Principy počítačů a operačních systémů

Operační systémy
Synchronizace procesů, zablokování

Zimní semestr 2011/2012

Přístup ke sdíleným datům

Terminologie: souběžné vs. paralelní zpracování

Paralelní provádění (parallel execution)

- více činností na různých místech současně
- v daném okamžiku více než 1 aktivní vlákno
- nastává pouze na víceprocesorovém stroji

Souběžné provádění (concurrent execution)

- více činností na různých místech prokládaně
- v daném okamžiku pouze 1 aktivní vlákno
- může nastat i na jednoprocesorovém stroji

Implicitní sdílení dat OS

Datové struktury OS

- souborový systém, síťový stack, správa procesů, ...

Modifikace datových struktur v reakci na události

- synchronní události: systémová volání
  - procesy volají služby operačního systému
- asynchronní události: přerušení
  - OS přijímá data ze sítě, klávesnice, ..., obsluhuje zařízení
- v kódu OS může být více vláken současně

Operační systém musí být reentrantní...

- podpora souběžného zpracování více událostí
- nutná podmínka pro paralelní zpracování

Explicitní sdílení dat procesu

Datové struktury procesů

- seznamy, stromy, grafy, fronty, ...

Souběžný přístup k datovým strukturám

- implicitně v rámci procesu
  - vlákna jednoho procesu "vidí" stejnou paměť
- explicitně mezi procesy
  - procesy komunikující prostřednictvím sdílené paměti

Přístup k datům je nutno synchronizovat...

- nutná podmínka pro korektní výpočet

Časové závislé chyby (race conditions)

Operace nad daty

- před začátkem a po skončení operace musí být data v konzistentním stavu, během provádění operace mohou být data dočasně v nekonzistentním stavu

Pokud operace sestává z více kroků...

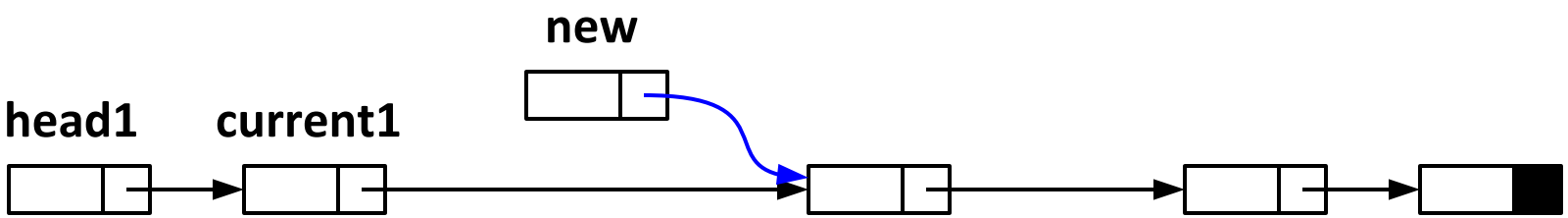
- při souběžném přístupu více vláken může dojít k promíchání kroků z různých operací
  - proces/vlákno přeplánován uprostřed operace
- problém pokud s jednou datovou strukturou pracuje více vláken/procesů
  - k přeplánování může dojít při nekonzistentním stavu dat
  - výsledek operací závisí na pořadí běhu vláken/procesů

Příklad: operace se spojovým seznamem

Operace vložení prvku do seznamu

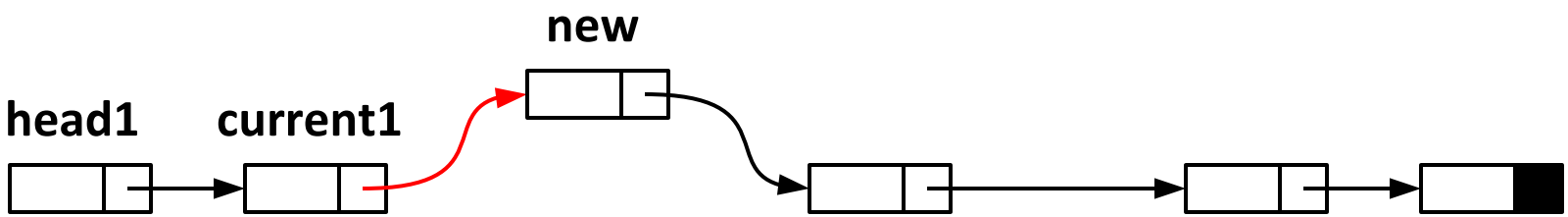
- přidání ***new*** do seznamu ***head1*** za ***current1***
- 1. krok

```
new->next = current->next;
```



- 2. krok

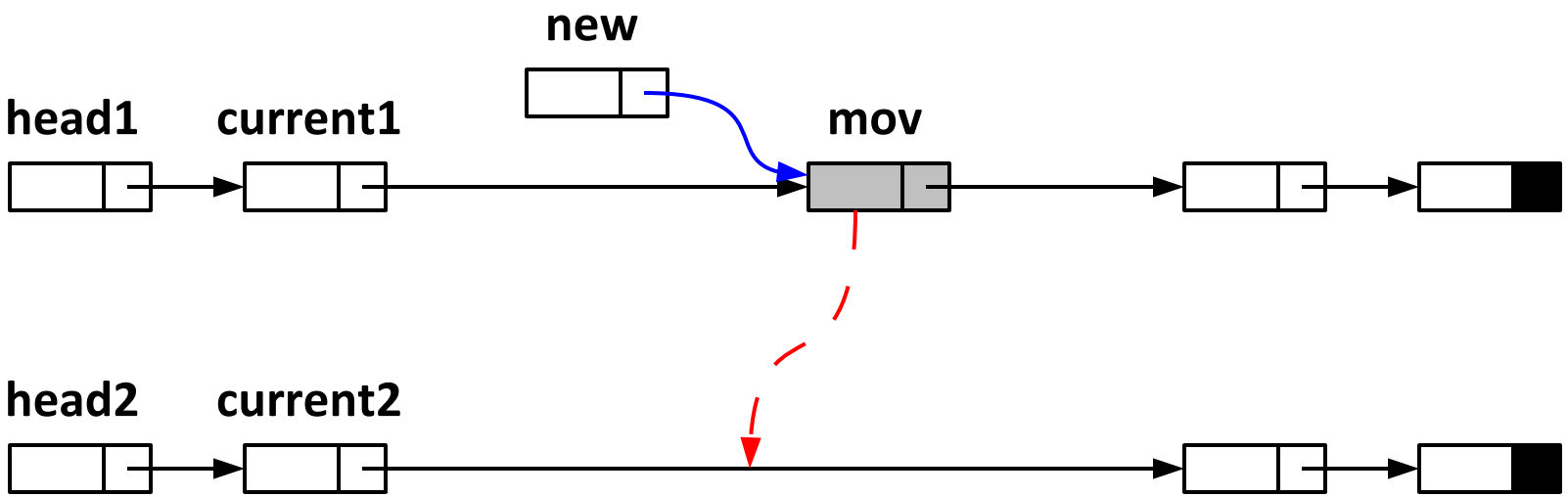
```
current1->next = new;
```



Příklad: operace se spojovým seznamem

1. vlákno
   - vložení ***new*** do ***head1*** za ***current1***
2. vlákno
   - přesun ***mov*** z ***head1*** do ***head2*** za ***current2***

1. vlákno přerušeno pro 1. kroku vkládání
   - naplánováno 2. vlákno realizující přesun



Příklad: operace se spojovým seznamem

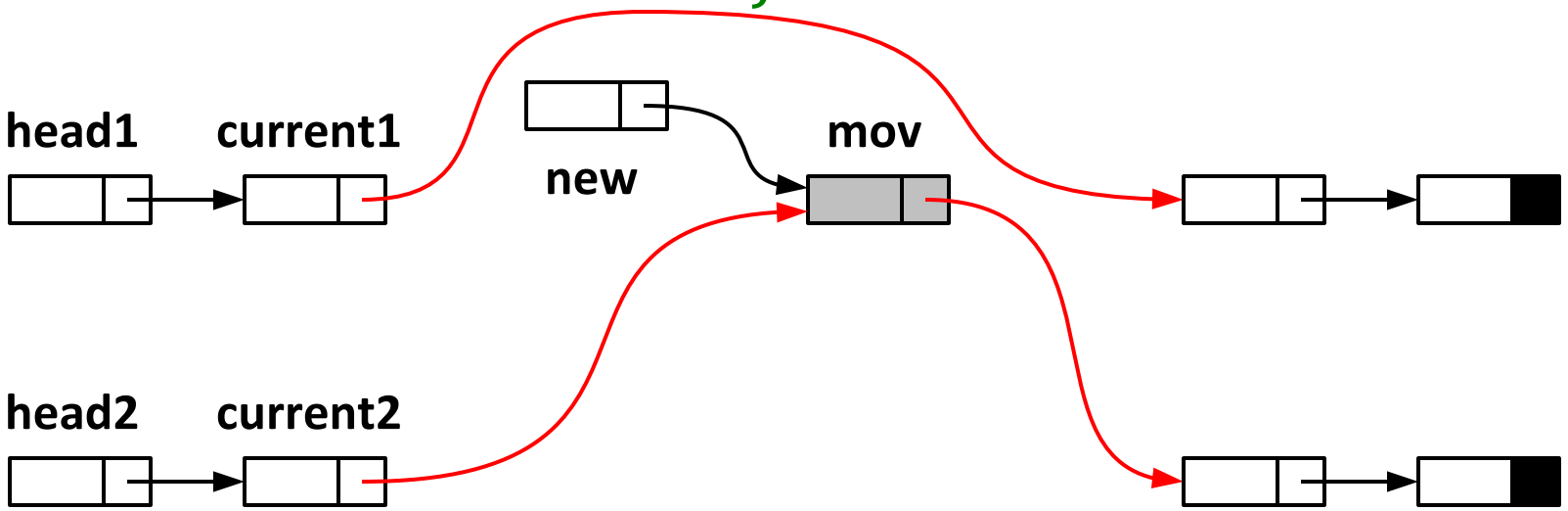
Operace přesunu prvku mezi seznamy

- vyjmutí z 1. seznamu + vložení do 2. seznamu

2. vlákno

- přesun ***mov*** z ***head1*** do ***head2*** za ***current2***

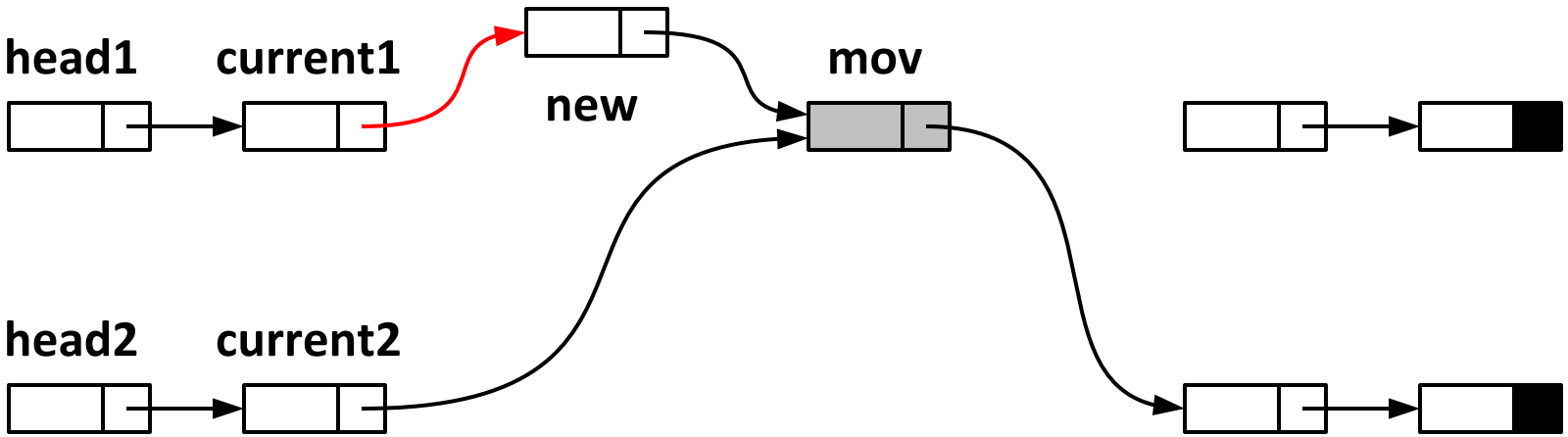
```
current1->next = mov->next;
mov->next = current2->next;
current2->next = mov;
```



Příklad: operace se spojovým seznamem

1. vlákno

- opět naplánováno po 2. vlákně
- dokončí vkládání ***new*** za ***current1***



Řešení problémů s race condition

Atomická změna stavu datové struktury

- bez ohledu na počet kroků nutných k realizaci
- dokud není operace dokončena, není možné začít jinou

Nutno identifikovat ***kritické sekce*** programu

```
begin_critical_section;
new->next = current1->next;
current1->next = new;
end_critical_section;
```

Pouze 1 vlákno smí provádět kód kritické sekce

- systém zajistí ***vzájemné vyloučení (mutual exlusion)***
  - co třeba zakázat přerušení?

Realizace vzájemného vyloučení, 1. pokus

Zkusíme použít sdílenou proměnnou...

```
boolean locked = FALSE;
...
while (locked);
locked = TRUE;
critical section
locked = FALSE;
```

Nefunguje!

- do programu jsme přidali novou race condition
- původní problém zůstal nevyřešen

Realizace vzájemného vyloučení, 2. pokus

Použijeme proměnnou pro každý proces...

Proces 1

```
p1_locked = TRUE;
while (p2_locked);
critical section
p1_locked = FALSE;
```

Proces 2

```
p2_locked = TRUE;
while (p1_locked);
critical section
p2_locked = FALSE;
```

Pořád nefunguje! Ale uz jsme blízko...

- funguje, když procesy vstupují do KS postupně
- co když do KS vstupují oba procesy ***najednou?***

Realizace vzájemného vyloučení, 3. pokus

Použijeme proměnnou pro každý proces...

- a navíc budeme sledovat "kdo je na tahu"

Proces 1

```
p1_locked = TRUE;
turn = P2;
while (p2_locked && turn = P2);
critical section
p1_locked = FALSE;
```

Proces 2

```
p2_locked = TRUE;
turn = P1;
while (p1_locked && turn == P1);
critical section
p2_locked = FALSE;
```

Konečně funguje!

- procesy si dávají přednost, což řeší předchozí deadlock
- Petersonův algoritmus – zobecnění pro N procesů
- v praxi se používají jiná řešení (s podporou HW)

Realizace vzájemného vyloučení s podporou HW

Použijeme sdílenou proměnnou...

- a také speciální instrukci procesoru

```
boolean locked = FALSE;
...
while (test_and_set (locked));
critical section
locked = FALSE;
```

Funguje na poprvé...

- funkci **test_and_set** odpovídá instrukce procesoru
  - přečte proměnnou, nastaví ji na **TRUE** a vrátí **původní hodnotu**
  - operace je atomická, čtení a zápis jsou neoddělitelné
- spin-lock: proměnná + operace lock/unlock

Problém spin-locků

Aktivní čekání (busy waiting)

- při zamčeném zámku procesor nedělá nic užitečného
  - obzvláště markantní na jednoprocesorovém systému, kde zámek nemůže nikdo jiný odemknout

Nešlo by to jinak?

- ***pasivní čekání***
  - pokud je zamčeno, vlákno se uspí (ready → blocked)
  - procesor může dělat něco jiného (užitečného)
  - ten kdo odemyká zámek vzbudí uspané vlákno
- uspání/vzbuzení procesu vyžaduje podporu OS
  - změna stavu vlákna, přeplánování

Pokus o realizaci pasivního čekání...

Použijeme spinlock + operace sleep/wakeup...

```
while (test_and_set (locked)) sleep (queue);
critical section
locked = FALSE;
wakeup (queue);
```

... pochopitelně nefunguje!

- test a uspání musí být atomické
  - je potřeba zámek k frontě, který se při sleep() odemkne
- to "umí" zařídit pouze operační systém

OS poskytuje synchronizační primitiva

- datová struktura + operace

Pasivní čekání pomocí zámku (mutexu)

Operace

- ***lock***
  - zamkne zámek pokud je odemčený, jinak čeká (pasivně) na odemčení
- ***unlock***
  - odemkne zámek a vzbudí čekající proces (pokud existuje)

Realizace

- celočíselná proměnná + fronta čekajících procesů
- zámek je odemčený při hodnotě == 0, jinak je zamčený

Typické použití

- implementace vzájemného vyloučení

Pasivní čekání pomocí semaforu

Operace

- ***down*** (původně P)
  - zabere semafor pokud je volný, jinak čeká na uvolnění
- ***up*** (původně V)
  - uvolní semafor, vzbudí čekající proces (pokud existuje)

Realizace

- celočíselná proměnná + fronta čekajících procesů
- semafor je zabraný při hodnotě < 1, jinak je volný
- zabrání semaforu snižuje hodnotu o 1, uvolnění zvyšuje

Typické použití

- reprezentace volných/přidělených prostředků

Příklad: použití semaforu

Problém producent/konzument

- producent dat, konzument dat, sdílený buffer
- pokud je buffer plný, producent musí počkat
- pokud je buffer prázdný, konzument musí počkat

```
#define N 100
semaphore mutex = 1;
semaphore empty = N, full = 0;
```

Producent

```
while (1) {
    // produce item
    down (empty);
    down (mutex);
    // put item in buffer
    up (mutex);
    up (full);
}
```

Konzument

```
while (1) {
    down (full);
    down (mutex);
    // get item from buffer
    up (mutex);
    up (empty);
    // consume item
}
```

Realizace semaforu

Operace *down*

- nutno vyřešit realizaci vnitřní kritické sekce

```
if (--value < 0)
    block_this_process();
```

Kontrola hodnoty + zablokování

- spinlock na strukturu + odemknutí při uspání
  - zavádí aktivní čekaní, kterému jsme se chtěli vyhnout
- zákaz přerušení na procesoru
  - zajistí atomicitu na velmi hrubé úrovni
  - nefunguje na více procesorech

Opět nutná podpora OS...

Pasivní čekání pomocí monitoru

Monitor

- datová struktura + operace pro čtení/změnu stavu
  - navíc zámek + fronta uspaných procesů
  - konstrukt programovacího jazyka
- operace ve stejné instanci se vzájemně vylučují
  - před "vstupem" do monitoru je zámek zamčen
  - po "opuštění" monitoru je zámek odemčen

Operace ***wait***

- zablokuje volající proces uvnitř monitoru a současně uvolní monitor (odemkne zámek) pro jiný proces

Operace ***signal***

- probudí zablokované procesy, ale ***neuvolňuje*** monitor
  - odblokované procesy musí zamknout zámek, aby mohly pokračovat v běhu (uvnitř monitoru)

Další synchronizační primitiva

Read/write zámky

- rozlišení typu přístupu k datové struktuře
  - čtenářů může být více současně, písař pouze jeden

Reentrantní zámky

- vícenásobné zamknutí v jednom vlákně
  - např. rekurzivní volání jedné funkce

Podmínkové proměnné

- fronta, operace wait + signal, parametrem zámek
- podobné monitoru: uspání + odemčení zámku
  - používá se explicitně, podobně jako zámky

Rendez-vous, bariéry

- synchronizace postupu vláken kódem
  - např. uspat vlákno, dokud ostatní vlákna nedojdou do stejného místa

Ekvivalence synchronizačních primitiv

Jakmile máme jedno primitivum...

- můžeme všechna ostatní implementovat pomocí něj...
  - implementace mutexu pomocí semaforu, implementace semaforu pomocí monitoru, ...

Obecně funguje pouze u sdílené paměti

- u distribuovaných systémů se situace komplikuje
  - místo datové struktury se ze semaforu stane server
  - požadavky na operace zasílány pomocí zpráv

Zablokování na sdílených prostředcích

Zablokování na sdílených prostředcích

Prostředky

- výpočetní – nutné k běhu programu
- synchronizační – nutné ke koordinaci konfliktů

Přidělování prostředků

- OS jako centrální správce prostředků, přiděluje právo používat nějaký prostředek (nebo jeho část, pokud je prostředek dělitelný)
- k zablokování může dojít v situaci, kdy procesy žádají současné přidělení více prostředků

Zablokování na sdílených prostředcích

Práce s prostředky

- žádost o prostředek (blokující)
- použití přiděleného prostředku
- odevzdání prostředku (dobrovolné, při skončení)

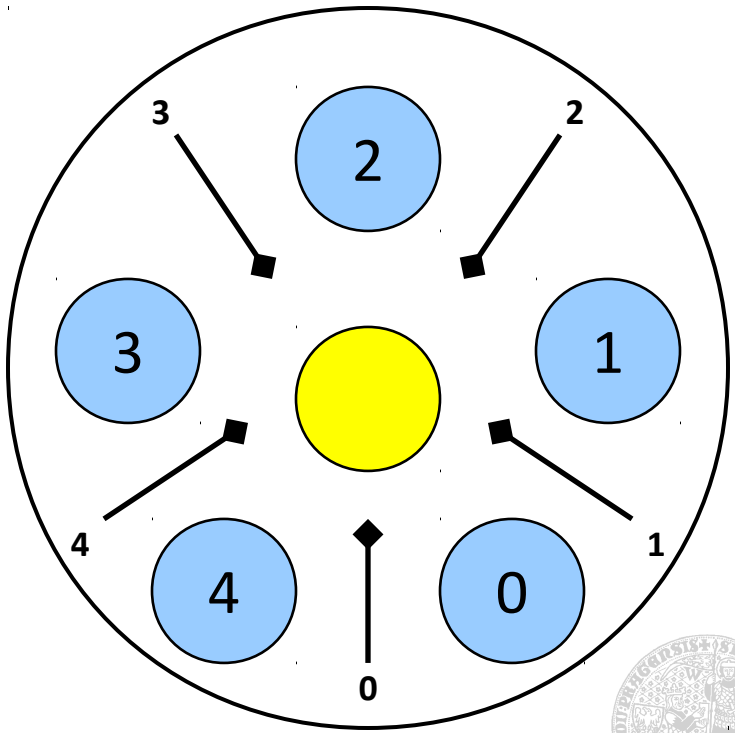
Zablokování (*deadlock*)

- Množina procesů je zablokována, jestliže každý proces z této množiny čeká na událost, kterou může způsobit pouze jiný proces z této množiny.

Příklad: večeřící filozofové

5 filozofů žije pohromadě, čas tráví přemýšlením

- pokud má filozof hlad, jde se do jídelny najíst
  - v jídelně je prostřeno pro 5 osob, na stole mísa špaget
- k jídlu je potřeba použít 2 vidličky...
  - ... na stole je ovšem pouze 5 vidliček!



V kontextu OS...

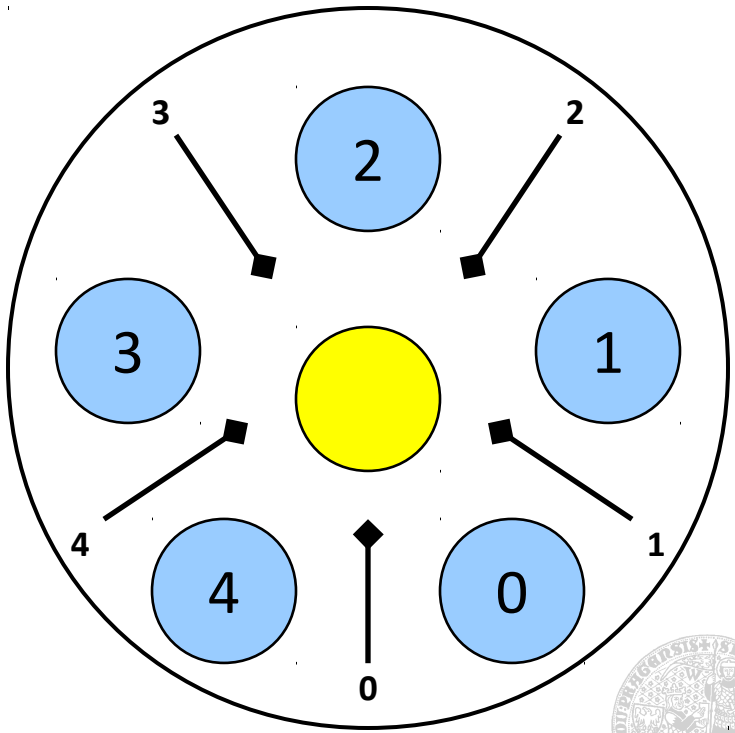
- 5 procesů soupeřících o prostředky
- každý prostředek sdílen pouze dvěma procesy

Příklad: večeřící filozofové

Co se stane, když...

- dostanou hlad všichni filozofové najednou?
- všichni si vezmou nejprve vidličku nalevo?

```
#define N 5
void philosopher (int i) {
  for (;;) {
    think ();
    take_fork (i);
    take_fork ((i + 1) % N);
    eat ();
    put_fork (i);
    put_fork ((i + 1) % N);
  }
}
```



Příklad: večeřící filozofové

Implementace funkce take_fork()

- take_fork() je blokovací
  - všichni najednou zvednou svoji levou a čekají na pravou
- take_fork() je opatrná
  - pokud nemohu vzít druhou vidličku, položím tu první
  - všichni zvednou levou, podívají se doprava, položí levou
  - filosofové "pracují" ale nenají se – livelock + vyhladovění

Možná řešení

- jeden z filozofů vezme vidličky v jiném pořadí
- do jídelny pustíme nejvýše 4 filozofy současně
- randomizace časů

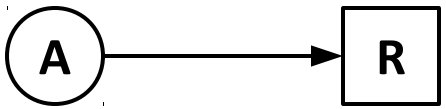
Formální model zablokování

Stav reprezentován orientovaným grafem

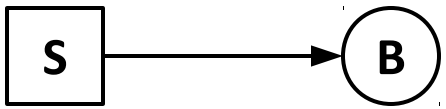
- prostředky

- procesy

- žádost o prostředek



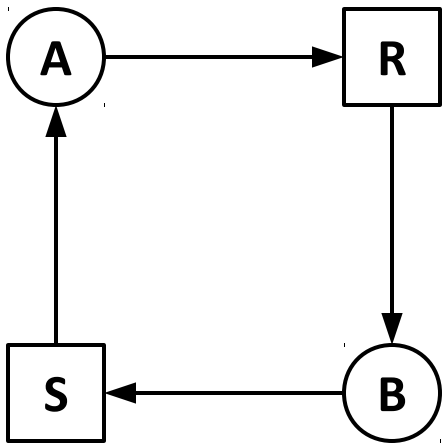
- vlastnění prostředku



Formální model zablokování

Cyklus v grafu indikuje potenciální deadlock

- požadavky vyjadřují skutečné potřeby procesu
- pokud nejsou požadavky uspokojeny, proces je zablokován a čeká na uvolnění prostředku
- pokud prostředek vlastní jiný proces, který je také zablokován, nemůže dojít k uvolnění prostředku
- původní proces zůstane zablokován



Vznik deadlocku

Coffmanovy podmínky

- při splnění všech podmínek dojde k zablokování

- ***Výlučný přístup (Exclusive use)***
  - prostředek je přidělen výhradně jednomu procesu
- ***Neodnímatelnost (No preemption)***
  - přidělené prostředky nemohou být odebrány
- ***Drž a čekej (Hold and wait)***
  - proces může zároveň držet prostředek a čekat na další
- ***Kruhová závislost (Cyclic dependency)***
  - procesy čekají na prostředky v kruhu

Řešení problému zablokování

(Statická) prevence deadlocku

- systém navržen tak, aby některá z Coffmanových podmínek nemohla platit

(Dynamické) vyhýbání se deadlocku

- kontrola žádostí o přidělení prostředků – žádosti, které by způsobily deadlock jsou zamítnuty

Deadlock detection & recovery

- umožňuje vznik deadlocku a řeší problém až při jeho vzniku – zvyšuje propustnost systému

"Pštrosí algoritmus"

- problém typicky vyřeší uživatel (kill -9)

Prevence deadlocku (*deadlock prevention*)

Zneplatnění některé z Coffmanových podmínek

- nelze aplikovat obecně, závisí na typu prostředku

Výlučný přístup (exclusive use)

- spooling – iluze výlučného přístupu

Neodnímatelnost (no preemption)

- odnímatelné prostředky lze odejmout bez následků
  - procesor (přeplánování), paměť (swapping)
- neodnímatelné prostředky nelze odejmout bez nebezpečí selhání výpočtu
  - obecně nevhodné z pohledu programátora

Prevence deadlocku (*deadlock prevention*)

Drž a čekej (hold and wait)

- OS vrátí chybu místo zablokování procesu
- nutno žádat o všechny prostředky najednou
- před žádostí nutno všechny prostředky uvolnit

Kruhová závislost (cyclic dependency)

- očíslování prostředků + možnost žádat pouze o prostředky s vyšším číslem
- pořadí nemusí být globální, ale pouze v rámci množiny prostředků sdílených současně v nějakém kontextu
  - zámky v subsystémech operačního systému

Vyhýbání se deadlocku (*deadlock avoidance*)

Výchozí stav

- počet dostupných a přidělených prostředků
- procesy nejsou zablokovány

Následující stav

- při přidělení dalších prostředků
- přechod pouze pokud je následující stav bezpečný

Bezpečný stav

- existuje pořadí, v jakém uspokojit všechny procesy

Nebezpečný stav

- uvedené pořadí přidělování prostředků neexistuje

Deadlock avoidance: bankéřův algoritmus

Základní koncept

- systém má k dispozici prostředky různých typů
- není možné uspokojit všechny požadavky najednou
- předpokládá se, že požadavky budou přicházet postupně, a že procesy budou prostředky vracet

Dodatečné informace

- max. počet jednotlivých typů prostředků, o které bude každý proces žádat

Bezpečný stav systému

- je možné plně uspokojit alespoň 1 proces
- takový proces časem prostředky vrátí

Příklad: bankéřův algoritmus

	Má	Max
A	0	6
B	0	5
C	0	4
D	0	7

Volné: 10

bezpečný

. .

	Má	Max
A	1	6
B	1	5
C	2	4
D	4	7

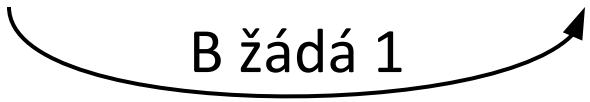
Volné: 2

bezpečný

	Má	Max
A	1	6
B	2	5
C	2	4
D	4	7

Volné: 1

nebezpečný



Detekce a zotavení z deadlocku (*detection & recovery*)

Problémy bankéřova algoritmu

- složité rozhodování o přidělení prostředků
  - algoritmus má navíc složitost $O(N^2)$
- požadované informace jsou typicky nedostupné
- efektivnější zpravidla bývá řešit až vzniklé problémy
  - typicky používané v databázových systémech
  - vyžaduje detekci a schopnost zotavení

Detekce deadlocku (*deadlock detection*)

- model závislostí mezi procesy ve formě grafu
- test na přítomnost kruhových závislostí
  - hledání cyklu v orientovaném grafu

Zotavení z deadlocku (*deadlock recovery*)

Odebrání prostředku

- na přechodnou dobu, pod dohledem operátora

Odstranění nepohodlných procesů

- proces z cyklu závislostí
- proces mimo cyklus vlastnící identický prostředek

Checkpointing/rollback

- OS ukládá stav procesů
- restart procesu v předchozím stavu

Transakční zpracování

- typické pro databázové systémy