

# 04. Mutexy, monitory

ZOS 2014, L. Pešička

---



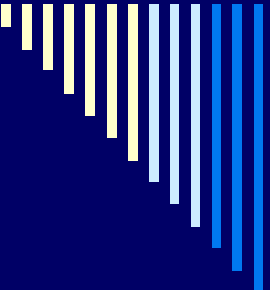
# Semaforý

## □ Ošetření kritické sekce

- ukázka – více nezávislých kritických sekcí x,y,z

## □ Synchronizace: Producent – konzument

- možnost procesu zastavit se a čekat na událost
- 2 události
  - buffer prázdný – čeká konzument
  - buffer plný – čeká producent
- „uspání procesu“ – operace semaforu P



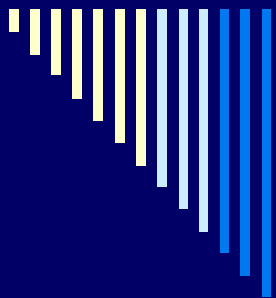
# Problém spícího holiče – jen zadání (The barbershop problem)

## Holičství

- čekárna s  $N$  křesly a holičské křeslo
- žádný zákazník – holič spí
- zákazník vstoupí
  - všechna křesla obsazena – odejde
  - holič spí – vzbudí ho
  - křesla volná – sedne si na jedno z volných křesel

Napsat program, koordinující činnost holiče a zákazníků

Je celá řada podobných synchronizačních úloh, cílem je pomocí synchronizačních mechanismů ošetřit úlohu, aby fungovala korektně a nedocházelo např. k vyhladovění ...



# Literatura pro samostatnou práci

The Little Book of Semaphores  
(Allen B. Downey)

kniha volně dostupná na netu:

<http://greenteapress.com/semaphores/>

**Serializace:** událost **A** se musí stát **před** událostí **B**

**Vzájemné vyloučení:** události **A** a **B** se nesmí stát ve stejný čas



# Mutexy, monitory

- Mutexy, implementace
- Implementace semaforů
- Problémy se semaforey
- Monitory
  - Různé reakce na signal
- Implementace monitorů

Obsah další části přednášky



# Mutexy

- Potřebujeme zajistit **vzájemné vyloučení**
  - chceme „**spin-lock**“ bez aktivního čekání
  - nepotřebujeme obecně schopnost semaforů čítat
- **mutex** – mutual exclusion
  - paměťový zámek

Mutex řeší vzájemné vyloučení a je k systému šetrnější než čistě aktivní čekání spin-lock, můžeme jej naimplementovat např. pomocí **TSL** instrukce a volání **yield**



# Implementace mutexu (!!)

## – s podporou jádra OS

Oblíbená  
instrukce  
TSL

**mutex\_lock:**TSL R, mutex

CMP R, 0

JE ok

**CALL** yield

Vzdát  
se CPU

JMP mutex\_lock

ok: RET

;; R:=mutex a mutex:=1

;; byla v mutex hodnota 0?

;; pokud byla skok na ok

;; vzdáme se procesoru -  
naplánuje se jiné vlákno

;; zkusíme znovu, později

**mutex\_unlock:**

LD mutex, 0

RET

;; ulož 0 do mutex



# Implementace mutexu

## – volání **yield**



Šetří CPU

- volající se **dobrovolně vzdává** procesoru ve prospěch jiných procesů (vláken,...)
- jádro OS přesune proces mezi **připravené** a časem ho opět **naplánuje**
- použití – např. **vzájemné vyloučení** mezi vlákny **stejného** procesu
- lze implementovat jako **knihovnu** prog. jazyka





# Moderní OS – semafony, mutexy

Moderní OS nám  
dávají k dispozici  
určitou množinu  
synchronizačních  
nástrojů, z nichž si  
programátor vybírá

## □ obecné (čítající) semafony

- obecnost
- i pro řešení problémů **meziprocesové komunikace**

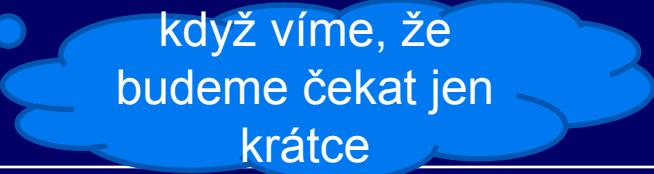
## □ mutexy, binární semafony

- paměťové zámkové, binární semafony
- pouze pro **vzájemné vyloučení**
- při vhodné implementaci efektivnější



# Spin-lock (aktivní čekání)

- **spin-lock** – vhodný, pokud je čekání **krátké** a procesy běží paralelně
- není vhodné pro použití v **aplikacích**
  - aplikace – doba čekání se může velmi lišit
- obvykle se používá uvnitř **jádra OS**,  
v knihovnách, ...



když víme, že  
budeme čekat jen  
krátce



# Mutex x binární semafor

- Společné – použití pro vzájemné vyloučení
- Často se v literatuře mezi nimi příliš nerozlišuje
- Někdy jsou zdůrazněny rozdíly

## Mutex

s koncepcí vlastnictví:

Odemknout mutex může jen stejné vlákno/proces, který jej zamkl (!!!!)

uvědomte si,  
kdy nám toto  
může vadit

pamatovat si co znamená  
pojem mutex s koncepcí  
vlastnictví



# Srovnání

## Příklad 1:

```
mutex_lock();  
// kritická sekce  
mutex_unlock();
```

**stejné vlákno**  
zamyká i odemyká

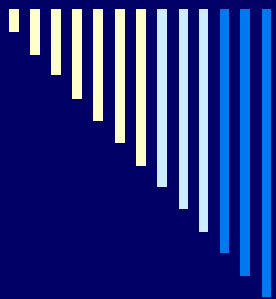
## Příklad 2:

```
semaphore s = 0;
```

```
v1:  
  P(s);  
  printf(" světe.");
```

```
.....  
v2:  
  printf("Ahoj, ");  
  V(s);
```

**jedno** vlákno zamkne,  
**druhé** mu signalizuje



# rozdíl mutex vs. binární semafor

<http://stackoverflow.com/questions/62814/difference-between-binary-semaphore-and-mutex>

a další:

<http://stackoverflow.com/questions/5454746/pthread-mutex-lock-unlock-by-different-threads/5492499#5492499>

a také:

<http://www.geeksforgeeks.org/mutex-vs-semaphore/>



# rozdíl mutex vs. semafor

Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource. Only **one task** (can be a thread or process based on OS abstraction) can **acquire the mutex**. It means there will be ownership associated with mutex, and **only the owner can release the lock** (mutex).

Semaphore is **signaling mechanism** ("I am done, you can carry on" kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend called you, an interrupt will be triggered upon which an interrupt service routine (ISR) will signal the call processing task to wakeup.

zdroj: <http://www.geeksforgeeks.org/mutex-vs-semaphore/>



# Reentrantní mutex

- Stejně vlákno může získat několikrát zámek (co by se stalo pokud není reentrantní?)
- Stejně tolikrát jej musí zas odemknout, aby mohlo mutex získat jiné vlákno
- Viz: [http://en.wikipedia.org/wiki/Reentrant\\_mutex](http://en.wikipedia.org/wiki/Reentrant_mutex)

Na ukázkou různých variant mutexů:  
reentrantní mutex, futex, ...



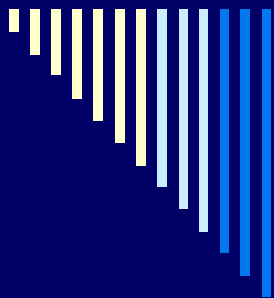
# Futex

- Userspace mutex, v Linuxu
- V **kernel** space: wait queue (fronta)
- V **user** space: integer (celé číslo, zámek)
- Vlákna/procesy mohou operovat nad číslem v **userspacu** s využitím **atomických** operací a **systémové volání** (které je drahé) jen pokud je třeba manipulovat s frontou čekajících procesů (vzbudit čekající proces, dát proces do fronty čekajících)
- Viz <http://en.wikipedia.org/wiki/Futex>

Vždy se řeší otázka rychlosti, ceny

Systémové volání je obvykle nákladná záležitost, proto snaha minimalizovat jejich počet





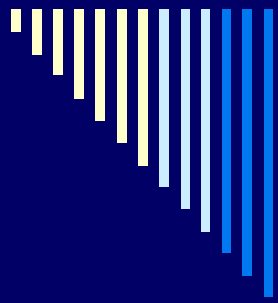
Dále bude ukázána obecná implementace semaforu a  
implementace semaforu s využitím mutexu



# Implementace semaforu obecná – datové struktury

```
typedef struct {  
    int value;                // hodnota semaforu  
    struct process *list;     // fronta zablokovaných  
}                             // procesů
```

Zatímco předpokládáme, že hodnota semaforu je  $\geq 0$   
pro vnitřní implementaci můžeme připustit i záporné  
hodnoty (udávají počet blokováných procesů)



# Implementace semaforu obecná - P

```
P (semaphore s) {  
    s.value--;  
    if (s.value < 0)  
        blokuj(s.list);  
}
```

blokuj – zablokuje volající proces, zařadí jej do fronty čekajících na daný semafor s.list



# Implementace semaforu obecná - V

```
V (semaphore s) {  
    s.value++;  
    if (s.value <= 0)  
        if (s.list != NULL) { // někdo spí nad S  
            vyjmi_z_fronty(p);  
            vzbud (p);         // blokováný -> připrav.  
        }  
}
```

---



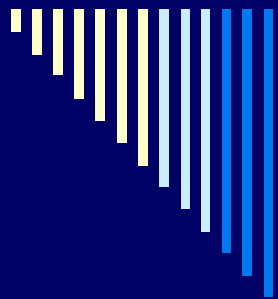
# Semaforey implementace s využitím mutexu

- S každým semaforem je sdruženo:
  - celočíselná proměnná **s.c**
    - pokud může nabývat i záporné hodnoty
    - $|s.c|$  vyjadřuje počet blokováných procesů
  - binární semafor **s.mutex**
    - vzájemné vyloučení při operacích nad semaforem
  - seznam blokováných procesů **s.L**



# Seznam blok. procesů

- Proces, který nemůže dokončit **operaci P** bude zablokován a uložen do seznamu procesů **s.L** blokováných na semaforu **s**
- Pokud při **operaci V** není seznam prázdný
  - vybere ze seznamu jeden proces a odblokuje se



# Uložení datové struktury semafor

- semafore v **jádře OS**
  - přístup pomocí služeb systému
  
- semafore ve **sdílené paměti**



# Popis implementace

```
type semaphore = record  
    m: mutex;           // mutex pro přístup k semaforu  
    c: integer;         // hodnota semaforu  
    L: seznam procesu  
end
```

---





# Popis implement. – operace P

```
P(s): mutex_lock(s.m);
```

```
    s.c := s.c - 1;
```

```
    if s.c < 0 then
```

```
        begin
```

```
            zařad' volající proces do seznamu s.L;
```

```
            označ volající proces jako "BLOKOVANY";
```

```
            naplánuj některý připravený proces;
```

```
            mutex_unlock(s.m);
```

```
            přepni kontext na naplánovaný proces
```

```
        end
```

```
    else
```

```
        mutex_unlock(s.m);
```

---



# Popis implement. – operace V

```
V(s): mutex_lock(s.m);  
      s.c := s.c + 1;  
      if s.c <= 0 then  
        begin  
          vyber a vyjmi proces ze sez. s.L;  
          odblokuj vybraný proces  
        end;  
      mutex_unlock(s.m);
```



# Popis implementace

- Pseudokód
- Skutečná implementace řeší i další detaily
  - Organizace datových struktur
    - Pole, seznamy, ...
  - Kontrola chyb
    - Např. jeli při operaci **V** záporné **s.c**  
a přitom **s.L** je prázdné



# Popis implementace

## □ Implementace v jádře OS

- Obvykle používá **aktivní čekání** (**spin-lock** nad s.mutex)
- Pouze **po dobu operace** nad obecným semaforem – max. desítky instrukcí - **efektivní**



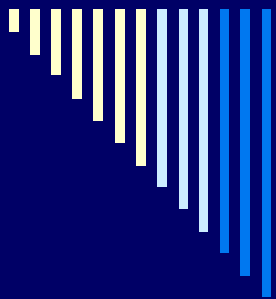
# Mutexy vs. semaforey

- **Mutexy** – vzájemné vyloučení vláken v jednom procesu
  - Např. knihovní funkce
  - Často běží v uživatelském režimu
- **Obecné semaforey** – synchronizace mezi procesy
  - Implementuje jádro OS
  - Běží v režimu jádra
  - Přístup k vnitřním datovým strukturám OS



# Problémy se semaforey

- primitiva **P** a **V** – použita **kdekoliv** v programu
- Snadno se udělá **chyba**
  - Není možné automaticky kontrolovat při překladu



# Chyby – přehození P a V

Přehození P a V operací nad mutexem:

1. V()
2. kritická sekce
3. P()

Důsledek – více procesů může vykonávat kritickou sekci současně



# Chyby – dvě operace P

1. P()
2. Kritická sekce
3. P()

Důsledek - deadlock





# Chyby – vynechání P, V

- Proces vynechá **P()**
- Proces vynechá **V()**
- Vynechá obě

Důsledek – porušení vzájemného vyloučení nebo deadlock



# Semaforey - Java

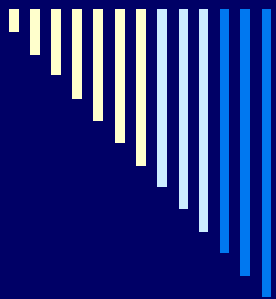
třída `java.util.concurrent.semaphore`

metody:

- `acquire()` - operace P()
- `release()` - operace V()

**java.util.concurrent** obsahuje celou řadu synchronizačních mechanismů, nejen semaforey

viz např. <http://tutorials.jenkov.com/java-util-concurrent/semaphore.html>



# Semaforey – Java - použití

```
Semaphore s1 = new Semaphore(1);
```

```
s1.acquire();
```

```
// kritická sekce
```

```
s1.release();
```



# Zámek - Java

□ třída `java.util.concurrent.locks.Lock`

```
Lock m = new ReentrantLock();
```

```
m.lock();
```

```
// kritická sekce
```

```
m.unlock();
```

---



# Semafore - C

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); // inicializuje na 1
sem_wait(&s);       // operace P()
// kritická sekce
sem_post(&s);       // operace V()

sem_destroy(&s);
```

---



# Mutexy - C

```
#include <pthread.h>
```

```
pthread_mutex_t m;
```

```
pthread_mutex_lock(&m);
```

```
// kritická sekce
```

```
pthread_mutex_unlock(&m);
```

---



# Monitory

- Snaha najít primitiva vyšší úrovně, která zabrání části potenciálních chyb
  - Hoare (1974) a Hansen (1973) nezávisle na sobě navrhli vysokoúrovňové synchronizační primitivum nazývané **monitor**
  - Odlišnosti v obou návrzích
-



# Monitor

- Monitor – na rozdíl od semaforů  
– **jazyková konstrukce**
  - Speciální typ modulu, sdružuje data a procedury, které s nimi mohou manipulovat
  - Procesy mohou volat proceduru monitoru, ale nemohou přistupovat přímo k datům monitoru
-





# Monitor

- V monitoru může být **v jednu chvíli**  
**AKTIVNÍ pouze jeden** proces !!
- Ostatní procesy jsou při pokusu o vstup do monitoru pozastaveny



# Terminologie OOP

- Snaha chápat kritickou sekci jako **přístup ke sdílenému objektu**
- Přístup k objektu pouze pomocí určených operací – **metod**
- Při přístupu k objektu vzájemné vyloučení, přístup **po jednom**



# Monitory

příklad se vztahuje k syntaxi Pascalu, tak jak monitor implementoval např. sw Baci

- Monitor – blok podobný proceduře nebo funkci
- Uvnitř monitoru definovány proměnné, procedury a funkce
- **Proměnné monitoru** – nejsou viditelné zvenčí
  - Dostupné pouze procedurám a funkcím monitoru
- **Procedury a funkce** – viditelné a volatelné vně monitoru



# Příklad monitoru

**monitor** m;

*var proměnné ...*

*podmínky ...*

**procedure** p; { procedura uvnitř monitoru }

begin

...

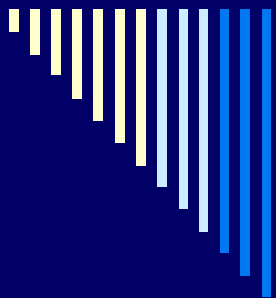
end;

begin

inicializace;

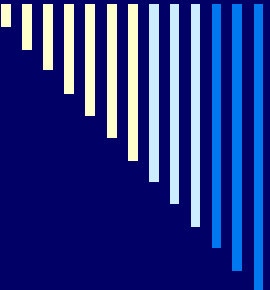
end;

---



# Příklad

Použití monitoru pro vzájemné vyloučení



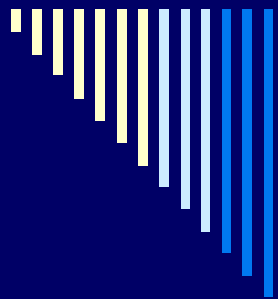
```
monitor m;                                // příklad – vzájemné vyloučení
  var x: integer;
  procedure inc_x;                          { zvětší x }
  begin
    x:=x+1;
  end;
  function get_x: integer;                  { vrací x }
  begin
    get_x:=x
  end
begin
  x:=0
end; { inicializace x };
```

---



# Problém dosavadní definice monitoru

- Výše uvedená definice (částečná) – dostačuje pro **vzájemné vyloučení**
- **ALE nikoliv pro synchronizaci** – např. řešení producent/konzument
- Potřebujeme **mechanismus**, umožňující procesu se **pozastavit** a tím **uvolnit vstup** do monitoru
- S tímto mechanismem jsou monitory **úplné**



# Synchronizace procesů v monitoru

- Monitory – speciální typ proměnné nazývané **podmínka** (condition variable)
- Podmínky
  - definovány a použity pouze uvnitř monitoru
  - Nejsou proměnné v klasickém smyslu, neobsahují hodnotu
  - Spíše odkaz na určitou událost nebo stav výpočtu (mělo by se odrážet v názvu podmínky)
  - Představují frontu procesů, které na danou podmínku čekají





# Operace nad podmínkami (!!)

- Definovány 2 operace – **wait** a **signal**

## **C.wait**

- Volající bude **pozastaven nad podmínkou C**
- Pokud je některý proces připraven vstoupit do monitoru, bude mu to dovoleno

často také najdeme ve tvaru: wait(c), signal(c)

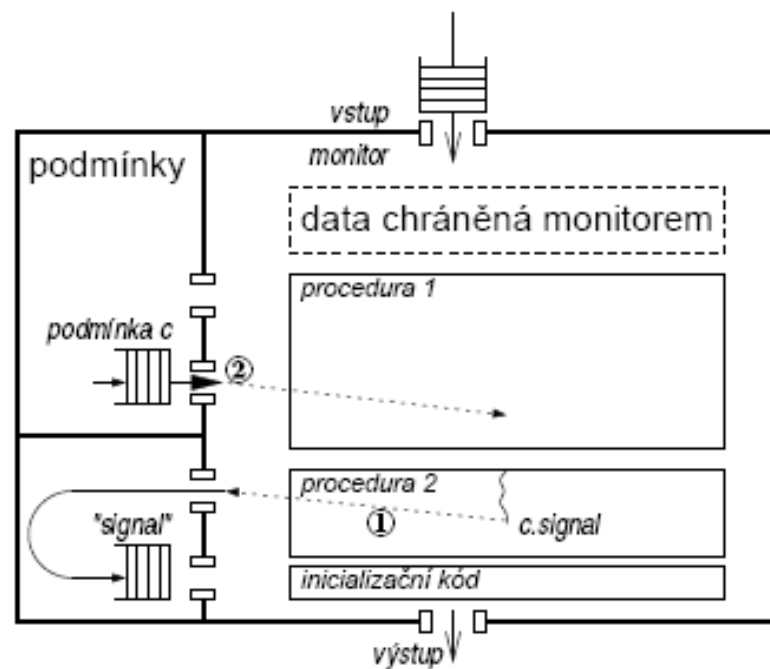
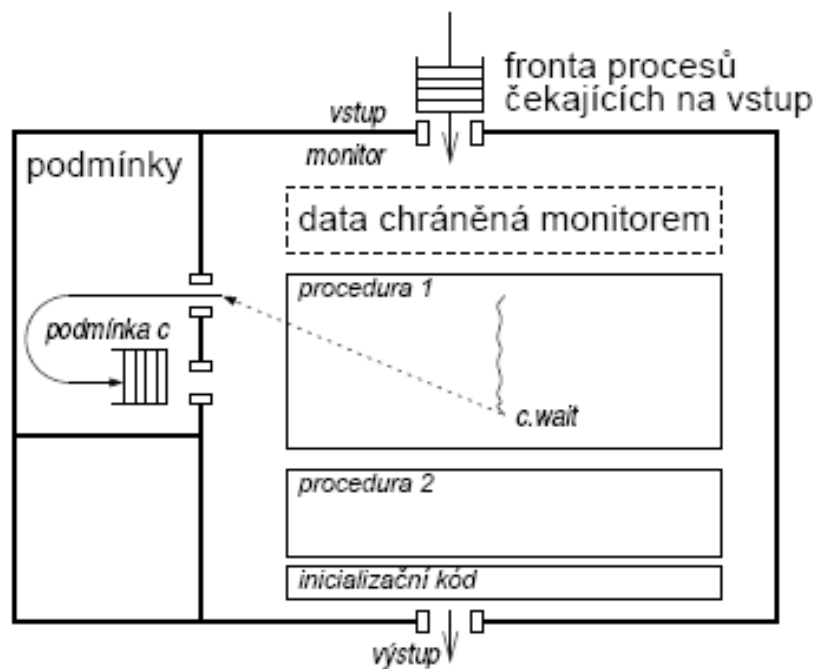


# Operace nad podmínkami

## C.signal

- Pokud existuje 1 a více procesů **pozastavených** nad podmínkou **C**, **reaktivuje** jeden z pozastavených procesů, tj. bude mu dovoleno pokračovat v běhu **uvnitř** monitoru
- Pokud nad podmínkou **nespí** žádný proces, **nedělá nic** 😊
  - Rozdíl oproti semaforové operaci V(sem), která si “zapamatuje”, že byla zavolána

# Schéma monitoru





# Problém s operací **signal**

- Pokud by signál pouze vzbudil proces, běžely by v monitoru dva
    - Vzbuzený proces
    - A proces co zavolal signal
  - **ROZPOR** s definicí monitoru
    - V monitoru může být v jednu chvíli **aktivní** pouze jeden proces
  - Několik řešení
-



# Řešení reakce na signal (!!)

## □ Hoare

- proces volající **c.signal** se **pozastaví**
- Vzbudí se až poté co předchozí rektivovaný proces opustí monitor nebo se pozastaví

## □ Hansen

- Signal smí být uveden pouze jako **poslední** příkaz v monitoru
  - Po volání signal musí proces **opustit** monitor
-



# Monitory v Javě – 2 typy

- zjednodušené monitory  
`synchronized`, `wait`, `notify`
- použití `java.util.concurrent`  
zámky + podmínkové proměnné



# Monitory v jazyce Java

- zjednodušené monitory
- S každým objektem je sdružen monitor, může být i prázdný
- Metoda nebo blok patřící do monitoru označena klíčovým slovem **synchronized**



# Monitory - Java

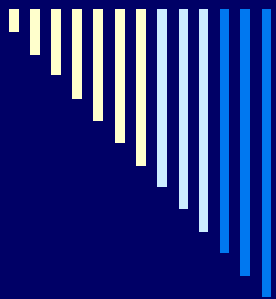
```
class jméno {  
    synchronized void metoda() {  
        ....  
    }  
}
```





# Monitory - Java

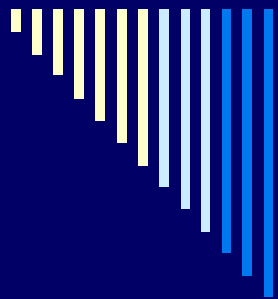
- S monitorem je sdružena **jedna** podmínka, metody:
- **wait()** – pozastaví volající vlákno
- **notify()** – označí jedno spící vlákno pro vzbuzení, vzbudí se, až **volající opustí** monitor (x c.signal, které pozastaví volajícího)
- **notifyAll()** – jako notify(), ale označí pro vzbuzení všechna spící vlákna



# Monitory - Java

Jde o třetí řešení problému, jak ošetřit volání signal (k Hoare, Hansen):

Čekající může běžet až poté, co proces (vlákno) volající signál opustí monitor



# Monitory Java – více podmínek

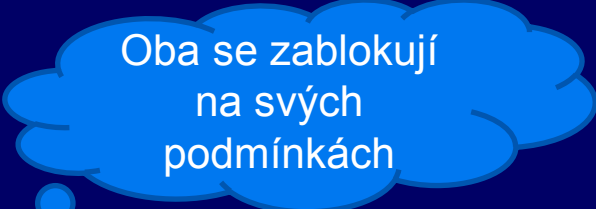
- Více podmínek, může nastat následující (x od Hoarovských monitorů)
- Pokud se proces pozastaví, protože proměnná B byla false, **nemůže** počítat s tím, že po vzbuzení bude B=true



# Více podmínek - příklad

□ 2 procesy, nastalo zablokování:

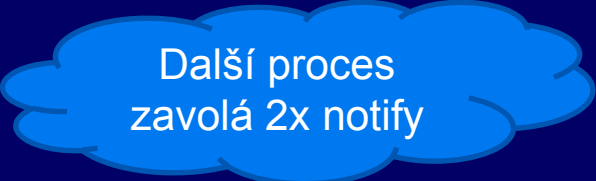
- P1: if not B1 then c1.**wait**;
- P2: if not B2 then c2.**wait**;



Oba se zablokují  
na svých  
podmínkách

□ Proces např. P3 běžící v monitoru způsobí  
splnění obou podmínek a oznámí to pomocí

- If B1 then c1.**notify**;
- If B2 then c2.**notify**;

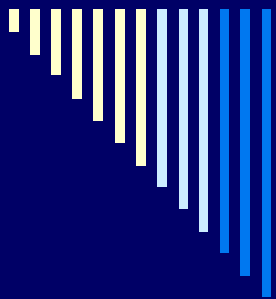


Další proces  
zavolá 2x notify



## Více podmínek - příklad

- Po opuštění monitoru se vzbudí P1
- Proces1 způsobí, že B2=false
- Po vzbuzení P2 bude B2 false, i když by logicky předpokládal, že tomu tak není
- Volání metody wait by mělo být v cyklu (x od Hoarovských)
- **While** not B do **c.wait**;



# Java – volatile proměnné

- poznámka
- Vlákno v Javě si může vytvořit soukromou pracovní kopii sdílené proměnné
- Zapiše zpět do sdílené paměti pouze při vstupu/výstupu z monitoru
- Pokud chceme zapisovat proměnnou při každém přístupu – deklarovat jako **volatile**



# Monitory v C

Úsek kódu ohraničený

- `pthread_mutex_lock(m)`
- ...
- `pthread_mutex_unlock(m)`

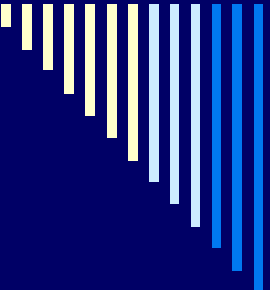
□ Uvnitř lze používat obdobu podmínek z monitorů



# Monitory v C

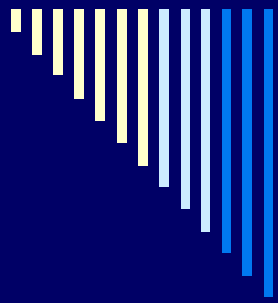
- `pthread_cond_wait(c, m)` - atomicky odemkne m a čeká na podmínku
- `pthread_cond_signal(c)` - označí 1 vlákno spící nad c pro vzbuzení
- `pthread_cond_broadcast(c)` - označí všechna vlákna spící nad c pro vzbuzení





# Shrnutí - monitory

- **Základní varianta** – Hoarovské monitory
- V reálných prog. jazycích **varianty**
  - Prioritní wait (např. BACI)
  - Primitiva wait a notify (Java, Borland Delphi)
- **Výhoda monitorů**
  - Automaticky řeší vzájemné vyloučení
  - Větší odolnost proti chybám programátora
- **Nevýhoda**
  - Monitory – koncepce programovacího jazyka, překladač je musí umět rozpoznat a implementovat



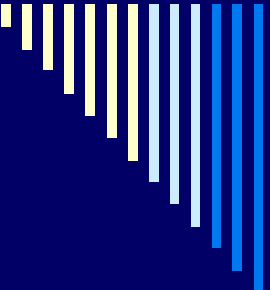
# Řešení producent/konzument pomocí: monitoru

Monitor ProducerConsumer

var

f, e: condition;

i: integer;



```
procedure enter;  
begin  
if i=N then wait(f);      { pamět je plná, čekám }  
enter_item;              { vlož položku do bufferu }  
i:=i+1;  
if i=1 then signal(e);    { první položka => vzbudím konz. }  
end;
```

```
procedure remove;  
begin  
if i=0 then wait(e);      { pamět je prázdná => čekám }  
remove_item;             { vyjmi položku z bufferu }  
i:=i-1;  
if i=N-1 then signal(f);  { je zase místo }  
end;
```



---

# Inicializační sekce

begin

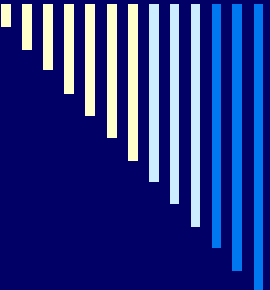
i:=0; { inicializace }

end

end monitor;

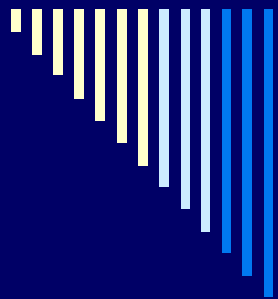
{ A vlastní použití monitoru dále: }

---



```
begin                                // začátek programu
  cobegin
    while true do    { producent }
      begin
        produkuj zaznam;
        ProducerConsumer.enter;
      end {while}
    ||
    while true do    { konzument }
      begin
        ProducerConsumer.remove;
        zpracuj zaznam;
      end {while}
    coend
  end.
```

---



# Implementace monitorů pomocí semaforů

- Monitory musí umět **rozpoznat** překladač programovacího jazyka
- **Přeloží** je do odpovídajícího kódu
- Pokud např. OS poskytuje semaforey může je využít pro implementaci monitoru



# Co musí implementace zaručit

1. Běh procesů v monitoru musí být vzájemně vyloučen (**nanejvýš 1 aktivní v monitoru**)
2. **Wait** musí blokovat aktivní proces v příslušné podmínce
3. Když proces opustí monitor, nebo je blokován podmínkou AND existuje  $>1$  procesů čekajících na vstup do monitoru  
=> musí být jeden z nich **vybrán**



# Implementace monitoru

- Existuje-li proces pozastavený jako výsledek operace **signal**, pak je vybrán
  - Jinak je vybrán jeden z procesů čekajících na vstup do monitoru
4. **Signal** musí zjistit, zda existuje proces čekající nad podmínkou
- **Ano** – aktuální proces pozastaven a jeden z čekajících reaktivován
  - **Ne** – pokračuje původní proces
-





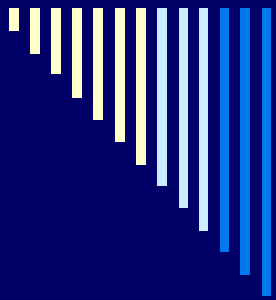
# Implementace monitoru

## Semaforey

```
m = 1;           // chrání přístup do monitoru
u = 0;           // pozastavení procesu při signal()
w[i] = 0;        // pozastavení při wait()
                // pole t semaforů, kolik je podmínek
```

## Čítače

```
ucnt = 0;        // počet pozastavení pomocí signal
wcnt[i]          // počet pozastavených na dané
                // podmínce voláním wait
```



# Vstup do monitoru, výstup z monitoru

Každý proces vykoná následující kód

```
P(m);                                // vstup – zamkne semafor
...                                  // tělo procedury v monitoru
                                     // výstupní kód
if ucnt > 0 then                     // byl někdo zablokován
    V(u);                            //že volal signal? Ano – pustíme ho
else                                 // jinak pustíme další
    V(m);                            // proces do monitoru
```



# Implementace volání c.**wait()**

```
wcnt [i] = wcnt [i] + 1;  
if ucnt > 0 then           // někdo bude pokračovat  
    V(u);                  // blokový na signál  
else                        // nebo ze vstupu  
    V(m);  
P(w[i]);                   // čekáme na podmínce  
wcnt [i] = wcnt [i] - 1;   // čekání skončilo
```



# Implementace volání c.**signal()**

```
ucnt = ucnt + 1;
```

```
If wcnt [i] > 0 then           // někdo čekal nad ci
```

```
begin
```

```
    V(w[i]);                   // pustíme čekajícího
```

```
    P(u);                       // sami čekáme
```

```
end;
```

```
ucnt = ucnt-1;                 // čekání skončilo
```

---