48) INTERAKCE MEZI PROCESY A TYPICKÉ PROBLÉMY PARALELISMU

- procesy spolu
  - kooperují (společně řeší nějaký problém – třeba každý řeší část)
  - soupeří (o přístup ke sdílené paměti, prostředkům, obecně do KS, ...)
  - → pro kooperaci je potřeba synchronizace
  - → při soupeření je nutné zajistit výlučný přístup a tedy vzájemné vyloučení
- synchronizace – zaručuje nějaké správné pořadí provádění operací
  - např. jeden proces stromu musí počkat až oba jeho synové zapíšou do svých buněk paměti a on je pak může přečíst a dále s nimi pracovat
  - nesmí to být naopak
- komunikace procesů
  - sdílenou pamětí – všechny procesy mají společnou paměť a společný adresný prostor
    - mohou skrz tu paměť simulovat zasílání zpráv
    - pokud je to nad více CPU tak jsou propojeny sdílenou sběrnicí k jedné paměti
    - pokud je jen 1CPU, pak se procesy střídají přepínáním kontextu na tom CPU a mají také jen jednu sdílenou paměť
      - → opět se může simulovat zasílání zpráv
    - přístupy
      - EREW – moc omezující
      - CREW – používá se
      - ERCW – hloupost
      - CRCW – náročný, více chce zapsat? (COMMON, ARBITRARY, PRIORITY)
  - zasíláním zpráv – každý proces/procesor má svůj vlastní adresný prostor a celkově paměť
    - komunikace pomocí kanálů
    - každý procesor je třeba jinde v síti
    - lze simulovat virtuální sdílenou paměť
- → v MIMD třeba (MULTIPLE INSTRUCTION MULTIPLE DATA)
  - multiprocesory – sdílená paměť – paralelní architektura
  - multicomputery – oddělená paměť a komunikace pomocí zpráv – distribuovaná architektura

Realizace vzájemného vyloučení

1) Pomocí HW

- v rámci jednoho CPU lze zakázat přerušení při vstupu do KS a opět jej povolit při výstupu
- uživatelská vlákna v userspace to však udělat nemohou → použití mutexu nebo futexu
- systémová vlákna v jádře OS to udělat mohou ale přináší to velkou režii při komunikaci s řadičem přerušení → použití mutexu

- jisti by šlo maskovat přerušení a obsluhovat ji později → není ideální
- na více CPU nám přerušení (ecklar) na jednom CPU nepomůže
- další možností jsou **atomické instrukce** – jsou implementovány v HW jako jediná atomická instrukce
  - provedou se buď celé nebo vůbec – nemohou být v půlce přerušeny
  - test and set a swap

```
int testAndSet(int *target){
    int tmp = *target;
    *target = 1;
    return tmp;
}
```

PSEUDOKÓDY VC

```
void swap(int *a, int *b){
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

- tyto instrukce lze použít i při **aktivním čekání – spin lock**

2) Pomocí nástrojů OS

- Obecný semafor
  - v POSIX je realizovaný pomocí monitoru – 1 mutex, 1 proměnná a 1 podmínka
  - má operace up() a down()
  - má nějakou počáteční kapacitu $N \geq 0$ – když ≥ 0 tak vstupují do KS
  - ~~down~~() (down) – sníží kapacitu N a pokud je < 0 tak se vloží do fronty čekajících procesů na uvolnění semaforu → je blokován a dá vědět OS že může přepnout kontext
  - up() – zvýší kapacitu N a pokud je ≤ 0 tak vezme proces z fronty blokovaných a vloží ho do fronty připravených
  - v KS může být naráz až N procesů
  - semafor nemá informaci o tom kdo jej vlastní – může jej odemknout kdokoliv

Použití:
- vzájemné vyloučení (NEVHODNÉ) – N=1 ale není vhodné viz POS, může být omylem navýšena kapacita a může se do KS vpustit více procesů × MUTEX
  - navíc nemáme info o tom kdo zamkl a pokud skončí tak mu tento zámek nemůžeme odebrat → deadlock při ukončení × MUTEX
- synchronizace/signalizace (VHODNÉ) – N=0 a jeden proces vpouští ostatní a synchronizuje je
  - jeden proces jim odemyká – up()
  - ostatní jim zamykají – down()
- simulování kapacity (VHODNÉ) – N = kapacita zdroje

producenti/konzumenti (neomezený buffer) – producent pořád konzumuje a zvyšuje tak N
- konzument pořád konzumuje a snižuje tak N
- zároveň je tam semafor S (0/1) který hlídá přístup k bufferu

producenti/konzumenti (omezený buffer) – je přidán další semafor E který je nastaven na max. kapacitu bufferu a pokud je vyčerpána tak producent nemůže produkovat další

pole kam se produkuje a odkud se čte

- **Binární semafor (+ mutex)** – má pouze dva stavy, zamknuto a odemknuto
  - zamykání je blokující a pokud se proces pokusí zamknout již zamčený semafor je tak je vložen do fronty blokovaných na tom semaforu
  - při odemčení se vezme proces z fronty blokovaných (pokud tam je) a dá se do fronty připravených
  - operace lock() a unlock()
  - pthread_mutex_t v POSIXu (je to rovnou MUTEX)
  - opět se neví kdo zamknul a může odemknout kdokoliv

  ⇒ **mutex** – binární semafor který nese identifikátor zamykajícího procesu a jen ten může odemknout
  → řeší problém deadlocku při ukončení

Použití:
- vzájemné vyloučení – ideální – hlavně MUTEX
- signalizace + kapacita – není vhodné – 2x unlock() když nikdo nechce dovnitř tak se jedno ztratí
  ⇒ na to lepší obecný semafor, respektive monitor

- **Monitor** – abstraktní datová struktura zapouzdřující data která tvoří kritickou sekci
  - data jsou přístupná pouze přes operace monitoru – rozhraní monitoru
  - operace zaručují vzájemné vyloučení – třeba pomocí mutexů
  - v monitoru může být v jednu chvíli jen jeden proces ↳ jinak s daty operoval nejdřív než těmito operacemi
  - jsou high-level a dosti abstraktní a tak je pro programátory práce s nimi snazší
  - je možné čekat na nějaké podmínění proměnné (pthread_cond_t *cond; + bool. prom.) uvnitř metody monitoru ⇒
    1) otestuji se podmíněnou proměnnou
    2) pokud není splněna tak proces uvolní zámek (vstup do monitoru) (pthread_cond_wait(&cond);)
    3) zavolá na podmínce wait() a začne čekat
  - v jiné (nebo stejné) metodě monitoru již proces uvolní čekající proces ⇒ budou tu 2 :O

  ⇒ to řeší dva přístupy:
  - **Hoare, Hanson = signal()** – volání je blokující a proces tedy uvolní čekajícího a sám začne čekat dokud uvolněný zase neuvolní prostředky/zámek (začne někde jinde čekat nebo opustí monitor)
    - proces který uvolnil má přednost před nově příchozími
  - **Lampson = notify()** – proces který uvolnil tak běží dál a jakmile začne někde čekat nebo odejde z monitoru (a uvolní tedy zámek) tak uvolněný soupeří s nově příchozími (nemá přednost) o zámek a až mu je přidělen tak musí znovu ověřit podmínku protože mu ji mohl někdo nově příchozí, kdo ho předběhl zneplatnit
    ⇒ může být hladovění

3) Pomocí SW (pomocí SW nástroje či nějakého algoritmu)

- **Operátor <await B -> S>**
  - problematická implementace, spíše teoretický
  - zkontroluje podmínku B a pokud platí tak vykoná atomicky sekvenci S
- **Kritické regiony (CR) a podmíněné kritické regiony (CCR)**
  - deklaruje se sdílená proměnná jako shared
  - nad takovou proměnnou je možné pracovat jen v kritickém regionu který implementačně zaručuje vzájemné vyloučení -> region (na pozadí dostane každá sdílená proměnná semafor který je ovládán vstupem do regionu)

```
var count : shared integer;
  :
region count do        <- producer třeba
      count += 1;
```

- procesy mohou chtít sdílené proměnné v opačném pořadí a může nastat deadlock
⇒ řešením jsou podmíněné kritické regiony

```
region count when B do
  :
```

- **Petersonův algoritmus**
  - udávají se kola podle procesů – máme-li 2 procesy tak je kolo 1 a kolo 2
  - proces nastaví svůj flag na to že by chtěl do KS ⇒ true
  - nastaví kolo na kolo jiného procesu
  - ve smyčce cyklí dokud je nastaveno kolo na ten jiný proces a zároveň on má nastavený flag, že chce do KS
  - jakmile jedna z toho neplatí (není kolo toho druhého nebo druhý nechce do KS) tak sám vstoupí do KS a na konci nastaví že už nechce do KS – tedy z ní vystupuje

```
  :
flag[i] = true;
turn = j;
while (turn == j && flag[j] == true);
  <KS>
flag[i] = false;
  :
```

Problémy paralelismu

- UVÁZNUTÍ (deadlock) – proces je pozastavený a čeká na přidělení prostředku (třeba na zámek) ale nikdy jej nedostane protože proces který jej má přidělený je stejným způsobem pozastavený a čeká
- UVÁZNUTÍ (livelock) – proces není pozastavený ale pouze přechází mezi vnitřními stavy a nikdy se nedostane do koncového – na to potřebuje prostředek, který mu ovšem nikdy přidělen nebude
- BLOKOVÁNÍ – proces zamkl KS ale pak už ji neodemkl a blokuje tak procesy které do ní chtějí vstoupit
- HLADOVĚNÍ – proces se snaží získat prostředek ale získávají jej jiné procesy (předbíhají ho) a na proces se nedostává -> hladoví

Synchronizační problémy

- 5 filosofů a 5 vidliček
  - každý chce nejdříve levou/pravou vidličku
- producenti/konzumenti
- čtenáři/písaři

Producent

```
while (1) {
    v = vyprodukuj_data();
    down (full);
    lock (mutex);
    buffer.add (v);
    unlock (mutex);
    up (empty);
}
```

SKRUHOVÝM BUFFREM
- bez něj by to šlo, ale o neomezeném a nebyl by tu full semafor

Konzument

```
while (1) {
    down (empty);
    lock (mutex);
    v = buffer.get()
    unlock (mutex);
    up (full);
    zpracuj_data (v);
}
```

buffer - obsahuje vyprodukovaná data
mutex - chrání buffer před víceuživatelským přístupem ⇒ vzájemné vyloučení
empty - obecný semafor
    - určuje, jestli v bufferu už něco je a konzument tedy může něco vzít
full - hlídá horní hranici bufferu - když je plný - konzument nic neodebírá a je plno tak ano, producent čeká

TICKET ALGORITHUS - další SW řešení jako Petersonův alg.

```
ticket_lock_init(int *next, int *serving) {
    *next = 0;
    *serving = 0;
}

ticket_lock-lock (int *next, int *serving) {
    int my_ticket = (*next)++;
    while (my_ticket != *serving);
}

ticket_lock-unlock (int *serving) {
    (*serving)++;
}
```