PB 169 Počítačové sítě a operační systémy

Synchronizace procesů

Uváznutí

Paralelní běh procesů

- 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
- 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

- 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

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
- Nechť je hodnota **counter** nastavena na 5. Může nastat:
  - producer: **register1 = counter** (*register1 = 5*)
  - producer: **register1 = register1 + 1** (*register1 = 6*)
  - consumer: **register2 = counter** (*register2 = 5*)
  - consumer: **register2 = register2 – 1** (*register2 = 4*)
  - producer: **counter = register1** (*counter = 6*)
  - consumer: **counter = register2** (*counter = 4*)
- Výsledná hodnota proměnné **counter** bude buďto 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

- 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

Podmínky řešení problému kritické sekce

- Podmínka vzájemného vyloučení (mutual exclusion), podmínka bezpečnosti, „safety“
  - jestliže proces P1 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
- 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

- 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 zprostředkované operačním systémem
  - potřebné funkce a datové struktury poskytuje OS
  - s pasivním čekáním
  - podpora v programovacím systému/jazyku
    - semafory, monitory, zasílání zpráv

- Sdílené proměnné
  - **int turn**;
    počátečně **turn = 0**
  - **turn = i** ⇒ P_i může vstoupit do KS
- Proces P_i

```
do {
        while (turn != i) ;
                critical section
        turn = j;
                reminder section
} while (1);
```

- Splňuje vzájemné vyloučení, ale ne trvalost postupu

Algoritmus 2

- Sdílené proměnné
  - **boolean flag[2]**;
    počátečně **flag [0] = flag [1] = false.**
  - **flag [i] = true** ⇒ P_i může vstoupit do své KS
- Process P_i

```
do {
        flag[i] := true;
        while (flag[j]) ;
critical section
        flag [i] = false;
                remainder section
} while (1);
```

- Splňuje vzájemné vyloučení, ale ne trvalost postupu

Algoritmus 3 (Petersonův)

- Kombinuje sdílené proměnné algoritmů 1 a 2
- Proces P_i

```
do {
        flag [i]:= true;
        turn = j;
        while (flag [j] and turn == j) ;
                critical section
        flag [i] = false;
                remainder section
} while (1);
```

- Jsou splněny všechny tři podmínky správnosti řešení problému kritické sekce

Synchronizační hardware

- 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;
    }
```

- 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 Pi

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

#

Využití Swap

- Sdílená data (inicializováno na false):

  boolean lock;

  boolean waiting[n];

- Proces Pi

```
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

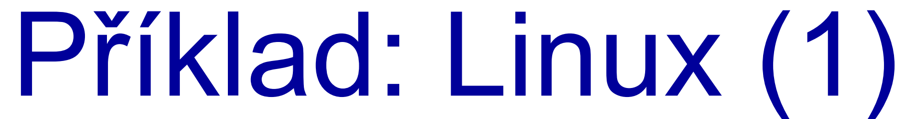
P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

- 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:

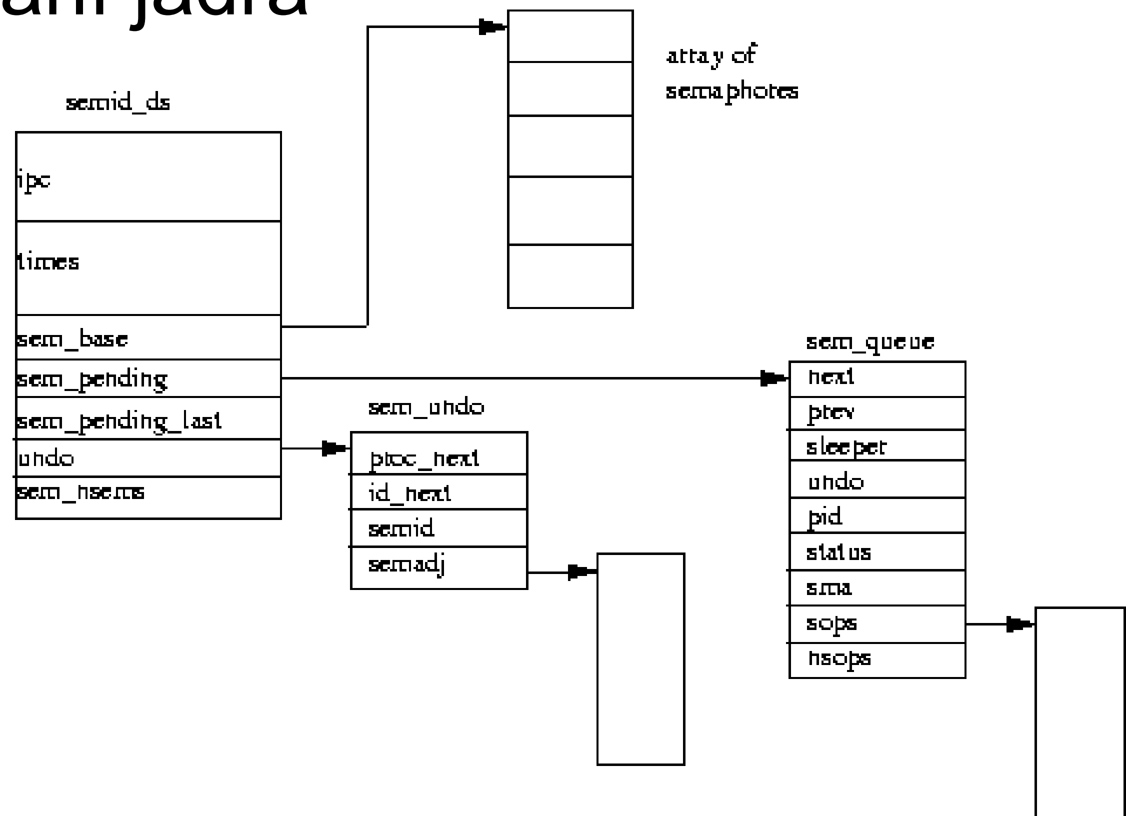
wait(x)	wait(x)
KS	KS
wait(x)	signal(y)



- IPC (InterProcess Communication)
  - signály (asynchronní události)
  - roury (ls|pr|lpr)
  - zprávy (messages)
  - semafory (semaphores)
  - sdílená paměť (shared memory)

Příklad: Linux (2)

- Semafory, volání jádra
  - semget
  - semctl
  - semop

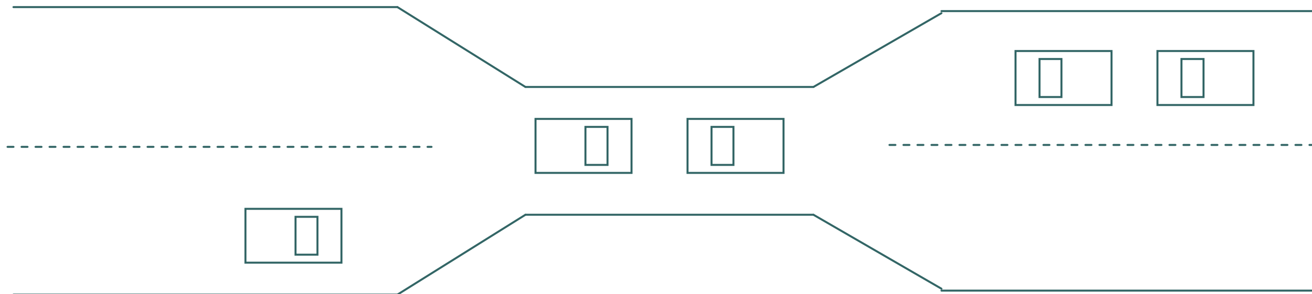


Problém uváznutí

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

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

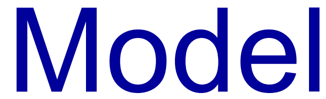
Příklad: úzký most



- 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

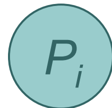
- Typy zdrojů R_1, R_2, …, R_m
  - tiskárna, paměť, I/O zařízení, …
- Každý zdroj R_i má W_i instancí
- Každý proces používá zdroj následujícím způsobem
  1. žádost
  2. použití
  3. uvolnění (v konečném čase)

Charakteristika uváznutí

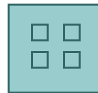
- K uváznutí dojde, když začnou současně platit 4 následující podmínky
  - vzájemné vyloučení (mutual exclusion)
    - sdílený zdroj může v jednom okamžiku používat pouze jeden proces
  - ponechání si zdroje a čekání na další (hold and wait)
    - proces vlastnící alespoň zdroj čeká na získání dalšího zdroje, dosud vlastněného jiným procesem
  - bez předbíhání (no preemption)
    - zdroj lze uvolnit pouze procesem, který ho vlastní, dobrovolně po té, 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

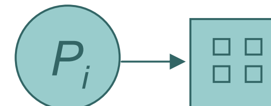
- Resource-Allocation Graph, RAG
- Množina uzlů V a množina hran E
- uzly jsou dvou typů:
  - $P = \{P_1, P_2, \ldots, P_n\}$, množina procesů existujících v systému
  - $R = \{R_1, R_2, \ldots, R_m\}$, množina zdrojů existujících v systému
- Hrana požadavku – orientovaná hrana $P_i \rightarrow R_j$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



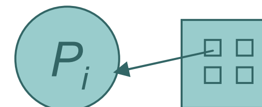
- Zdroj se 4 instancemi:



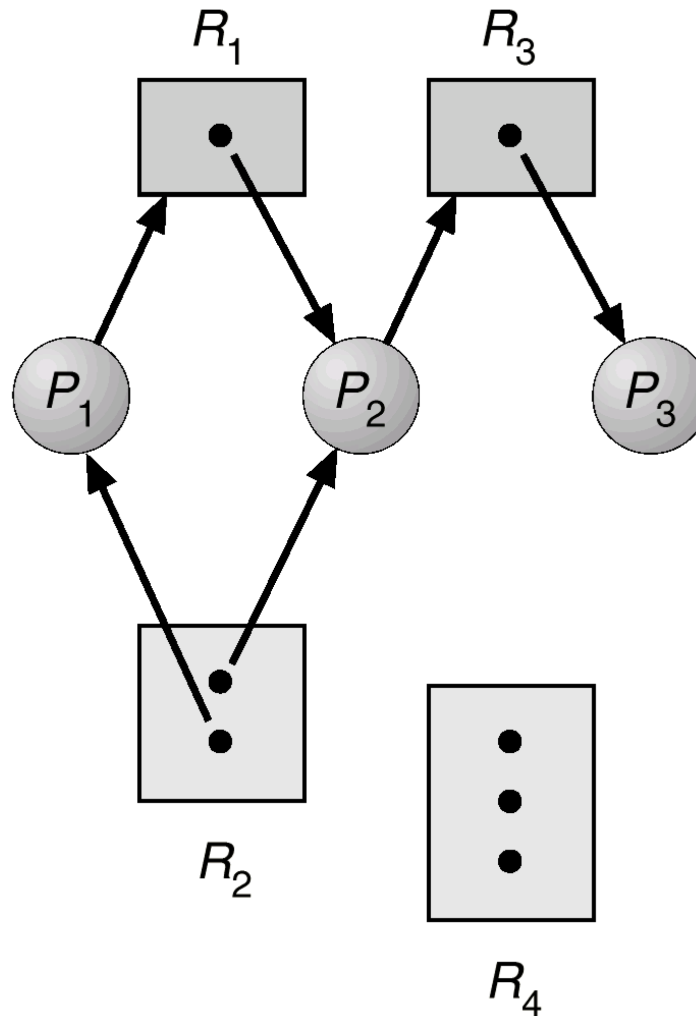
- Proces P_i požadující prostředek R_j:



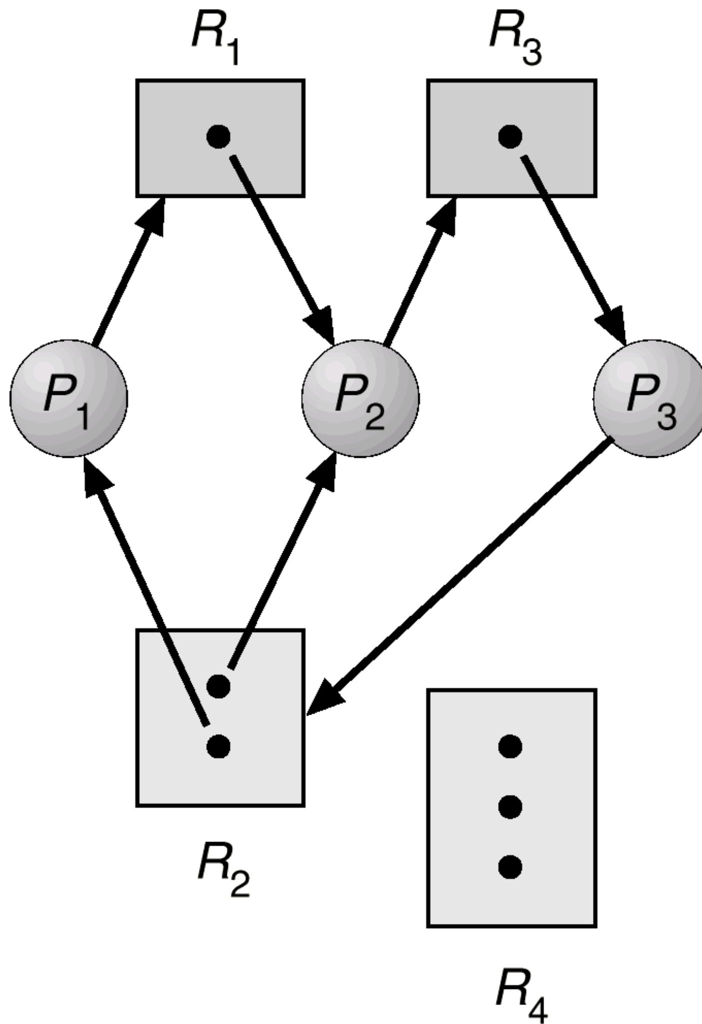
- Proces P_i vlastnící prostředek R_j :



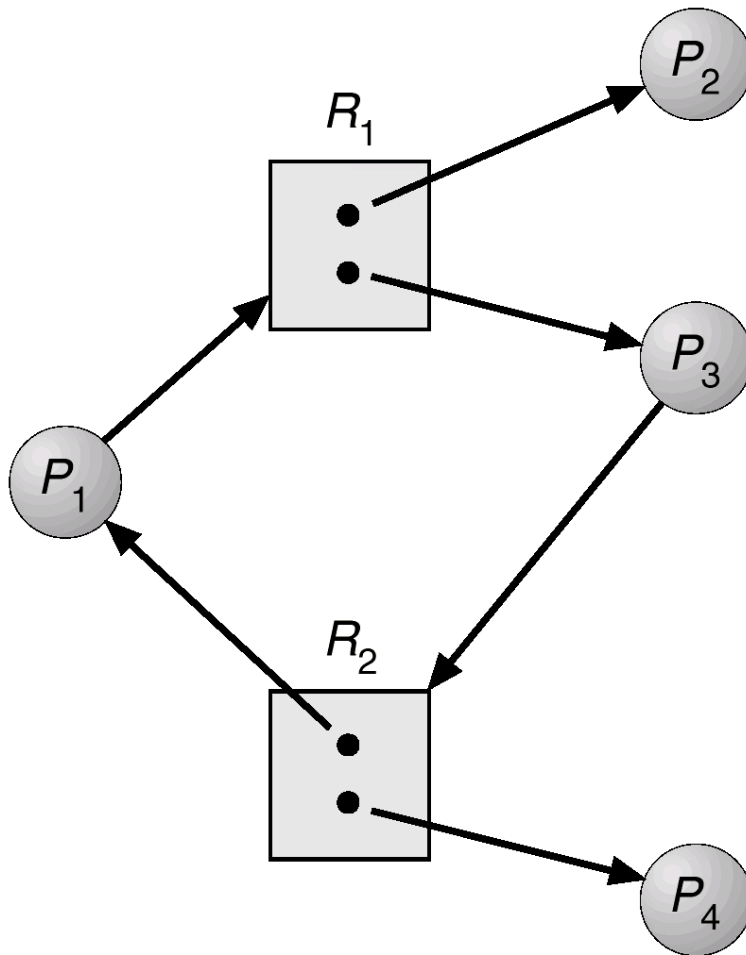
Příklad RAG (bez cyklu)



Příklad RAG (s uváznutím)



Příklad RAG (bez uváznutí)



- 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

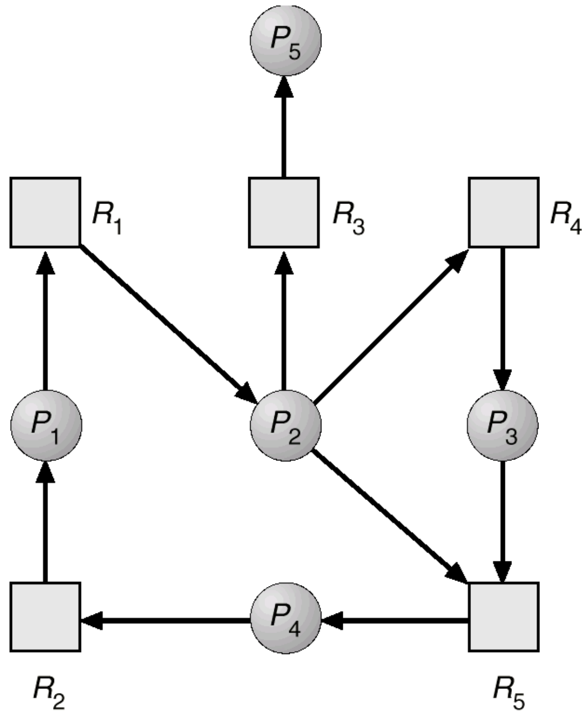
- Umožníme, aby došlo k uváznutí
- Ale toto uváznutí detekujeme
- Aplikujeme plán obnovy

1 instance prostředku každého typu

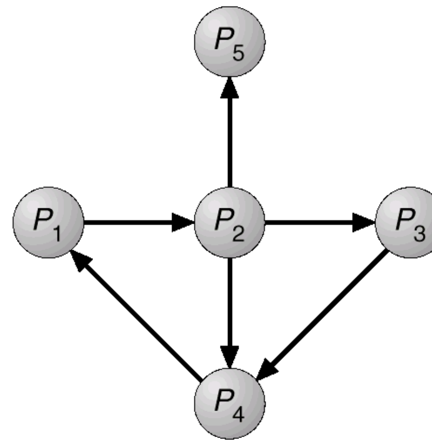
- Udržuje se graf čekání (wait-for graph)
  - uzly jsou procesy
  - $P_i \rightarrow P_j$ jestliže P_i čeká na P_j
- Periodicky se provádí algoritmus, který v grafu hledá cykly
- Algoritmus pro detekci cyklu v grafu požaduje provedení n^2 operací, kde *n* je počet uzlů v grafu

Grafy

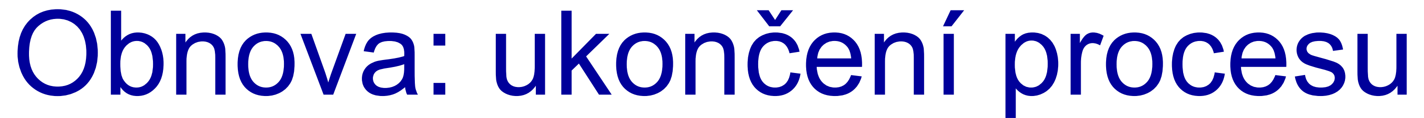
Graf přidělení zdrojů | Odpovídající graf čekání



(a)



(b)



- 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)