58. Interakce mezi procesy a typické problémy paralelismu (synchronizační a komunikační mechanismy).

Interakce mezi procesy

Interakci mezi procesy rozdělujeme na **soupeření** (competition) a **spolupráci** (cooperation).
Soupeření: obecně se k jednomu zdroji v daný okamžik může snažit přistupovat více procesů. Je třeba zajistit vzájemné vyloučení.
Spolupráce: procesy spolupracují a domlouvají se na něčem (synchronizace, problém rozdělení úloh, detekce správného nebo i nesprávného ukončení výpočtu, ...).

Pojmy

Kritická sekce je část programu, ve které dochází k přístupu ke sdíleným zdrojům a jejíž provádění jedním procesem vylučuje současné provádění ostatními procesy.
Vzájemné vyloučení (mutual exclusion): nanejvýš jeden (obecně k) proces je v daném okamžiku v kritické sekci.

Problémy vznikající na kritické sekci

Data race (časově závislá chyba nad daty, souběh nad daty): dva přístupy ke zdroji s výlučným přístupem ze dvou procesů bez synchronizace, alespoň jeden z těchto přístupů je pro zápis.
Uváznutí (deadlock): situace, kdy je každý proces z určité množiny procesů pozastaven a čeká na uvolnění zdroje s výlučným přístupem vlastněného nějakým procesem z dané množiny, který jediný může tento zdroj uvolnit
Blokování přístupu do kritické sekce: situace, kdy proces, jenž žádá o vstup do kritické sekce, musí čekat, přestože je kritická sekce volná.
Hladovění (stárnutí, starvation): situace, kdy proces čeká na podmínku, která nemusí nastat. V případě kritické sekce je touto podmínkou umožnění vstupu do kritické sekce. Zvláštním případem hladovění je také livelock, kdy všechny procesy z určité množiny běží, ale provádějí jen omezený úsek kódu, ve kterém opakovaně žádají o určitý zdroj s výlučným přístupem, který vlastní některý z procesů dané skupiny. Je to situace podobná uváznutí, ale s aktivním čekáním.

Požadavky pro vzájemné vyloučení (mutual exclusion)

- Pouze 1 proces se může v daném okamžiku nacházet v kritické sekci
- Proces, který není v kritické sekci, neinteraguje s ostatními procesy
- Nedochází k uváznutí ani vyhladovění procesů

- Pokud chce proces vstoupit do kritické sekce a kritická sekce je volná, tak do ní hned vstupuje
- Proces je v kritické sekci pouze časově omezenou dobu

Přístupy ke kritickým sekcím

- SW přístup - bez podpory programovacího jazyka nebo operačního systému
- HW přístup - zakázání přerušení nebo speciální instrukce
- Podpora poskytovaná operačním systémem nebo programovacím jazykem - semafory, monitory, ...

SW řešení vzájemného vyloučení

Předpoklady: jedno čtení z paměti je atomické, jeden zápis do paměti je atomický a souběžné čtení/zápisy budou v nějakém pořadí proloženy.

Dekkerův algoritmus

Dva procesy chtějí vstoupit do kritické sekce. Pokud chce proces i vstoupit do kritické sekce, je nastaveno flag[i] na true. Sdílená proměnná turn značí, který z procesů má prioritu (který je aktuálně na řadě).
Na začátku proces i nastaví svůj flag na true a chce vstoupit do kritické sekce. Pokud je flag druhého procesu false, tak druhý proces do kritické sekce vstoupit nechce. Proces tedy na nic nečeká a do kritické sekce vstupuje. Po jejím provedení nastaví prioritu na druhý proces a nastaví svůj flag na false. V případě, že ale ve stejném okamžiku chce do kritické sekce vstoupit i druhý proces, vstoupí do této sekce proces s větší prioritou. Pokud je řada na procesu i, vstoupí do kritické sekce. Pokud je řada na druhém procesu, tak proces nastaví svůj flag na false a čeká dokud nebude řada opět na něm. Až se tak stane, nastaví svůj flag opět na true.

```
shared var flag: array [0..1] of boolean;
    turn: 0..1;
repeat
 flag[i] := true;
  while flag[j] do
    if turn=j then                  // Only one process releases flag
       flag[i] := false;
       while turn=j do skip;        // wait
       flag[i] := true;
    endif
  endwhile
 critical section
 turn := j;
 flag[i] := false;
 remainder section
until false;
```

Petersonův algoritmus

Algoritmus využívá stejných proměnných jako Dekkerův algoritmus, ale funguje na trošku jiném principu. Když chce proces vstoupit do kritické sekce, nastaví svůj flag na true. Následně nastaví prioritu druhému procesu. Dokud chce druhý proces vstoupit do kritické sekce a je na řadě, proces čeká. Následně vstoupí do kritické sekce a po jejím vykonání nastaví svůj flag na false.

```
shared var flag: array [0..1] of boolean;
    turn: 0..1;
repeat
 flag[i] := true;
 turn := j;
  while (flag[j] and turn=j) do skip;
 critical section
 flag[i] := false;
 remainder section
until false;
```

Bakery (Ticket) algoritmus

Algoritmus pro vzájemné vyloučení *n* procesů. Před vstupem do kritické sekce získá každý proces přístupový lístek, jehož číselná hodnota je větší než čísla přidělená již čekajícím procesům (resp. procesu, který je již v kritické sekci). Držitel nejmenšího čísla a s nejmenším PID může vstoupit do kritické sekce (více procesů může získat lístek současně). Čísla přidělovaná procesům mohou teoreticky neomezeně růst.

HW řešení vzájemného vyloučení

Zakázání přerušení

Za normálních okolností proces běží, dokud nezavolá službu operačního systému, nebo není přerušen. Pokud zakážeme přerušení, tak zajistíme vzájemné vyloučení. Omezíme tím ale možnost procesoru prokládat jednotlivé procesy a tím se může snížit rychlost provádění.

Speciální instrukce

Jiná realizace mechanismu vzájemného vyloučení vede přes speciální instrukce, které atomicky vykonávají několik operací, konkrétně čtení a zápis. Dvěmi takovými instrukcemi jsou **test-and-set** a **swap**.

Test-and-set

Instrukce test-and-set vrátí původní hodnotu v target a zároveň ji nastaví na 1.

Test-and-set

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

Řešení přístupu do kritické sekce: máme sdílenou proměnnou lock, která bude nabývat hodnot 0 (zámek není zamčený) a 1 (zámek je zamčený). Když chce proces vstoupit do kritické sekce, tak musí počkat, až je zámek odemčený. Až je odemčený, je mu umožněn vstup do kritické sekce a zároveň se zámek uzamče pro další procesy. Po vystoupení z kritické sekce je zámek odemčen pro další procesy.

Test-and-set solution

```
shared var lock = 0;
repeat
        while testAndSet(&lock) do skip;
        critical section
        lock = 0;
        remainder section
until false;
```

Swap

Instrukce swap atomicky prohodí hodnoty v a a b.

Swap

```
void Swap(a, b)
int *a;
int *b;
{       int temp = *a;
        *a = *b;
        *b = temp;
}
```

Řešení přístupu do kritické sekce: opět máme sdílenou proměnnou lock, jejíž hodnota udává, zda je možné vstoupit do kritické sekce. Kromě toho má také každý proces svou lokální proměnnou key, která je na začátku nastavena na 1. Když chce proces vstoupit do kritické sekce, tak opakovaně prohazuje hodnotu svého klíče s hodnotou v lock. Jakmile je hodnota jeho klíče 0, vstupuje do kritické sekce. Znamená to, že lock byl nastaven na nulu, tzn. kritická sekce byla otevřená, a pomocí klíče se zase zavřela. Po provedení kritické sekce je zámek otevřen.

```
Atomic Swap Solution
shared var lock = 0;
repeat
 key := 1;
 repeat
 swap (&lock, &key);
 until key=0;
 critical section
 lock := 0;
 remainder section
until false;
```

Synchronizační mechanismy

Všechna doteď uvedená řešení využívala pro realizaci vzájemného vyloučení **aktivní čekání** (žhavení železa). Procesy, které nemohou vstoupit do kritické sekce, probíhají, čekají ve smyčce a spotřebovávají systémový čas. Následující synchronizační mechanismy jsou bez aktivního čekání.

Semafory

Semafor je synchronizační nástroj, který nevyžaduje aktivní čekání. Jedná se o celočíselnou proměnnou, která je přístupná dvěma základními atomickými operacemi:

- **lock** (také P či down) - zamknutí/obsazení semaforu, volající proces čeká, dokud není možné operaci úspěšně dokončit
- **unlock** (také V či up) - odemknutí/uvolnění semaforu

Pokud proces musí čekat na odemčení semaforu, je vložen do fronty procesů, které na toto odemčení také čekají. Tím je zamezeno aktivnímu čekání.
Pokud pro proměnnou S odpovídající semaforu platí S>0, pak je semafor odemknut (hodnota S>1 se užívá u zobecněných semaforů, jež mohou propustit do kritické sekce více než jeden proces). Pokud platí $S \leq 0$, pak je semafor zamknut a pokud $S < 0$, pak $|S|$ udává počet procesů čekajících na semaforu.
Binární semafory jsou speciálním případem obecných semaforů. Čítač v tomto případě může obsahovat jenom dva stavy a proto může být nahrazen booleovskou hodnotou.

Využití semaforů pro synchronizaci na kritické sekci:

```
semaphore S; // shared semaphore
init(S, 1); // initially S = 1
//...
lock(S);
// critical section
unlock(S);
//...
```

Semafor je struktura obsahující čítač a frontu procesů

Operace semaforu:

```
typedef struct {
  int value;
  process_queue *queue;
} semaphore;

lock(S){
  S.value--;
  if (S.value < 0){
    // remove the process calling lock(S) from the ready_queue
    C = get(ready_queue);
    // add the process calling lock(S) to S.queue
    append(S.queue, C);
    // switch context, the current process has to wait to get
    // back to the ready queue
    switch();
  }
}

unlock(S){
  S.value++;
  if (S.value <= 0){
    // get and remove the first waiting process from S.queue
    P = get(S.queue);
    // enable further execution of P by adding it into
    // the ready queue
    append(ready_queue, P);
  }
}
```

Kritické regiony (critical regions, CR) a podmíněné kritické regiony (CCR)

Kritické regiony: deklarujeme proměnnou *v* jako sdílenou proměnnou. Proměnná je přístupná pouze uvnitř příkazu *region*. Pro příkaz *region v do S;* platí, že dokud je prováděno *S*, žádný jiný proces nemůže přistoupit k proměnné *v*.

Implementace kritického regionu:
Každé sdílenné proměnné je přiřazen binární semafor (mutex), který je inicializován na 1. Operace lock a unlock semaforu jsou provedeny kolem každého regionu, který obsahuje příslušnou proměnnou.

```
- var x: shared T;
            region x do S;
- Becomes:
            var x: T;
            var xmutex: semaphore;

            P(xmutex);
            S;
            V(xmutex);
```

Problém:
Použitím více kritických regionů může dojít k uváznutí.

- 1. Proces performs : "**region x do region y do S1;**"
- 2. Proces performs : "**region y do region x do S2;**"

```
Becomes:

Process1:

   "P(x.mutex); P(y.mutex);"
   "S1;"
   "V(y.mutex); V(x.mutex);"
```

```
Process2:

   "P(y.mutex); P(x.mutex);"
   "S2;"
   "V(x.mutex); V(y.mutex);"
```

Řešení:
Řešením je použití podmíněných kritických regionů (CCR). Používáme příkazy tvaru *region v when B do S;* kde platí, že *B* je booleovský výraz, který musí platit předtím, než je provedeno *S*.

Monitory

Monitor je jeden z vysokoúrovňových synchronizačních prostředků. Zapouzdřuje data, má definované operace a jen jeden proces může provádět nějakou operaci nad chráněnými daty. Monitor poskytuje stejnou funkcionalitu jako semafory, ale je jednodušší s ním manipulovat. Lze jej implementovat pomocí semaforů. Monitor obsahuje jednu nebo více procedur, inicializační sekvenci a lokální datové proměnné.

```
monitor monitor_name{
  shared variable declarations

  procedure body P1 (...){...}
  procedure body P2 (...){...}
  ...

  {
    initialization code
  }
}
```

Ke sdíleným proměnným se dostaneme pouze přes nějakou proceduru monitoru. Proces vstoupí do monitoru zavoláním nějaké z jeho procedur. V monitoru může být v jeden okamžik maximálně jeden proces. Monitor zajišťuje vzájemné vyloučení - není třeba jej explicitně programovat. Sdílená data jsou tedy chráněna tím, že je zapouzdříme do monitoru. Synchronizace procesů je provedena programátorem pomocí používání *podmínek* - speciálních datových struktur, nad kterými je možné provádět následující operace:

- **wait(a)** - proces se chce synchronizovat na podmínce a, proces čeká a pokračuje, až když nějaký jiný proces pošle uvolňující signál *signal(a)* pro danou podmínku
- **signal(a)** - proces uvolní proces, který čeká na podmínce a, tento uvolněný proces poté běží
- **notify(a)** - podobně jako signal, ale uvolněný proces neběží hned, ale čeká, až doběhne proces, který jej uvolnil (uvolněný proces je mezitím v pomocné čekací frontě uvnitř monitoru)

Příklady synchronizačních problémů

Producent-konzument

Úloha demonstrující problém zápisu a čtení položek z bufferu. Jeden proces - producent - zapisuje položky do bufferu a druhý proces - konzument - položky z bufferu čte. K dispozici může být neomezený buffer (v takovém případě producent může vždy produkovat a může se stát, že konzument někdy musí čekat, protože nemá co konzumovat) nebo omezený buffer (v takovém případě se může stát, že oba procesy musí čekat - když konzument nemá co konzumovat, nebo když je naplněna kapacita bufferu a producent nemá kam produkovat). Základní problém je ten, že může docházet ke konfliktu při souběžném působení konzumenta i producenta. Kromě samotného bufferu přistupují oba procesy i k dalším sdíleným zdrojům, jako například k proměnným obsahujícím index, ze kterého má být čtena další položka a do kterého má být zapsána další položka.

```
Circular Array of Buffers
typedef ... item;
item buffer[n-1];
int in = 0;
int out = 0;
```

```
Producent
while( 1 ) {
      item nextp;

      produce( nextp );
      while( (in+1 % n) == out) ;
      buffer[in] = nextp;
      in = in+1 % n;
}
```

```
Konzument
while( 1 ) {
      item nextc;

      while(in == out) ;
      nextc = buffer[out];
      out = out+1 % n;
      consume( nextc );
}
```

Souběh instrukcí, kde jeden proces proměnnou modifikuje a druhý čte (např. in=in+1 % n a while(in == out)) může způsobit nežádoucí chování celého systému (procesy pak mohou pracovat s neplatnou hodnotou těchto proměnných). Proto potřebujeme zabezpečit kritickou sekci tak, aby bylo zajištěno **vzájemné vyloučení** procesů.

```
counter = counter + 1;

        load   A,counter
        add    1,A
        store  A,counter

counter = counter - 1;

        load   A,counter
        sub    1,A
        store  A,counter
```

```
.Producer : "load A,counter"
.Consumer : "load A,counter"
.Producer : "add 1,A"
.Consumer : "sub 1,A"
.Producer : "store A,counter"
.Consumer : "store A,counter"
```

Řešení pomocí semaforů:

Potřebujeme semafor S pro vzájemné vyloučení nad bufferem a semafor N pro vzájemné vyloučení nad počtem prvků v bufferu. Pokud máme konečný buffer, musíme ještě přidat semafor E, který bude synchronizovat přístup k proměnné vyjadřující počet prázdných míst v bufferu. Řešení pro nekonečný buffer:

```
                    Initialization:
                      S.count:=1;
                      N.count:=0;
                      in:=out:=0;

append(v):
b[in]:=v;
in++;

                    Producer:           Consumer:
                    repeat              repeat
                      produce v;          P(N);
                      P(S);               P(S);
take():               append(v);          w:=take();
w:=b[out];            V(S);               V(S);
out++;                V(N);               consume(w);
return w;           forever             forever
```

Řešení pomocí kritického regionu:

```
var count: shared integer;

producer {
      .
      .
      region count do
          count = count + 1;
      .
      .
}
```

```
consumer {
      .
      .
      region count do
          count = count - 1;
      .
      .
}
```

Večeřící filozofové

5 filozofů sedí u kulatého stolu. Každý má po levé i pravé ruce jednu hůlku. Aby se najedl, potřebuje obě hůlky. Tento problém reprezentuje situaci, kdy je mezi synchronizujícími se procesy cyklická závislost.

Řešení pomocí semaforů:

Máme pět filozofů. Každý filozof je proces. Máme semafor pro každou hůlku. Každý filozof vezme do ruky nejprve levou a poté pravou hůlku, nají se a poté obě hůlky odloží.

```
Process Pi:
repeat
 think;
 P(fork[i]);
 P(fork[i+1 mod 5]);
 eat;
 V(fork[i+1 mod 5]);
 V(fork[i]);
forever
```

V tomto případě může dojít k uváznutí, pokud by si každý z filozofů vzal do ruky nejprve svou levou hůlku. Možné řešení této situace je například získávání obou hůlek současně, získávání hůlek asymetricky, nebo omezení maximální počtu filozofů, kteří zaráz chtějí jíst, na 4 (viz následující příklad, T.value je inicializováno na 4).

```
Process Pi:
repeat
 think;
 P(T);
 P(fork[i]);
 P(fork[i+1 mod 5]);
 eat;
 V(fork[i+1 mod 5]);
 V(fork[i]);
 V(T);
forever
```

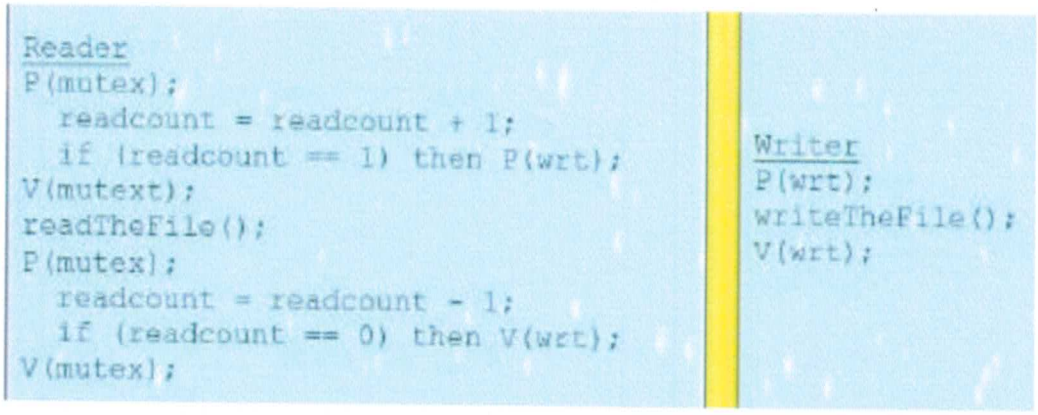
Čtenáři a písaři

Libovolný počet čtenářů může číst, ale pokud někdo píše, nikdo další nesmí psát ani číst.

Řešení pomocí semaforů:

V proměnné readcount uchováváme aktuální počet čtenářů. Použijeme dva semafory: mutex pro výlučný přístup k proměnné readcount a wrt pro výlučný přístup písařů. Písař čeká na odemknutí wrt, je jediný, který v tu chvíli přistupuje do kritické sekce. Pokud je wrt odemknutý a čtenář chce číst, zamkne jej. Každý další čtenář může začít číst a je inkrementována proměnná readcount. Jakmile poslední čtenář dočte (readcount == 0), je wrt opět odemknut.

V tomto řešení je sice zajištěn výlučný přístup, ale může dojít k vyhladovění písařů. Řešením je například přidat další semafor, který zajistí, že jakmile se objeví nějaký písař, tak čtenáři dočtou, ale nemůže už přijít nový čtenář.



```
Reader
P(mutex);
  readcount = readcount + 1;
  if (readcount == 1) then P(wrt);
V(mutext);
readTheFile();
P(mutex);
  readcount = readcount - 1;
  if (readcount == 0) then V(wrt);
V(mutex);
```

```
Writer
P(wrt);
writeTheFile();
V(wrt);
```

Pokud v textu najdete chybu, nebudete něčemu rozumět nebo budete mít dojem, že by bylo vhodné něco doplnit, kontaktujte na discordu uživatele barborasmahlikova.