OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

Meziprocesní komunikace a synchronizace procesů

Ing. Pavel Smolka, Ph.D.

OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

Východisko

- Souběžný přístup ke sdíleným datům může způsobit jejich nekonzistenci ⇒ nutná kooperace 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
  - Může dojít k uváznutí
    - Každý proces v jisté skupině procesů čeká na zprávu od jiného procesu v téže skupině
  - Může dojít ke stárnutí
    - Dva procesy si vzájemně posílají zprávy, zatímco třetí proces čeká na zprávu nekonečně dlouho
- 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

Definice souběhu

- Zejména u složitějších datových struktur (obousměrné spojové seznamy, složité dynamické struktury uložené v souborech apod.) dochází často k tomu, že v určitém stadiu zpracování jsou data dočasně nekonzistentní
- Pokud v tom okamžiku dojde k přepnutí kontextu na proces, který tato data také používá, může nastat souběh.
- Souběh (race condition) je situace, kdy při přístupu dvou nebo více procesů ke sdíleným datům dojde k chybě, přestože každý z procesů samostatně se chová korektně.

Příklad souběhu 1

1. proces (výběr)

pom:=konto;

pom:=pom-10000;

-> (context switch) ->

2. proces (vklad)

pom:=konto;

pom:=pom+20000;

konto:=pom;

<- (context switch) <-

konto:=pom;

Řešení problému 1

1. proces (výběr)

konto:=konto-10000;

-> (context switch) ->

<- (context switch) <-

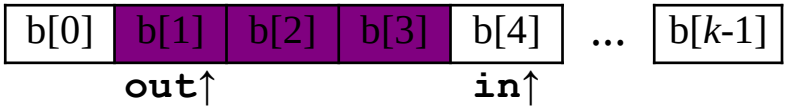
konto:

2. proces (vklad)

konto:=konto+20000;

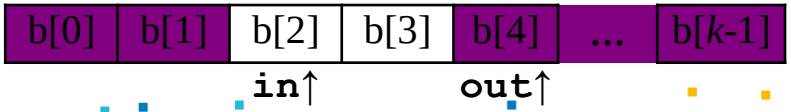
Úloha Producent-Konzument

- 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`**
  - Pokud je v poli nějaká položka, konzument při jejím vyjmutí dekrementuje **`count`**



`in` je privátní proměnná producenta

`out` je privátní proměnná konzumenta



OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

Problém soupeření (*race condition*)

- count++ bude implementováno asi takto:

P_1:	$registr_1$	← count	`move count,D0`
P_2:	$registr_1$	← $registr_1$ + 1	`add D0,#1`
P_3:	count	← $registr_1$	`move D0,count`

- count-- bude zřejmě implementováno jako:

K_1:	$registr_2$	← count	`move count,D4`
K_2:	$registr_2$	← $registr_2$ – 1	`sub D4,#1`
K_3:	count	← $registr_2$	`move D4,count`

- Tam, kde platí Murphyho zákony, **<u>může</u>** nastat následující posloupnost prokládání producenta a konzumenta (nechť na počátku count = 3)

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

Na konci může být
count == 2 nebo 4,
ale programátor zřejmě chtěl mít 3 (i to se může podařit)

Je to 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

OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

Požadavky na řešení problému kritických sekcí

1. Vzájemné vyloučení – podmínka bezpečnosti (*Mutual Exclusion*)
   - Pokud proces P_i je ve své kritické sekci, pak žádný další 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 zdrojem a existuje alespoň jeden proces, který si přeje vstoupit do kritické sekce sdružené se tímto zdrojem, pak výběr procesu, který do takové kritické sekce vstoupí, se nesmí odkládat nekonečně dlouho.
3. Konečné čekání – podmínka spravedlivosti (*Fairness*)
   - Proces smí čekat na povolení vstupu do kritické sekce jen konečnou dobu.
   - Musí existovat omezení počtu, kolikrát může být povolen vstup do kritické sekce sdružené se jistým prostředkem 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 žádnou další podporu
  - Základní (a problematické) řešení s aktivním čekáním (*busy waiting*)
- Hardwarové řešení
  - Pomocí speciálních instrukcí procesoru
  - Stále ještě s aktivním čekáním
- Softwarové řešení zprostředkované operačním systémem
  - Potřebné služby a datové struktury poskytuje OS (např. semafory)
  - Tím je umožněno pasivní čekání – proces nesoutěží o procesor
  - Podpora volání synchronizačních služeb v programovacích systémech/jazycích (např. monitory, zasílání zpráv)

OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

Prostředky pro zajištění výlučného přístupu - obecně

- Zákaz přerušení
- Instrukce Test and set lock
- Semafory

Zákaz přerušení

- Zákaz přerušení znemožní přepnutí kontextu.
- Nebezpečné - proces může zakázat přerušení a zhavarovat nebo se dostat do nekonečného cyklu -> celý systém je mrtvý -> u většiny systémů může přerušení zakázat pouze jádro OS.
- Zákaz přerušení je privilegovaná instrukce. OS zpravidla vnitřně používá zákaz přerušení, aby zajistil nedělitelnost posloupností instrukcí používaných při implementaci jiných synchronizačních konstrukcí.

Vzájemné vyloučení s aktivním čekáním

Zamykací proměnné

- Kritickou sekci „ochráníme" sdílenou zamykací proměnnou přidruženou ke sdílenému prostředku (iniciálně = 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*).
  - `while(lock != 0)`
    `; /* Nedělej nic a čekej */`
- Při opouštění kritické sekce proces tuto proměnnou opět nuluje.
  - `lock = 0;`
- Čeho jsme dosáhli? Nevyřešili jsme nic: souběh jsme přenesli na zamykací proměnnou
- Myšlenka zamykacích proměnných však není zcela chybná

Vzájemné vyloučení střídáním procesů

- 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:** Nechť P_0 proběhne svojí kritickou sekcí velmi rychle, *turn* = 1 a oba procesy jsou v nekritických částech. 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 2 (Trvalost postupu)
    - Navíc řešení nepřípustně závisí na rychlostech procesů

OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

Petersonovo řešení

- Řešení pro dva procesy P_i (i = 0, 1) – dvě globální proměnné:

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

  - 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

```
void proc_i () {
   do {
        j = 1 – i;
        turn = j;
        interest[i] = TRUE;
        while (interest[j] && turn == j) ;   /* 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

Hardwarová podpora pro synchronizaci

- Zamykací proměnné rozumné, avšak je nutná atomicita
- 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 přerušení?
- Moderní systémy nabízejí speciální nedělitelné instrukce
  - Tyto instrukce mezi paměťovými cykly „nepustí“ sběrnici pro jiný procesor (dokonce umí pracovat i s víceportovými pamětmi)
  - Instrukce TestAndSet atomicky přečte obsah adresované buňky a bezprostředně poté změní její obsah (tas – MC68k, tsl)
  - Instrukce Swap (xchg) atomicky prohodí obsah registru procesoru a adresované buňky
    - Instrukce xchg a zejména její rozšířená verze cmpxchg (I586+) umožňuje i implementaci tzv. ***neblokující synchronizace*** na multiprocesorech
  - 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)

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() místo aktivního čekání – proces se zablokuje
  - 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 */
        }
}
```

OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

Synchronizace bez aktivního čekání (2)

- Předešlý kód však také není řešením:
  - Je zde 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, který 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--;
}
```

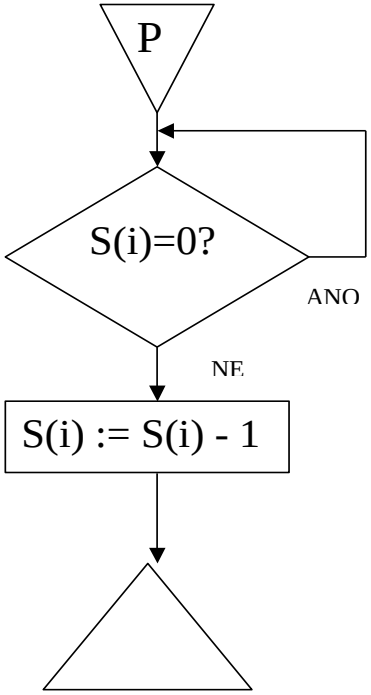
```
signal(S) {
    S++;
        // Čeká-li proces před
                // semaforem, pusť ho dál
}
```

  - Tato sémantika stále obsahuje aktivní čekání

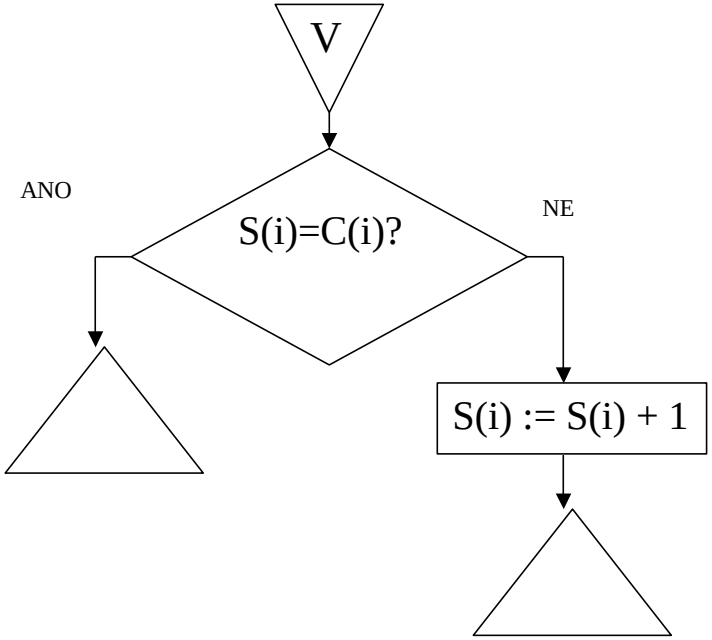
Semafory

- Instrukce "test and set lock" můžeme zobecnit takovým způsobem, že dvoustavovou proměnnou typu boolean nahradíme čítačem (proměnnou typu integer). Operace byly původně nazvány P a V, podle dánských slov proberen (testovat) a verhogen (zvětšit). Některé prameny tyto operace nazývají down a up.

Princip operace P



Princip operace V



Dvě sémantiky vzhledem k hodnotám čítače:

- 1. čítač >= 0
  - Operace **DOWN** zkontroluje hodnotu semaforu. Jestliže je větší než 0, sníží hodnotu semaforu o 1 a operace skončí. Jestliže je rovna 0, operace DOWN se zablokuje a příslušný proces je přidán do fronty čekajících procesů na daném semaforu.
  - Operace **UP** zjistí, zda je fronta čekajících procesů na daném semaforu neprázdná. Pokud ano, vybere jeden z čekajících procesů (např. nejdéle čekající) a ten odblokuje (tj. pokračuje za svou operací DOWN). Pokud je fronta prázdná, zvětší čítač semaforu o 1.

- 2. čítač libovolný (záporná hodnota je počet zablokovaných procesů)
  - Operace **DOWN** sníží hodnotu semaforu o 1. Jestliže je větší nebo roven 0, operace skončí. Jestliže je menší než 0, operace DOWN se zablokuje a příslušný proces je přidán do fronty čekajících procesů na daném semaforu.
  - Operace **UP** zvětší čítač semaforu o 1. Pokud je hodnota menší rovna 0, zkontroluje, zda je fronta čekajících procesů na daném semaforu neprázdná. Pokud ano, vybere jeden z čekajících procesů a ten odblokuje.

Operace P

```
procedure Down(var S: semaphore);
begin
    DisableInterrupts;
    while S<=0 do begin
            EnableInterrupts;
            DisableInterrupts;
    end;
    S:=S-1;
    EnableInterrupts;
end;
```

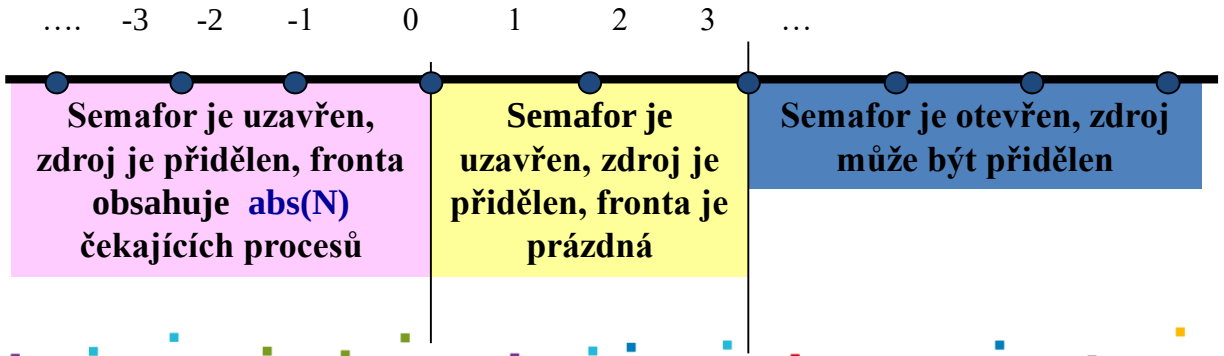
Operace V

```
procedure Up(var S:semaphore);
begin
    DisableInterrupts;
    S:=S+1;
    EnableInterrupts;
end;
```

Operační systémy - procesy

Semafory

- **Semafor** je celočíselná proměnná s hodnotou **N –** počet současně operujících procesů nad zdrojem. Hodnotu semaforu N nastavuje inicializační operace ***sem_init***
- Nad semaforem provádějí atomické operace funkce ***sem_down*** (snižuje hodnotu N o 1) a ***sem_up*** (zvyšuje hodnotu N o1)



Operační systémy - procesy

Jak semafor funguje?

- Před vstupem do kritické oblasti sníží proces funkcí ***sem_down*** hodnotu N o 1. Pokud je N=0, proces je „uspán“ a zařazen do fronty semaforu. Pokud je N>0, proces dostane přístup ke zdroji
- Před výstupem z kritické oblasti proces funkcí ***sem_up*** zvýší N o 1. Pokud je N=0, vybere se z fronty semafory zablokovaný proces a „probudí se“. Pokud je fronta prázdná a N=0, funkcí ***sem_up*** se nastaví N na hodnotu 1.

Použití semaforů

- Semafory se používají podobně jako instrukce "test nad set" - před vstupem do kritické sekce se vyvolá Down, po výstupu Up. Semafory popsané výše také nesplňují podmínku omezeného čekání a neodstraňují aktivní čekání.

Implementace a užití semaforů

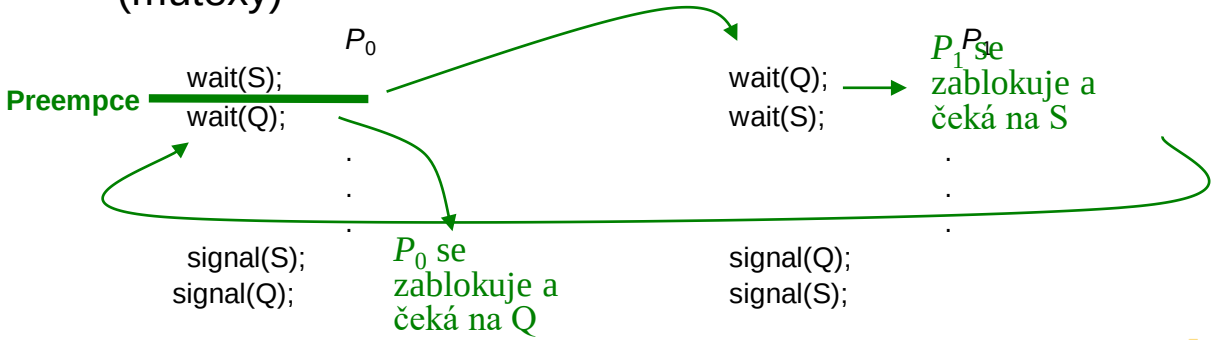
- Implementace musí zaručit aby žádné dva procesy neprováděly operace wait() a signal() se stejným semaforem současně
- Implementace semaforu je problémem kritické sekce
  - Operace wait() a signal() musí být atomické
  - Aktivní čekání není plně eliminováno, je ale přesunuto z aplikační úrovně (kde mohou být kritické sekce dlouhé) do úrovně jádra OS pro implementaci atomicity operací se semafory

- Užití:

```
mutex mtx;     // Volání systému o vytvoření semaforu,
                                        //   inicializován na hodnotu 1
        wait(mtx);                      // Volání akce nad semaforem, která může
                                        //   proces zablokovat
        Critical_Section;
        signal(mtx);                    // Volání akce nad semaforem, která může
                                        //   ukončit blokování procesu čekajícího
                                        //   „před“ semaforem
```

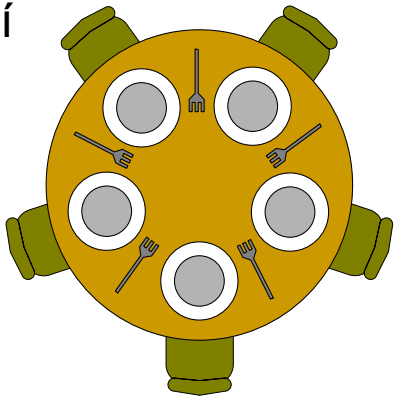
Semafory a uváznutí

- 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 pro řešení uváznutí
    - 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?



Producent-Konzument se semafory

OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

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

OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

Sdílená data

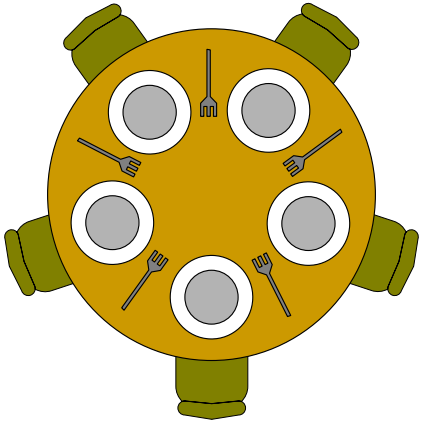
- semaphore chopStick[ ] = new Semaphore[5];

Inicializace

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

Implementace *i*-tého filozofa:

```
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) ;
```



Toto řešení nepočítá s žádnou ochranou proti uváznutí

- Rigorózní ochrana proti uváznutí je značně komplikovaná

OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

Filozofové, ochrana proti uváznutí

Zrušení symetrie úlohy
- Jeden z filozofů bude levák a ostatní praváci
  - Levák se liší pořadím zvedání vidliček

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 vidličky pouze tehdy, jsou-li obě (tedy ta vpravo i vlevo) volné
- Musí je uchopit uvnitř kritické sekce
- Příklad obecnějšího řešení – tzv. skupinového zamykání prostředků

OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

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 7/10

Problémy s použitím semaforů

Semafory s explicitním ovládáním operacemi wait(S) a signal(S) představují synchronizační nástroj nízké úrovně

Avšak

- Jsou-li operace wait(S) a signal(S) prováděny více procesy, jejich účinek nemusí být zcela determinován, pokud procesy nepřistupují k semaforům ve společně definovaném pořadí
- Chybné použití semaforů v jednom procesu naruší souhru všech procesů

Příklady chybného použití semaforů:

```
wait(S);
...
wait(S);
```

```
wait(S);
...
signal(T);
```

```
signal(S);
...
wait(S);
```

OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

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.

OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

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
        }
}
```

OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

Podmínkové proměnné monitorů

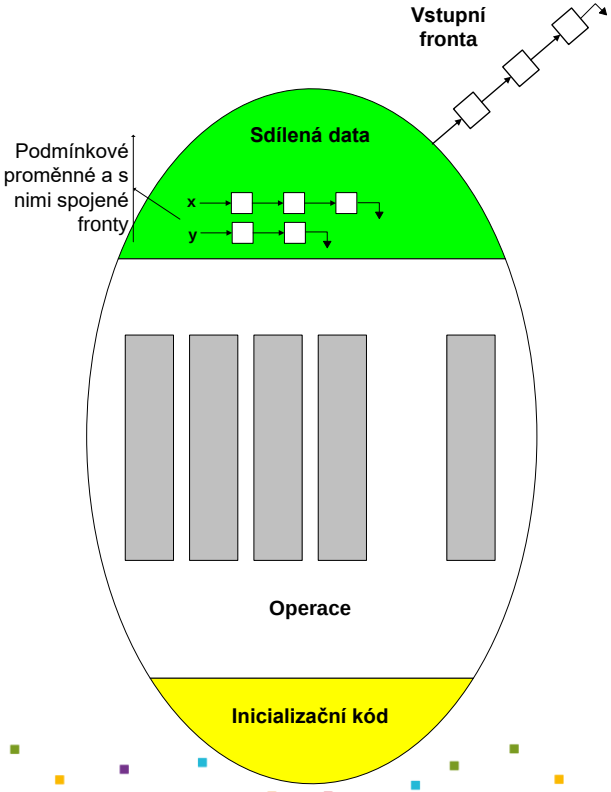
Pro účely synchronizace mezi vzájemně exkluzivními monitorovými procedurami se zavádějí tzv. podmínkové proměnné

- 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

OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

Filozofové pomocí monitoru

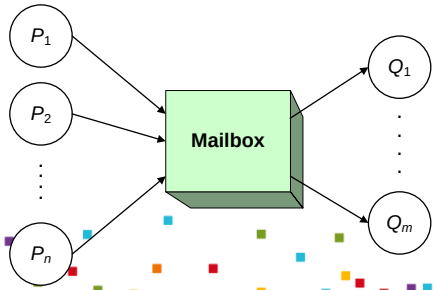
- Bez hrozby uváznutí
  - Smí uchopit vidličku, jen když jsou volné obě potřebné
- Filozof se může nacházet ve 3 stavech:
  - Myslí – nesoutěží o vidličky
  - Hladoví – čeká na uvolnění obou vidliček
  - Jí – dostal se ke dvěma vidličkám
  - Jíst může jen když oba jeho sousedé nejedí
  - Hladovějící 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

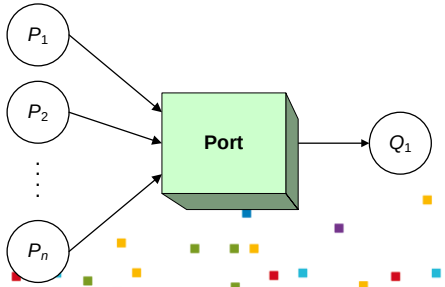
Implementace filozofů s monitorem

```
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;
        }
}
```

Synchronizace pomocí zasílání zpráv

- Mechanismus mailboxů a portů
- Mailbox je schránka pro předávání zpráv
  - Může být soukromá pro dvojici komunikujících procesů nebo sdílená více procesy
  - JOS vytvoří mailbox na žádost procesu a tento proces je pak jeho vlastníkem
  - Vlastník může mailbox zrušit nebo bude automaticky zrušen při skončení vlastníka
- Port je schránka vlastněná jediným příjemcem
  - Zprávy do portu může zasílat více procesů
  - V modelu klient/server je přijímacím procesem server
  - Port zaniká ukončením přijímacího procesu

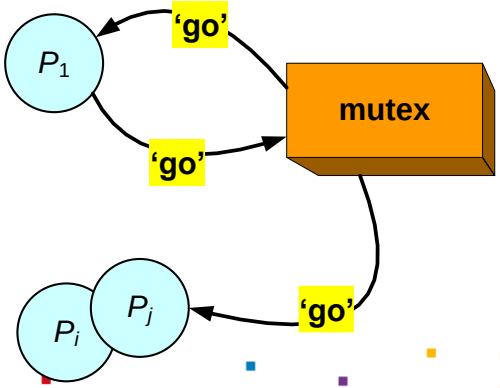




Vzájemné vyloučení pomocí zpráv

Mailbox se použije jako zamykací mechanismus (mutex)

- Operace send(mbx, msg) je asynchronní operací, končící odesláním zprávy
- Operace receive(mbx, msg) je synchronní a způsobí zablokování volajícího procesu, pokud je mbx prázdný
- Inicializace: send(mutex, ´go´);
- Před vstupem do svých kritických sekcí volají procesy receive(mutex, msg) a ten, který dostane zprávu ´go´ jako první, vstoupí. Vzhledem k synchronnosti receive() postoupí jen jeden proces
- Při výstupu z kritické sekce procesy volají send(mutex, ´go´);



Producent/Konzument se zasíláním zpráv

Sdílená data

```
mailbox mayconsume, mayproduce;
```

Inicializace

```
for(i=0; i<BUF_SZ; i++)  send(mayproduce, ‘free‘);
```

Implementace:

```
void producer() {
    message pmsg;
    while (1) {
        receive(mayproduce, &pmsg);
            /* Vygeneruj položku do proměnné pmsg */
        send(mayconsume, pmsg);
    }
}
void consumer() {
    message cmsg;
    while (1) {
        receive(mayconsume, cmsg);
            /*  Zpracuj položku z proměnné cmsg */
        send(mayproduce, ‘free‘);
    }
}
```

Synchronizace v různých OS

- Synchronizace ve Windows
  - Na monoprocesorech se používá vypínání přerušení při přístupu ke globálním prostředkům, na multiprocesorech se využívají v jádře spin-lock semafory
  - Jsou podporovány i tzv. dispečerské objekty, připomínající a fungující jako obecné nebo binární semafory
  - Dispečerské objekty mohou podporovat i tzv. events (události)
    - Events pracují obdobně jako monitory. Používají rovněž podmínkové proměnné a fronty s nimi sdružené.
- Synchronizace v Linuxu
  - Krátké kritické sekce na jednoprocesorovém Linuxu se řeší zamaskováním přerušení
  - Multiprocesing je podporován pomocí spin-locks
  - Linux poskytuje:
    - Semafory pro řešení dlouhých kritických sekcí a spin-locks pro krátké
- Synchronizace v Pthreads
  - Pthreads podporují
    - binární semafory
    - monitory s podmínkovými proměnnými

OSTRAVSKÁ UNIVERZITA
PŘÍRODOVĚDECKÁ FAKULTA

Děkuji za pozornost