6. Synchronizační nástroje - semafor

- zobecnění zámků pro vzájemné vyloučení
- Dijkstra
- abstraktní, procesorově nezávislé
- implementace v jádře nebo knihovnách

6.1 Binární semafor (zámek, binary semaphore):

init(sem, v)	inicializace semaforu *sem* na hodnotu *v*=0,1
lock(sem)	zamčení, *atomická operace* čekání na nulovou hodnotu a nastavení *v* na 1
unlock(sem)	odemčení, nastavení *v* na 0 a odblokování procesů čekajících v operaci *lock()*

- jako zámek strážící kritickou sekci, pouze dva stavy
- **nelze číst hodnotu** (nemělo by ani smysl ji číst, zdůvodněte!)
- čekání v operaci *lock(sem)* je pasivní
- operace *lock(sem)* a *unlock(sem)* jsou atomické
- odemykat může jiný proces než zamknul (předávání zámku)
- **Silný/slabý semafor** - nepodléhá/podléhá stárnutí
- **Mutex** – speciální binární semafor určený pouze pro vzájemné vyloučení, při zamčení má identifikovaného vlastníka, pouze vlastník ho může odemknout (nutné pro řešení inverze priority)

Použití:

a) pro vzájemné vyloučení:

```
    init(sem, 0);       /* volný */
    while (1) {
        lock(sem);                          ENTRY
        kritická sekce
        unlock(sem);                        EXIT
        výpočet
    }
```

b) pro signalizaci událostí (nevhodné):

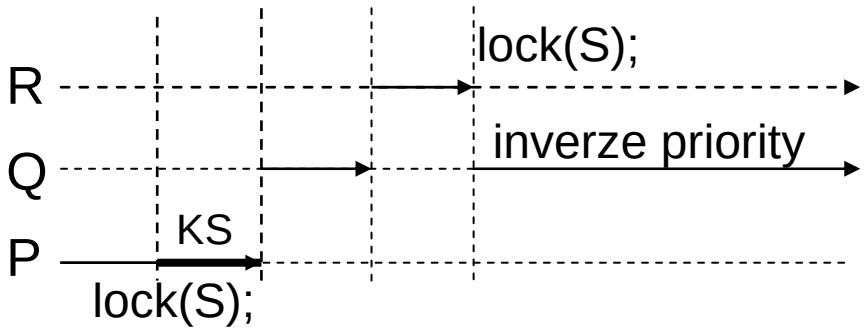
```
    init(sem, 1);              /* zamčený */

P1:                          P2:
    ...                      unlock(sem);
    lock(sem);         /* čeká až  P2 provede unlock() */
```

Problém: co když P2 provede *unlock()* vícekrát a P1 nestihne *unlock()*? → signalizace události se ztratí → obecný semafor

Problematika implementace binárních semaforů:

- *rekurzivní* binární semafor (mutex)
  Motivace: složité knihovny, např. standardní V/V pro C:
  printf() - musí zamknout *stdout*
  volá *putchar()* - musí zamknout *stdout*, uváznutí
  → pokud je zámek zamčen stejným procesem, pouze se inkrementuje čítač úrovně rekurze, při odemykání se zmenšuje, zámek se odemkne až při 0
- *implementace na úrovni uživatelského režimu (viz kap. 5)*
- *inverze priority* - proces s nižší prioritou blokuje provádění procesu s vyšší prioritou

Kdy nastává inverze priority?
Proces s vyšší prioritou (h) nemůže vstoupit do kritické sekce, protože je v kritické sekci proces s nižší prioritou (l) a neběží, protože jsou v systému procesy s prioritou $p > l$ (pokud $p < h$, jedná se o inverzi priority těchto procesů proti h).



priorita P=l, Q=p, p>l &&p<h, R = h

Řešení:

- *dědění priority (priority inheritance)* - po dobu provádění kritické sekce je priorita procesu v kritické sekci zvýšena na max. prioritu všech čekajících:
  + pokud žádný proces nečeká, zůstává priorita procesu v kritické sekci nezměněná (neovlivňuje chování systému)
  - musí se dynamicky upravovat při každém blokujícím zamčení
- *horní mez priority (priority ceiling)* - po dobu provádění kritické sekce je nastavena vždy statická pevná priorita:
  + pevně deklarovaná priorita je jednoduchá na implementaci
  - procesu se musí zvyšovat priorita vždy (i když to není nutné)

Pro které prostředky?

binární semafor, mutex, monitor - ano
obecný semafor, condition - ne, není jasné, kdo prostředek vlastní

Binární semafor ve vláknech POSIX 1003.1c:

```
#include <pthread.h>
typ pthread_mutex_t
```

inicializace statická:

```
pthread_mutex_t mutex =
                      PTHREAD_MUTEX_INITIALIZER;
```

dynamická:

```
int pthread_mutex_init(pthread_mutex_t *m,
                       pthread_mutexattr_t *attr);

pthread_mutex_t mutex;
...
pthread_mutex_init(&mutex, NULL);
```

použití:

```
int pthread_mutex_lock(pthread_mutex_t *m);
int pthread_mutex_unlock(pthread_mutex_t *m);
int pthread_mutex_trylock(pthread_mutex_t *m);
```

```
    pthread_mutex_lock(&mutex);     /* acquire */
    ...       /* kritická sekce */
    pthread_mutex_unlock(&mutex); /* release */
```

Pořadí zamčení není definováno, záleží na plánovacím algoritmu.

zrušení:

```
    pthread_mutex_destroy(&mutex);
```

Podrobněji viz studijní opora k vláknům na privátních stránkách.

6.2 Obecný semafor (číselný, counting, general)

Počáteční hodnota určuje „kapacitu“ semaforu – kolik jednotek zdroje chráněného semaforem je k dispozici. Jakmile se operací *down()* zdroj vyčerpá, jsou další operace *down()* blokující, dokud se operací *up()* nějaká jednotka zdroje neuvolní.

Definice:

init(sem, v)	inicializace semaforu *sem* na hodnotu *v*>=0
down(sem)	zamčení, *atomická operace* čekání na hodnotu > 0 a pak zmenšení o 1
up(sem)	odemčení, zvětšení o 1 a odblokování případného čekajícího procesu

Edsger W. Dijkstra (1965) – P(rolaag) = down, V(erhoog) = up,
Per Brinch Hansen - **wait**, **signal**
Pamatuje si počet *up()* a *down()*

Variantní definice (povoluje zápornou hodnotu):

P(S):	zmenšení hodnoty o 1, pokud je hodnota < 0, čekání
V(S):	zvětšení hodnoty o 1, pokud je hodnota <= 0, odblokování čekajícího procesu

Použití:

a) použití pro vzájemné vyloučení:

```
    init(sem, 1);
    while (1) {
        down(sem);                    ENTRY
        kritická sekce
        up(sem);                      EXIT
        výpočet
    }
```

V tomto použití je „ekvivalentní“ binárnímu semaforu. Nicméně jej v praxi takto používat nesmíme!

Důvody proč mutex:

1. **Inverze priority** při zamykání (nelze řešit).
2. **Rekurzivní deadlock** – co když zamkne zámek ten samý proces, který už ho má zamčený? U binárního zámku lze detekovat (je majitel zámku), u obecného nelze detekovat (z definice může kdokoli dělat libovolně krát operaci *down()*).
3. **Efektivnější** u relaxovaných paměťových modelů (acq/rel).
4. **Deadlock při ukončení procesu** – co když proces, který má zámek zamčen, skončí bez uvolnění zámku? U binárního lze detekovat a řešit, u obecného nelze (není řečeno, kdo v případě zablokovaného semaforu provede operaci *up()*, může to být kterýkoli jiný proces).
5. **Náhodné uvolnění** – co když se párová operace *down()* zratí? Obecný semafor pak pustí příště dva procesy do kritické sekce, u binárního semaforu se nic nestane.

b) použití pro signalizaci událostí:

```
    init(sem, 0);

P1:                         P2:
    ...                     up(sem);
    down(sem);          /* čeká až  P2 provede up() */
```

Oproti binárnímu semaforu bezpečné, žádná událost se nemůže ztratit!

c) hlídání zdroje s definovanou kapacitou N:

```
    init(sem, N);

Pi:
    down(sem);          /* pokud je volno, pokračujeme dále */
    ...                 /* nejedná se o kritickou sekci,
                           je zde až N procesů současně! */
    up(sem);            /* uvolníme místo */
```

Například restaurace s počtem N míst.

Implementace (povoluje pouze kladné hodnoty):

```
struct SEMA {
    struct SLOCK lock;      /* binární zámek */
    int value;              /* hodnota */
    queue_t queue;          /* fronta procesů */
};

void down(struct SEMA *s)
{
    zakázání přerušení
    spin_lock(&s->lock);
    while (s->value <= 0) {
        append(s->queue, current_pcb);
        spin_unlock(&s->lock);
        switch();   /* pozastavení, plánovač */
        spin_lock(&s->lock);
    }
    --s->value;
    spin_unlock(&s->lock);
    povolení přerušení
}

void up(struct SEMA *s)
{
    zakázání přerušení
    spin_lock(&s->lock);
    if (!empty(s->queue)) {    /* while */
        append(ready_q, get(s->queue));
    }
    ++s->value;
    spin_unlock(&s->lock);
    povolení přerušení
}
```

Číselný x binární semafor

binární = 2 číselné + sdílené proměnné (domácí úkol)
Lze naopak?

```
semaphore S1,S2,S3;
int value;              /* hodnota č.semaforu */

init:
    init(S1, 0);
    init(S2, 1);
    init(S3, 0);

down:
    lock(S3);
    lock(S1);
    value = value - 1;
    if (value < 0) {
        unlock(S1); /* povolit přístup k value */
    /* pouze jeden proces může být zde, pokud
       bude unlock(S2) proveden dříve, nevadí -
       počáteční stav je 1, jednou může být
       signalizováno bez ztráty */
        lock(S2);   /* čekání na uvolnění */
    } else unlock(S1);
    unlock(S3)

up:
    lock(S1);
    value = value+1;
    if (value <= 0) unlock(S2);
    unlock(S1);
```

1. Proč je zde S1?
2. K čemu je S2?
3. Je nutný S3?

6.4 Klasické synchronizační úlohy

1. Vzájemné vyloučení (Mutual Exclusion)
2. Producent/konzument (Producer/Consumer, Bounded Buffer)
3. Čtenáři/písaři (Readers/Writers)
4. Pět filozofů (Dining philosophers)

2. Producent/konzument

- producenti produkují data do sdílené paměti, konzumenti je z ní odebírají
- konzumenti musí čekat, pokud nic není vyprodukováno
- producenti musí čekat, pokud je paměť plná
- operace s pamětí musí být synchronizovány

```
semaphore_t empty, full;
mutex_t mutex;
DATA v;
shared buffer[N];
shared int get, put;

init(empty, 0);
init(full, N);
init(mutex, 0);
...
producent                       konzument
while (1) {                     while(1) {
    v = produkuj data;              down(empty);
    down(full);                     lock(mutex);
    lock(mutex);                    v = buffer[put];
    buffer[get] = v;                put = (put+1)%N;
    get = (get+1)%N;                unlock(mutex);
    unlock(mutex);                  up(full);
    up(empty);                      zpracuj data v;
}                               }
```

Je důležité pořadí? Je nutný mutex? Jsou nutné empty a full?

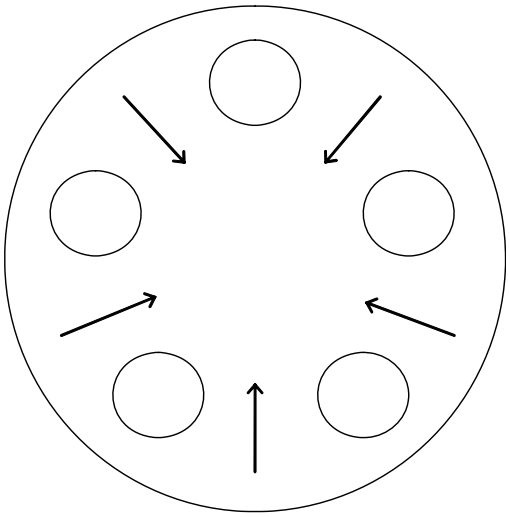
3. Čtenáři/písaři

- přístup ke sdíleným datům
- čtenář pouze čte data
- písař čte a zapisuje
- vzájemné vyloučení všech je příliš omezující:
    - více čtenářů současně
    - pouze jeden písař

```
mutex_t read, write;
int readers;

init(read, 0);
init(write, 0);
...
čtenář                          písař
while (1) {                     while(1) {
    lock(read);
    if (++readers == 1)
        lock(write);                lock(write);
    unlock(read);

    čtení dat                       operace s daty

    lock(read);
    if (--readers == 0)
        unlock(write);              unlock(write);
    unlock(read);
}                               }
```

Problém: stárnutí písaře - omezení předbíhání čtenáři
Domácí úkol: úplné řešení (je třeba použít více sdílených proměnných a semaforů) – poměrně složité
Speciální synchronizační nástroje – rwlock, RCU (read-copy-update)

4. Pět filozofů



5 filozofů, 5 vidliček, 5 talířů

Problémy:

- vyhladovění – je třeba zajisti, že když chce jíst, dostane v konečném čase najíst a nebude systematicky předbíhán jinými filozofy
- uváznutí – všichni přijdou ke stolu a uchopí levou vidličku, nikdo nemůže jíst

```
semaphore_t fork[5];
init(fork[*], 1);
while (1) {          /* i = číslo filozofa */
     myslí
     down(fork[i]);
     down(fork[(i+1)%5];
     jí
     up(fork[i]);
     up(fork[(i+1)%5];
}
```

Nastává uváznutí - všechny procesy provedou první *down(fork[i])*

Bez uváznutí:

```
mutex_t mutex;
while (1) {
    myslí
    lock(mutex);    // atomické získání vidliček
    down(fork[i]);
    down(fork[(i+1)%5];
    unlock(mutex);
    jí
    up(fork[i]);
    up(fork[(i+1)%5];
}
```

Pokud je mutex silný, nenastává hladovění, ale je neefektivní (může být dostatek vidliček pro jiného filozofa u stolu, ale *mutex* zůstane zamčený a tím brání ostatním jíst, např. vidl. 2/3 zrovna jí, vidl. 1 dostanu, na přidělení vidl. 2 zůstanu čekat, vidl. 4/5 jsou volné, ale nebudou použity, dokud se neuvolní vidl.2)

Lepší řešení – nepustíme ke stolu více než 4:

```
semaphore_t fork[5], total;
init(fork[*], 1)
init(total, 4);    // povolí max. 4 přidělení
while (1) {
    myslí
    down(total);
    down(fork[i]);
    down(fork[(i+1)%5];
    jí
    up(fork[i]);
    up(fork[(i+1)%5];
    up(total);
}
```

Pokud jsou semafory silné, řeší i hladovění, jinak je třeba doplnit (použít sdílené proměnné).