



Počítačové sítě a operační systémy

Synchronizace procesů Uvážnutí

Jaromír Plhák
xplhak@fi.muni.cz

Paralelní běh procesů (1)

- Synchronizace běhu procesů
 - Jeden proces čeká na událost z druhého procesu
- Komunikace mezi procesy
 - Komunikace – způsob synchronizace, koordinace různých aktivit
 - Může dojít k uváznutí
 - Každý proces čeká na zprávu od nějakého jiného procesu
 - Může dojít ke stárnutí
 - Dva procesy si opakovaně posílají zprávy zatímco třetí proces čeká na zprávu nekonečně dlouho

Paralelní běh procesů (2)

- Sdílení prostředků
 - Procesy používají a modifikují sdílená data
 - Operace zápisu musí být vzájemně výlučné
 - Operace zápisu musí být vzájemně výlučné s operacemi čtení
 - Operace čtení mohou být realizovány souběžně
 - Pro zabezpečení integrity dat se používají kritické sekce

Soupeření souběžných aktivit

- Souběžné procesy se ucházejí o zdroje – procesor, FAP, globálně dostupné periferie, soubory dat, ...
- Zdroje soupeřícím procesům typicky přiděluje OS
- OS efektivně izoluje soupeřící procesy, aby se chybně neovlivňovaly
- Soupeřící procesy se vzájemně neznají, soupeřící proces si není vědom existence ostatních soupeřících procesů
- Realizace procesů musí být deterministická, reprodukovatelná, procesy musí být rušitelné a restartovatelné bez bočních efektů

Kooperace souběžných aktivit

- Kooperující procesy sdílí jistou množinu zdrojů, vzájemně se znají
- Kooperace se dosahuje buďto implicitním sdílením zdrojů nebo explicitní komunikací kooperujících procesů
- Proč vlákna/procesy kooperují
 - Aby mohly sdílet jisté zdroje
 - Aby se mohly nezávislé akce řešit souběžně, např. čtení příštího bloku dat během zpracovávání již přečteného bloku dat
- Vlákna jednoho procesu obvykle kooperují, nesoupeří
- Procesy mohou jak kooperovat, tak i soupeřit

Problém konzistence

- Paralelní přístup ke sdíleným údajům může být příčinou nekonzistence dat
- Udržování konzistence dat vyžaduje používání mechanismů, které zajistí patřičné provádění spolupracujících procesů
- Problém komunikace procesů v úloze typu Producent-Konzument přes vyrovnávací paměť s omezenou kapacitou

Problém konzistence – příklad (1)

- Souběžný přístup ke sdíleným údajům se musí mnohdy provádět neatomickými operacemi
- Příklad neatomické operace nad sdílenými proměnnými

```
void echo()  
{  
    chin = getchar();  
    chout = chin;  
    putchar( chout) ;  
}
```

- Procesy P_1 a P_2 provádějí tutéž proceduru *echo* a operují se sdílenými proměnnými *chin*, *chout*
- Oba procesy lze přerušit v kterémkoliv místě
- O rychlosti postupu každého z procesů nelze nic předpovědět

Problém konzistence – příklad (2)

- Příklad možného průběhu procesu P_1 a P_2

| P_1 | P_2 |
|-------------------|-------------------|
| ... | ... |
| chin = getchar(); | ... |
| ... | chin = getchar(); |
| chout = chin; | ... |
| ... | chout = chin; |
| putchar(chout); | ... |
| ... | putchar(chout); |
| ... | ... |

- Znak načtený v P_1 se ztrácí dříve než je zobrazený
- Znak načtený v P_2 se vypisuje v P_1 i P_2



Producent-konzument (1)

- Sdílená data

```
#define BUFFER_SIZE 10  
typedef struct {  
    ...  
} item;  
item buffer[BUFFER_SIZE];  
int in = 0;  
int out = 0;  
int counter = 0;
```



Producent-konzument (2)

- Producent

```
item nextProduced;
```

```
while (1) {  
    while (counter == BUFFER_SIZE)  
        ; /* do nothing */  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```



Producent-konzument (3)

- Konzument

```
item nextConsumed;
```

```
while (1) {  
    while (counter == 0)  
        ; /* do nothing */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
}
```

Producent-konzument (4)

- Atomická operace je taková operace, která vždy proběhne bez přerušení
- Následující příkazy musí být atomické
 - counter++;
 - counter--;
- count++ v assembleru může vypadat
 - register₁ = counter
 - register₁ = register₁ + 1
 - counter = register₁
- count-- v assembleru může vypadat
 - register₂ = counter
 - register₂ = register₂ - 1
 - counter = register₂

Producent-konzument (5)

- Protože takto implementované operace `count++` a `count--` nejsou atomické, můžeme se dostat do problémů s konzistencí
- Necht' je hodnota counter nastavena na 5.
 - Producer – $\text{register}_1 = \text{counter}$ ($\text{register}_1 = 5$)
 - Producer – $\text{register}_1 = \text{register}_1 + 1$ ($\text{register}_1 = 6$)
 - Consumer – $\text{register}_2 = \text{counter}$ ($\text{register}_2 = 5$)
 - Consumer – $\text{register}_2 = \text{register}_2 - 1$ ($\text{register}_2 = 4$)
 - Producer – $\text{counter} = \text{register}_1$ ($\text{counter} = 6$)
 - Consumer – $\text{counter} = \text{register}_1$ ($\text{counter} = 4$)
- Výsledná hodnota proměnné counter bude 4 (nebo 6), zatímco správný výsledek má být 5.

Čtenáři a písaři (1)

- Operace zápisu do sdíleného zdroje musí být exklusivní, vzájemně vyloučené s jakoukoli jinou operací
- Operace čtení mohou čtený zdroj sdílet
- Libovolný počet procesů-čtenářů může číst jeden a tentýž zdroj současně
- V jednom okamžiku smí daný zdroj modifikovat pouze jeden proces-písař
- Jestliže proces-písař modifikuje zdroj, nesmí ho současně číst žádný proces-čtenář
- Čtenář není konzument, písař není producent
 - Jde o jinou úlohu

Čtenáři a písaři (2)

- Čtenáři a písaři s prioritou čtenářů
 - První čtenář přistupující ke zdroji zablokuje všechny písaře
 - Poslední z čtenářů končící čtení zdroje uvolní přístup ke zdroji, případně čekajícímu některému písaři
 - Písaři mohou stárnout
- Čtenáři a písaři s prioritou písařů
 - První čtenář přistupující ke zdroji zablokuje všechny písaře
 - První písař žádající o vstup do kritické sekce dalším čtenářům zakáže přístup ke zdroji
 - Čtenáři mohou stárnout

Race condition

- Race condition (podmínka soupeření)
 - Více procesů současně přistupuje ke sdíleným zdrojům a manipulují s nimi
 - Konečnou hodnotu zdroje určuje poslední z procesů, který zdroj po manipulaci opustí
- Ochrana procesů před negativními dopady race condition
 - Je potřeba procesy synchronizovat



Problém kritické sekce (KS)

- N procesů soupeří o právo používat jistá sdílená data
- V každém procesu se nachází segment kódu programu nazývaný kritická sekce, ve kterém proces přistupuje ke sdíleným zdrojům
- Problém
 - Je potřeba zajistit, že v kritické sekci, sdružené s jistým zdrojem, se bude nacházet nejvýše jeden proces

Modelové prostředí

- Pro hledání řešení problému kritické sekce
 - Předpokládá se, že každý proces běží nenulovou rychlostí
 - Nic se nepředpokládá o relativní rychlosti procesu
 - Žádný proces nezůstane v kritické sekci nekonečně dlouho

Podmínky řešení problému KS (1)

- Podmínka **vzájemného vyloučení** (mutual exclusion), podmínka bezpečnosti, „safety“
 - Jestliže proces P_1 provádí svoji kritickou sekci, žádný jiný proces nemůže provádět svoji kritickou sekci sdruženou se stejným zdrojem
- Podmínka **trvalosti postupu** (progress), podmínka živosti, „liveliness“
 - Jestliže žádný proces neprovádí svoji sekci sdruženou s jistým zdrojem a existuje alespoň jeden proces, který si přeje vstoupit do kritické sekce sdružené s tímto zdroje, pak výběr procesu, který do takové kritické sekce vstoupí, se nesmí odkládat nekonečně dlouho

Podmínky řešení problému KS (1)

- Podmínka **konečnosti doby čekání** (bounded waiting), podmínka spravedlivosti, „fairness“
 - Musí existovat horní mez počtu, kolikrát může být povolen vstup do kritické sekce sdružené s jistým zdrojem jiným procesům než procesu, který vydal žádost o vstup do kritické sekce sdružené s tímto zdrojem, po vydání takové žádosti a před tím, než je takový požadavek uspokojen
 - Předpokládáme, že každý proces běží nenulovou rychlostí
 - O relativní rychlosti procesů nic nevíme

Počáteční návrhy řešení

- Máme pouze 2 procesy, P_0 a P_1
- Generická struktura procesu P_i

do {

entry section

critical section

exit section

reminder section

} while (1);

- Procesy mohou za účelem dosažení synchronizace svých akcí sdílet společné proměnné
- Činné čekání na splnění podmínky v „entry section“ – „busy waiting“

Řešení problému KS (1)

- Softwarová řešení
 - Algoritmy, jejichž správnost se nespolehá na žádné další předpoklady
 - S aktivním čekáním „busy waiting“
- Hardwarová řešení
 - Vyžadují speciální instrukce procesoru
 - S aktivním čekáním



Řešení problému KS (2)

- Řešení zprostředkované operačním systémem
 - Potřebné funkce a datové struktury poskytuje OS
 - S pasivním čekáním
 - Podpora v programovacím systému/jazyku
 - Semafory, monitory, zasílání zpráv

Petersonův algoritmus

- Proces P_i
do {
 flag [i] := true;
 turn = j;
 while (flag [j] and turn == j) /* do nothing */;
 critical section
 flag [i] = false;
 remainder section
} **while (1);**
- Jsou splněny všechny první dvě podmínky správnosti řešení problému kritické sekce

Synchronizační hardware (1)

- Speciální instrukce strojového jazyka
 - Test_and_set, exchange / swap, ...
- Stále zachována idea používání „busy waiting“
- Test_and_set
 - Testování a modifikace hodnoty proměnné – atomicky

```
boolean TestAndSet (boolean &target) {  
    boolean rv = target;  
    target = true;  
    return rv;  
}
```

Synchronizační hardware (2)

- Swap

- Atomická výměna dvou proměnných

```
void Swap (boolean &ra, boolean &rb) {  
    boolean temp = a;  
    a = b;  
    b = temp;  
}
```

Využití TestAndSet

- Sdílená data (inicializováno na false)
boolean lock = false;
- Proces P_i
do {
 while (TestAndSet(lock)) ;
 critical section
 lock = false;
 remainder section
}

Využití Swap

- Sdílená data (inicializováno na false)
 boolean lock = false;
 boolean waiting[n];
- Proces P_i

do {
 key = true;
 while (key == true)
 swap(lock, key);
 critical section
 lock = false;
 remainder section
}

Situace bez podpory OS

- Nedostatek softwarového řešení
 - Procesy, které žádají o vstup do svých KS to dělají metodou „busy waiting“
 - Po nezanedbatelnou dobu spotřebovávají čas procesoru
- Speciální instrukce
 - Výhody
 - Vhodné i pro multiprocesory (na rozdíl od prostého maskování/povolení přerušení)
 - Nevýhody
 - Opět „busy waiting“
 - Možnost stárnutí – náhodnost řešení konfliktu
 - Možnost uváznutí v prioritním prostředí (proces v KS nedostává čas CPU)

Semaforey

- Synchronizační nástroj, který lze implementovat i bez „busy waiting“
 - Proces je (operačním systémem) „uspán“ a „probuzen“
 - Tj. řešení na úrovni OS
- Definice
 - Semaphore S : integer
- Lze ho zpřístupnit pouze pomocí dvou atomických operací

wait (S):

```
while  $S \leq 0$  do no-op;  
S--;
```

signal(S):

```
S++;
```

Kritická sekce

- Sdílená data
semaphore mutex; // počátečně mutex = 1
- Proces P_i :
do {
 wait(mutex);
 critical section
 signal(mutex);
 remainder section
} while (1);

Uvážnutí a stárnutí

- Uvážnutí

- Dva nebo více procesů neomezeně dlouho čekají na událost, kterou může generovat pouze jeden z čekajících procesů
- Necht' S a Q jsou dva semaforey inicializované na 1

| | |
|------------|------------|
| P_0 | P_1 |
| wait(S); | wait(Q); |
| wait(Q); | wait(S); |
| ⋮ | ⋮ |
| signal(S); | signal(Q); |
| signal(Q) | signal(S); |

- Stárnutí

- Neomezené blokování, proces nemusí být odstraněný z fronty na semafor nikdy (předbíhání vyššími prioritami, ...)

Problémy se semaforem

- Semaforem jsou mocný nástroj pro dosažení vzájemného vyloučení a koordinaci procesů
- Operace `wait(S)` a `signal(S)` jsou prováděny více procesy a jejich účinek nemusí být vždy explicitně zřejmý
 - Semafor s explicitním ovládáním `wait/signal` je nástroj nízké úrovně
- Chybné použití semaforu v jednom procesu hroutí souhru všech spolupracujících procesů
- Patologické případy použití semaforů

`wait(x)`

...

`wait(x)`

`wait(x)`

...

`signal(y)`

`signal(x)`

...

`signal(x)`

Problém uváznutí (1)

- Existuje množina blokováných procesů, každý proces vlastní nějaký prostředek (zdroj) a čeká na zdroj držený jiným procesem z této množiny
- Příklad 1
 - V systému existují 2 páskové mechaniky
 - Procesy P_1 a P_2 chtějí kopírovat data z pásky na pásku, každý z procesů „vlastní“ jednu mechaniku a požaduje alokaci druhé

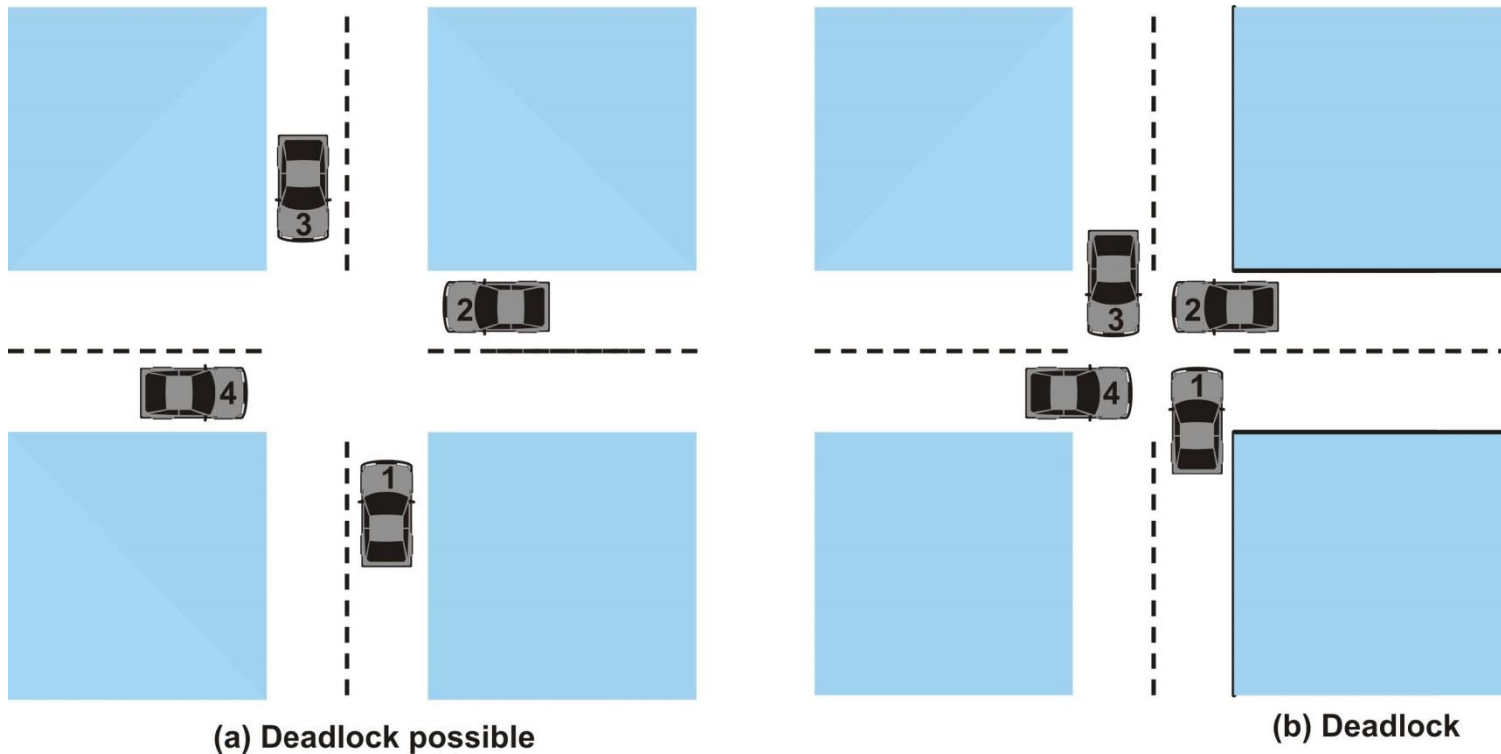
Problém uváznutí (2)

- Příklad 2
 - Semaforey A a B, inicializované na 1

| | |
|-----------|---------|
| P_1 | P_2 |
| wait (A); | wait(B) |
| wait (B); | wait(A) |

Problém uváznutí – soupeření o zdroj (1)

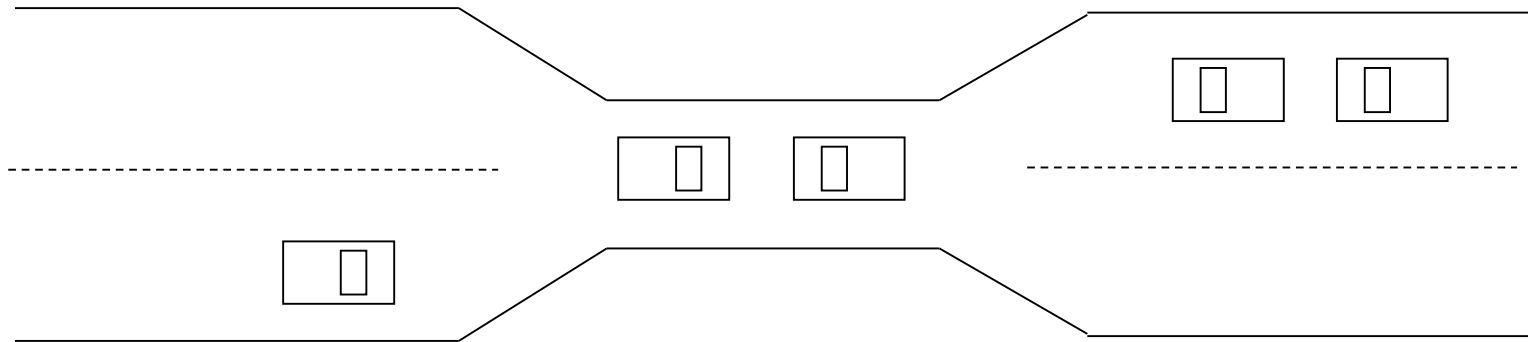
- Auta (procesy) soupeří o výhradní získání prostoru pro přejezd křižovatky (zdroj)



Problém uváznutí – soupeření o zdroj (2)



Příklad – úzký most



- Most s jednosměrným provozem
- Každý vjezd mostu lze chápat jako zdroj
- Dojde-li k uváznutí, lze ho řešit tím, že se jedno auto vrátí
 - Preempce zdroje (přivlastnění si zdroje, který vlastnil někdo jiný) a vrácení soupeře do situace před žádostí o přidělení zdroje (preemption a rollback)
- Při řešení uváznutí se může vrátet i více vozů
- Může docházet ke stárnutí

Definice uváznutí a stárnutí

- Uváznutí
 - Množina procesů P uvázla, jestliže každý proces P_i z P čeká na událost (uvolnění prostředku, zaslání zprávy), kterou vyvolá pouze některý z procesů P
- Stárnutí
 - Požadavky 1 nebo více procesů z P nebudou splněny v konečném čase
 - Z důvodů vyšších priorit jiného procesu
 - Z důvodů prevence uváznutí apod.

Model

- Typy zdrojů R_1, R_2, \dots, R_m
 - Tiskárna, paměť, I/O zařízení, ...
- Každý zdroj R_i má W_i instancí
- Každý proces používá zdroj následujícím způsobem
 - 1. žádost
 - 2. použití
 - 3. uvolnění (v konečném čase)

Charakteristika uváznutí (1)

- K uváznutí dojde, když začnou současně platit 4 následující podmínky
 - Vzájemné vyloučení (mutual exclusion)
 - Sdílený zdroj může v jednom okamžiku používat pouze jeden proces
 - Ponechání si zdroje a čekání na další (hold and wait)
 - Proces vlastníci alespoň zdroj čeká na získání dalšího zdroje, dosud vlastněného jiným procesem

Charakteristika uváznutí (2)

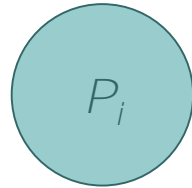
- Bez předbívání (no preemption)
 - Zdroj lze uvolnit pouze procesem, který ho vlastní, dobrovolně poté, co daný proces zdroj dále nepotřebuje
- Kruhové čekání (circular wait)
 - Existuje takový seznam čekajících procesů (P_0, P_1, \dots, P_n), že P_0 čeká na uvolnění zdroje držného P_1 , P_1 čeká na uvolnění zdroje držného P_2 , ..., P_{n-1} čeká na uvolnění zdroje držného P_n , a P_n čeká na uvolnění zdroje držného P_0

Graf přidělení zdrojů (1)

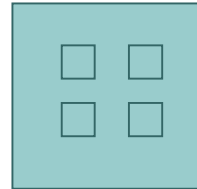
- Resource-Allocation Graph, RAG
- Množina uzlů V a množina hran E
- Uzly jsou dvou typů
 - $P = \{P_1, P_2, \dots, P_n\}$, množina procesů existujících v systému
 - $R = \{R_1, R_2, \dots, R_m\}$, množina zdrojů existujících v systému
- Hrana požadavku – orientovaná hrana $P_i \rightarrow R_j$
- Hrana přidělení – orientovaná hrana $R_j \rightarrow P_i$

Graf přidělení zdrojů (2)

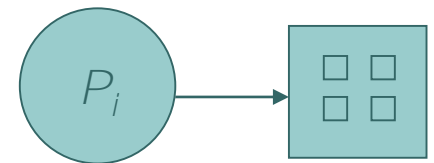
- Proces



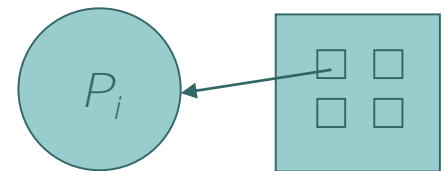
- Zdroj se 4 instancemi



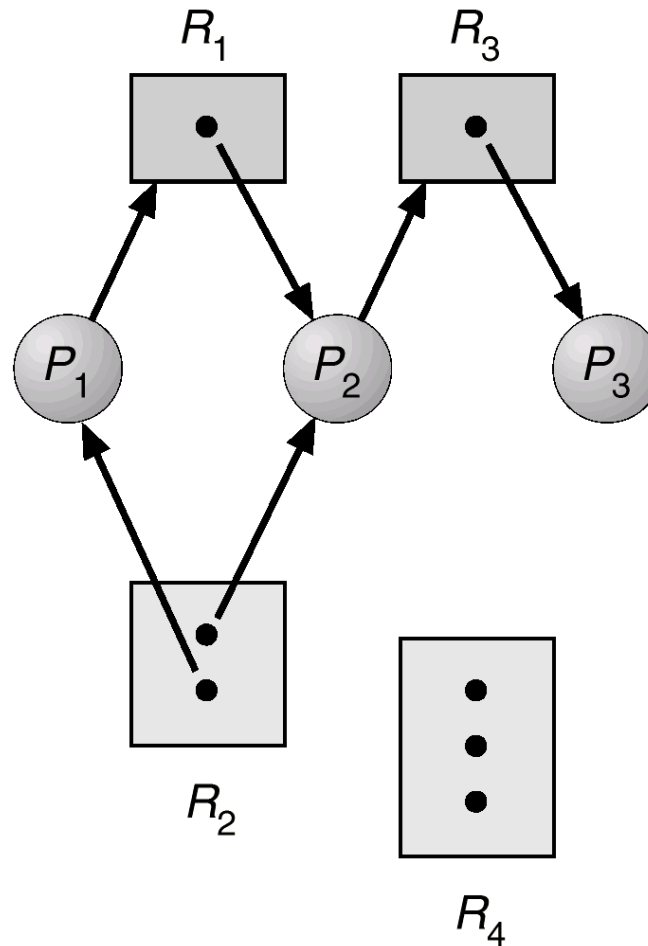
- Proces P_i požadující prostředek R_j



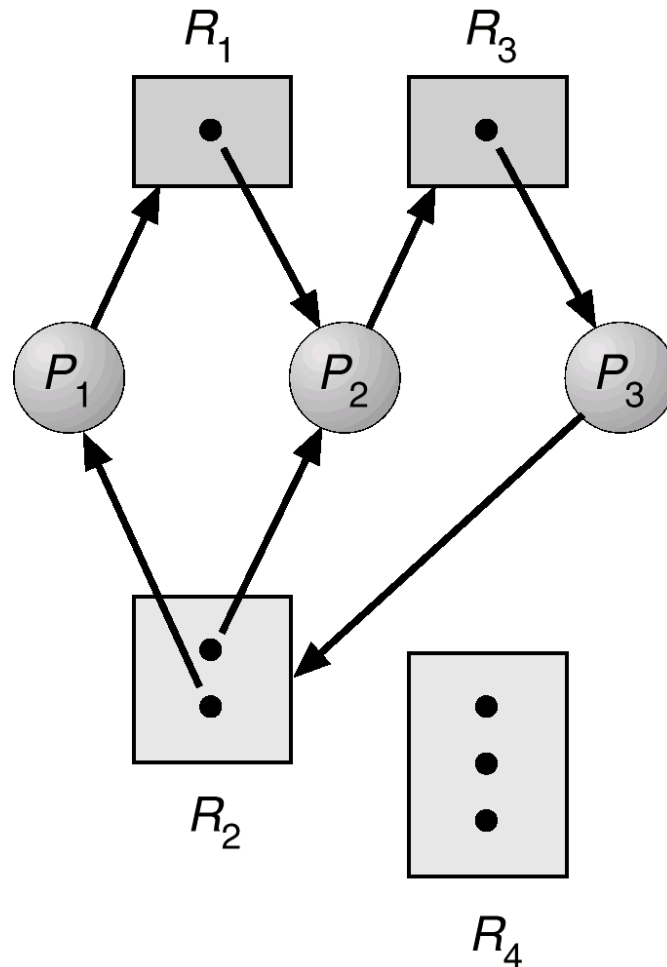
- Proces P_i vlastníci prostředek R_j



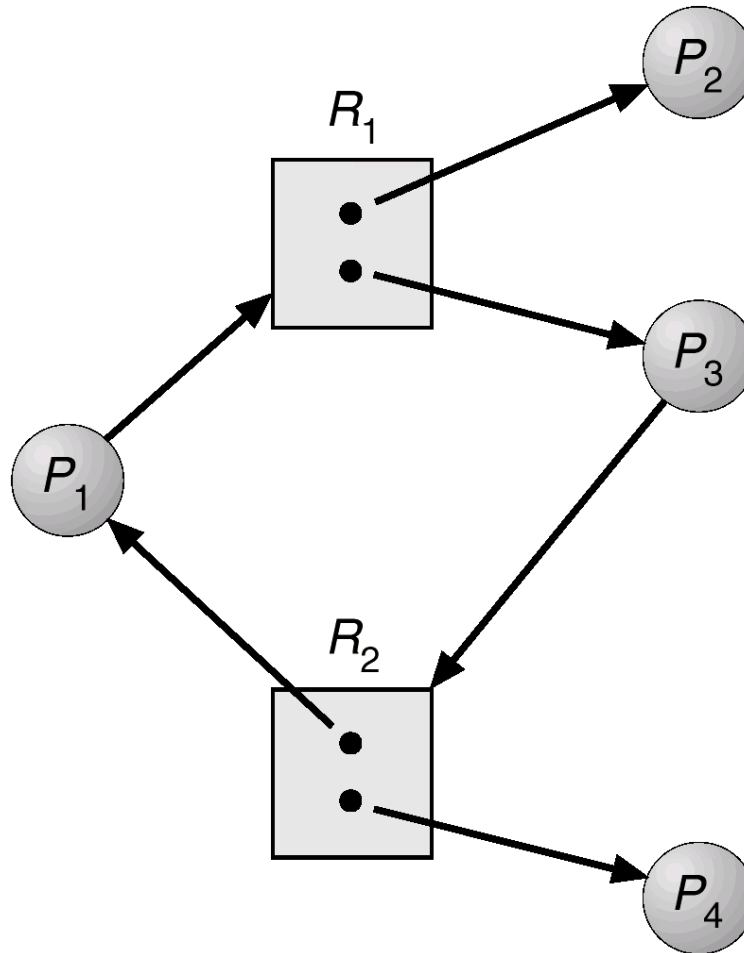
Příklad RAG (bez cyklu)



Příklad RAG (s uváznutím)



Příklad RAG (bez uváznutí)



RAG – závěry

- Jestliže se v RAG nevyskytuje cyklus – k uvážnutí nedošlo
- Jestliže se v RAG vyskytuje cyklus
 - Existuje pouze jedna instance zdroje daného typu
→ k uvážnutí došlo
 - Existuje více instancí zdroje daného typu
→ k uvážnutí může (ale nemusí) dojít

Problém uváznutí

- Ochrana před uváznutím prevencí
 - Zajistíme, že se systém nikdy nedostane do stavu uváznutí
 - Zrušíme platnost některé nutné podmínky
- Obcházení uváznutí
 - Detekce potenciální možnosti vzniku uváznutí a nepřipuštění takového stavu
 - Zamezujeme současné platnosti všech nutných podmínek
 - Prostředek se nepřidělí, pokud by hrozilo uváznutí (hrozí stárnutí)
- Obnova po uváznutí
 - Uváznutí povolíme, ale jeho vznik detekujeme a řešíme
- Ignorování hrozby uváznutí
 - Uváznutí je věc aplikace ne systému
 - Způsob řešení zvolený většinou OS

Ochrana prevencí

- Nepřímé metody
 - Zneplatnění některé nutné podmínky
 - Virtualizací prostředků, ruším nutnost vzájemné výlučnosti při přístupu
 - Požadováním všech prostředků najednou
 - Odebíráním prostředků
- Přímé metody
 - Nepřipuštění platnosti postačující podmínky (cyklus v grafu)
 - Uspořádání pořadí vyžadování prostředků

Prevence uváznutí (1)

- **Vzájemné vyloučení**
 - Podmínka není nutná pro sdílené zdroje
 - U nesdílených zdrojů musí podmínka platit
 - Řeší se např. virtualizací prostředků (např. tiskárny)
- **Ponechání zdrojů a čekání na další**
 - Při žádosti o zdroje proces žádné zdroje „vlastnit“ nesmí
 - Proces musí požádat o zdroje a obdržet je dříve než je spuštěn běh procesu
 - Důsledkem je nízká efektivita využití zdrojů a možnost stárnutí

Prevence uváznutí (2)

- Zakázané předbíhání
 - Jestliže proces držící nějaké zdroje a požadující přidělení dalšího zdroje, nemůže zdroje získat okamžitě, pak se uvolní všechny tímto procesem držené zdroje
 - „Odebrané“ zdroje se zapíší do seznamu zdrojů, na které proces čeká
 - Proces bude obnoven, pouze jakmile může získat jak jím původně držené zdroje, tak jím nově požadované zdroje
- Zabránění kruhovému pořadí
 - Zavedeme úplné uspořádání typů zdrojů a každý proces bude žádat o prostředky v pořadí daném vzrůstajícím pořadí výčtu

Obcházení uváznutí

- Systém musí mít nějaké dodatečné apriorní informace
- Nejjednodušší a nejužitečnější model požaduje, aby každý proces udal maxima počtu prostředků každého typu, které může požadovat
- Algoritmus řešící obcházení uváznutí dynamicky zkouší, zda stav systému přidělování zdrojů zaručuje, že se procesy v žádném případě nedostanou do cyklické fronty čekání
- Stav systému přidělování zdrojů se definuje počtem dostupných a přidělených zdrojů a maximem žádostí procesů



Bankéřův algoritmus (1)

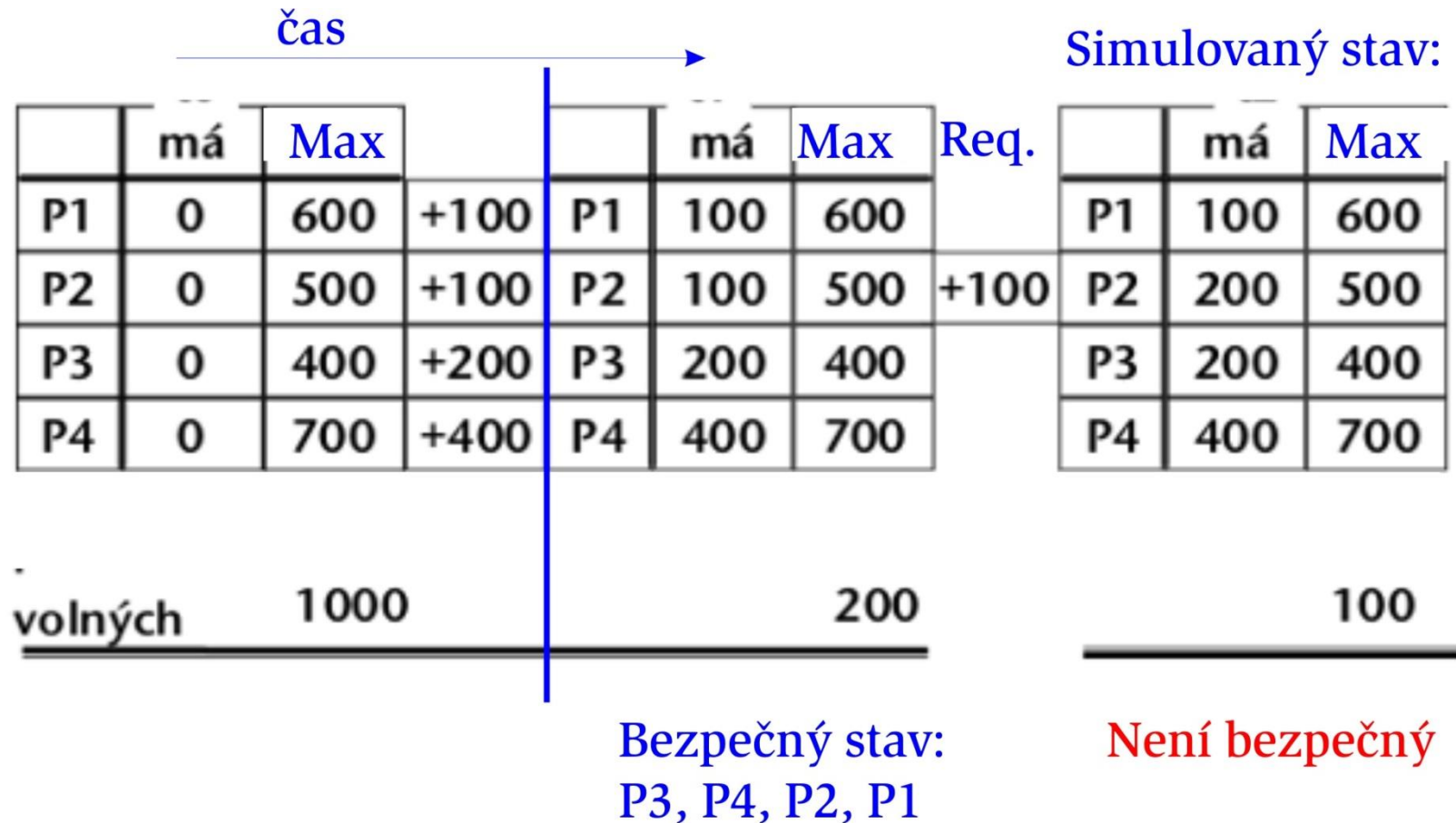
- Princip
 - Bankéř = OS
 - Zákazník = proces
 - Peníze = prostředky
 - Měna = typ prostředku
 - Suma = počet instancí prostředků



Bankéřův algoritmus (2)

- Nový zákazník deklaruje max. částku v každé měně, kterou si chce půjčit
- Pokud zákazník dostal všechny peníze, vrátí je v konečném čase
- Půjčku může zákazník čerpat postupně. Pokud bankéř zjistí, že půjčení právě požadované částky nevede k bezpečnému stavu pokladny, zákazník musí čekat, dokud jiný zákazník nevrátí dostatek peněz
- Stav je bezpečný, pokud bankéř dokáže aplikací pravidel 2. a 3. uspokojit potřeby všech zákazníků v konečném čase

Obcházení – příklad





Detekce uváznutí

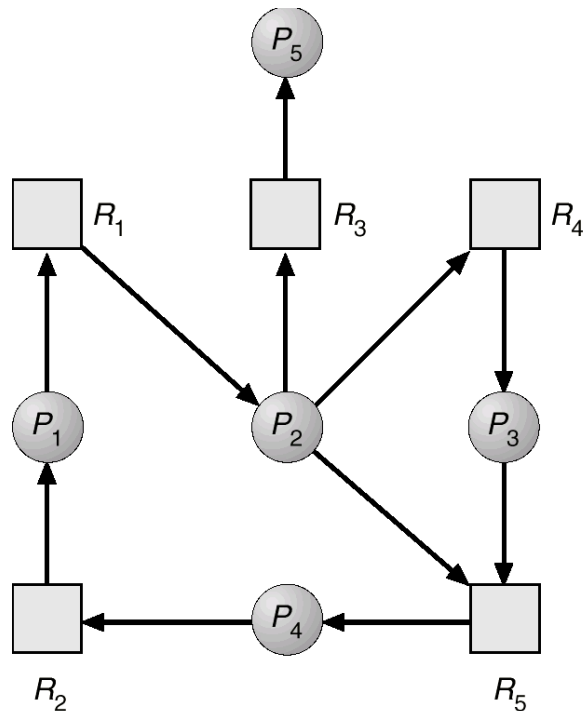
- Umožníme, aby došlo k uváznutí
- Ale toto uváznutí detekujeme
- Aplikujeme plán obnovy

1 instance prostředku každého typu

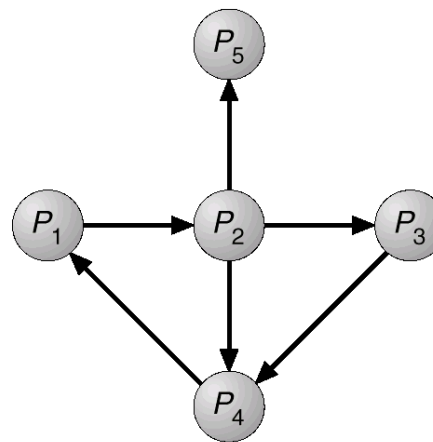
- Udržuje se graf čekání (wait-for graph)
 - Uzly jsou procesy
 - $P_i \rightarrow P_j$ jestliže P_i čeká na P_j
- Periodicky se provádí algoritmus, který v grafu hledá cykly
- Algoritmus pro detekci cyklu v grafu požaduje provedení n^2 operací, kde n je počet uzlů v grafu

Grafy

- (a) Graf přidělení zdrojů
- (b) Odpovídající graf čekání



(a)



(b)

Obnova – ukončení procesu

- Násilné ukončení uváznutých procesů
- Násilně se ukončuje jednotlivě proces po procesu, dokud se neodstraní cyklus
- Čím je dáno pořadí násilného ukončení?
 - Priorita procesu
 - Doba běhu procesu, doba potřebná k ukončení procesu
 - Prostředky, které proces použil
 - Prostředky, které proces potřebuje k ukončení
 - Počet procesů, které bude potřeba ukončit
 - Preference interaktivních nebo dávkových procesů



Obnova – nové rozdělení prostředků

- Výběr oběti – minimalizace ceny
- Návrat zpět (rollback) – návrat do některého bezpečného stavu, proces restartujeme z tohoto stavu
- Stárnutí – některý proces může být vybírán jako oběť trvale
 - Řešení – do cenové funkce zahrneme počet restartů (rollbacků)

Večeřící filozofové (1)

- 5 filozofů buď myslí nebo jí
 - Jí špagety jen 2 vidličkami
 - Co se stane, když se všech 5 filozofů najednou chopí např. levé vidličky ?
 - ... zemřou hladem
- Hledáme řešení – rituál/protokol zajišťující ochranu před uváznutím a stárnutím

Večeřící filozofové (2)

- Zrušení symetrie
 - Jeden filozof je levák, ostatní jsou praváci
 - Levák se liší pořadím získávání vidliček
- Strava se podává n filozofům v jídelně s $(n - 1)$ židlemi
 - Vstup do jídelny hlídá obecný semafor iniciálně nastavený na kapacitu $n - 1$
 - Řešení chrání jak před uváznutím, tak i před stárnutím

Večeřící filozofové (3)

- Filozof smí uchopit vidličky pouze když jsou obě (jeho levá i pravá) volné
 - Musí je uchopit uvnitř kritické sekce, pro řešení lze použít monitor
 - Definuje se vektor n podmínkových proměnných (čekání na vidličku)
 - N dle počtu filozofů
 - Definuje se vektor indikující stav vidličky (true = volná)
 - Definují se dvě monitorové procedury pro získání a uvolnění 2 vidliček
 - Uváznutí nehrozí, v monitoru může by pouze jeden filozof