36. SEMAFORY

- synchronizace – kooperace procesů pomocí synchronizačních okamžiků
  - procesy na sebe čekají, nebo čekají až jeden dokončí nějakou operaci
- race condition – chyba která vznikla rozdílnou rychlostí vykonávání procesů, obvykle nad sdílenou pamětí
  - proces čte data ze sdílené paměti dříve než je tam jiný proces stihl zapsat
  - → nedeterminismus v pořadí provádění operací
- atomické instrukce / operace – vykoná se celá, nebo se nevykoná vůbec
  - např. read a write v CPU a nebo i všechny operace nad semafory a mutexy musí být atomické: lock(), unlock(), up(), down(), ..
- vzájemné vyloučení – způsob synchronizace (tu kooperace procesů) k tomu aby v jednom místě byl pouze jeden proces nebo aby pouze jeden proces prováděl nějakou operaci
- kritická sekce – část kódu či programu, kde musí být zaručeno vzájemné vyloučení
  - nesmí tam proces skončit a nebo když ano, tak musí uvolnit všechny prostředky a zámky atd...
- bezpečný algoritmus – zaručuje vzájemné vyloučení v kritických sekcích
- živý algoritmus – neobsahuje uváznutí (deadlock), blokování ani stárnutí / hladovění
  - ↳ čekají na uvolnění zdroje které alokoval proces který taky čeká
  - ↳ neuvolnil zámek a blokuje ostatní a KS je volná
  - čeká na podmínku, která nenastává (je předbíhán) ↲
- synchronizace
  - techniky: čtení a zápis, přerušení (zakazování), ...
  - nástroje: bin. semafor, obecný semafor, mutex, futex, monitor, signály, ...

Vzájemné vyloučení pro jeden CPU

- není zde souběžné provádění, pouze pseudosouběžné – procesům/vláknům je v rychlém sledu přidělován procesor vždy na krátkou dobu (střídají se na něm) a je neustále přepínán kontext a z dlouhodobého hlediska to vypadá že tam jsou 2 CPU a že ty procesy běží paralelně ale ve skutečnosti ne
- odpadá zde tedy problém toho že běží dva procesy na dvou CPU a vzájemně si třeba přepisují paměť sdílenou a nebo chtějí oba naráz vykonávat nějakou operaci
- je zde ale problém že když je proces v KS a třeba vypisuje něco na výstup a najednou je vyvoláno přerušení a je přepnut kontext a do té KS přijde jiný proces a ten taky vypíše něco na výstup
- musí se tedy v těch KS zajistit taky to vzájemné vyloučení

- uživatelské procesy / vlákna (userspace) – zde nelze využít zakazování přerušení a tedy přepnutí kontextu
  - musí se použít zámky – mutexy
  - jsou to ale nástroje jádra OS a jejich používání způsobuje volání jádra a to má velkou režii ⇒ používají se futexy – volání jádra jen při zamykání, při testování uzamčenosti ne
- jaderné procesy / vlákna
  - zakázání přerušení a tedy zakázání přepnutí kontextu je řešení ale má to režii protože probíhá komunikace s řadičem přerušení a navíc nám nepojedou I/O operace
  - když nepřijde proces přerušit tak si vždy dokončí kritickou sekci ale to řešení není ideální a lepší je použít mutexy – zde jsme v jádře OS přímo takže není nutné používat futexy

Vzájemné vyloučení pro více CPU

- zde než zakazování přerušení a přepnutí kontextu na CPU nijak nepomůže jelikož naráz do KS mohou vstupovat procesy z jiných CPU

- plánování přidělování procesoru
  - preemptivní – OS má nad přidělováním plnou kontrolu a může přerušit (přepnout kontext) kdykoliv
  - nepreemptivní – kontext se přepne až když proces dokončí co chtěl a řekne OS že může přepnout

- krátkodobé vyloučení – lze i aktivní čekání a použití atomických instrukcí a třeba spin locku s použitím atomické instrukce test and set

```
Bool testAndSet (bool flag) {
    bool tmp = flag;
    flag = true;
    return tmp;
}
```

```
void spinlock (bool flag) {
    while (testAndSet(flag));
}                  ↳ aktivní čekání
```

- je bez blokování a uvíznutí ale je zde stárnutí

- dlouhodobé vyloučení – používá se binární semafor a mutex

Binární semafor

- ideální pro vzájemné vyloučení, ne pro signalizaci (zámky budou zamčené a proces je bude jen odemykat až jiným procesům něco signalizoval – když dovolí odemkne a nikdo zrovna nechtěl zamykat tak se to ztratí jak to nemá počítadlo) a taky na vracování nějaké kapacity ⟹ na tyto dvě věci je obecný semafor
- pouze dva stavy – odemčeno a zamčeno a obsahuje frontu čekatelů – i pro bariéru je lepší obecný
- operace lock(), unlock() a init() a destroy() (atomické)

$$\left.\begin{array}{l}\text{lock()}\\ \quad\vdots\\ \text{unlock()}\end{array}\right\}\text{KS}$$

- odemknout může někdo jiný než ten co zamknul → třeba procesy jen vstupují do KS a něco obslouží a odejdou a jeden proces je tam pouští třeba když přijde paket atd...
- chybí ukazatel na toho kdo zamknul → jen mutex to má
- pthread_mutex_t – v POSIXu

Mutex

- odemknout může jen ten kdo zamknul, tato zodpovědnost tedy není přenositelná
- obsahuje identifikátor toho co zamknul

Obecný semafor

- v UNIXu není a nahrazuje se monitory → testování hodnoty čítače + spinlock
- operace init(N), down() a up() → nebo také wait() a signal()
- obsahuje číselnou hodnotu která se nastavuje při inicializaci (N) a ta určuje jakou má daný semafor kapacitu
- do KS tedy může naráz vstoupit až N procesů
- je to velmi vhodné pro signalizaci, pro simulování kapacity nebo pro bariéru kde na sebe procesy čekají než se všichni sejdou
- obsahuje také frontu čekatelů ale neobsahuje identifikátory těch co zamknuli, to má jen mutex
- up() zvyšuje to počítadlo a pokud někdo je pozastavený a čeká ve frontě tak jej vyjme a vloží do fronty připravených
- down() se dívá jestli je N>0 a když jo tak sníží a pokračuje a když ne tak se vloží do fronty čekajících a dá OS vědět, že bude čekat a že může přepnout kontext

– obecný semafor není vhodný pro vzájemné vyloučení protože když se někde omylem provede dvakrát up() tak se pak do KS vpustí dva procesy naráz → na tohle je nejlepší binární semafor nebo mutex – než když zamkne 2x jeden proces – to nezjistíme komu ten zámek patří

– pokud by měl jeden a ten samý proces vícekrát (rekurzivně) zamykat jeden zámek tak se to realizuje jako čítač rekurze – semafor se může odemknout když je čítač rekurze = 0

→ obecný semafor pro vzájemné vyloučení by měl N=1 ale jak píšu, vhodné to není hned z několika důvodů – i třeba když proces zamkne a skončí tak nezjistíme že to zamknul proces co skončil protože tam není uveden kdo zamknul a nemůže se to vyřešit – u mutexů ano

⟹ deadlock při ukončení – u mutexů se to řeší

Problém inverze priorit

– může nastat problém že do ~~systému~~ KS přijde proces a zamkne jí ale má nízkou prioritu a tak se přepne kontext a do KS chce přijít proces s vyšší prioritou ale nemůže protože ji zamknul ten proces s nižší prioritou a ten jí ale neodemkne protože nedostane procesor protože má nízkou prioritu

– nejde použít zakázat přerušení – více CPU

– řešení:

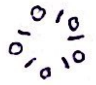
dědění priorit – pokud jsou v plánovači procesy tak se vezme ten s největší prioritou a ta se přidělí tomu procesu co jde do KS
– nevýhodou je to že se to musí hledat ta priorita no
– výhoda je že když tam žádné procesy nejsou tak se nic nenastavuje a odpadá režije

horní mez priorit – priorita se nastavuje vždy na nějakou nejvyšší hodnotu kterou nemůže mít žádný proces
– výhodou je že se nic nehledá
– nevýhodou je že se to nastavuje neustále i když žádné další procesy nejsou v pořadí → režije

– algoritmy: ◦ čtenáři a písaři – více čte a nikdo nesmí zapisovat
– jen jeden zapisuje a nikdo nesmí číst ani zapisovat

◦ konzumenti a producenti – producenti produkují a konzumenti konzumují dokud tam něco je
– když je plno tak producent nesmí produkovat

◦ 5 filozofů večeřících – 5 vidliček



OBECNÝ SEMAFOR V POSIX – pomocí monitoru

— je to teda implementováno jako monitor – jeden mutex a jedna podmínka na čekání a jedna hodnota

↙ měla by tu být queue

```
typedef struct {
    pthread_cond_t cond;
    pthread_mutex_t mutex;
    int value
} 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 up(sema_t *sema){
  pthread_mutex_lock(&sema->mutex);
  ++sema->value;
  pthread_cond_signal(&sema->cond);
  pthread_mutex_unlock(&sema->mutex);}
```

že se vybere z fronty

```
void 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);
}
```

PSEUDOKÓD SEMAFORU (obecného)

```
STRUKTURA SEMAFOR {
        MUTEX M;
        CISLO N;
        FRONTA Q;
}

                       CISLO
INIT-SEMAFOR (SEMAFOR,       ) {
      ~~M~~ SEMAFOR -> M~~UTEX~~ -> ODEMKNI();
  SEMAFOR -> ~~CISLO~~ N = CISLO
      SEMAFOR -> Q -> INICIALIZUJ();
}

UP (SEMAFOR) {
   SEMAFOR -> M -> ZAMKNI()
   SEMAFOR -> Q -> VZOMY-Z-FRONTY();
   SEMAFOR -> N++;
   SEMAFOR -> M -> ODEMKNI();
}
```

zakáž přerušení

```
DOWN
~~UP~~ (SEMAFOR) {
      SEMAFOR -> M -> ZAMKNI ();
      IF (SEMAFOR -> N ≤ 0) {
         SEMAFOR -> Q -> PRIDEJ-SEBE();
         SEMAFOR -> M -> ODEMKNI();
         USPI_SE_A_CEKEJ_NA_PREPNUTI_KONTEXTU();
         SEMAFOR -> ~~Q~~ -> ZAMKNI();
      }
      SEMAFOR -> N--;
      SEMAFOR -> M -> ODEMKNI();
}
```

povol přerušení

⟹ jsou to atomické instrukce / operace

nebo taky:

```
up(s) {
  s.n++;
  vyjmi-z-front-cekajicich;
  dej-do_fronty-pripravenych;
}

down(s) {
  if(s.n <= 0) {      / a zablokuj proces
  dej-do-fronty-cekajicich;
  }
  s.n --;
}
```

– Jak zaručit atomicitu instrukcí/operací jako lock(), unlock(), up(), down() atd.? ⟹ při jejich provádění se zakáže přerušení – jsou to jaderné (OS) instrukce a tak se může, navíc se při nich nedělá 10 let to ani nevadí, jen je trochu režie při komunikaci s řadičem přerušení

– deadlock při ukončení → u mutexů není – víme o tom pořád
– uvolní se zámky a posléze všech ~~ott~~ ukončujících se procesů