Paralelní programování

přednášky

Jan Outrata

únor–duben 2011

Monitor

Semafor

- vedle aktivní (čekací smyčka, busy-wait) i „pasivní“ implementace (změna stavu procesu)
- středněúrovňové synch. primitivum – nestrukturované $\rightarrow$ náročné použití pro rozsáhlejší programy

Monitor

- Hoare, Hansen

$=$ **strukturované** synch. primitivum – synchronizace v **„objektu/modulu“**

- zobecnění jádra/supervisoru v OS (centralizace kritických sekcí)
- pro každý „objekt/modul“ vyžadující synchronizaci

Monitor

- **operace na** (stejném) **monitoru prováděné** více procesy **se vzájemným vyloučením**
  - = pouze jeden proces může provádět operaci na monitoru (v daném čase)
    - ostatní procesy čekají na (vstupu do) monitoru
    - řešení problému **kritické sekce**, **atomické provádění operací**
  - = implicitní synchronizace – „zámek“ na vstupu do monitoru
    - není určeno pořadí uvolňování čekajících procesů $\Rightarrow$ může dojít k vyhladovění procesu
- – zobecnění objektu z objektově orientovaného programování (OOP) pro **zapouzdření synchronizačních dat a operací** pomocí třídy
- data monitoru privátní (přístupná pouze z monitoru) = zapouzdření sdílených proměnných
- různé implementace (dat. typu/třídy) napříč prog. jazyky nebo systémy – pozor na sémantiku!

Monitor

Atomická operace monitoru

```
monitor CS
    int n ← 0

    operation incr()
        int temp
        temp ← n
        n ← temp + 1
```

A	*B*
1: CS.incr()	1: CS.incr()

Obrázek: Atomická inkrementace pomocí monitoru

Podmíněná proměnná (condition variable, event)

- monitor = implicitní vzájemné vyloučení
- mnoho synch. problémů vyžaduje explicitní synchronizaci = čekání na splnění podmínky, např. problém producent-konzument

= proměnná, na které proces „čeká“, dokud není splněna podmínka (**čekání na podmínce**); při naplnění podmínky uvolnění procesu – po explicitní **signalizaci podmínky**

- test podmínky a čekání – atomické, **součást operace na monitoru** $\Rightarrow$ hodnota se mezi testováním a čekáním nemůže změnit
- při čekání procesu „opuštění“ monitoru – **atomicky**, pro umožnění operací na monitoru (zejm. signalizace) jiným procesům
- pasivní implementace: proměnná = množina (fronta) blokovaných procesů
- možné implementace i mimo monitor, např. PThreads – potřeba zajistit vzájemné vyloučení monitoru, např. mutexem
- rozdíly oproti semaforu: *waitC* vždy čeká, *signalC* bez efektu, pokud žádný proces nečeká

Podmíněná proměnná (condition variable, event)

(Atomické) operace čekání na podmínce proměnné *cond*, signalizace podmínky a testování na čekající procesy (prováděné procesem *A*):

waitC(cond)
proces A
monitor mon
$cond \leftarrow cond \cup A$
A.state $\leftarrow$ blocked
unlock(mon.lock)

signalC(cond)
proces B
if $cond \neq \emptyset$
B $\leftarrow$ libovolný prvek cond
$cond \leftarrow cond \setminus \{B\}$
B.state $\leftarrow$ ready

emptyC(cond)
return $cond = \emptyset$

Obrázek: Operace *waitC*(*cond*), *signalC*(*cond*) a *emptyC*(*cond*) na podmíněné proměnné *cond*

Podmíněná proměnná (condition variable, event)

Simulace semaforu pomocí monitoru

```
monitor Sem
    int s ← k
    condition notZero

    operation wait()
        if s = 0
            waitC(notZero)
        s ← s - 1

    operation signal()
        s ← s + 1
        signalC(notZero)
```

A	*B*
loop forever	loop forever
nekritická sekce	nekritická sekce
1: Sem.wait()	1: Sem.wait()
kritická sekce	kritická sekce
2: Sem.signal()	2: Sem.signal()

Obrázek: Simulace semaforu (na řešení problému krit. sekce)

Podmíněná proměnná (condition variable, event)

- stavový diagram: celé operace monitoru $\sim$ jediný krok (vzájemné vyloučení, žádné prolnutí)

Obr. 151

Obrázek: Stavový diagram simulace semaforu (na řešení problému krit. sekce)

  - pozn.: uvolněný proces ihned pokračuje, v rámci kroku operace se signalizací

- **problém**: při uvolnění čekajícího procesu pokračování a „vstup“ do monitoru, signalizující proces ale také pokračuje „v“ monitoru = neplatný stav
- **řešení**: jeden z procesů musí počkat na „opuštění“ monitoru druhým procesem – (uvolněný) čekající W nebo signalizující S nebo libovolný?
- navíc ještě čekají procesy E na „vstupu“ do monitoru

Podmíněná proměnná (condition variable, event)

- klasická priorita: $E < S < W$ (**požadavek okamžitého pokračování, signal and urgent wait**) – při signalizaci podmínka platí a uvolněný proces může pokračovat bez jejího opětovného testování, viz např. simulace semaforu
- při $W < S$ (např. v Javě) může pokračující signalizující proces podmínku před pokračováním uvolněného procesu znovu zneplatnit $\rightarrow$ uvolněný proces musí podmínku **znovu otestovat** a případně čekat, např. pro simulaci semaforu:

```
while s = 0
    waitC(notZero)
s ← s - 1
```

- $S < E$ nebo $W < E$ nevhodné (*proč?*)

Problém producenta a konzumenta

- podmíněné proměnné místo semaforů, operace monitoru

Producent-konzument

```
monitor PC
    dataType buffer[N] ← nil
    int i ← 0, j leftarrow 0
    condition notEmpty, notFull

    operation put(dataType item)
        if i + 1 mod N = j
            waitC(notFull)
        buffer[i] ← item
        i ← i + 1 mod N
        signalC(notEmpty)

    operation get()
        dataType item
        if i = j
            waitC(notEmpty)
        item = buffer[j]
        j ← j + 1 mod N
        signalC(notFull)
        return item
```

producent	konzument
dataType item	dataType item
loop forever	loop forever
1: item ← produce()	1: item ← PC.get()
2: PC.put(item)	2: consume(item)

Obrázek: Řešení problému producenta a konzumenta s omezeným bufferem

Problém čtenářů a písařů

Čtenáři-písaři

```
monitor RW
  int readers ← 0
  bool writing ← false
  condition OKtoRead, OKtoWrite

  operation StartRead()
    if writing or not emptyC(OKtoWrite)
      waitC(OKtoRead)
    readers ← readers + 1
    signalC(OKtoRead)

  operation EndRead()
    readers ← readers - 1
    if readers = 0
      signalC(OKtoWrite)

  operation StartWrite()
    if writing or readers ≠ 0
      waitC(OKtoWrite)
    writing ← true

  operation EndWrite()
    writing ← false
    if emptyC(OKtoRead)
      signalC(OKtoWrite)
    else
      signalC(OKtoRead)
```

čtenář		písař	
	loop forever		loop forever
1:	RW.StartRead()	1:	RW.StartWrite()
2:	čtení	2:	zápis
3:	RW.EndRead()	3:	RW.EndWrite()

Obrázek: Řešení problému čtenářů a písařů (monitor s podmíněnými proměnnými)

Problém čtenářů a písařů

- přednost čekajících písařů před novými čtenáři (*jak?*) a čekajících čtenářů před čekajícími písaři (*jak?*)
- čtenář při vstupu do krit. sekce uvolní (všechny) ostatní čekající čtenáře a umožní jim vstup (*jak?*) = kaskádové uvolnění, ale ne pro nové čtenáře (*proč?*)

$\Rightarrow$ absence vyhladovění čtenářů i písařů (čekajících na vstup do krit. sekce, ne na „vstup“ do monitoru)

Problém večeřících filozofů

- atomické testování a čekání na dostupnost obou hůlek
- podmíněné proměnné pro filozofy místo semaforů pro hůlky, pole počtů volných hůlek pro každého filozofa

Večeřící filozofové

```
monitor Phils
  int forks[N] ← [2,...,2]
  condition OKtoEat[N]

  operation takeForks(int i)
    if forks[i] ≠ 2
      waitC(OKtoEat[i])
    forks[i+1] ← forks[i+1] - 1
    forks[i-1] ← forks[i-1] - 1

  operation releaseForks(int i)
    forks[i+1] ← forks[i+1] + 1
    forks[i-1] ← forks[i-1] + 1
    if forks[i+1] = 2
      signalC(OKtoEat[i+1])
    if forks[i-1] = 2
      signalC(OKtoEat[i-1])
```

filozof *i*

```
    loop forever
1:  filozofování
2:  Phils.takeForks(i)
3:  jezení
4:  Phils.releaseForks(i)
```

Obrázek: Řešení problému večeřících filozofů

- může dojít k vyhladovění procesu: *scénář?*

Chráněný objekt (protected object)

- u (klasického) monitoru s podmíněnými proměnnými synchronizace (operace testování podmínky, waitC, signalC, emptyC) explicitní

= **monitor s implicitní synchronizací** – před a po operacích monitoru = **chráněných operací**

- před operací: **podmínka zahájení operace** (jen nad proměnnými objektu) = „bariéra“, při nesplnění čekání
- po operaci: otestování všech podmínek operací („bariér“) a při naplnění podmínky uvolnění jednoho čekajícího procesu

Chráněný objekt (protected object)

Čtenáři-písaři

```
protected object RW
   int readers ← 0
   bool writing ← false

   operation StartRead() when not writing
      readers ← readers + 1

   operation EndRead()
      readers ← readers - 1

   operation StartWrite()
         when not writing and
         readers = 0
      writing ← true

   operation EndWrite()
      writing ← false
```

čtenář		písař	
	loop forever		loop forever
1:	RW.StartRead()	1:	RW.StartWrite()
2:	čtení	2:	zápis
3:	RW.EndRead()	3:	RW.EndWrite()

Obrázek: Řešení problému čtenářů a písařů (chráněný objekt)

- může dojít k vyhladovění procesu: *scénář?*

Chráněný objekt (protected object)

- **efektivnější implementace** implicitní synchronizace snižující počet přepnutí procesů (ze signalizujícího na čekající a zpět)
  - procesy vykonávají operace monitoru i za jiné procesy (díky zapouzdření proměnných a vzájemnému vyloučení)!
  - podmínky operací („bariéry“) nesmí záviset na (lokálních) parametrech operace, jinak čekání na část podmínky bez parametru a pak v chráněné operaci otestování dat a případné další čekání

Monitory

- monitory, podmíněné proměnné a chráněné objekty dnes klasická **vysokoúrovňová sychronizační primitiva**

= strukturované dat. typy/třídy – vhodná pro použití v OOP

- na nich založené další synch. kontrukty v moderních (OO) prog. jazycích
- **centralizované** – jako všechna synch. primitiva založená na sdílení dat
- pro distribuované architektury vhodnější synchronizace založená na komunikaci