Univerzita Hradec Králové
Fakulta informatiky a managementu

Synchronizace

Mgr. Josef Horálek

FIM UHK

Synchronizace procesu

- Kooperující proces je proces, který může ovlivnit nebo být ovlivněn jiným procesem právě spuštěným v systému
- Spolupracující procesy mohou sdílet:
  - logický adresový prostor (jak kód, tak i data);
  - data pouze prostřednictvím sdílených souborů;

Pozadí

Připomeňme problém omezeného bufferu

Producent:

```
repeat {vytvor dalsi polozku do
    nova_hodnota}
while citac = n do nic_nedelej;
    buffer(in) := nova_hodnota;
in := in + 1 mod n;
citac := citac + 1;
until false;
```

Příjemce:

```
repeat while citac = 0 do
    nic_nedelej;
prijata_hodnota := buffer(out);
out := out + 1 mod n;
citac := citac - 1;
{zpracuj prijatou polozku v
    prijata_hodnota}
until false;
```

- Je-li buffer velikosti n, dovoluje uložit maximálně n-1 položek současně
  - nedostatek řešíme zavedením proměnné čítač
    - hodnota se na počátku definuje jako 0 a je inkrementována při každém přidání nové položky do bufferu a dekrementována při odebrání položky .
- Jak příjemce, tak producent fungují správně, jsou-li spuštěni samostatně.
  - k chybám může dojít poběží-li oba procesy současně
    - např. hodnota proměnné citac je 5 a producent i příjemce běží současně;
    - v tom případě může být zcela náhodné výsledná hodnota v proměnné citac bud 4, 5 nebo 6;
    - správná hodnota je však samozřejmě jen 5 a ta by vznikla pokud by oba procesy běžely samostatně;

Pozadí

- Jak k chybě může dojít
  - možná implementace přiřazení citac := citac + 1 může být:

```
registr1 := citac;
registr1 := registr1 + 1;
citac := registr1;
```

  - registr1 je lokální registr CPU;
  - stejně tak přiřazení citac := citac - 1 může být implementováno jako:

```
registr2 := citac;
registr2 := registr2 - 1;
citac := registr2;
```

  - registr2 je jiný lokální registr CPU;

Pozadí

- Registr1 i Registr2 mohou být fyzicky jeden registr (akumulátor)
  - tehdy se však operace provedou správně díky logice CPU a mechanismu přerušení;
- Současné spuštění příkazu:
  - `citac := citac + 1;`
  - `citac := citac – 1;`
    - je ekvivalentní s postupným spouštěním jednotlivých příkazů nižší úrovně tak jak byly uvedeny v předem neurčeném pořadí;
    - jedno z možných proložení příkazů nižší úrovně může být:

```
T0 producent spouští registr1 := citac {registr1 = 5}
T1 producent spouští registr1 := registr1 + 1 {registr1 = 6}
T2 příjemce spouští registr2 := citac {registr2 = 5}
T3 příjemce spouští registr2 := registr2 - 1 {registr2 = 4}
T4 producent spouští citac := registr1 {citac = 6}
T5 příjemce spouští citac := registr2 {citac = 4}
```

Pozadí

- Souběh (race condition):
  - situace, kdy různé procesy přistupují a mění sdílená data současně;
  - výsledek jejich činnosti závisí na pořadí v jakém jsou jejich jednotlivé příkazy prováděny;
- Pro ochranu před souběhem musíme mít jistotu, že pouze jeden proces v daném čase může měnit proměnnou citac
  - K tomu je nutná synchronizace obou procesů.

FIM UHK

Problém kritické sekce

- Uvažujme systém obsahující n procesů {P0, P1, ..., Pn-1}
  - každý proces má část kódu zvanou kritická sekce
    - v ní může měnit společné proměnné, provést update tabulky, zapisovat do souboru apod;
- Pokud proces spouští svou kritickou sekci, nesmí žádný jiný proces mít možnost spustit ji také
  - spouštění kritických sekcí procesu je vzájemně jedinečné v čase (mutually exlusive in time);

FIM UHK

Problém kritické sekce

- Problémem kritické sekce je vytvořit protokol, který procesům umožní spolupracovat
  - každý proces musí mít právo provést svou kritickou sekci;
    - repeat
    - vstupní sekce;
    - kritická sekce;
    - ukončovací sekce;
    - zbývající sekce;
    - until false
- Řešení problému kritické sekce musí zohledňovat následující 3 požadavky:
  - vzájemnou jedinečnost (mutual exclusion);
  - progress;
  - omezené čekání (bounded waiting);

Problém kritické sekce
řešení pro dva procesy

- Nyní se podíváme na algoritmy aplikovatelné pouze pro dva procesy v čase
  - procesy budou označeny P0 a P1;
  - v obecném označení procesu užijeme Pi a Pj (j = 1 – i);

FIM UHK

Problém kritické sekce řešení pro dva procesy

- Algoritmus 1
  - spočívá ve sdílené celočíselné proměnné otáčka, která bude moci nabývat hodnot pouze 0 a 1;
  - jestliže otáčka = i, potom proces Pi může spouštět svou kritickou sekci;

```
repeat
while otacka <> i do nic_nedelej;
kriticka sekce otacka := j;
zbyvajici sekce
until false;
```

  - toto řešení zajišťuje, že pouze jeden proces ve stejné době může mít spuštěnou kritickou sekci;
  - je-li otáčka = 0, proces P1 svou kritickou sekci spustit nemůže;
  - Problém je v tom, že nedrží dostatečné informace o stavu každého procesu, rozhoduje pouze který proces může spustit kritickou sekci;

Problém kritické sekce řešení pro dva procesy

- Algoritmus 2
  - řešení zmíněného problému tkví v nahrazení proměnné otáčka polem:

```
var priznak: array (0..1) of boolean;
```

  - prvky pole příznak jsou na počátku inicializovány do hodnoty false;
    - je-li P(i) = true, potom proces Pi je připraven vstoupit do své kritické sekce;

```
repeat
priznak(i) := true;
while priznak(j) do nic_nedelej;
kriticka sekce priznak(i) := false;
zbyvajici sekce
until false;
```

Problém kritické sekce řešení pro dva procesy

- je-li proces Pi připraven vstoupit do své kritické sekce, nastaví hodnotu příznak(i) na true a kontroluje příznak(j), zdali proces Pj neprovádí právě svou kritickou sekci
- pokud ano, Pi musí čekat až bude kritická sekce procesu Pj dokončena a potom může spustit svou
- po jejím ukončení nastaví hodnotu příznak(i) na false, čímž dává na vědomí, že další proces může začít se svou kritickou sekcí;

- U tohoto algoritmu je požadavek vzájemné jednoznačnosti vyřešen
  - není vyřešena progresivnost;
    - pro ilustraci problému užijme následující spouštěnou sekvenci:

```
T0: ... P0 nastavuje priznak(0) = true
T1: ... P1 nastavuje priznak(1) = true
```

    - v tomto případě nastává u obou procesů zacyklení v cyklu while;

Problém kritické sekce řešení pro dva procesy

Algoritmus 3

- kombinuje myšlenku algoritmů 1 i 2
- výsledkem je řešení problému kritické sekce, které splňuje všechny tři dříve uvedené požadavky
- oba procesy pak sdílejí 2 proměnné:

```
var priznak: array (0..1) of boolean;
otacka: 0..1;
```

```
repeat priznak(i) := true;
otacka := j; while (priznak(j) and otacka = j) do nic_nedelej;
kriticka sekce priznak(i) := false;
zbyvajici sekce
until false;
```

- před vstupem do kritické sekce procesu Pi se nejprve nastaví proměnná příznak(i) na hodnotu true;
- potom tvrdí, ze druhý proces nechce vstoupit do kritické sekce (otacka := j);
- jestliže se oba procesy pokouší současně spustit své kritické sekce, proměnná otáčka určí, který z obou procesů může spustit svou kritickou sekci jako první;
  - rozhodne o tom poslední přiřazení hodnoty proměnné otáčka, předcházející bude přepsáno;

Synchronizační hardware

- Stejně jako jinde, i v synchronizaci procesů může hardware zjednodušit softwarovou implementaci a zvýšit efektivitu systému
  - jednoduchým řešením problému kritické sekce je zakázat ošetření přerušení v době, kdy jsou modifikovány systémové proměnné;
    - není vždy proveditelné;
  - mnoho počítačů provozuje hardwarovou instrukci, která umožňuje testovat a měnit obsah slova a další, která vymění obsah dvou slov;
    - těchto speciálních instrukcí můžeme využít k relativně jednoduchému řešení problému kritické sekce;

Semafory

- Semafor S je celočíselná proměnná, jejíž hodnota může být změněna pouze dvěma standardními atomickými operacemi:
  - čekej;
  - signál;

```
Čekej(S):
while S <= 0 do nic_nedelej;
S := S - 1;

Signál(S):
S := S + 1;
```

FIM UHK

Semafory - Využití

- Semafory je možno užít pro řešení problému kritické sekce n-procesů
  - N procesů sdílí semafor vzajed (zkratka ze vzájemná jedinečnost) inicializovanou do hodnoty 1;
  - každý proces Pi je organizován:

```
repeat cekej(vzajed);
kritická sekce
signal(vzajed);
zbyvajici sekce
until false;
```

Semafory - Využití

- Semafory lze užít pro řešení různých synchronizačních problémů
  - např. dva současně probíhající procesy: P1 s instrukcí I1 a P2 s instrukci I2
    - instrukce I2 může být spuštěna pouze po dokončení instrukce I1;
    - řešení můžeme implementovat s pomocí sdílené proměnné synch inicializované na počátku do 0 a vložit do programu procesu P1 a P2 tyto instrukce:

```
P1:
I1;
signal(synch);

P2:
cekej(synch);
 I2;
```

Semafory - Využití

- Proměnná synch je na začátku inicializovaná do hodnoty 0
  - P2 může spustit instrukci S2 pouze pokud P1 vykoná instrukci signal(synch), která se spustí po dokončení instrukce S1;
  - semafory, ve kterých proces „krouží “ v cyklu while, jsou nazývány kruhové blokování (spinlock);
    - kruhové blokování je výhodné v multiprocesorových systémech;
    - výhodou kruhového čekání je, že v jeho průběhu není třeba žádné přepínání kontextu;
    - k překonání kruhového blokování je třeba modifikovat definici operací čekej a signál;
    - pokud proces spustí operaci čekej a zjistí, že semafor mu není nakloněn, musí čekat;
    - proces, který je blokován a čeká na semafor S může být restartován pokud jiný proces užije operaci signál;
      - proces je restartován operací probuď_se, který změní stav procesu z čekající na probíhající;
      - poté je proces přesunut do fronty připravených;.

FIM UHK

Semafory - Implementace

- K implementaci semaforu je pak potřeba definovat semafor jako záznam:

```
type semafor = record
hodnota: integer;
L: list of proces;
end;
```

  - každý semafor má celočíselnou hodnotu a seznam procesů;
  - musí-li proces čekat na semafor, je přidán do seznamu procesů;
  - operace signál odstraní proces ze seznamu čekajících procesů a „vzbudí“ proces;

Semafory - Implementace

= Operace na semaforu jsou definovány následovně:

čekej(S):

```
S.hodnota := S.hodnota - 1;
if S.hodnota < 0 then
begin
"pridej tento proces do
  S.L";
blokuj;
 end;
```

signál(S):

```
S.hodnota := S.hodnota + 1;
if S.hodnota <= 0 then
Begin
 "odeber proces P z S.L";
probud_se(P);
end;
```

- = operace block dočasně pozastaví proces, který ji vyvolá; operace probuď_se obnoví běh blokovaného procesu P;
- = seznam čekajících procesů může být jednoduše implementován jako odkaz na seznam PCB bloku;

FIM UHK

Semafory - Implementace

- Základní rys semaforu je, že operace na nich musí být spouštěny atomicky
  - je třeba zaručit, aby žádné dva procesy najednou nespustily operace čekej a signál na stejný semafor;
    - tato situace je problémem kritické sekce a může být řešen jedním z následujících dvou způsobů:
      - zakázat přerušení během výpočtu funkcí čekej a signál;
      - užitím libovolného z fungujících algoritmů pro softwarové řešení problému kritické sekce;
  - kruhové čekání odstraníme z řízení vstupu do kritické sekce u aplikačních programů;
    - implementace semaforu s frontou čekajících může vést k situaci, kde dva nebo více procesů neomezeně čekají na událost, kterou může vyvolat pouze některý z čekajících procesů;
    - další problém spojený se zablokováním procesu je umoření (starvation) procesu,
      - situace kdy proces čeká neomezeně dlouho na semaforu;

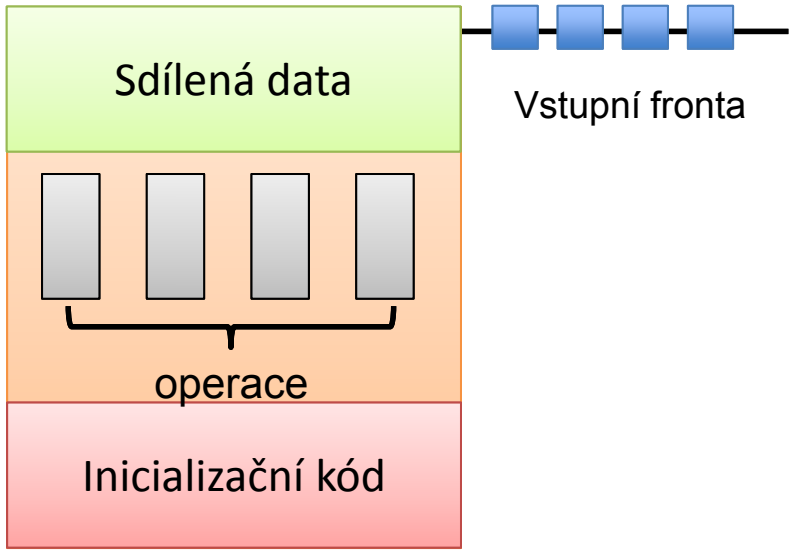
Monitory

- Další synchronizační konstrukcí vyšší úrovně jsou monitory
  - monitor je charakterizován množinou programově definovaných operátorů;
  - reprezentace typu monitor obsahuje deklarace proměnných jejichž hodnoty definuje procedura či funkce;

```
type jmeno_monitoru = monitor
deklarace promennych
procedure entry P1 (...);
begin ... end;
  .
procedure entry Pn (...);
begin ... end;
begin
inicializacni kod
end.
```

Monitory

- Konstrukce monitoru zajišťuje
  - pouze jeden proces může být v daném čase na monitoru aktivní;
    - programátor se nemusí zabývat synchronizačním kódem;



Univerzita Hradec Králové
Fakulta informatiky a managementu

Děkuji za pozornost…