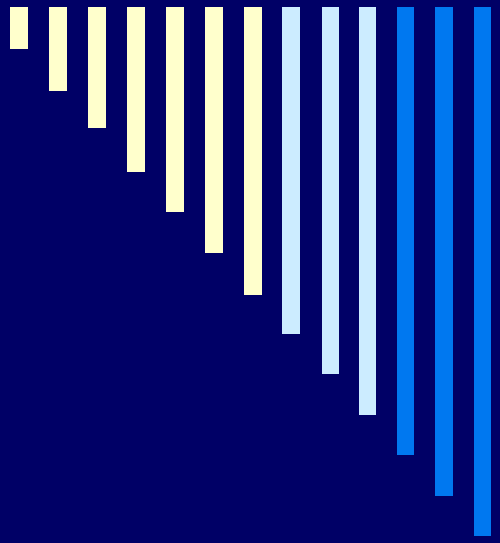
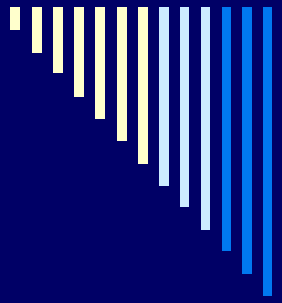


04. Mutexy, monitory

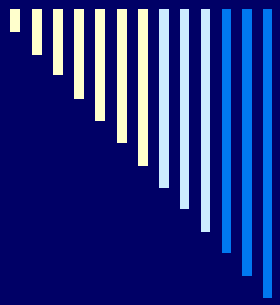
ZOS 2006, L. Pešička



Administrativa ☺

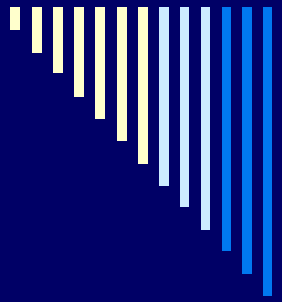
změna termínů zápočtových testů

- 7.11.2006 (út), EP130, 18:30
- 12.12.2006 (út), EP130, 18:30



Semafory

- Ošetření kritické sekce
  - ukázka – více nezávislých kritických sekcí
- Producent – konzument
  - možnost procesu zastavit se a čekat na událost
  - 2 události
    - buffer prázdný – spí jeden proces
    - buffer plný – spí jiný proces
  - „uspání procesu“ – operace semaforu P

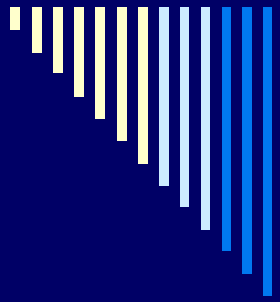


Problém spícího holiče (The barbershop problem)

Holičství

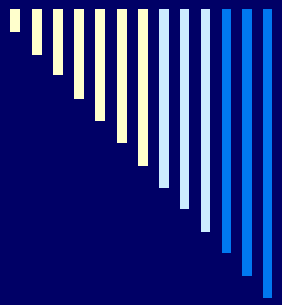
- čekárna s N křesly a holičské křeslo
- žádný zákazník – holič spí
- zákazník vstoupí
  - všechna křesla obsazena – odejde
  - holič spí – vzbudí ho
  - křesla volná – sedne si na jedno z volných křesel

Napsat program, koordinující činnost holiče a zákazníků



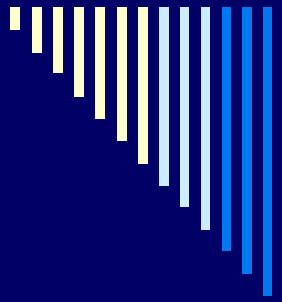
Problém spícího holiče

- proces zákazníka volá funkci getHairCut
- proces holiče volá funkci cutHair
- když holič volá cutHair, je právě jeden proces zákazníka volající getHairCut současně



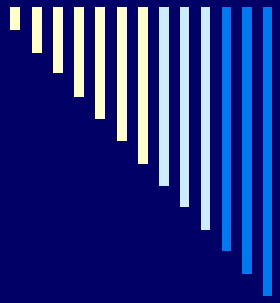
Spící holič – datové struktury

N	počet čekacích křesel
customers = 0	počet zákazníků
mutex = Semaphore(1)	chrání přístup mutexem
customer = Semaphore(0)	holič čeká na zákazníka (má prázdný obchod)
barber = Semaphore(0)	zákazník čeká na holiče než mu řekne že má volno



Proces - zákazník

```
1.  mutex.wait ()
2.  if customers == N+1
3.        { mutex.signal (); odchazim_neostrihan_zklaman(); }
4.  else {
5.    customers += 1;
6.    mutex.signal ();
7.    customer.signal();          // hlásíme příchod zákazníka
8.    barber.wait ();             // čekáme, až má holič čas
9.    getHairCut ();
10.   mutex.wait (); customers -= 1; mutex.signal ();
11.  }
```



Proces – holič

```
1.  while (1) {
2.      customer.wait ();          // čekám na zákazníka
3.      barber.signal ();          // signalizuji, jsem free
4.      cutHair();
5.  }
```

Hilzer's Barbershop

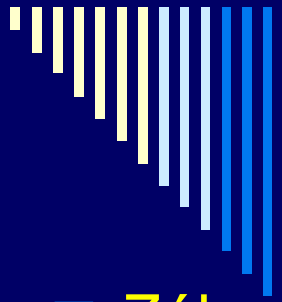
Holičství

- 3 holičská křesla, 3 holiči
- čekárna – 4 sedící a dále stojící zákazníci
- požární předpisy – maximálně 20 zákazníků v holičství

Zákazník

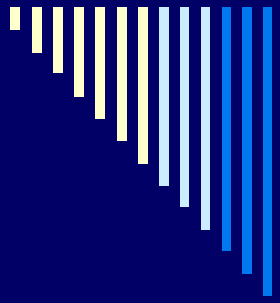
- vstoupí, není-li překročena kapacita
- sedne si na pohovku, nebo stojí, je-li plná
- volný holič – zákazník co nejdéle sedí na pohovce je obsloužen a stojící si sedne
- ostříhaný zákazník – holič vezme platbu (jen jedna pokladna – v daném čase může platit jen jeden zákazník)

Holič – stříhá vlasy, přebírá platbu, spí čekaje na zákazníka



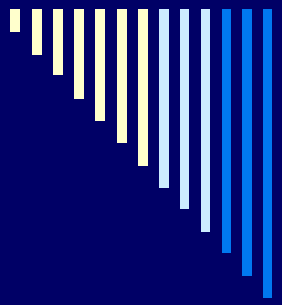
Synchronizace

- Zákazník – následující funkce v daném pořadí
  - enterShop, sitOnSofa, sitInBarberChair, pay, exitShop
- Holič
  - volá cutHair, acceptPayment
- Zákazník nemůže enterShop, je-li plná kapacita
- Když je pohovka plná – nemůže zavolat sitOnSofa dokud jeden ze zákazníků co sedí na pohovce nezavolají sitInBarberChair
- Dokud všichni tři holiči mají práci, zákazník nemůže zavolat sitInBarberChair, dokud někdo ze zákazníků v křesle nezavolá pay
- Zákazník musí pay, než holič acceptPayment
- holič musí acceptPayment, než zákazník může exitShop



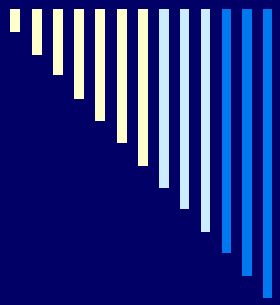
FIFO přístup

- čekající oblasti – „na stojáka“, pohovka
- spravedlivý přístup k zákazníkům
- obsluha zákazníků first-in-first-out
- spravedlivé semafory – využívají FIFO



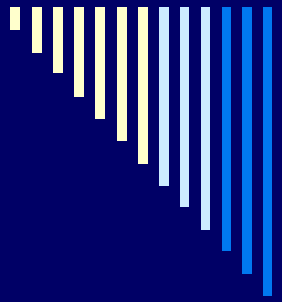
Datové struktury

customers = 0	počet zákazníků
mutex = Semaphore(1)	
standingRoom = Fifo(16)	čekání ve stoje
sofa = Fifo(4)	čekání v sedě
chair = Semaphore(3)	volná hol. křesla
barber = Semaphore(0)	
customer = Semaphore(0)	
cash = Semaphore(0)	zaplatí
receipt = Semaphore(0)	stvrzenka



Holič

```
customer.wait()          // žádný zákazník,spí
barber.signal()          // signalizace, holič je volný
cutHair()

cash.wait()              // čekáme až zákazník zapl.
acceptPayment()
receipt.signal()         // zaplaceno, zákazník ok
```



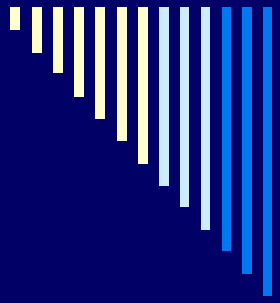
Zákazník

… delší kód

uveden v The Little Book of Semaphores (Allen B. Downey), str. 134 (resp. 146)

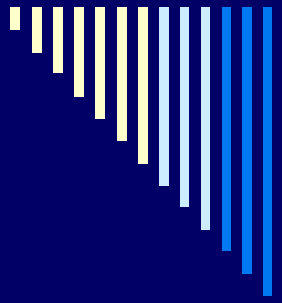
kniha volně dostupná na netu:
http://greenteapress.com/semaphores/

knihu doporučuji zájemcům o problematiku synchronizace v OS



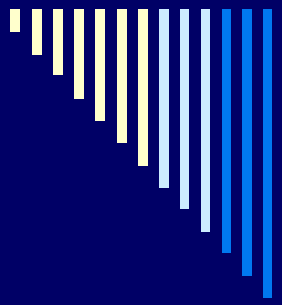
Mutexy, monitory

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



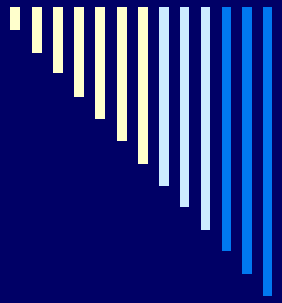
Mutexy

- Potřeba zajistit vzájemné vyloučení
  - spin-lock bez aktivního čekání
  - nepotřebujeme obecně schopnost semaforů čítat
- mutex – mutual exclusion
  - paměťový zámek



Implementace mutexu – podpora jádra OS

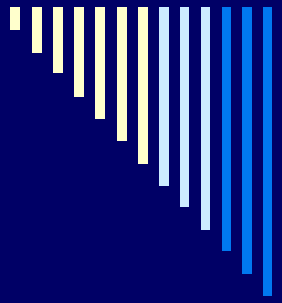
```
mutex_lock:TSL R, mutex     ;;  R:=mutex a mutex:=1
CMP R, 0                    ;; byla v mutex hodnota 0?
JE ok                       ;; pokud byla na OK
CALL yield                  ;; vzdáme se procesoru -
                               naplánuje se jiné vlákno
JMP mutex_lock              ;; zkusíme znovu, pozdější
ok: RET


mutex_unlock:
LD mutex, 0                 ;; ulož  0 do  mutex
RET
```



Implementace mutexu – volání yield

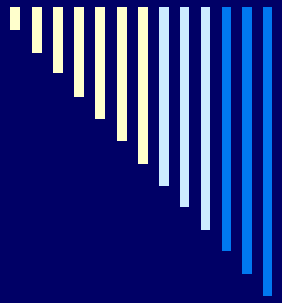
- volající se dobrovolně vzdává procesoru ve prospěch jiných procesů
- jádro OS přesune proces mezi připravené a časem ho opět naplánuje

- použití – např. vzájemné vyloučení mezi vlákny stejného procesu
- lze implementovat jako knihovna prog. jazyka



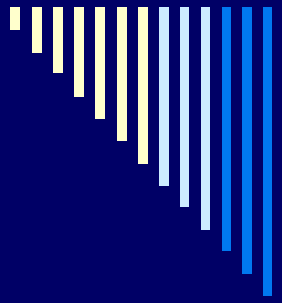
Moderní OS – semafory i mutexy

- obecné (čítající) semafory
  - obecnost
  - i pro řešení problémů meziprocesové komunikace
- mutexy
  - paměťové zámky, binární semafory
  - pouze pro vzájemné vyloučení
  - při vhodné implementaci efektivnější



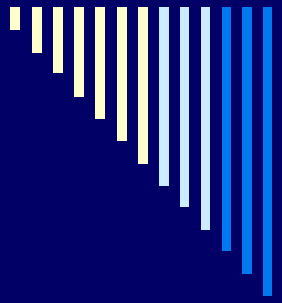
Srovnání – spin-lock

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



Implementace semaforů

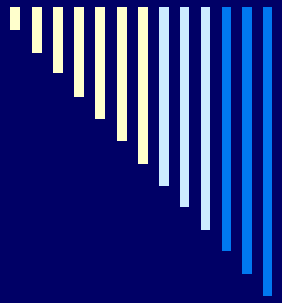
- Procesům poskytneme možnost se pozastavit (zablokovat) při operaci P
- aktivace operací V



Semafory

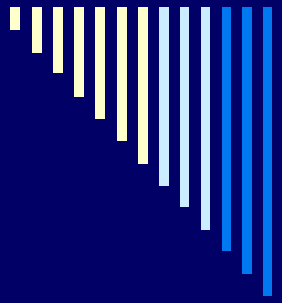
- S každým semaforem je sdruženo:

- celočíselná proměnná **s.c**
  - pokud může nabývat i záporné hodnoty
  - |s.c| vyjadřuje počet blokovaných procesů
- binární semafor **s.mutex**
  - vzájemné vyloučení při operacích nad semaforem
- seznam blokovaných procesů **s.L**



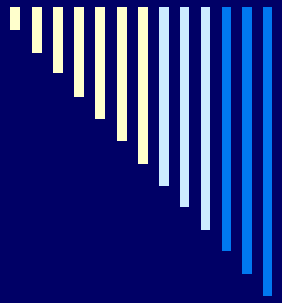
Seznam blok. procesů

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



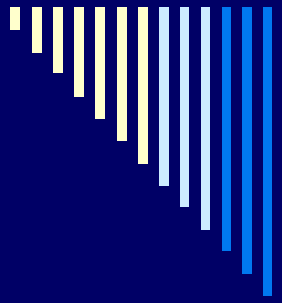
Uložení datové struktury semafor

- semafory v jádře OS
  - přístup pomocí služeb systému
- semafory ve sdílené paměti



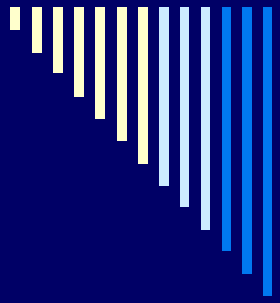
Popis implementace

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



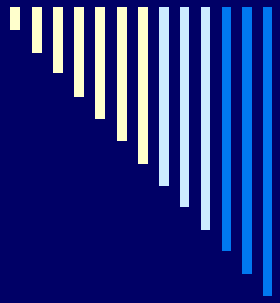
Popis implement. – operace P

```
P(s): mutex_lock(s.m);
    s.c := s.c - 1;
    if s.c < 0 then
        begin
            zařaď volající proces do seznamu s.L;
            označ volající proces jako ”BLOKOVANY´”;
            naplánuj některý připravený proces;
            mutex_unlock(s.mutex);
            přepni kontext na naplánovaný proces
        end
    else
        mutex_unlock(s.m);
```



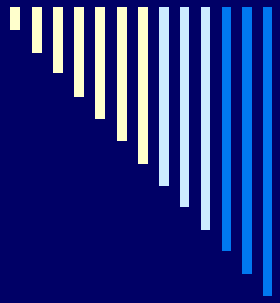
Popis implement. – operace V

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



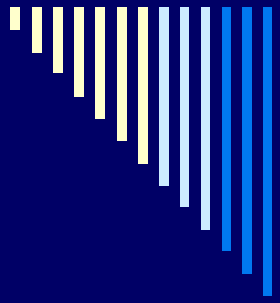
Popis implementace

- 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



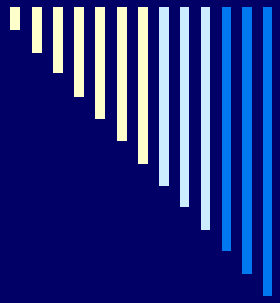
Popis implementace

- Implementace v jádře OS
  - Obvykle používá aktivní čekání (spin-lock nad s.mutex)
  - Pouze po dobu operace nad obecným semaforem – max. desitky instrukcí - efektivní



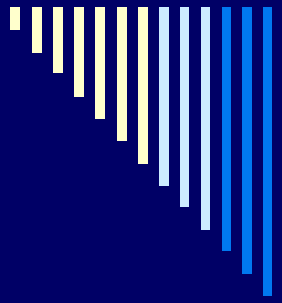
Mutexy vs. semafory

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



Problémy se semafory

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

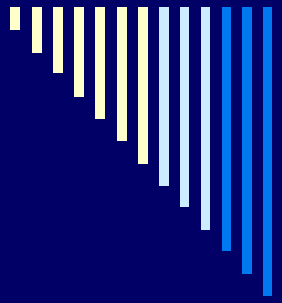


Chyby – přehození P a V

- Přehození P a V operací nad mutexem:

- Mutex.V()
- kritická sekce
- Mutex.P()

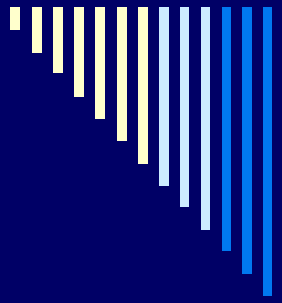
- Důsledek – více procesů může vykonávat kritickou sekci současně



Chyby – dvě operace P

- Mutex.P()
- Kritická sekce
- Mutex.P()

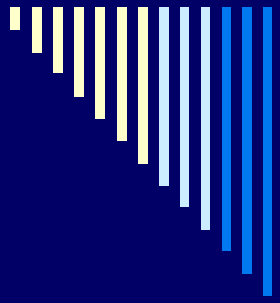
- Důsledek - deadlock



Chyby – vynechání P, V

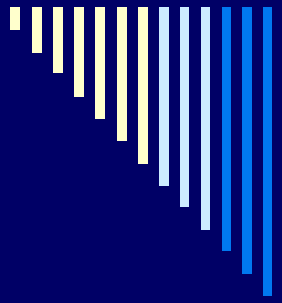
- Proces vynechá mutex.P()
- Proces vynechá mutex.V()
- Vynechá obě

- Důsledek – porušení vzájemného vyloučení nebo deadlock



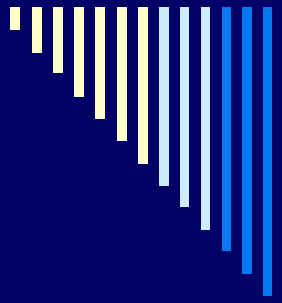
Monitory

- 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



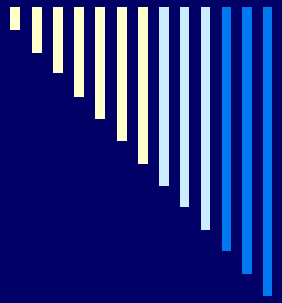
Monitor

- 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



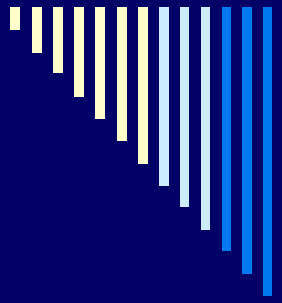
Monitor

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



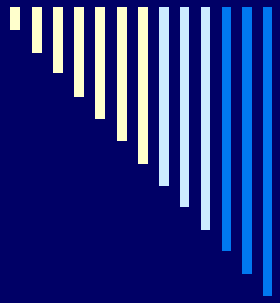
Terminologie OOP

- 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



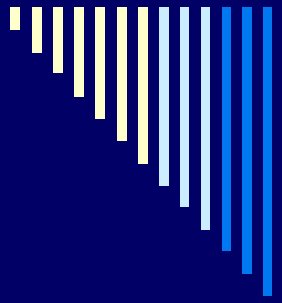
Monitory v BACI

- Monitor – Pascalský 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



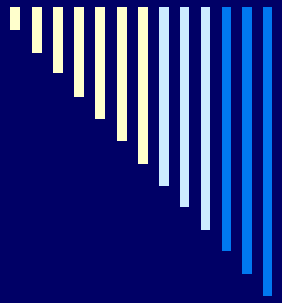
Příklad monitoru

```
monitor m;
    var proměnné ...
        podmínky ...

    procedure p; { procedura uvnitř monitoru }
    begin
        ...
    end;
begin
    inicializace;
end;
```

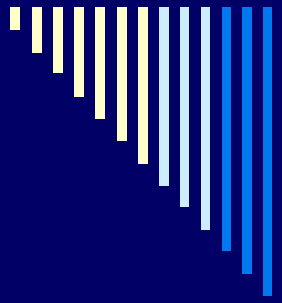


Příklad

- Použití pro vzájemné vyloučení

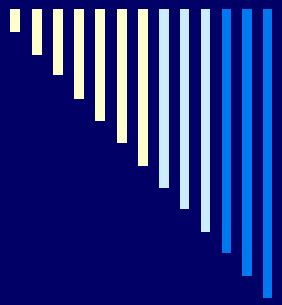


```
monitor m;                  // příklad – vzájemné vyloučení
    var x: integer;
    procedure inc_x;                    { zvětší x }
    begin
        x:=x+1;
    end;
    function get_x: integer;            { vrací x }
    begin
        get_x:=x
    end
begin
    x:=0
end; { inicializace x };
```



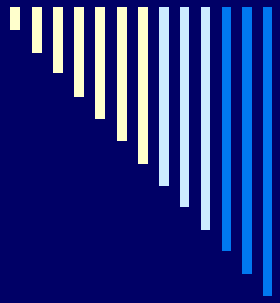
Problém dosavadní definice

- 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

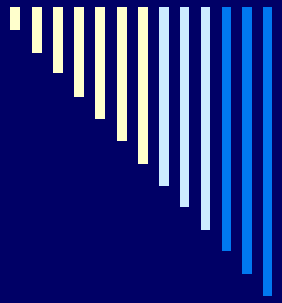
- Monitory – speciální typ proměnné nazývané podmínka (condition variable)
- Podmínky
  - definovány a použity pouze uvnitř monitoru
  - Nejsou proměnné v klasickém smyslu, neobsahují hodnotu
  - Spíše odkaz na určitou událost nebo stav výpočtu (mělo by se odrážet v názvu podmínky)



Operace nad podmínkami

- Definovány 2 operace – wait a signal

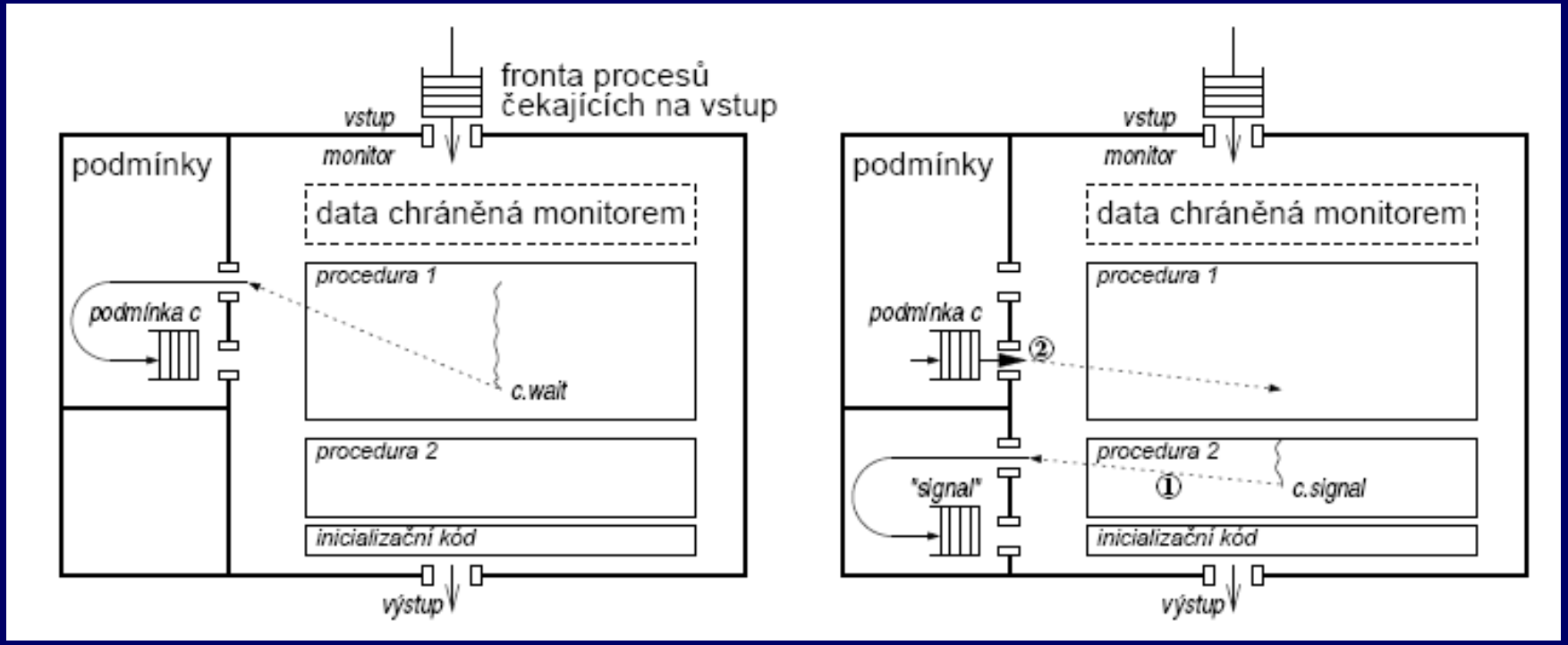
- **C.wait**
- Volající bude pozastaven nad podmínkou C
- Pokud je některý proces připraven vstoupit do monitoru, bude mu to dovoleno

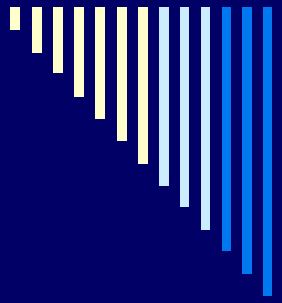


Operace nad podmínkami

- **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 V(sem), která si “zapamatuje”, že byla zavolána

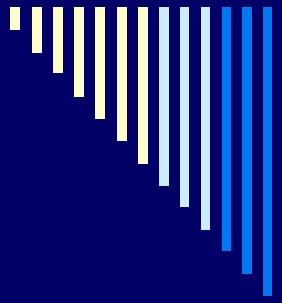
Schéma monitoru





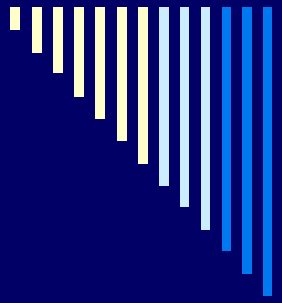
Problém s operací signal

- 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



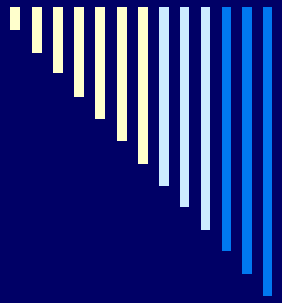
Řešení reakce na signal

- 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



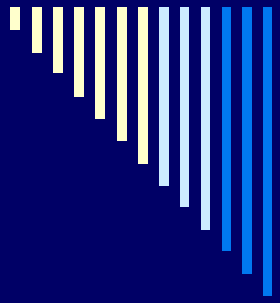
Jak je to v BACI?

- Monitory podle Hoara
- Waitc (cond: condition)
- Signalc (cond: condition)
  - semantika dle Hoara
- Waitc (cond: condition, prio: integer)
  - Čekajícímu je možné přiřadit prioritu
  - Vzbuzen bude ten s nejvyšší prioritou
    Nejvyšší priorita – nejnižší číslo prio



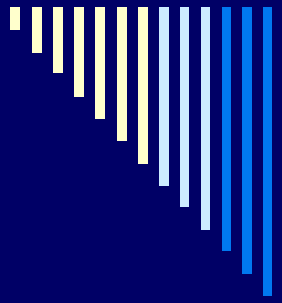
Monitory v jazyce Java

- Existují i jiné varianty monitorů, např. zjednodušené monitory s primitivy wait a notify v jazyce Java a dalších
- S každým objektem je sdružen monitor, může být i prázdný
- Metoda nebo blok patřící do monitoru označena klíčovým slovem synchronized



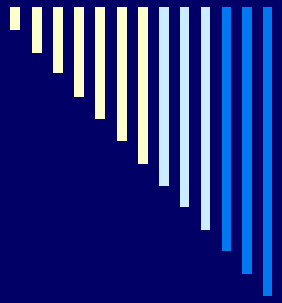
Monitory - Java

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



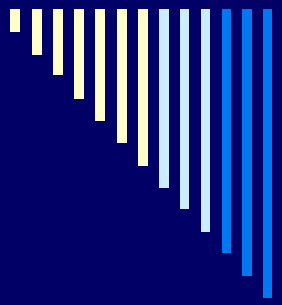
Monitory - Java

- 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



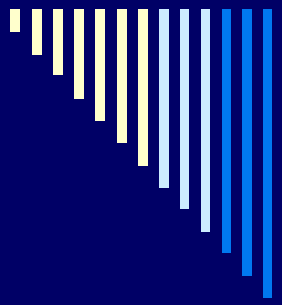
Monitory - Java

- Pozn. Jde vlastně o třetí řešení problému, jak ošetřit volání signal
- Čekající může běžet až poté, co proces volající signál opustí monitor



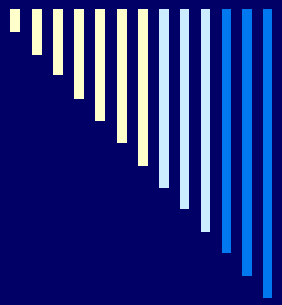
Monitory Java – více podmínek

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



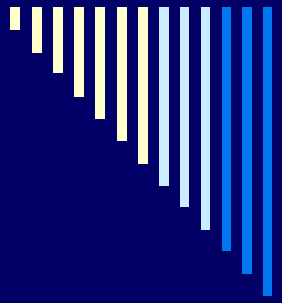
Více podmínek - příklad

- 2 procesy, nastalo zablokování:
  - P1: if not B1 then c1.wait;
  - P2: if not B2 then c2.wait;
- Proces běžící v monitoru, způsobí splnění obou podmínek a oznámí to pomocí
  - If B1 then b1.notify;
  - If B2 then b2.notify;



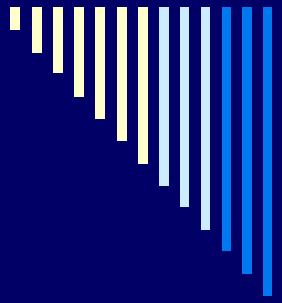
Více podmínek - příklad

- Po opuštění monitoru se vzbudí proces1
- Proces1 způsobí, že B2=false
- Po vzbuzení P2 bude B2 false
- Volání metody wait by mělo být v cyklu (x od Hoarovskych)
- While not B do c.wait;



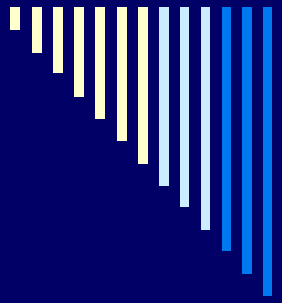
Java – volatile proměnné

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



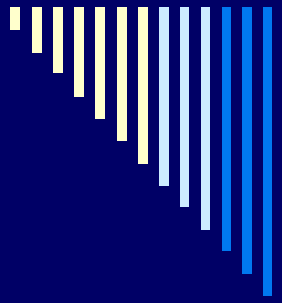
Shrnutí - monitory

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



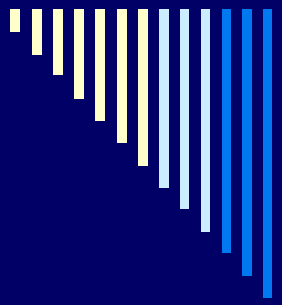
Práce s vlákny v C

- Některé myšlenky i v rozhraní LINUXu pro práci s vlákny
- Úsek kódu ohraničený
  - pthread_mutex_lock(m) .. pthread_mutex_unlock(m)
- Uvnitř lze používat obdobu podmínek z monitorů



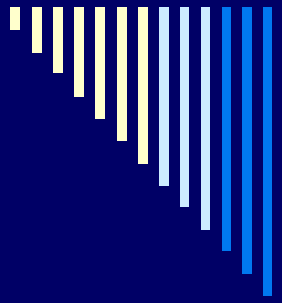
Práce s vlákny v C

- 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

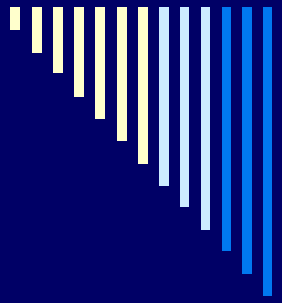


Řešení producent/konzument pomocí monitoru

```
Monitor ProducerConsumer
var
    f, e: condition;
    i: integer;
```



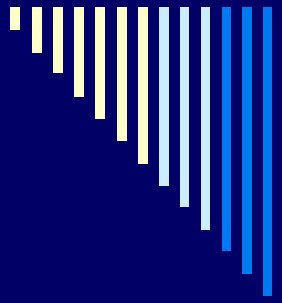
```
procedure enter;
begin
if i=N then wait(f);          {pamět je plná, čekám }
enter_item;                   { vlož položku do bufferu }
i:=i+1;
if i=1 then signal(e);        { prvnií položka => vzbudím konz. }
end;

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;
```



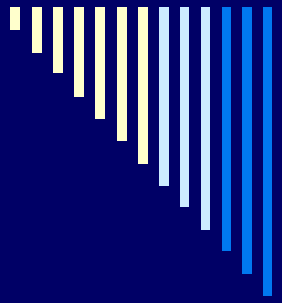
Inicializační sekce

- begin
- i:=0; { inicializace }
- end
- end monitor;

- A vlastní použití monitoru



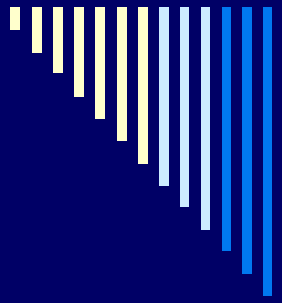
```
begin                              // začátek programu
    cobegin
            while true do       { producent}
            begin
              produkuj zaznam;
              ProducerConsumer.enter;
            end {while}
            ||
            while true do      { konzument }
            begin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



Implementace monitorů pomocí semaforů

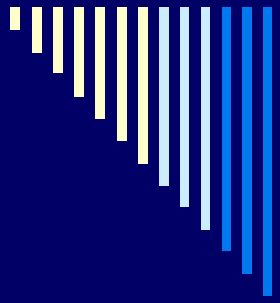
- Monitory musí umět rozpoznat překladač programovacího jazyka
- Přeloží je do odpovídajícího kódu

- Pokud např. OS poskytuje semafory
- Jak by potom implementace vypadala



Co musí implementace zaručit

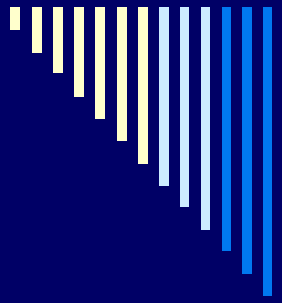
1. Běh procesů v monitoru musí být vzájemně vyloučen (pouze 1 v monitoru)
2. Wait musí blokovat aktivní proces v příslušné podmínce
3. Když proces opustí monitor, nebo je blokován podmínkou AND existuje >1 procesů čekajících na vstup do monitoru
   => musí být jeden z nich vybrán



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 - poznámky

- Podrobný popis – viz *p4monitor.pdf*

- Semafory – pozastavení a odblokování procesů
- Pole semaforů – pro každou podmínku jeden

- Výstupní kod = dovolíme někomu pokračovat
  - Podmínka 3 a, b v tomto pořadí
  - Využívá čítač ucnt