Téma 5 – Synchronizace procesů a problém uváznutí

Obsah

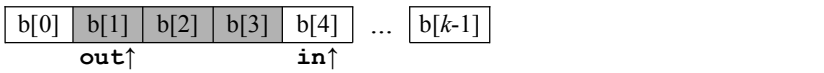
1. Problém soupeření, kritické sekce
2. Vzájemné vyloučení
3. Semafory
4. Klasické synchronizační úlohy
5. Problém uváznutí a časově závislých chyb
6. Graf přidělování zdrojů
7. Přístupy k řešení problému uváznutí
8. Algoritmy řešení problému uváznutí
9. Detekce uváznutí a možnosti obnovy po uváznutí

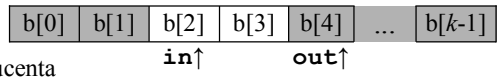
Podstata problému

- Souběžný přístup ke sdíleným datům může způsobit jejich nekonzistenci
  - nutná koordinace procesů
- Synchronizace běhu procesů
  - Čekání na událost vyvolanou jiným procesem
- Komunikace mezi procesy (IPC = *Inter-process Communication*)
  - Výměna informací (zpráv)
  - Způsob synchronizace, koordinace různých aktivit
- Sdílení prostředků – problém soupeření či souběhu (*race condition*)
  - Procesy používají a modifikují sdílená data
  - Operace zápisu musí být vzájemně výlučné
  - Operace zápisu musí být vzájemně výlučné s operacemi čtení
  - Operace čtení (bez modifikace) mohou být realizovány souběžně
  - Pro zabezpečení integrity dat se používají kritické sekce

Úloha Producent-Konzument

- Ilustrační příklad
  - Producent generuje data do vyrovnávací paměti s konečnou kapacitou (*bounded-buffer problem*) a konzument z této paměti data odebírá
    - V podstatě jde o implementaci komunikačního kanálu typu „roura“
  - Zavedeme celočíselnou proměnnou **`count`**, která bude čítat položky v poli. Na počátku je **`count = 0`**
  - Pokud je v poli místo, producent vloží položku do pole a inkrementuje **`count`** (modulo velikost_vyrovnávací_paměti)
  - Pokud je v poli nějaká položka, konzument při jejím vyjmutí dekrementuje **`count`** (modulo velikost_vyrovnávací_paměti)





`in` ... privátní proměnná producenta
`out` ... privátní proměnná konzumenta

Kód „Producent-Konzument“

Sdílená data:

```
#define BUF_SZ = 20
typedef struct { … } item;
item buffer[BUF_SZ];
int count = 0;
```

Producent:

```
void producer() {
  int in = 0;
  item nextProduced;
  while (1) {  /*nekonečná smyčka*/
  /*     Vygeneruj novou položku do
         proměnné nextProduced */
         while (count == BUF_SZ);
              /* nedělej nic */
         buffer[in] = nextProduced;
         in = (in + 1) % BUF_SZ;
         count = count + 1;
  }
}
```

Privátní proměnné

Konzument:

```
void consumer() {
  int out = 0;
  item nextConsumed;
  while (1) {  /*nekonečná smyčka*/
     while (count == 0) ;
          /* nedělej nic */
     nextConsumed =  buffer[out];
     out = (out + 1) % BUF_SZ;
     count = count - 1;
     /*  Zpracuj položku z
          proměnné nextConsumed */
  }
}
```

- Je to korektní řešení?

Problém soupeření (*race condition*)

- count = count + 1 bude obvykle implementováno takto:
  P_1: $registr_0 \leftarrow count$
  P_2: $registr_0 \leftarrow registr_0 + 1$
  P_3: $count \leftarrow registr_0$
- count = count – 1 bude zřejmě implementováno jako:
  K_1: $registr_1 \leftarrow count$
  K_2: $registr_1 \leftarrow registr_1 - 1$
  K_3: $count \leftarrow registr_1$
- Vlivem Murphyho zákonů, **může** nastat následující posloupnost prokládání producenta a konzumenta (nechť na počátku count = 3)

Interval	Běží	Akce	Výsledek
P_1	producent	$registr_1 \leftarrow count$	$registr_1 = 3$
P_2	producent	$registr_1 \leftarrow registr_1 + 1$	$registr_1 = 4$
K_1	konzument	$registr_2 \leftarrow count$	$registr_2 = 3$
K_2	konzument	$registr_2 \leftarrow registr_2 - 1$	$registr_2 = 2$
P_3	producent	$count \leftarrow registr_1$	$count = 4$
K_3	konzument	$count \leftarrow registr_2$	$count = 2$

- Na konci může být count == 2 nebo 4, ale programátor zřejmě chtěl mít 3 (ale i to se může podařit)
- Vše je důsledkem **nepředvídatelného** prokládání procesů vlivem možné preempce

Kritická sekce

- Problém lze formulovat obecně:
  - Jistý čas se proces zabývá svými „obvyklými“ činnostmi a jistou část své aktivity věnuje sdíleným prostředkům.
  - Část kódu programu, kde se přistupuje ke sdílenému prostředku, se nazývá kritická sekce procesu vzhledem k tomuto sdílenému prostředku (nebo také sdružená s tímto prostředkem).
- Je potřeba zajistit, aby v kritické sekci sdružené s jistým prostředkem, se nacházel nejvýše jeden proces
  - Pokud se nám podaří zajistit, aby žádné dva procesy nebyly současně ve svých kritických sekcích sdružených s uvažovaným sdíleným prostředkem, pak je problém soupeření vyřešen.
- Modelové prostředí pro řešení problému kritické sekce
  - Předpokládá se, že každý z procesů **běží** nenulovou rychlostí
  - Řešení **nesmí** záviset na relativních rychlostech procesů

Požadavky na řešení problému kritických sekcí

1. Vzájemné vyloučení – podmínka bezpečnosti (*Mutual Exclusion*)
   - Pokud proces P_i je ve své kritické sekci, pak žádný jiný proces nesmí být ve své kritické sekci sdružené s týmž prostředkem
2. Trvalost postupu – podmínka živosti (*Progress*)
   - Jestliže žádný proces neprovádí svoji kritickou sekci sdruženou s jistým prostředkem a existuje alespoň jeden proces, který si přeje vstoupit do kritické sekce sdružené s tímto prostředkem, pak výběr procesu, který do takové kritické sekce vstoupí, se nesmí odkládat
3. Konečné čekání – podmínka spravedlivosti (*Fairness*)
   - Procesu musí být umožněno vstoupit do kritické sekce v konečném čase
   - Musí existovat omezení počtu, kolikrát smí být povolen vstup do kritické sekce jiným procesům než procesu požadujícímu vstup v době mezi vydáním žádosti a jejím uspokojením

Možnosti řešení problému kritických sekcí

- Základní struktura procesu s kritickou sekcí

```
do {
      enter_cs();
          critical section
      leave_cs ();
          non-critical section
  } while (TRUE);
```

Korektní implementace enter_cs() a leave_cs() je klíčem k řešení celého problému kritických sekcí.

- Čistě softwarové řešení na aplikační úrovni
  - Algoritmy, jejichž správnost se nespoléhá na další podporu
  - Základní (a problematické) řešení s aktivním čekáním (*busy waiting*)
- Hardwarové řešení
  - Pomocí speciálních instrukcí CPU
  - Stále ještě s aktivním čekáním
- Softwarové řešení zprostředkované operačním systémem
  - Potřebné služby a struktury poskytuje JOS (např. semafory)
  - Tím je umožněno pasivní čekání – proces nesoutěží o procesor
  - Podpora volání synchronizačních služeb v programovacích systémech/jazycích (např. monitory, zasílání zpráv)

Řešení na aplikační úrovni (1)

- Vzájemné vyloučení s aktivním čekáním
- Zamykací proměnné
  - Kritickou sekci „ochráníme" sdílenou zamykací proměnnou **lock** přidruženou ke sdílenému prostředku (iniciálně lock == 0)
  - Před vstupem do kritické sekce proces testuje tuto proměnnou a, je-li nulová, nastaví ji na 1 a vstoupí do kritické sekce. Neměla-li proměnná hodnotu 0, proces čeká ve smyčce (aktivní čekání – *busy waiting*).

```
void enter_cs(void) {
    while(lock != 0);  /* Nedělej nic a čekej */
    lock++;
}
```

  - Při opouštění kritické sekce proces tuto proměnnou opět nuluje

```
void leave_cs(void) {
     lock = 0;  /* Odemkni přístup ke sdílenému prostředku */
}
```

  - Nevyřešili jsme však nic: souběh jsme přenesli na zamykací proměnnou
    - Myšlenka zamykacích proměnných však není úplně chybná (➡)

Řešení na aplikační úrovni (2)

- Striktní střídání dvou procesů nebo vláken
  - Zaveďme proměnnou *turn*, jejíž hodnota určuje, který z procesů smí vstoupit do kritické sekce. Je-li *turn* == 0, do kritické sekce může P_0, je-li == 1, pak P_1.

P_0

```
while(TRUE) {
        while(turn!=0);   /* čekej */
        critical_section();
        turn = 1;
        noncritical_section();
}
```

P_1

```
while(TRUE) {
        while(turn!=1);   /* čekej */
        critical_section();
        turn = 0;
        noncritical_section();
}
```

  - **Problém:** P_0 proběhne svojí kritickou sekcí velmi rychle, *turn*==1 a žádný proces není v kritické sekci. P_0 je rychlý i ve své nekritické části a chce vstoupit do kritické sekce. Protože však *turn* == 1, bude čekat, přestože kritická sekce je volná.
    - Je porušen požadavek Trvalosti postupu
    - Navíc řešení nepřípustně závisí na rychlostech procesů

Řešení na aplikační úrovni (3)

- Petersonovo řešení střídání dvou procesů nebo vláken
  - Řešení pro dva procesy P_i ($i = 0, 1$) – dvě globální proměnné:

```
int turn;     boolean interest[2];
```

    - Proměnná turn udává, který z procesů je na řadě při přístupu do kritické sekce
    - V poli interest procesy indikují svůj zájem vstoupit do kritické sekce; interest[i] == TRUE znamená, že P_i tuto potřebu má
      - Prvky pole interest nejsou sdílenými proměnnými

```
void proc_i () {
    do {
       j = 1 – i;
       turn = j;
       interest[i] = TRUE;
       while (interest[j] && turn == j) ;     /* aktivní čekání */
                                  /*        KRITICKÁ SEKCE              */
       interest[i] = FALSE;
                                  /*        NEKRITICKÁ ČÁST PROCESU     */
     } while (TRUE);
    }
```

    - Proces bude čekat jen pokud druhý z procesů je na řadě a současně má zájem do kritické sekce vstoupit
- Všechna řešení na aplikační úrovni působí aktivní čekání

Hardwarová podpora pro synchronizaci

- Zamykací proměnné mají smysl, avšak je nutná atomicita přístupu k nim
- Jednoprocesorové systémy mohou vypnout přerušení
  - Při vypnutém přerušení nemůže dojít k preempci
    - Nelze použít na aplikační úrovni (vypnutí přerušení je privilegovaná akce)
  - Nelze jednoduše použít pro víceprocesorové systémy
    - Který procesor přijímá a obsluhuje přerušení?
- Moderní systémy nabízejí speciální nedělitelné (atomické) instrukce
  - Tyto instrukce mezi paměťovými cykly „nepustí" sběrnici pro jiný procesor (dokonce umí pracovat i s tzv. víceportovými pamětmi)
  - Instrukce TestAndSet atomicky přečte obsah adresované buňky a bezprostředně poté změní její obsah (tas – MC68k, tsl – Intel)
  - Instrukce Swap (xchg) atomicky prohodí obsah registru procesoru a adresované buňky
  - Např. IA32/64 (I586+) nabízí i další atomické instrukce
    - Prefix „LOCK" pro celou řadu instrukcí typu *read-modify-write* (např. ADD, AND, … s cílovým operandem v paměti)

Hardwarová podpora pro synchronizaci (2)

- Příklad použití instrukce tas – Motorola 68000

```
enter_cs: tas   lock        // Kopíruj lock do CPU a nastav lock na 1
          bnz   enter_cs    // Byl-li lock nenulový,
                            //   skok na opakované testování = aktivní čekání
          ret               // Byl nulový – návrat a vstup do kritické sekce

leave_cs: mov   lock, #0    // Vynuluj lock a odemkni kritickou sekci
          ret
```

- Příklad použití instrukce xchg – IA32

```
enter_cs: mov   EAX, #1     // 1 do registru EAX
          xchg  lock, EAX   // Instrukce xchg lock, EAX atomicky prohodí
                            //   obsah registru EAX s obsahem lock.
          jnz   enter_cs    // Byl-li původní obsah proměnné lock nenulový,
                            //   skok na opakované testování = aktivní čekání
          ret               // Nebyl – návrat a vstup do kritické sekce

leave_cs: mov   lock, #0    // Vynuluj lock a odemkni tak kritickou sekci
          ret
```


Synchronizace bez aktivního čekání

- Aktivní čekání mrhá strojovým časem
  - Může způsobit i nefunkčnost při rozdílných prioritách procesů
    - Např. vysokoprioritní producent zaplní pole, začne aktivně čekat a nedovolí konzumentovi odebrat položku (samozřejmě to závisí na strategii plánování procesů)
- Blokování pomocí systémových atomických primitiv
  - sleep() volající proces je zablokován
  - wakeup(process) probuzení spolupracujícího procesu při opouštění kritické sekce

```
void producer() {
    while (1) {
        /* Vygeneruj položku do proměnné nextProduced */
        if (count == BUFFER_SIZE)  sleep();                  // Je-li pole plné, zablokuj se
        buffer[in] = nextProduced;  in = (in + 1) % BUFFER_SIZE;
        count++ ;
        if  (count == 1) wakeup(consumer);                   // Bylo-li pole prázdné, probuď konzumenta
    }
}

void consumer() {
    while (1) {
        if (count == 0) sleep();                             // Je-li pole prázdné, zablokuj se
        nextConsumed =  buffer[out];  out = (out + 1) % BUFFER_SIZE;
        count-- ;
        if (count == BUFFER_SIZE-1) wakeup(producer);  // Bylo-li pole plné, probuď producenta
        /*  Zpracuj položku z proměnné nextConsumed */
    }
}
```


Synchronizace bez aktivního čekání (2)

- Předešlý kód není řešením:
  - Zůstalo konkurenční soupeření – count je opět sdílenou proměnnou
    - Konzument přečetl count == 0 a než zavolá sleep(), je mu odňat procesor
    - Producent vloží do pole položku a count == 1, načež se pokusí se probudit konzumenta. Ten ale ještě nespí!
    - Po znovuspuštění se konzument domnívá, že pole je prázdné a volá sleep()
    - Po čase producent zaplní pole a rovněž zavolá sleep() – spí oba!
  - Příčinou této situace je ztráta budícího signálu
- Lepší řešení: Semafory

Semafory

- Obecný synchronizační nástroj (Edsger Dijkstra, NL, [1930−2002])
- Semafor *S*
  - Systémem spravovaný objekt
  - Základní vlastností je celočíselná proměnná (obecný semafor)
    - Též čítající semafor
  - Binární semafor (mutex) = zámek – hodnota 0 nebo 1
- Dvě standardní atomické operace nad semaforem
  - wait(*S*) [někdy nazývaná acquire() nebo down(), původně P (*proberen*)]
  - signal(*S*) [někdy nazývaná release() nebo up(), původně V (*vrhogen*)]
- Sémantika těchto operací:

```
wait(S) {
    while (S <= 0)
                ;   // čekej
    S--;
}
```

```
signal(S) {
    S++;
    // Čeká-li proces před
    // semaforem, pusť ho dál
}
```

  - Tato sémantika stále obsahuje aktivní čekání
  - Skutečná implementace však aktivní čekání obchází tím, že spolupracuje s plánovačem CPU, což umožňuje blokovat a reaktivovat procesy (vlákna)
    - S plánovačem spolupracovat může, neboť semafor je spravován JOS

Implementace a užití semaforů

- Implementace musí zaručit:
  - Žádné dva procesy nebudou provádět operace wait() a signal() se stejným semaforem současně
- Implementace semaforu je problémem kritické sekce
  - Operace wait() a signal() musí být atomické
  - Aktivní čekání není plně eliminováno, je ale přesunuto z aplikační úrovně (kde mohou být kritické sekce dlouhé) do úrovně jádra OS, kde je atomicita operací se semafory implementována
- Užití:

```
mutex mtx;              // Volání systému, žádost o vytvoření semaforu,
                        //  inicializovaného na hodnotu 1
    wait(mtx);          // Volání akce nad semaforem, která může
                        //  proces zablokovat
    Critical_Section;   // Práce s prostředkem krytým binárním semaforem mtx
    signal(mtx);        // Volání akce nad semaforem, která může
                        //  ukončit blokování „jiného“ procesu, jenž čeká
                        //  „před“ semaforem
```


Implementace semaforů

- Struktura semaforu

```
typedef struct {
    int value;              // „Hodnota“ semaforu
    struct process *list;   // Fronta procesů stojících „před semaforem“
} semaphore;
```

- Operace nad semaforem jsou pak implementovány jako nedělitelné s touto sémantikou

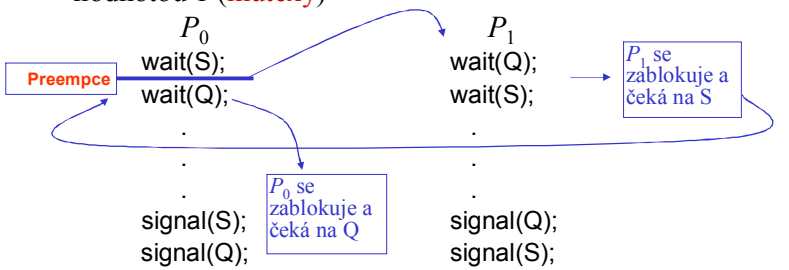
```
void wait(semaphore S) {
    S.value= S.value - 1;
    if (S.value < 0)            // Je-li třeba, zablokuj volající proces a zařaď ho
        block(S.list);          //  do fronty před semaforem (S.list)
}
void signal(semaphore S) {
    S.value= S.value + 1
    if (S.value <= 0) {
        if(S.list != NULL) {    // Je-li fronta neprázdná
            ...                 //  vyjmi proces P z čela fronty
            wakeup(P);          //  a probuď P
        }
    }
}
```


Implementace semaforů (2)

- Záporná hodnota S.value udává, kolik procesů „stojí“ před semaforem
- Fronty před semaforem: Vždy FIFO
  - Nejlépe bez uvažování priorit procesů, jinak vzniká problém se stárnutím
- Operace wait(S) a signal(S) musí být vykonány atomicky
  - Jádro bude používat atomické instrukce či jiný odpovídající hardwarový mechanismus

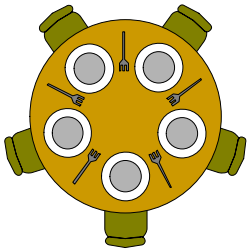
Semafory a uváznutí

- Semafory s explicitním ovládáním operacemi wait(S) a signal(S) představují synchronizační nástroj nízké úrovně
- Nevhodné použití semaforů je nebezpečné
- **Uváznutí** (*deadlock*) – dva či více procesů čeká na událost, kterou může vyvolat pouze proces, který také čeká
  - Jak snadné: Nechť S a Q jsou dva semafory s iniciální hodnotou 1 (mutexy)



Klasické synchronizační úlohy

- Producent – konzument (*Bounded-Buffer Problem*)
  - předávání zpráv mezi 2 procesy
- Čtenáři a písaři (*Readers and Writers Problem*)
  - souběžnost čtení a modifikace dat (v databázi, ...)
  - pro databáze zjednodušený případ!
- Úloha o večeřících filozofech (*Dining Philosophers Problem*)
  - zajímavý ilustrační problém souběhu
    - 5 filozofů buď přemýšlí nebo jí
    - Jedí rozvařené a tedy klouzavé špagety, a tak každý potřebuje 2 vidličky
    - Co se stane, když se všech 5 filozofů najednou uchopí např. své pravé vidličky? *„Přece časem všichni umřou hladem", milé děti*



Producent-Konzument se semafory

- Tři semafory
  - mutex s iniciální hodnotou 1 – pro vzájemné vyloučení při přístupu do sdílené paměti
  - used – počet položek v poli – inicializován na hodnotu 0
  - free – počet volných položek – inicializován na hodnotu BUF_SZ

```
void producer() {
    while (1) {   /* Vygeneruj položku do proměnné nextProduced */
                  wait(free);
                  wait(mutex);
                  buffer [in] = nextProduced;     in = (in + 1) % BUF_SZ;
                  signal(mutex);
                  signal(used);
    }
}
void consumer() {
    while (1) {   wait(used);
                  wait(mutex);
                  nextConsumed =  buffer[out]; out = (out + 1) % BUF_SZ;
                  signal(mutex);
                  signal(free);
                  /*  Zpracuj položku z proměnné nextConsumed */
    }
}
```


Čtenáři a písaři

- Úloha: Několik procesů přistupuje ke společným datům
  - Některé procesy data jen čtou – čtenáři
  - Jiné procesy potřebují data zapisovat – písaři
  - Souběžné operace čtení mohou čtenou strukturu sdílet
    - Libovolný počet čtenářů může jeden a tentýž zdroj číst současně
  - Operace zápisu musí být exklusivní, vzájemně vyloučená s jakoukoli jinou operací (zápisovou i čtecí)
    - V jednom okamžiku smí daný zdroj modifikovat nejvýše jeden písař
    - Jestliže písař modifikuje zdroj, nesmí ho současně číst žádný čtenář
- Dva možné přístupy
  - Přednost čtenářů
    - Žádný čtenář nebude muset čekat, pokud sdílený zdroj nebude obsazen písařem. Jinak řečeno: Kterýkoliv čtenář čeká pouze na opuštění kritické sekce písařem.
    - Písaři mohou stárnout
  - Přednost písařů
    - Jakmile je některý písař připraven vstoupit do kritické sekce, čeká jen na její uvolnění (čtenářem nebo písařem). Jinak řečeno: Připravený písař předbíhá všechny připravené čtenáře.
    - Čtenáři mohou stárnout

Čtenáři a písaři s prioritou čtenářů

Sdílená data

- semaphore wrt, readcountmutex;
- int readcount

Inicializace

- wrt = 1; readcountmutex = 1; readcount = 0;

Implementace

Písař:

```
wait(wrt);
....
    písař modifikuje zdroj
....
signal(wrt);
```

Čtenář:

```
wait(readcountmutex);
readcount++;
if (readcount==1) wait(wrt);
signal(readcountmutex);

    ... čtení sdíleného zdroje ...

wait(readcountmutex);
readcount--;
if (readcount==0) signal(wrt);
signal(readcountmutex);
```

Čtenáři a písaři s prioritou písařů

Sdílená data
- semaphore wrt, rdr, readcountmutex, writecountmutex;
  int readcount, writecount;

Inicializace
- wrt = 1; rdr = 1; readcountmutex = 1; writecountmutex = 1;
  readcount = 0; writecount = 0;

Implementace

Čtenář:

```
wait(rdr);
wait(readcountmutex);
readcount++;
if (readcount == 1) wait(wrt);
signal(readcountmutex);
signal(rdr);

   ... čtení sdíleného zdroje ...

wait(readcountmutex);
readcount--;
if (readcount == 0) signal(wrt);
signal(readcountmutex);
```

Písař:

```
wait(writecountmutex);
writecount++;
if (writecount==1) wait(rdr);
signal(writecountmutex);
wait(wrt);

                ... písař modifikuje zdroj ...

signal(wrt);
wait(writecountmutex);
writecount--;
if (writecount==0) release(rdr);
signal(writecountmutex);
```

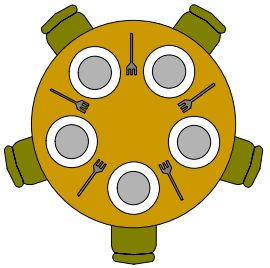

Problém večeřících filozofů

Sdílená data
- semaphore chopStick[ ] = new Semaphore[5];

Inicializace
- for(i=0; i<5; i++) chopStick[i] = 1;

Implementace filozofa *i*:

```
do {
    chopStick[i].wait;
    chopStick[(i+1) % 5].wait;
        eating();          // Teď jí
    chopStick[i].signal;
    chopStick[(i+1) % 5].signal;
        thinking();        // A teď přemýšlí
} while (TRUE) ;
```



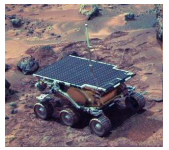
- Možné ochrany proti uváznutí
  - Zrušení symetrie úlohy
    - Jeden filozof bude levák a ostatní praváci (levák zvedá vidličky opačně)
  - Jídlo se *n* filozofům podává v jídelně s *n*+1 židlemi
    - Vstup do jídelny se hlídá čítajícím semaforem počátečně nastaveným na kapacitu *n*. To je ale jiná úloha
  - Filozof smí uchopit vidličku jen, když jsou obě volné
    - Příklad obecnějšího řešení – tzv. skupinového zamykání prostředků

Spin-lock

- Spin-lock je obecný (čítající) semafor, který používá aktivní čekání místo blokování
  - Blokování a přepínání mezi procesy či vlákny by bylo časově mnohem náročnější než ztráta strojového času spojená s krátkodobým aktivním čekáním
- Používá se ve víceprocesorových systémech pro implementaci krátkých kritických sekcí
  - Typicky uvnitř jádra
    - např. zajištění atomicity operací se semafory
- Užito např. v multiprocesorových Windows 2k/XP/7 i v mnoha Linuxech

Negativní důsledky použití semaforů

- Fakt, že semafory mohou blokovat, může způsobit:
  - uváznutí (*deadlock*)
    - Proces je blokován čekáním na prostředek vlastněný jiným procesem, který čeká na jeho uvolnění dalším procesem čekajícím z téhož důvodu atd.
  - stárnutí (*starvation*)
    - Dva procesy si prostřednictvím semaforu stále vyměňují zabezpečený přístup ke sdílenému prostředku a třetí proces se k němu nikdy nedostane
  - aktivní zablokování (*livelock*)
    - Speciální případ stárnutí s efektem podobným uváznutí, kdy procesy sice nejsou zablokovány, ale nemohou pokročit, protože se neustále snaží si vzájemně vyhovět
      - Dva lidé v úzké chodbičce se vyhýbají tak, že jeden ukročí vpravo a ten protijdoucí ukročí stejným směrem. Poté první uhne vlevo a ten druhý ho následuje ...
  - inverze priorit (*priority inversion*)
    - Proces s nízkou prioritou vlastní prostředek požadovaný procesem s vysokou prioritou, což vysokoprioritní proces zablokuje. Proces se střední prioritou, který sdílený prostředek nepotřebuje (a nemá s ním nic společného), poběží stále a nedovolí tak nízkoprioritnímu procesu prostředek uvolnit.



Mars Pathfinder
1996

Monitory

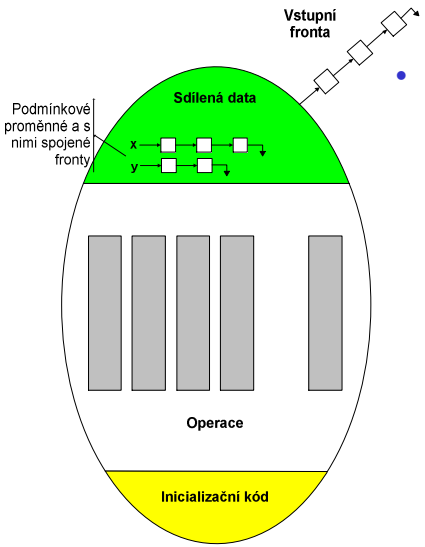
- Monitor je synchronizační nástroj vysoké úrovně
- Umožňuje bezpečné sdílení libovolného datového typu
- Monitor je jazykový konstrukt v jazycích „pro paralelní zpracování“
  - Podporován např. v Concurrent Pascal, Modula-3, C#, ...
  - V Javě může každý objekt fungovat jako monitor (viz Object.wait() a klíčové slovo synchronized)
- Procedury definované jako monitorové procedury se vždy vzájemně vylučují

```
monitor monitor_name {
    int i;                  // Deklarace sdílených proměnných
    void p1(...) { ... }    // Deklarace monitorových procedur
    void p2(...) { ... }
    {
       inicializační kód
    }
}
```


Podmínkové proměnné monitorů

- Pro účely synchronizace mezi vzájemně exkluzivními monitorovými procedurami se zavádějí tzv. podmínkové proměnné
  - datový typ **condition**
  - condition x, y;
- Pro typ condition jsou definovány dvě operace
  - x.wait();
    Proces, který zavolá tuto operaci je blokován až do doby, kdy jiný proces provede x.signal()
  - x.signal();
    Operace x.signal() aktivuje právě jeden proces čekající na splnění podmínky x. Pokud žádný proces na x nečeká, pak x.signal() je prázdnou operací

Struktura monitoru



- V monitoru se v jednom okamžiku může nacházet nejvýše jeden proces
  - Procesy, které mají potřebu vykonávat některou monitorovou proceduru, jsou řazeny do vstupní fronty
  - S podmínkovými proměnnými jsou sdruženy fronty čekajících procesů
  - Implementace monitoru je systémově závislá a využívá prostředků JOS
    - obvykle semaforů

Filozofové pomocí monitoru

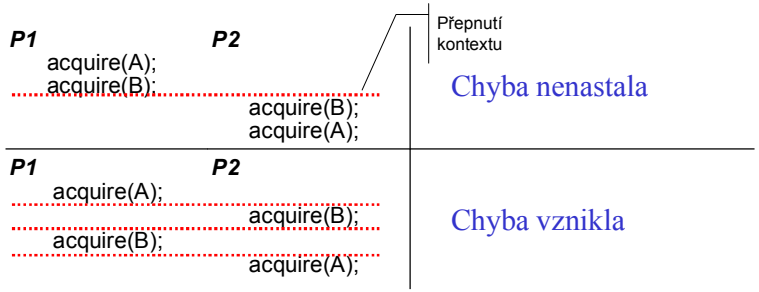
- Bez hrozby uváznutí
  - Smí uchopit vidličku, jen když jsou volné obě potřebné
- Filozof se může nacházet ve 3 stavech:
  - Myslí – nesoutěží o vidličky
  - Hladoví – čeká na uvolnění obou vidliček
  - Jí – dostal se ke dvěma vidličkám
  - Jíst může jen když oba jeho sousedé nejedí
  - Hladovějící filozof musí čekat na splnění podmínky, že žádný z obou jeho sousedů nejí
- Když bude chtít *i*-tý filozof jíst, musí zavolat proceduru pickUp(i), která se dokončí až po splnění podmínky čekání
- Až přestane filozof *i* jíst bude volat proceduru putDown(i), která značí položení vidliček; pak začne myslet
  - Uváznutí nehrozí, filozofové však mohou stárnout, a tak zcela vyhladovět

Implementace filozofů s monitorem

```
monitor DiningPhilosophers {
      enum {THINKING, HUNGRY, EATING}  state [5] ;
      condition self [5];

      void pickUp(int i) {
            state[i] = HUNGRY;
            test(i);
            if (state[i] != EATING) self [i].wait;
      }

      void putDown (int i) {
            state[i] = THINKING;
                // test left and right neighbors
            test((i + 4) % 5);
            test((i + 1) % 5);
      }

      void test(int i) {
            if (
                  (state[(i + 4) % 5] != EATING) &&
                  (state[i] == HUNGRY) &&
                  (state[(i + 1) % 5] != EATING)
               ) {
                  state[i] = EATING ;
                  self[i].signal () ;
            }
      }

      initialization_code() {
            for (int i = 0; i < 5; i++)
                  state[i] = THINKING;
      }
}
```

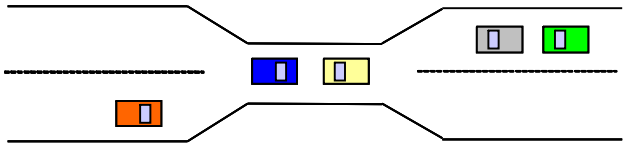

Časově závislé chyby

- Příklad časově závislé chyby
  - Procesy *P1* a *P2* spolupracují za použití mutexů *A* a *B*



- Nebezpečnost takových chyb je v tom, že vznikají jen zřídkakdy za náhodné souhry okolností
  - Jsou fakticky neodladitelné

Uváznutí na mostě



- Most se střídavým provozem
  - Každý z obou směrů průjezdu po mostě lze chápat jako sdílený prostředek (zdroj)
  - Dojde-li k uváznutí, situaci lze řešit tím, že se jedno auto vrátí – preempce zdroje (přivlastnění si zdroje, který byl vlastněn někým jiným) a vrácení soupeře před žádost o přidělení zdroje (*rollback*)
  - Při řešení uváznutí může dojít k tomu, že bude muset couvat i více aut
  - Riziko stárnutí (hladovění)

Definice uváznutí a stárnutí

- Uváznutí (*deadlock*):
  - Množina procesů $\mathscr{P}$ uvázla, jestliže každý proces $P_i \in \mathscr{P}$ čeká na událost (zaslání zprávy, uvolnění prostředku, ...), kterou může vyvolat pouze proces $P_j \in \mathscr{P},\ j \neq i$
  - Prostředek: paměťový prostor, V/V zařízení, soubor nebo jeho část, ...

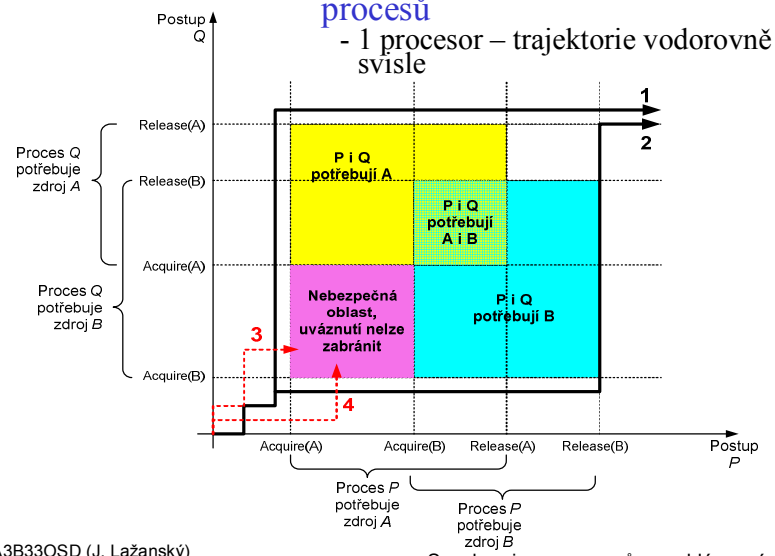
- Stárnutí:
  - Požadavky jednoho nebo více procesů z $\mathscr{P}$ nebudou splněny v konečném čase
    - např. z důvodů priorit, opatření proti uváznutí, atd.

Použitý model systému

- Typy prostředků (zdrojů) R_1, R_2, ... R_m
  - např. úseky v paměti, V/V zařízení, ...
- Každý typ prostředku R_i má W_i instancí
  - např. máme 4 magnetické pásky a 2 CD mechaniky
  - často $W_i = 1$ – tzv. *jednoinstanční prostředky*
- Každý proces používá potřebné zdroje podle schématu
  - žádost – acquire, request, wait
  - používání prostředku po konečnou dobu (kritická sekce)
  - uvolnění (navrácení) – release, signal

Bezpečné a nebezpečné trajektorie

- Bezpečné a nebezpečné trajektorie procesů
  - 1 procesor – trajektorie vodorovně nebo svisle

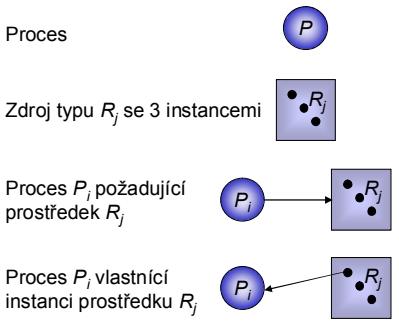


Charakteristika uváznutí

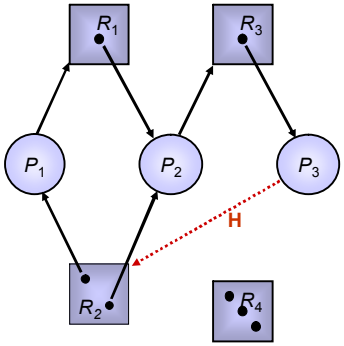
- Coffman formuloval čtyři podmínky, které musí platit **současně**, aby uváznutí **mohlo** vzniknout
  1. Vzájemné vyloučení, *Mutual Exclusion*
     - sdílený zdroj může v jednom okamžiku používat nejvýše jeden proces
  2. Postupné uplatňování požadavků, *Hold and Wait*
     - proces vlastnící alespoň jeden zdroj potřebuje další, ale ten je vlastněn jiným procesem, v důsledku čehož bude čekat na jeho uvolnění
  3. Nepřipouští se odnímání zdrojů, *No preemption*
     - zdroj může uvolnit pouze proces, který ho vlastní, a to dobrovolně, když již zdroj nepotřebuje
  4. Zacyklení požadavků, *Circular wait*
     - Existuje množina čekajících procesů $\{P_0, P_1, \ldots, P_k, P_0\}$ takových, že P_0 čeká na uvolnění zdroje drženého P_1, P_1 čeká na uvolnění zdroje drženého P_2, ... , P_{k-1} čeká na uvolnění zdroje drženého P_k, a P_k čeká na uvolnění zdroje drženého P_0.
     - V případě jednoinstančních zdrojů splnění této podmínky značí, že k uváznutí již došlo.

Graf přidělování zdrojů

- Modelování procesů a zdrojů pomocí Grafu přidělování zdrojů (*Resource Allocation Graph*, RAG):
- Množina uzlů $\boldsymbol{V}$ a množina hran $\boldsymbol{E}$
- Uzly dvou typů:
  - $\mathscr{P} = \{P_1, P_2, \ldots, P_n\}$, množina procesů existujících v systému
  - $\mathscr{R} = \{R_1, R_2, \ldots, R_m\}$, množina zdrojů existujících v systému
- Hrany:
  - hrana požadavku – orientovaná hrana $P_i \rightarrow R_j$
  - hrana přidělení – orientovaná hrana $R_i \rightarrow P_j$
- Bipartitní graf



Příklad RAG

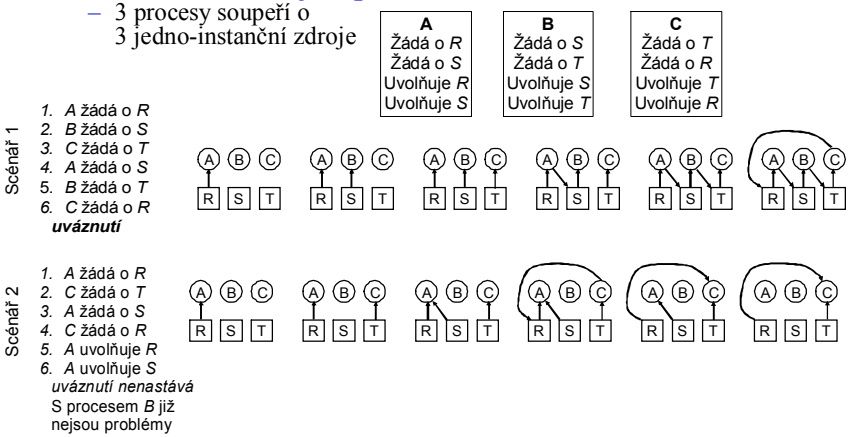


- Proces P_1 vlastní zdroj R_2 a požaduje zdroj R_1
- Proces P_2 vlastní zdroje R_1 a R_2 a ještě požaduje zdroj R_3
- Proces P_3 vlastní zdroj R_3
- Zdroj R_4 není nikým vlastněn ani požadován
- Jednoinstanční zdroje R_1 a R_3 jsou obsazeny
- Instance zdroje R_2 jsou vyčerpány
- Přidání hrany H, kdy proces P_3 zažádá o přidělení zdroje R_2 a zablokuje se, způsobí uváznutí

- V RAG není cyklus
  - K uváznutí nedošlo a zatím ani nehrozí
- V RAG se cyklus vyskytuje
  - Jsou-li součástí cyklu pouze zdroje s jednou instancí, pak došlo k uváznutí
  - Mají-li dotčené zdroje více instancí, pak k uváznutí může dojít

Plánování procesů a uváznutí

- Uvažme následující příklad a 2 scénáře:
  - 3 procesy soupeří o 3 jedno-instanční zdroje



- Lze vhodným plánováním předejít uváznutí?
  - Za jakých podmínek?
  - Jak to algoritmizovat?

Co lze činit s problémem uváznutí?

Existují čtyři přístupy

- Zcela ignorovat hrozbu uváznutí
  - Pštrosí algoritmus – strč hlavu do písku a předstírej, že se nic neděje
  - Používá mnoho OS včetně mnoha implementací UNIXů
- Prevence uváznutí
  - Pokusit se přijmout taková opatření, aby se uváznutí stalo vysoce nepravděpodobným
- Vyhýbání se uváznutí
  - Zajistit, že k uváznutí *nikdy* nedojde
  - Prostředek se nepřidělí, pokud by hrozilo uváznutí
    - hrozí stárnutí
- Detekce uváznutí a následná obnova
  - Uváznutí se připustí, detekuje se jeho vznik a zajistí se obnova stavu před uváznutím

Prevence uváznutí

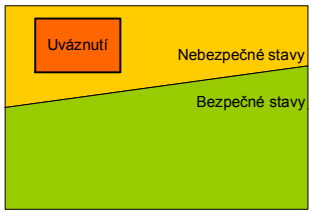
- Konzervativní politikou se omezuje přidělování prostředků
  - Přímá metoda – plánovat procesy tak, aby nevznikl cyklus v RAG
    - Vzniku cyklu se brání tak, že zdroje jsou očíslovány a procesy je smějí alokovat pouze ve vzrůstajícím pořadí čísel zdrojů
      - Nerealistické – zdroje vznikají a zanikají dynamicky
  - Nepřímé metody (narušení některé Coffmanovy podmínky)
    - Eliminace potřeby vzájemného vyloučení
      - Nepoužívat sdílené zdroje, virtualizace (spooling) periferií
      - Mnoho činností však sdílení nezbytně potřebuje ke své funkci
    - Eliminace postupného uplatňování požadavků
      - Proces, který požaduje nějaký zdroj, nesmí dosud žádný zdroj vlastnit
      - Všechny prostředky, které bude kdy potřebovat, musí získat naráz
      - Nízké využití zdrojů
    - Připustit násilné odnímání přidělených zdrojů (preempce zdrojů)
      - Procesu žádajícímu o další zdroj je dosud vlastněný prostředek odňat
        - » To může být velmi riskantní – zdroj byl již zmodifikován
      - Proces je reaktivován, až když jsou všechny potřebné prostředky volné
        - » Metoda inkrementálního zjišťování požadavků na zdroje – nízká průchodnost
- Kterákoliv metoda prevence uváznutí způsobí výrazný pokles průchodnosti systému

Vyhýbání se uváznutí

- Základní problém: Systém musí mít dostatečné apriorní informace o požadavcích procesů na zdroje
  - Nejčastěji se požaduje, aby každý proces udal maxima počtu prostředků každého typu, které bude za svého běhu požadovat
- Algoritmus:
  - Dynamicky se zjišťuje, zda stav subsystému přidělování zdrojů zaručuje, že se procesy v žádném případě nedostanou do cyklu v RAG
- Stav systému přidělování zdrojů je popsán
  - Počtem dostupných a přidělených zdrojů každého typu a
  - Maximem očekávaných žádostí procesů
  - Stav může být bezpečný nebo nebezpečný

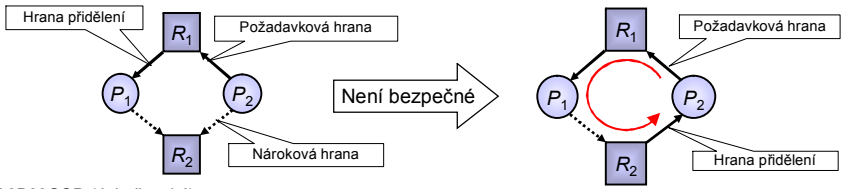
Vyhýbání se uváznutí – bezpečný stav

- Systém je v bezpečném stavu, existuje-li „*bezpečná posloupnost procesů*“
  - Posloupnost procesů $\{P_0, P_1, \ldots, P_n\}$ je bezpečná, pokud požadavky každého P_i lze uspokojit právě volnými zdroji a zdroji vlastněnými všemi P_k , $k < i$
    - Pokud nejsou zdroje požadované procesem P_i volné, pak P_i bude čekat dokud se všechny P_k neukončí a nevrátí přidělené zdroje
    - Když P_{i-1} skončí, jeho zdroje může získat P_i, proběhnout a jím vrácené zdroje může získat P_{i+1}, atd.
    - Je-li systém v bezpečném stavu (*safe state*) k uváznutí nemůže dojít. Ve stavu, který není bezpečný (*unsafe state*), přechod do uváznutí hrozí
  - Vyhýbání se uváznutí znamená:
    - Plánovat procesy tak, aby systém byl stále v bezpečném stavu
      - Nespouštět procesy, které by systém z bezpečného stavu mohly vyvést
      - Nedopustit potenciálně nebezpečné přidělení prostředku



Vyhýbání se uváznutí – algoritmus

- Do RAG se zavede „nároková hrana“
  - Nároková hrana $P_i \rightarrow R_j$ značí, že někdy v budoucnu bude proces P_i požadovat zdroj $P_i \rightarrow R_j$
    - V RAG hrana vede stejným směrem jako požadavek na přidělení, avšak kreslí se čárkovaně
  - Nároková hrana se v okamžiku vzniku žádosti o přidělení převede na požadavkovou hranu
  - Když proces zdroj získá, požadavková hrana se změní na hranu přidělení
  - Když proces zdroj vrátí, hrana přidělení se změní na požadavkovou hranu
  - Převod požadavkové hrany v hranu přidělení nesmí v RAG vytvořit cyklus (včetně uvažování nárokových hran)



Bankéřský algoritmus

- Chování odpovědného bankéře:
  - Klienti žádají o půjčky do určitého limitu
  - Bankéř ví, že ne všichni klienti budou svůj limit čerpat současně a že bude půjčovat klientům prostředky postupně
  - Všichni klienti v jistém okamžiku svého limitu dosáhnou, avšak nikoliv současně
  - Po dosažení přislíbeného limitu klient svůj dluh v konečném čase vrátí
  - Příklad:
    - Ačkoliv bankéř ví, že všichni klienti budou dohromady potřebovat 22 jednotek, na celou transakci má jen 10 jednotek

Klient	Užito	Max.
Adam	0	6
Eva	0	5
Josef	0	4
Marie	0	7

K dispozici: 10
Počáteční stav (a)

Klient	Užito	Max.
Adam	1	6
Eva	1	5
Josef	2	4
Marie	4	7

K dispozici: 2
Stav (b)

Klient	Užito	Max.
Adam	1	6
Eva	2	5
Josef	2	4
Marie	4	7

K dispozici: 1
Stav (c)

Bankéřský algoritmus (2)

- Činnost
  - Zákazníci přicházející do banky pro úvěr předem deklarují maximální výši, kterou si budou kdy chtít půjčit
  - Úvěry v konečném čase splácí
  - Bankéř úvěr neposkytne, pokud si není jist, že uspokojí všechny zákazníky
- Analogie
  - Zákazník = proces
  - Úvěr = přidělovaný prostředek
- Vlastnosti
  - Procesy musí deklarovat své potřeby předem
  - Proces požadující přidělení může být zablokován
  - Proces vrátí všechny přidělené zdroje v konečném čase
  - Nikdy nedojde k uváznutí
    - Proces bude spuštěn jen, pokud bude možno uspokojit všechny jeho požadavky
  - Sub-optimální pesimistická strategie
    - Předpokládá se, že nastane nejhorší případ

Bankéřský algoritmus (3)

- Datové struktury
  - n ... počet procesů
  - m ... počet typů zdrojů
  - Vektor available[m]
    - available[j] == k značí, že je *k* instancí zdroje typu R_j je volných
  - Matice max[n, m]
    - Povinná deklarace procesů:
    - max[i, j] == k znamená, že proces P_i bude během své činnosti požadovat až *k* instancí zdroje typu R_j
  - Matice allocated[n, m]
    - allocated[i, j] = k značí, že v daném okamžiku má proces P_i přiděleno *k* instancí zdroje typu R_j
  - Matice needed[n, m] (needed[i, j] = max[i, j] – allocated[i, j])
    - needed[i, j] = k říká, že v daném okamžiku procesu P_i chybí ještě *k* instancí zdroje typu R_j

Bankéřský algoritmus (4)

- Test bezpečnosti stavu
  1. Inicializace
     - work[m] a finish[n] jsou pracovní vektory
     - Inicializujeme work = available; finish[i] = false; *i*=1, ... , *n*
  2. Najdi *i*, pro které platí
     - (finish[i] == false) && (needed[i] <= work[i])
     - Pokud takové *i* neexistuje, jdi na krok 4
  3. Simuluj ukončení procesu *i*
     - work[i] = work[i] + allocated[i]; finish[i] = true;
     - Pokračuj krokem 2
  4. Pokud platí
     - finish[i] == true pro všechna *i*, pak stav systému je bezpečný

Postup přidělení zdroje bankéřským algoritmem

Proces P_i formuje vektor request:

request[j] == k znamená, že proces P_i žádá o *k* instancí zdroje typu R_j

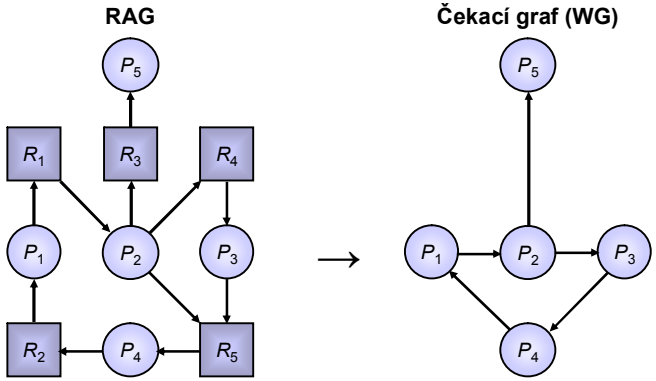
1. if(request[j] >= needed[i, j]) error;
   - Deklarované maximum překročeno!
2. if(request[j] <= available[j]) goto 3;
   - Jinak zablokuj proces P_i – požadované prostředky nejsou volné
3. Namodeluj přidělení prostředku a otestuj bezpečnost stavu:
   - available[j] = available[j] – request[j];
   - allocated[i, j] = allocated[i, j] + request[j];
   - needed[i, j] = needed[i, j] – request[j];

   } Akce 3

   - Spusť test bezpečnosti stavu
     - Je-li bezpečný, přiděl požadované zdroje
     - Není-li stav bezpečný, pak vrať úpravy „Akce 3“ a zablokuj proces P_i, neboť přidělení prostředků by způsobilo nebezpečí uváznutí

Detekce uváznutí s následnou obnovou

- Strategie připouští vznik uváznutí:
  - Uváznutí je třeba detekovat
  - Vznikne-li uváznutí, aplikuje se plán obnovy systému
  - Aplikuje se zejména v databázových systémech



Detekce uváznutí – postup

- Případ jednoinstančního zdroje daného typu
  - Udržuje se čekací graf – uzly jsou procesy
  - Periodicky se provádí algoritmus hledající cykly
  - Algoritmus pro detekci cyklu v grafu má složitost $O(n^2)$, kde n je počet hran v grafu
- Případ více instancí zdrojů daného typu
  - n ... počet procesů
  - m ... počet typů zdrojů
  - Vektor available[m]
    - available[j] = k značí, že je k instancí zdroje typu R_j je volných
  - Matice allocated[n, m]
    - allocated[i, j] = k značí, že v daném okamžiku má proces P_i přiděleno k instancí zdroje typu R_j
  - Matice request[n, m]
    - Indikuje okamžité požadavky každého procesu:
    - request[i, j] = k znamená, že proces P_i požaduje dalších k instancí zdroje typu R_j

Detekce uváznutí – algoritmus

1. Nechť
   - work[m] a finish[n] jsou pracovní vektory
   - Inicializujeme work = available; finish[i] = false; i=1, ... , n
2. Najdi i, pro které platí
   - (finish[i] == false) && (request[i] <= work[i])
   - Pokud takové i neexistuje, jdi na krok 4
3. Simuluj ukončení procesu i
   - work[i] += allocated[i]; finish[i] = true;
   - Pokračuj krokem 2
4. Pokud platí
   - finish[i] == false pro některé i, pak v systému došlo k uváznutí. Součástí cyklů ve WG jsou procesy P_i, kde finish[i] == false

Algoritmus má složitost $O(m\ n^2)$

m a n mohou být veliká a algoritmus časově značně náročný

Použitelnost detekčního algoritmu a obnova

- Kdy a jak často algoritmus vyvolávat? (Detekce je drahá)
  - Jak často bude uváznutí vznikat?
  - Kterých procesů se uváznutí týká a kolik jich „likvidovat“?
    - Minimálně jeden v každém disjunktním cyklu ve WG
- Násilné ukončení všech uváznutých procesů
  - velmi tvrdé a nákladné
- Násilně se ukončují dotčené procesy dokud cyklus nezmizí
  - Jak volit pořadí ukončování
    - Kolik procesů bude nutno ukončit
    - Jak dlouho už proces běžel a kolik mu zbývá do ukončení
    - Je to proces interaktivní nebo dávkový (dávku lze snáze restartovat)
    - Cena zdrojů, které proces použil
  - Výběr oběti podle minimalizace ceny
  - Nebezpečí stárnutí
    - některý proces bude stále vybírán jako oběť

Alternativa: Neblokující synchronizace

- Princip
  - Sériový přístup ke sdílenému zdroji řízený blokujícími „zámky“ se nahradí „neřízeným přístupem“ a vždy se kontroluje, zda nedošlo k nechtěnému prokládání procesů (či vláken)
  - Ilustrativní příklad:

```
void increment_counter(int *counter) {
    do {
        int oldvalue = *counter;
        int newvalue = oldvalue + 1;
        /* BEGIN ATOMIC COMPARE-AND-SWAP INSTRUCTION */
        if (*counter == oldvalue) { *counter = newvalue; success = true; }
        else { success = false; }
        /* END COMPARE-AND-SWAP INSTRUCTION */
    } while (!success);
}
```

- Potřeba hardwarové podpory
  - Např. Intel Pentium (a vyšší) má atomickou instrukci cmpxchg, která pracuje přesně tak, jak je uvedeno v příkladu

Neblokující synchronizace - hodnocení

- Výhody
  - Menší režie se správou zámků a s nimi spojených front
  - Odstraňuje nebezpečí vzniku uváznutí
  - Není problém s „inverzí priorit“
  - Automatická (hardwarově zajištěná) synchronizace se subsystémem přerušení
- Nevýhody
  - Potřeba hardwarové podpory
  - Vlivem neomezeného počtu pokusů o opakování přístupu může způsobovat značné mrhání strojovým časem
  - Vyžaduje velmi pečlivou správu paměti, kdy je nutno zajistit, aby nedošlo k odejmutí paměti v době, kdy ji vlákno referencuje (*counter v předchozím příkladu)

Závěrečné úvahy o uváznutí

- Metody popsané jako „prevence uváznutí“ jsou velmi restriktivní
  - ne vzájemnému vyloučení, ne postupnému uplatňování požadavků, preempce prostředků
- „Vyhýbání se uváznutí“ nemá dost apriorních informací
  - zdroje dynamicky vznikají a zanikají (např. úseky souborů)
- Detekce uváznutí a následná obnova
  - jsou vesměs velmi drahé – vyžadují restartování aplikací
- Smutný závěr
  - Problém uváznutí je v obecném případě efektivně neřešitelný
- Realistické východisko
  - Distribuované systémy, jejichž komponenty jsou jednoúčelové a nesoutěží o sdílené prostředky
    - Předávání zpráv na "hardwarové úrovni" pomocí protokolů vytvořených pro počítačové sítě a jiné zabezpečené sběrnice
- Existuje však řada algoritmů pro speciální situace
  - Používá se zejména v databázových systémech ➡

Dotazy