arguments into the kernel
- obtain return value from the kernel
- all of this must be done safely

11.4.2 System Calls We would like system calls to behave more-or-less like standard subroutines (e.g. those provided by system libraries): this means that we want to pass arguments to the subroutine and obtain its return value. Like with the transfer of control flow, we need the argument passing to be safe: the user-space side of the call must not be able to read or modify kernel memory.

- details are very architecture-specific
- the kernel sets a fixed entry address
- uses a specific instruction
- change the CPU mode + jump into kernel

11.4.3 Trapping into the Kernel Security from execution of arbitrary code by the application is achieved by tying the privilege escalation (i.e. the entry into the privileged CPU mode) to a simultaneous transfer of execution to a fixed address, which the application is unable to change. The exact mechanism is highly architecture-dependent, but the principle outlined here is universal.

Example: x86

- there is an int instruction on those CPUs
- this is called a **software interrupt**
  - interrupts are normally a **hardware** thing
  - interrupt **handlers** run in **privileged mode**
- it is also synchronous
- the handler is set in IDT (interrupt descriptor table)

On traditional (32 bit) x86 CPUs, the preferred method of implementing the system call trap was through **software interrupts**. In this case, the application uses an int instruction, which causes the CPU to perform a process analogous to a hardware interrupt. The two important aspects are:

1. the CPU switches into privileged mode to execute the **interrupt handler**,
2. reads the address to jump to from an **interrupt handler table**, which is a data structure stored in RAM, at an address given by a special register.

The kernel sets up the interrupt handler table in such a way that user-level code cannot change it (via standard MMU-based memory protection). The register which holds its address cannot be changed outside of privileged mode.

- those are available on a range of CPUs
- generally not very efficient for system calls
- the handler address is retrieved from memory
- a lot of CPU state needs to be saved

11.4.4 Aside: Software Interrupts A similar mechanism is available on many other processor architectures. There are, however, some downsides to using this approach for system calls, the main being their poor performance. Since the mechanism piggy-backs on the hardware variety of interrupts, the CPU usually saves a lot more computation state than would be required. As an additional inconvenience, there are multiple entry-points, which must therefore be stored in RAM (instead of a register), causing additional delays when the CPU needs to read the interrupt table. Finally, arguments must be passed through memory, since registers are reset by the interrupt, again contributing to increased latency.

Example: On the x86 architecture, software interrupts were the preferred mechanism to provide services to application programs until the end of the 32-bit x86 era. Interestingly, x86 CPUs since 80386 offer a mechanism that was directly intended to implement operating system services (i.e. syscalls), but it was rather complex and largely ignored by operating system programmers.

- those are used even in **real mode**
  - legacy 16-bit mode of 80x86 CPUs
  - BIOS (firmware) routines via int 0x10 & 0x13
  - MS-DOS API via int 0x21
- and on older CPUs in 32-bit **protected mode**
  - Windows NT uses int 0x2e
  - Linux uses int 0x80

11.4.5 Syscall instructions: amd64 / x86_64 When x86 switched to a 64-bit address space, many new instructions found their way into the instruction set. Among those was a simple, single-entrypoint privilege escalation instruction. This mechanism avoids most of the overhead associated with software interrupts: computation state is managed in software, allowing compilers to only save and restore a small number of registers across the system call (instead of having the CPU automatically save its entire state into memory).

- sysenter and syscall instructions
  - and corresponding sysexit / sysret
- the entry point is in a machine state register
- there is only one entry point
  - unlike with software interrupts
- quite a bit faster than interrupts

11.4.6 Which System Call? Usually, there is only a single entry point (address) shared by all system calls. However, the kernel needs to be able to figure out which service the application program requested.
This is achieved by simply sending the **syscall number** as an argument in a specific CPU register. The kernel can then decide, based on this number, which kernel routine to execute on behalf of the program.

- often there are many system calls
  - there are more than 300 on 64-bit Linux
  - about 400 on 32-bit Windows NT
- but there is only a handful of interrupts
  - and only one sysenter address

11.4.7 System Call Sequence The first stage of a system call is executed in user mode, and is usually implemented in libc. After the CPU is switched into privileged mode, a kernel routine – the syscall handler – starts executing:
After the switch to privileged mode, the kernel needs to make sense of the arguments that the user program provided, and most importantly, decide which system call was requested. The code to do this in the kernel might look like the above switch statement:

- first, libc prepares the system call arguments
- puts the syscall number in the correct register
- then the CPU is switched into privileged mode
- this also transfers control to the syscall handler

```
switch ( sysnum )
{
   case SYS_write: return syscall_write();
   case SYS_read: return syscall_read();
   /* many more */
}
```

Since different system calls expect different arguments, the specific argument processing is done after the system call is dispatched based on its number. In modern systems, arguments are passed in CPU registers, but this was usually not possible with protocols based on software interrupts (instead, arguments would be passed through memory, usually at the top of the user-space stack).
For example, on amd64 Linux, all system call arguments go into **registers** – there can be up to 6 of them. Of course, some of those arguments can in fact be addresses which hold additional data (like the buffer for read or write).

11.5: Kernel Services

Finally, we will revisit the services offered by monolithic kernels, and look at how they are realized in microkernel operating systems.

11.5.1 What Does a Kernel Do? The first two points are a core responsibility of the kernel: those are rarely 'outsourced' into external services. The remaining services are a core part of an **operating system**, but not necessarily of a kernel. However, it is hard to imagine a modern, general-purpose operating system which would omit any of them. In traditional (monolithic) designs, they are all part of the kernel.

- memory & process management
- task (thread) scheduling
- device drivers
- file systems
- networking

11.5.2 Additional Services A monolithic kernel may provide a number of additional services, with varying importance. Not all systems provide all the services, and the implementations can look quite different across operating systems. Out of this (incomplete) list, IPC (inter-process communication) is the only item that is quite universally present, in some form, in microkernels. Moreover, while dedicated IPC mechanisms are common in monolithic kernels, they are more important in a microkernel.

- inter-process communication
- timers and time keeping
- process tracing, profiling
- security, sandboxing
- cryptography

11.5.3 Reminder: Microkernel Systems Recall that a microkernel is small: it only provides services that cannot be reasonably implemented outside of it. Of course, the operating system as a whole still needs to implement those services. Two basic strategies are available:

1. a single program, running in a single process, implements all the missing functionality: this program is called a superserver, and internally has an architecture that is rather similar to that of a standard monolithic kernel,
2. each service is provided by a separate, specialized program, running in its own process (and hence, address space) – this is characteristic of so-called 'true' microkernel systems.

- the kernel proper is very small
- it is accompanied by servers
- microkernel systems have many servers
- each device, filesystem, etc. is separate
- in hybrid systems, there is one, or a few

There are of course different trade-offs involved in those two basic designs. A hybrid system (i.e. one with a superserver) is easier to initially design and implement (for instance, persistent storage drivers, the block layer, and the file system all share the same address space, simplifying the implementation) and is often considerably faster, since communication between components does not involve context switches. On the other hand, a true microkernel system with services and drivers all strictly separated into individual processes is more robust, and in theory also easier to scale to large SMP systems.

- programs don't care which server provides what
- for services, we take a monolithic view
- services are used through system libraries
- mechanics of the call are abstracted away

11.5.4 **Kernel Services** From a user-space point of view, the specifics of kernel architecture should not matter. Applications use system libraries to talk to the kernel in either case: it is up to the libraries in question to implement the protocol for locating relevant servers and interacting with them.

- not all device drivers are part of the kernel
- case in point: printer drivers
- also some USB devices (but not the USB bus)
- part of the GPU/graphics stack
  - memory and output management in kernel
  - most of OpenGL in user space

11.5.5 **User-Space Drivers** While user-space drivers are par for the course in microkernel systems, there are also certain cases where drivers in operating systems based on monolithic kernels have significant user-space components. The most common example is probably printer drivers: low-level communication with the printer (at the USB level) is mediated by the kernel, but for many printers, document processing comprises a large part of the functionality of the driver. In some cases, this involves format conversion (e.g. PCL printers) but in others, the input document is rasterised by the driver on the main CPU: instead of sending text and layout information to the printer, the driver sends pixel data, or even a stream of commands for the printing head.
The situation with GPUs is somewhat analogous: low-level access to the hardware is provided by the kernel, but again, a large part of the driver is dedicated to data manipulation: dealing with triangle meshes, textures, lighting and so on. Additionally, modern GPUs are invariably *programmable*: a shader compiler is also part of the driver, translating high-level shader programs into instruction streams that can be executed by the CPU.

Část 12: OS Virtualization

This lecture will focus on running multiple operating systems on the same physical computer. Until now, we have always assumed that the operating system (in particular the kernel) has direct control over physical resources. This week, we will see that this does not always need to be the case (in fact, it is increasingly rare in production systems). Instead, we will see that multiple operating systems may share a single computer in a manner similar to how multiple applications (processes) co-exist within an operating system.
We will also explore a compromise approach, known as **containers**, where only the user-space parts of the operating system are duplicated and isolated from each other, while the kernel remains shared and retains direct control of the underlying machine.

12.1: Hypervisors

In the domain of hardware-accelerated virtualisation, a **hypervisor** is the part of the VM software that is roughly equivalent to an operating system kernel.

- also known as a Virtual Machine Monitor
- allows execution of multiple operating systems
- like a kernel that runs kernels
- improves hardware utilisation

12.1.1 **What is a Hypervisor** While hypervisor itself behaves a bit like a kernel, standing as it does between the hardware and the virtualised operating systems, the virtualised operating systems running on top are, in a sense, like processes (including their kernels). In particular, they are isolated in physical memory (by using either regular MMU and a bit of software magic, or using an MMU capable of second-level translation) and they time-share on the available processors.

- OS-level sharing is tricky
  - user isolation is often insufficient
  - only root can install software
- the hypervisor/OS interface is simple
  - compared to OS-application interfaces

12.1.2 **Motivation** Virtualised operating systems allow a degree of autonomy that is not usually possible when multiple users share a single operating system. This is partially due to the simplicity of the interface between the hypervisor and the operating system: there are no file systems, in fact no communication between the operating systems (other than through standard networking), no user management and so on. Virtual machines simply bundle up some resources and make them available to the operating system.

- many resources are "virtualised"
  - physical memory by the MMU
  - peripherals by the OS
- makes resource management easier
- enables isolation of components

12.1.3 **Virtualisation in General** Operating systems (or computers, if you prefer) are of course not the only thing that can be (or is) virtualised. If you think about it, a lot of operating system itself is built around some sort of virtualisation: virtual memory, file systems, network stack,

device drivers – they all, in some sense, virtualise hardware resources. This in turn makes it possible for multiple programs, and multiple users, to share those resources safely and fairly.

12.1.4 Hypervisor Types There are two basic types of hypervisors, based on how the overall system is layered. In type 1, the hypervisor is at the bottom of the stack (just above hardware), and is responsible for management of the basic resources (a bit like a simple microkernel): processor and RAM (scheduling and memory management, respectively).
On the other hand, type 2 hypervisors run on top of an operating system and reuse its scheduler and memory management: the virtual machines appear as actual processes of the host system.

- type 1: bare metal
  - standalone, microkernel-like
- type 2: hosted
  - runs on top of normal OS
  - usually need kernel support

12.1.5 Examples & History The first foray into running multiple operating systems on the same hardware was made by IBM in the late 60s and was made, on big iron, a rather standard feature soon after.

- bare metal: z/VM, Xen, Hyper-V, ESX
- hosted: VMWare, VirtualBox
- Linux KVM, FreeBSD BHyve, OpenBSD VMM
- CP/CMS 1968, VM/370 1972, z/VM 2000

12.1.6 Desktop Virtualisation Small (personal) computers, for a long time, did not offer any OS virtualisation capabilities. Performance of PC processors became sufficient to do PC-on-PC emulation in mid-90s, but the performance penalty was initially huge and was only suitable to run legacy software (which was designed for much slower hardware).

- x86 hardware lacks virtual supervisor mode
- software-only solutions viable since late 90s
- Bochs 1994, VMWare 1999, QEMU 2003

12.1.7 Paravirtualisation A decade later, VMWare has made a breakthrough in software-based virtualisation technology, by inventing paravirtualisation: this required modifications to the guest operating system, but by the time, open-source operating systems were gaining a foothold – and porting open-source systems to a paravirtualising hypervisor was not too hard.

- introduced as VMI in 2005 by VMWare
- alternative approach in Xen in 2006
- relies on modification of the guest OS
- near-native speed without HW support

12.1.8 The Virtual x86 Revolution Around the same time, vendors of desktop CPUs started to incorporate virtualization extensions, which in turn made it unnecessary to modify the guest operating system (at least in principle). By 2008, mainstream desktop processors offered MMU virtualisation, further simplifying x86 hypervisor design (and making it more efficient at the same time).

- 2005: virtualisation extensions on x86
- 2008: MMU virtualisation
- unmodified guest at near-native speed
- most software-only solutions became obsolete

12.1.9 Paravirtual Devices However, paravirtualisation made a quick and dramatic comeback: while virtualisation of CPU and memory was, for the most part, handled by the hardware itself, a hardware-based approach is not economical for virtualisation of peripherals.
Additionally, paravirtualised peripherals do not need changes in the guest operating system: all that is required is a quite regular device driver that targets the respective protocol. The virtual peripherals offered by the host system then simply appear as regular devices through an appropriate device driver running in the guest.

- special drivers for virtualised devices
  - block storage, network, console
  - random number generator
- faster and simpler than emulation
  - orthogonal to CPU/MMU virtualisation

12.1.10 Virtual Computers The entire system running under a virtualised operating system is known as a virtual machine (or, sometimes, a virtual computer), not to be confused with program-level VMs like the Java Virtual Machine.
Most virtual computers only provide a few essential resources:

- the CPU and RAM
- persistent (block) storage
- network connection
- a console device

A typical virtual machine will offer at least a processor, memory, block storage (on which the operating system will store a file system), a network connection and a console for management. While other peripherals are possible, they are not very common, at least not on servers.

- usually known as Virtual Machines
- everything in the computer is virtual
  - either via hardware (VT-x, EPT)
  - or software (QEMU, virtio, ...)
- much easier to manage than actual hardware

12.1.11 CPU Sharing Most instructions (specifically those available to user-space programs) are simply executed without additional overhead by the host CPU, without direct involvement of the hypervisor. However, the hypervisor does manage the virtualised MMU. However, just as importantly, when the CPU encounters certain types of privileged instructions, it will invoke the hypervisor to perform the required actions in software.

- same principle as normal processes
- there is a scheduler in the hypervisor
  - simpler, with different trade-offs
- privileged instructions are trapped

12.1.12 RAM Sharing Like CPU virtualisation, memory sharing is built on the same basic principles that standard operating systems use to isolate processes from each other. Memory is sliced into pages and the MMU does the heavy lifting of address translation.
Software solution: Shadow Page Tables

- the **guest** system **cannot** access the MMU
- set up **shadow table**, invisible to the guest

- very similar to standard paging
- software (shadow paging)
- or hardware (second-level translation)
- fixed amount of RAM for each VM

- guest page tables are sync'd to the sPT by VMM
- the gPT can be made read-only to cause traps

The trap can then synchronise the gPT with the sPT, which are translated versions of each other. The 'physical' addresses stored in the gPT are virtual addresses of the hypervisor. The sPT stores real physical addresses, since it is used by the real MMU.
Hardware solution: Second-Level Translation

- hardware-assisted MMU virtualisation
- adds guest-physical to host-physical layer
- greatly **simplifies** the VMM
- also much **faster** than shadow page tables

Shadow page tables cause a lot of overhead, trapping every change of the guest page table into the hypervisor. Unfortunately, page tables are rearranged by the guest operating system rather often (on real hardware, this is comparatively cheap).
However, modern processors offer another level of translation, which is inaccessible to the guest operating system. Since the MMU is aware of virtualisation, the guest can directly modify its page tables, without compromising isolation of VMs from each other (and from the hypervisor).

- usually a paravirtualised NIC
  - transports frames between guest and host
  - usually connected to a SW bridge in the host
  - alternatives: routing, NAT
- a single physical NIC is used by everyone

12.1.13 **Network Sharing** In contemporary virtualisation solutions, networking uses a paravirtual NIC (network interface card) which is connected to an Ethernet tunnel pseudo-device in the host system (essentially a virtual network interface card that handles Ethernet frames). The frames sent on the paravirtual device appear on the virtual NIC in the host and vice versa. The pseudo-device is then either software-bridged to the hardware NIC (and hence to the outside ethernet), or alternatively, routing (layer 3) is set up between the pseudo-device and the hardware NIC.

- usually also paravirtualised
- often backed by normal files
  - maybe in a special format
  - e.g. based on copy-on-write
- but can be a real block device

12.1.14 **Virtual Block Devices** Like networking, block storage is typically based on paravirtualisation. In this case, the host side of the device is either backed by a regular file in the file system of the host, or sometimes it is backed by a block device on the same (often virtualised, e.g. through LVM/device-mapper or similar technology, but sometimes backed directly by a hardware block device).

- mainly useful in desktop systems
- GPU / graphics hardware
- audio equipment
- printers, scanners, …

12.1.15 **Special Resources** Now that we have covered the essentials, let's briefly look at other classes of hardware. However, with the possible exception of compute GPUs, peripherals are only useful on desktop systems, which are a tiny market compared to server virtualisation.

- an anti-virtualisation technology
- based on an IO-MMU (VT-d, AMD-Vi)
- a virtual OS can touch real hardware
  - only one OS at a time, of course

12.1.16 **PCI Passthrough** Let's first mention a very generic, but very un-virtualisation method of giving hardware access to a virtual machine, that is, exposing a PCI device to the guest operating system directly, via IO-MMU-mapped memory. An IO-MMU must be involved, because otherwise the guest OS could direct the hardware to overwrite physical memory that belongs to the host, or to another VM running on the same system. With that covered, though, there is nothing that stops the host system from handing over control of specific PCI endpoints to a guest (of course, the host system must not attempt to communicate with those devices though its own drivers, else chaos would ensue).

- can be assigned (via VT-d) to a single OS
- or time-shared using native drivers (GVT-g)
- paravirtualised
- shared by other means (X11, SPICE, RDP)

12.1.17 **GPUs and Virtualisation** Of course, since a GPU is attached through PCI, it can be shared using the IO-MMU (VT-d) approach described above. However, modern GPUs all support time-sharing (i.e. they allow contexts to be suspended and resumed, just like threads and processes on a CPU). For this to work, the hypervisor (or the host OS) must provide drivers for the GPU in question, so that it can mediate access to individual VMs.
Another solution, is paravirtualisation: the guest uses a vendor-neutral protocol to send a command stream to the driver running in the hypervisor, which in turn does the multiplexing. The guest system still needs the userspace part of the GPU driver to generate the command stream and to compile shaders.
Finally, existing network graphics protocols can be, of course, used between a guest and the host, though they are never quite as efficient as one of the specialised options.

- useful either via passthrough
  - audio, webcams, …
- or standard sharing technology
  - network printers & scanners
  - networked audio servers

12.1.18 **Peripherals** Finally, there is a wide array of peripherals that can be attached to a PC. Some of them, like printers and scanners, and in some cases (or rather, in some operating systems) audio hardware, can be shared over standard networks, and hence also between guests and the host over a virtual network. For this type of peripherals, there is either no loss in

performance (printers, scanners) or possibly a small increase in latency (this mainly affects audio devices).
Peripheral passthrough is essentially bus virtualisation:

- **virtual** PCI, USB or SATA bus
- **forwarding** to a real device
  - e.g. a single USB stick
  - or a single SATA drive

Of course, network-based sharing is not always practical. Fortunately, most peripherals attach to the host system through a handful of standard buses, which are not hard to either pass through, or paravirtualise. The devices then appear as endpoints on the virtual bus of the requisite type exposed to the guest operating system.

12.1.19 **Suspend & Resume** An important feature available in most virtualisation solutions is the ability to suspend the execution of a VM and store its state in a file (i.e. create an image of the running virtualised OS). Of course this is only useful if the image can later be loaded and resumed 'as if nothing happened'.
On the outside, this looks rather like what happens when a laptop's lid is closed: the computer stops (in this case to save energy) and when it is opened again, continues where it left off. An important difference here is that in a VM, the guest operating system does not need to cooperate, or even be aware of the suspend/resume operation.

- the VM can be quite easily stopped
- the RAM of a stopped VM can be copied
  - e.g. to a file in the host filesystem
  - along with registers and other state
- and also later loaded and resumed

12.1.20 **Migration Basics** Of course, if an image can be stored in a file, it can just as well be sent over a network. Resuming an image on a different host is called a 'paused' migration, since the VM is paused for the duration of the network transfer: depending on the size of the image, this can be long enough to time out TCP connections or application-level protocols. Of course, even if this does not happen, there will be a noticeable lag for any interactive use of such a system.
Of course, the operation is predicated on the requirement that the supporting environment **on the outside** of the VM is sufficiently compatible between the hosts: in particular, the backing storage for virtualised block storage, and the virtual networking infrastructure need to match.

- the stored state can be sent over network
- and resumed on a different host
- as long as the virtual environment is same
- this is known as paused migration

12.1.21 **Live Migration** Live migration is an improvement over paused migration above in that it does not cause noticeable lag and does not endanger TCP or other stateful connections that use timeouts to detect broken connections.
The main idea that enables live migration is that the VM can continue to run as normal while its memory is being copied, with the provision that any subsequent writes must be tracked by the hypervisor: this is achieved through the standard 'copy-on-write' trick, where pages are marked read-only right before they are copied, and the hypervisor traps faults. As appropriate, it allows the write to proceed, but also marks the page as dirty. When the initial sweep is finished, another pass is made but this time only through dirty pages, marking them as clean.
Finally:

- the VM is paused
- registers and last few pages are sent
- the VM is **resumed** at the remote end
- usually within a **few milliseconds**

When the number of dirty pages is sufficiently small at the end of an iteration, the VM is paused, the remaining dirty pages and the CPU context are copied over and the VM is immediately resumed. Since the last transfer is only a few hundred kilobytes, the switchover latency is almost negligible.

- uses asynchronous memory snapshots
- host copies pages and marks them read-only
- the snapshot is sent as it is constructed
- changed pages are sent at the end

12.1.22 **Memory Ballooning** One final consideration is that the hypervisor allocates memory to the guest VMs on demand, but normally, operating systems don't have a concept of 'deallocating' physical memory that they are not actively using. In these circumstances, if the VM sees a spike in memory use, this memory will be indefinitely locked by that VM, even though it has no use for it.
A commonly employed solution is a so-called 'memory ballooning driver' which runs on the guest side and returns unmapped 'physical' (from the point of view of the guest) memory to the host operating system. The memory is unmapped on the host side (i.e. the content of the memory is lost to the guest) and later mapped again if the demand arises.

- how to deallocate "physical" memory?
  - i.e. return it to the hypervisor
- this is often desirable in virtualisation
- needs a special host/guest interface

12.2: Containers

While hardware-accelerated virtualisation is rather efficient when it comes to CPU overhead, there are other costs associated. Some of them can be mitigated by clever tricks (like memory ballooning, TRIM, copy-on-write disk images, etc.) but others are harder to eliminate. When maximal resource utilization is a requirement, containers can often outperform full virtualisation, without significantly compromising other aspects, like maintainability, isolation, or security.

- OS-level virtualisation
  - e.g. virtualised network stack
  - or restricted file system access
- not a complete virtual computer
- turbocharged processes

12.2.1 What are Containers? Containers use virtualisation (in the broad sense of the word) already built into the operating system, mainly based on processes. This is augmented with additional separation, where groups of processes can share, for instance, a network stack which is separate from the network stack available to a different set of processes. While both stacks use the same hardware, they have separate IP addresses, separate routing tables, and so on. Likewise, access to the file system is partitioned (e.g. with chroot), the user mapping is separated, as are process tables.

- virtual machines take a while to boot
- each VM needs its own kernel
  - this adds up if you need many VMs
- easier to share memory efficiently
- easier to cut down the OS image

12.2.2 Why Containers There are two main selling points of containers:

1. so-called 'provisioning speed' – the time it takes from 'I want a fresh system' to having one booted,
2. more efficient resource use.

Both are in large part enabled by sharing a kernel between the containers: in the first case, there is no need to initialize (boot) a new kernel, which saves non-negligible amount of time. For the second point, this is even more important: within a single kernel, containers can share files (e.g. through common mounts) and processes across containers can still share memory – especially executable images and shared libraries that are backed by common files. Achieving the same effect with virtual machines is quite impossible.
One reason for reduced resource use is kernel sharing:

- multiple containers share a **single kernel**
- but not user tables, process tables, …
- the kernel must explicitly support this
- another level of **isolation**

Of course, since a single kernel serves multiple containers, the kernel in question must support an additional isolation level (on top of processes and users), where separate containers have also separate process tables and so on.
Levels of isolation: thread, process, user, container, virtual computer.

- a light virtual machine takes a second or two
- a container can take under 50ms
- but VMs can be suspended and resumed
- but dormant VMs take up a lot more space

Even discounting issues like preparation of disk images, on boot time alone, a container can be 20 times faster than a conventional virtual machine (discounting exokernels and similar tiny operating systems).

- the origin of container systems
- not very sophisticated or secure
- but allows multiple OS images under 1 kernel
- everything else is shared

12.2.3 Boot Time **12.2.4 chroot** The chroot system call can be (ab)used to run multiple OS images (the user-space parts thereof, to be more specific) under a single kernel. However, since everything besides the file system is fully shared, we cannot really speak about containers yet.

- process tables, network, etc. are shared
- the superuser must also be shared
- containers have their **own view** of the filesystem
- including **system libraries** and **utilities**

Since the process tables, networking and other important services are shared across the images, there is a lot of interference. For instance, it is impossible to run two independent web servers from two different chroot pseudo-containers, since only one can bind to the (shared) port 80 (or 443 if you are feeling modern).
Another implication is that the role of the super-user in the container is not contained: the root on the inside can easily become root on the outside.

12.2.5 BSD Jails The jail mechanism on FreeBSD is an evolution of chroot that adds what is missing: separation users, process tables and network stacks. The jail also limits what the 'inside' root can do (and prevents them from gaining privileges outside the jail). It is one of the oldest open-source containerisation solutions.

- an evolution of the chroot container
- adds user and process table separation
- and a virtualised network stack
  - each jail can get its own IP address
- root in the jail has limited power

12.2.6 Namespaces An approach similar to BSD jails was done on the Linux kernel a year later, but was not accepted into the official version of the kernel and was long distributed as a set of third-party patches.

- VServer: like BSD jails but on Linux
  - FreeBSD jail 2000, VServer 2001
- not part of the mainline kernel
- jailed root user is partially isolated

The solution that was eventually added to official Linux kernels is based around **namespaces** which handle each aspect of containerisation separately: when a new process is created (with a fork-like system call, called clone), the parent can specify which aspects are to be shared with the parent, and which are to be separated.

- visibility compartments in the Linux kernel
- virtualizes common OS resources
  - the filesystem hierarchy (including mounts)
  - process tables
  - networking (IP address)

12.2.7 cgroups The other important component in Linux containers are 'control groups' which limit resource usage of a process sub-tree (which can coincide with the process sub-tree that belongs to a single container). This allows containers to be isolated not only with respect to their access to OS-level objects, but also with respect to resource consumption.

- controls HW resource allocation in Linux
- a CPU group is a fair scheduling unit
- a memory group sets limits on memory use
- mostly orthogonal to namespaces

12.2.8 LXC LXC is a suite of user-space tools for management of containers based on Linux namespaces and control groups. Since version 1.0 (circa 2014), LXC also offers separation of the in-container super user, and also unprivileged containers which can be created and managed by regular users (limitations apply).

- mainline Linux way to do containers
- based on namespaces and cgroups
- relative newcomer (2008, 7 years after vserver)
- feature set similar to VServer, OpenVZ &c.

12.2.9 User-Mode Kernels Ports of kernels 'to themselves' so to speak: a regime where the kernel runs as an ordinary user-space process on top of a different configuration the same kernel, are somewhere between containers and full virtual machines. They rely quite heavily on paravirtualisation techniques, although in a rather unusual fashion: since the kernel is a standard process, it can directly access the POSIX API of the host operating system, for instance directly sharing the host file system.
On Linux, the user-mode linux is known as User-Mode Linux and was integrated into Linux 2.6 in 2003. A similar approach was adopted by DragonFlyBSD a few years later, under the name 'virtual kernels'. In this case, the user-mode port of kernel even uses the standard libc, just like any other program. Unfortunately, no direct access to the host file system is possible, making this approach closer to standard VMs.
Comparison of user-mode kernels with containers:

- easier to retrofit securely
  - uses existing security mechanisms
  - for the host, mostly a standard process
- the kernel needs to be ported though
  - analogous to a new hardware platform

When it comes to implementation effort, user-mode kernels are simpler than containers, and offer better host-side security, since they appear as regular processes, without special status.

- inbetween a container and a virtual machine
- an early fully paravirtualised system
- a kernel runs as a process of another kernel
- first production-capable versions around 2003

12.2.10 Migration One major drawback of both containers and user-mode kernels is lack of support for suspend and resume, and hence for migration. In both cases, this comes down to the much more complex state of a process, as opposed to a virtual machine, though the issue is considerably more serious for containers (the user-mode kernel is often just a single process on the host, whereas processes in containers are, in fact, real host-side processes).

- not widely supported, unlike in hypervisors
- process state is much harder to serialise
  - file descriptors, network connections &c.
- mitigated by fast shutdown/boot time