Semafory

Z FITwiki

Semafory, vlastnosti a typické použití (binární, obecné).

Obsah

Semafor Obecně

- Synchronizační nástroj poskytovaný operačním systémem
- Nepotřebuje aktivní čekání
- Nejjednodušší implementace je struktura:

```
type semaphore = record
  count: integer;
  queue: list of process
end;
var S: semaphore;
```

Typy semaforů

Binární semafor

je zámek střežící kritickou sekci. Pouze dvě hodnoty zamčeno/odemčeno.

- Operace `init()`, `lock()` a `unlock()`.
- Nelze číst jeho hodnotu (nemá smysl, mohla by se změnit potom).
- Pasivní čekání ve funkci `lock()`.
- Operace `lock()` a `unlock()` jsou atomické.
- Odemykat může i jiný proces než ten, co zamknul (předávání zámku).
- Silný/slabý semafor - má/nemá stárnutí
- Majitel zámku

- použití:
  - pro vzájemné vyloučení:

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

  - 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

Mutex

je speciální binární semafor pro vzájemné vyloučení, který nelze předávat (pouze vlastník ho může odemknout)

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.

- Operace `init(v)` - inicializace semaforu sem na hodnotu v>=0
- Operace `down()` - zamčení, atomická operace čekání na hodnotu > 0 a pak zmenšení o 1, potom může pokračovat.
- Operace `up()` - odemčení, zvýší hodnotu o 1 a tím případně odblokuje další proces.
- Používá pasivní čekání ve funkci `down()`.
- Variantní definice (povoluje zápornou hodnotu):
  - down() - zmenšení hodnoty o 1, pokud je hodnota < 0, čekání
  - up() - zvětšení hodnoty o 1, pokud je hodnota <= 0, odblokování čekajícího procesu
- Implementace operace down:

```
down(S):
  S.count--;
  if (S.count<0) {
    block this process
    place this process in S.queue
  }
```

- Implementace operace up:

```
up(S):
  S.count++;
  if (S.count<=0) {
    remove a process P from S.queue
    place this process P on ready list
  }
```

- Použití:

- pro vzájemné vyloučení - nepoužívat, důvody:
  - **inverze priority** při zamykání (nelze řešit).
  - **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())
  - **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).
  - **náhodné uvolnění** – co když se operace down() ztratí? Obecný semafor pak pustí příště dva procesy do kritické sekce, u binárního semaforu se nic nestane.

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

- pro signalizaci událostí (udrží i počet neobsloužených, oproti binárnímu semaforu bezpečné, žádná událost se nemůže ztratit)

```
init(sem, 0);
P1:      P2:
...      up(sem);
down(sem);  /* čeká až  P2 provede up() */
```

- 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 */
```

Problémy implementace semaforů

v UNIXových systémech není, používá se monitor. Obecně je implementován pomocí spinlocku a testování hodnoty čítače. Pozastavené procesy se přidávají na konec fronty.

Rekurzivní binární semafor

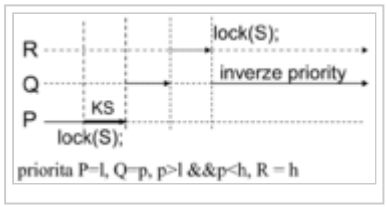
- řeší problém pokud se v rámci funkce volá jiná funkce a obě zamykají semafor (printf zamkne stdout, volá putchar, který chce taky zamknout stdout)
- implementuje se jako čítač rekurze. Pokud zámek zamkl stejný proces tak se pouze mění tento čítač a ne zámek.
- semafor může být odemknut pouze pokud je hodnota čítače rovna nule.

Implementace na úrovní už. režimu

- jako služba jádra je pomalé
- nelze použít aktivní čekání
- Řešení: jednoduchý zámek na uživatelské úrovni plus pasivní čekání na změnu hodnoty v jádře

Inverze priority

- proces s nižší prioritou blokuje proces s vyšší prioritou



- Formálněji: 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).
- Řešení:
  - **dědění priority** (priority inheritance) - po dobu provádění KS je priorita prováděného procesu zvýšena na max. prioritu všech čekajících procesů (**pozitivum:** pokud žádný proces nečeká, zůstává priorita procesu v kritické sekci nezměněná (neovlivňuje chování systému), **negativum:** musí se dynamicky upravovat při každém blokujícím zamčení)
  - **horní mez priority** (priority ceiling) - po dobu provádění KS je prováděnému procesu nastavena statická priorita (**pozitivum:** pevně deklarovaná priorita je jednoduchá na implementaci, **negativum:** procesu se musí zvyšovat priorita vždy (i když to není nutné))
  - Řešení funguje pouze pro binární semafor, mutex, monitor. U obecného semaforu a condition to nelze, protože není jasné, kdo prostředek vlastní.

Binární semafor lze simulovat dvěma číselnými se sdílenými proměnnými; obecný lze simulovat třemi binárními a sdílenou hodnotou

Použití semaforů

Podmíněná kritická sekce

- Použití semaforů řeší výlučný přístup do kritické sekce:

```
Process Pi:
repeat
  down(S);
    CriticalSection
  up(S);
    RemainderSection
forever
```

Bariéra

je místo, na kterém čekají procesy, dokud se nesejdou všechny.

- typické u paralelních algoritmů
- čítač s počátečním stavem N (počet procesů), proces v bariéře dekrementuje čítač a čeká, až se čítač posune na 0 proces pokračují dále

Implementace semaforu v POSIX

```
typedef struct {
  pthread_cond_t cond;   /* pro čekání */
  pthread_mutex_t mutex; /* vzájemné vyloučení*/
  int value;             /* hodnota semaforu */
} sema_t;
```

```
void sema_init(sema_t *sema, int value)
{
  pthread_mutex_init(&sema->mutex, NULL);
  pthread_cond_init(&sema->cond, NULL);
  sema->value = value;
}
```

```
void sema_down(sema_t *sema)
{
  pthread_mutex_lock(&sema->mutex);
  while (sema->value <= 0) {
      pthread_cond_wait(&sema->cond,
                                &sema->mutex);
  }
  --sema->value;
  pthread_mutex_unlock(&sema->mutex);
}
```

```
void sema_up(sema_t *sema)
{
  pthread_mutex_lock(&sema->mutex);
  ++sema->value;
  pthread_cond_signal(&sema->cond);
  pthread_mutex_unlock(&sema->mutex);
}
```

Implementace semaforu ze slajdů

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
}
```

Synchronizacne ulohy

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

- Řešení pomocí kruhového bufferu:

```
semaphore_t empty, full;
mutex_t mutex;
DATA v;
shared buffer[N];
shared int get, put;
init(empty, 0);
init(full, N);
init(mutex, 1);
```

```
producent
while (1) {
  v = produkuj data;
  down(full);
  lock(mutex);
  buffer[get] = v;
  get = (get+1)%N;
  unlock(mutex);
  up(empty);
}
```

```
konzument
while(1) {
  down(empty);
  lock(mutex);
  v = buffer[put];
  put = (put+1)%N;
  unlock(mutex);
  up(full);
  zpracuj data v;
}
```

Uvedené řešení je bez deadlocku, ještě se dá dělat pomocí neomezeného bufferu (není zde modulo, ale prostě se stálé zvětšuje index...), kde deadlock hrozí, pokud je buffer prázdný a konzument provede down(empty) před provedením down(full) producentem.

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 Obecně lze řešit tento problém prioritou čtenářu, nebo písařu. Je tu však i třetí algoritmus, který hladovění odstraňuje. Lze ho přečíst na wiki (http://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem) .

Pět filozofů

```
semaphore 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). Uvedené řešení je bez deadlocku, pokud by zde nebyly operace down(total) a up(total), tak by mohl nastat deadlock, pokud by všichni filozofové zvedli svoji levou vidličku.

Alternativa: Nechat liché filozofy uchvátit nejdříve levou vidličku a sudé filozofy nejdříve pravou.

Kategorie: Státnice 2011 | Pokročilé operační systémy

- Stránka byla naposledy editována 31. 5. 2016 v 10:47.