Synchronizace procesů
Uváznutí

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 isoluje 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 ve 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

```
. . .
chin = getchar();
. . .
chout = chin;
. . .
putchar(chout);
. . .
. . .
```

P_2

```
. . .
. . .
chin = getchar();
. . .
chout = chin;
. . .
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
  - $register_1$ = counter
  - $register_1$ = $register_1$ + 1
  - counter = $register_1$
- count-- v assembleru může vypadat
  - $register_2$ = counter
  - $register_2$ = $register_2$ – 1
  - counter = $register_2$

Producent-konzument (5)

- Protože takto implementované operace count++ a count-- nejsou atomické, můžeme se dostat do problémů s konzistencí
- Nechť je hodnota counter nastavena na 5. Může nastat:
  - producer: $register_1$ = counter ($register_1$ = 5)
  - producer: $register_1$ = $register_1$ + 1 (register1 = 6)
  - consumer: $register_2$ = counter (register2 = 5)
  - consumer: $register_2$ = $register_2$ – 1 (register2 = 4)
  - producer: counter = $register_1$ (counter = 6)
  - consumer: counter = $register_1$ (counter = 4)
- Výsledná hodnota proměnné counter bude 4 (nebo 6), zatímco správný výsledek má být 5.

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 hledaní řesení problému kritické sekce
  - Předpokládá se, že každý proces beží 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 nespoléhá 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:
- Proces P_i

```
do {
    while (TestAndSet(lock)) ;
        critical section
    lock = false;
        remainder section
}
```

Využití Swap

- Sdílená data (inicializováno na false):
  boolean lock;
  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)

Semafory

- 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 ≤ 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);
```

Uváznutí a stárnutí

- Uváznutí
  - Dva nebo více procesů neomezeně dlouho čekají na událost, kterou může generovat pouze jeden z čekajících procesů
  - Nechť S a Q jsou dva semafory 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 fonty na semafor nikdy (předbíhání vyššími prioritami, …)

Problémy se semafory

- Semafory 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)	signal(x)
...	...	...
wait(x)	signal(y)	signal(x)

Problém uváznutí (1)

- Existuje množina blokovaných 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 alokací druhé

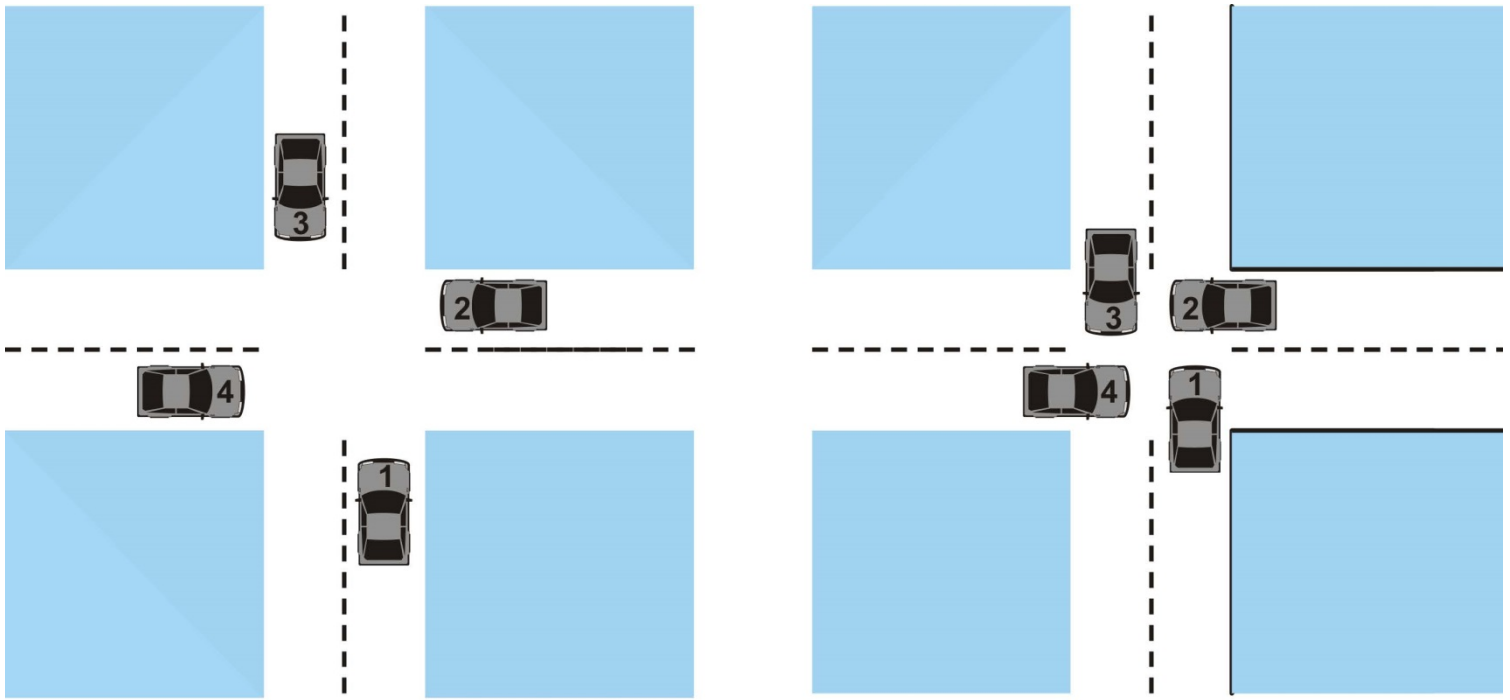
Problém uváznutí (2)

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

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

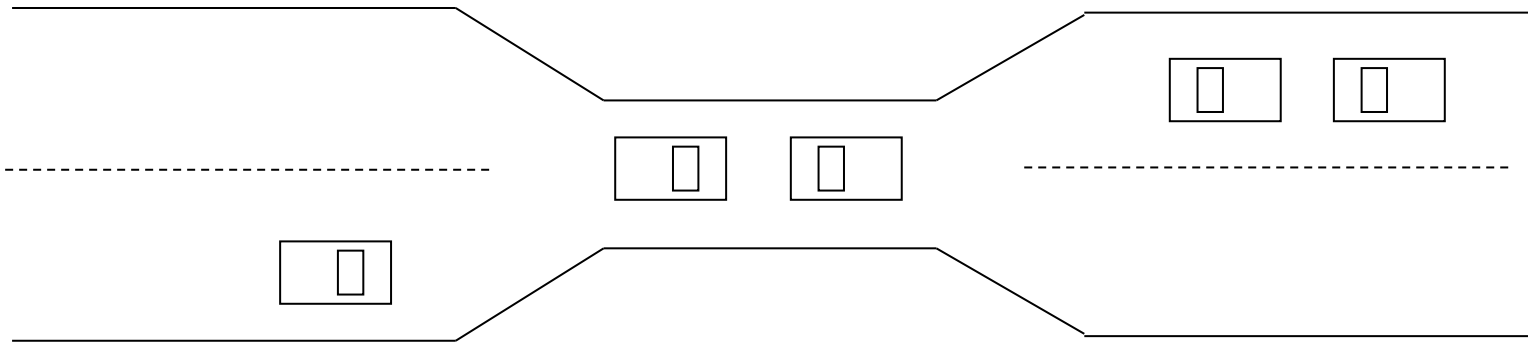
Problém uváznutí - soupeření o zdroj (1)

- 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 vracet 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, …, 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 vlastnící alespoň zdroj čeká na získání dalšího zdroje, dosud vlastněného jiným procesem

Charakteristika uváznutí (2)

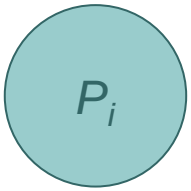
- Bez předbíhání (no preemption)
  - Zdroj lze uvolnit pouze procesem, který ho vlastní, dobrovolně poté, co daný proces zdroj dále nepotřebuje
- Kruhové čekání (circular wait)
  - Existuje takový seznam čekajících procesů (P_0, P_1, …, P_n), že P_0 čeká na uvolnění zdroje drženého P_1, P_1 čeká na uvolnění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

Graf přidělení zdrojů (1)

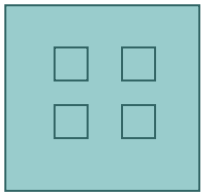
- Resource-Allocation Graph, RAG
- Množina uzlů V a množina hran E
- Uzly jsou dvou typů:
  - P= {P_1, P_2, …, P_n}, množina procesů existujících v systému
  - R = {R_1, R_2, …, R_m}, množina zdrojů existujících v systému
- Hrana požadavku – orientovaná hrana $P_i \rightarrow R_j$
- Hrana přidělení –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

Graf přidělení zdrojů (2)

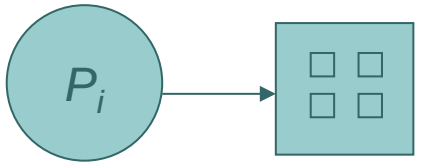
- Proces:



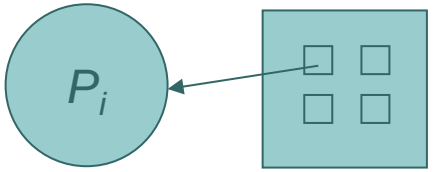
- Zdroj se 4 instancemi:



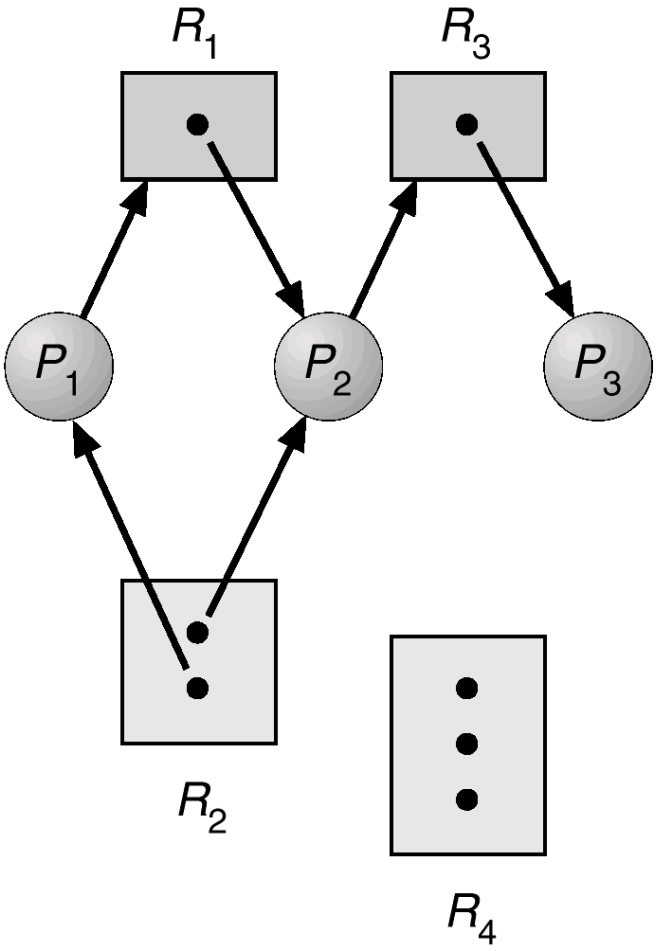
- Proces P_i požadující prostředek R_j:



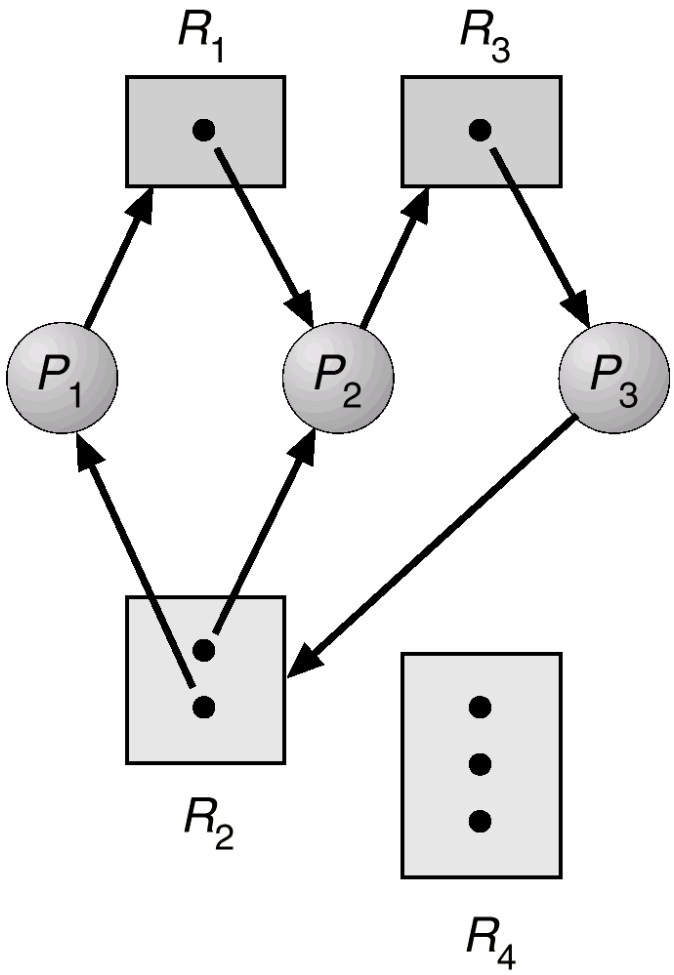
- Proces P_i vlastnící prostředek R_j :



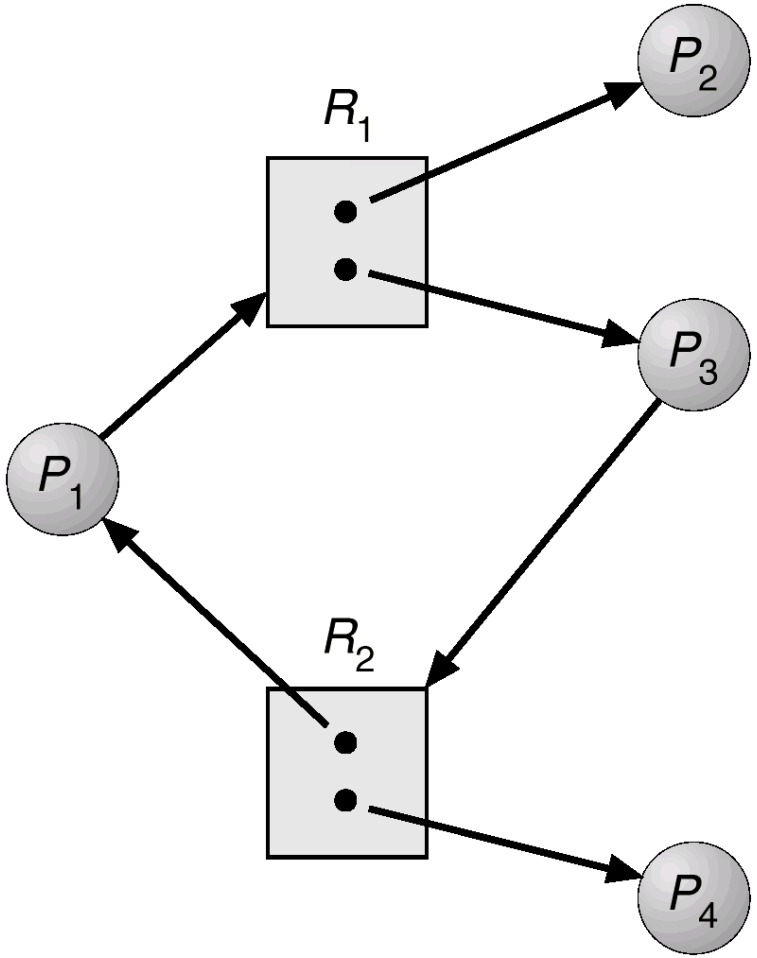
Příklad RAG (bez cyklu)



Příklad RAG (s uváznutím)



Příklad RAG (bez uváznutí)



RAG: závěry

- Jestliže se v RAG nevyskytuje cyklus – k uváznutí nedošlo
- Jestliže se v RAG vyskytuje cyklus
  - Existuje pouze jedna instance zdroje daného typu → k uváznutí došlo
  - Existuje více instancí zdroje daného typu → k uváznutí 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

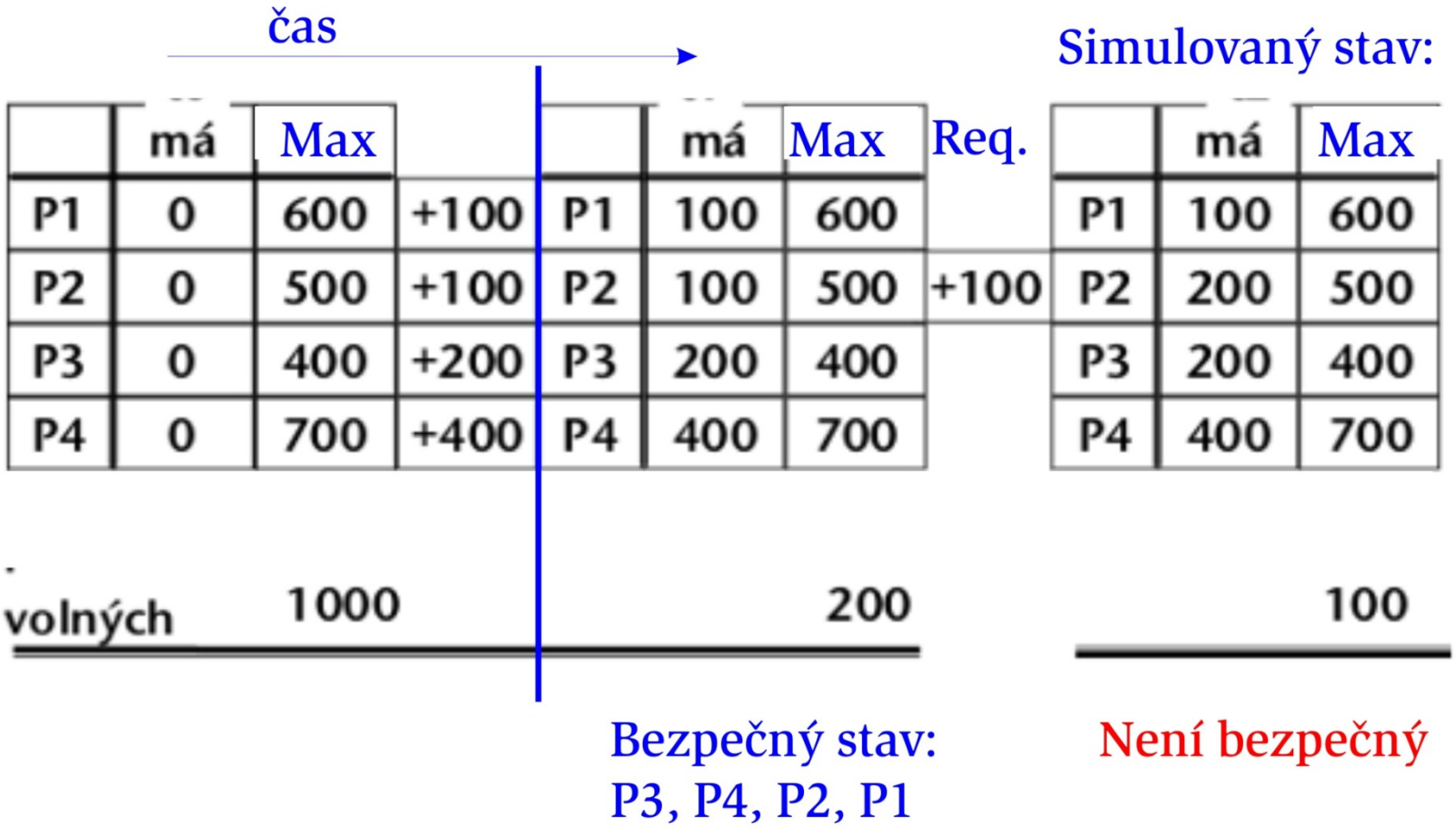
Prevence uváznutí (2)

- 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

Obcházení - příklad



	má	Max			má	Max	Req.		má	Max
P1	0	600	+100	P1	100	600		P1	100	600
P2	0	500	+100	P2	100	500	+100	P2	200	500
P3	0	400	+200	P3	200	400		P3	200	400
P4	0	700	+400	P4	400	700		P4	400	700

Detekce uváznutí

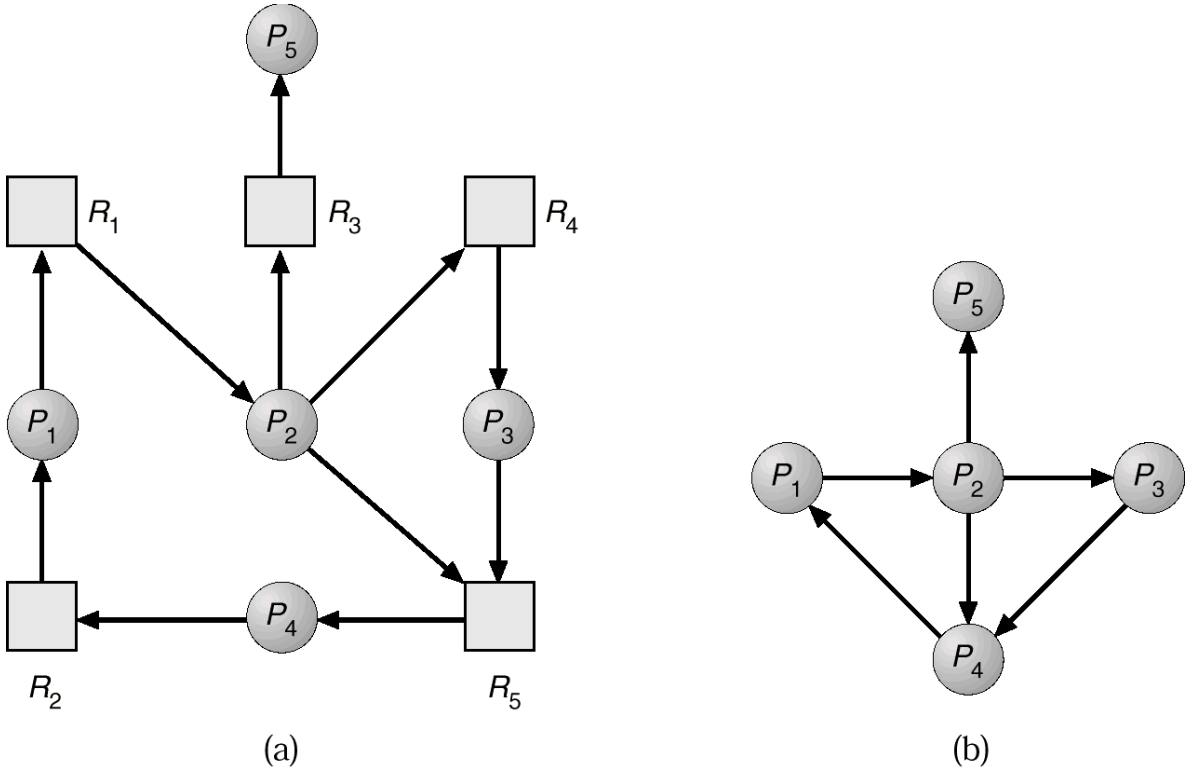
- Umožníme, aby došlo k uváznutí
- Ale toto uváznutí detekujeme
- Aplikujeme plán obnovy

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



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)