

# 04. Mutexy, monitory

---

ZOS 2016, L. PEŠIČKA

# Semaforey – co znát

---

## Datové struktury

- `int s;` (0, 1, 2, ...)
- fronta procesů (vláken) blokováných na semaforu

## Operace nad semaforem

- `P()` – blokující
- `V()`
- a popis jejich implementace

# Semaforey – základní použití

---

## 1. Ošetření kritické sekce

- semaphore  $s = 1$ ;
- $P(s)$ ; Kritická sekce;  $V(s)$ ;

## 2. Synchronizace

- Semaphore  $s = 0$ ;
- 1. vlákno  $P(s)$
- 2. vlákno  $V(s)$

## 3. Složitější synchronizační úlohy

- Producent - konzument

# Semaforey

---

## 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

# Synchronizační úlohy

---

- modelové příklady
- testují se na nich synchronizační mechanismy, ale i praktické použití
- knížky, které se těmto úlohám věnují
- typické úlohy:
  - Producent – konzument
  - Čtenář – písař
  - Spící holič
  - Večeřící filozofové

# Problém spícího holiče

---

## 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 k synchronizaci

---

The Little Book of Semaphores (Allen B. Downey)

Kniha je volně dostupná na Internetu

<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

# Další části přednášky

---

- ❑ Mutexy, implementace
- ❑ Implementace semaforů
- ❑ Problémy se semaforey
- ❑ Semaforey v C a Javě
- ❑ Monitory
- ❑ Implementace monitorů



# Mutexy (!!)

---

**mutex** – mutual exclusion

- **synchronizační prostředek pro vzájemné vyloučení (!)**
- paměťový zámek, méně náročný než semafor
- nepotřebujeme schopnost semaforů počítat, jen vzájemné vyloučení
- chceme „**spin-lock**“ bez aktivního čekání“

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

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

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



Oblíbená  
instrukce TSL



Vzdát se CPU

# Implementace mutexu

## – volání `yield`



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

podívejte se: [http://linux.die.net/man/2/sched\\_yield](http://linux.die.net/man/2/sched_yield)

# Moderní OS – semafony, mutexy

---

## obecné (čítající) semafony

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

## mutexy, binární semafony

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

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

# Spin-lock (aktivní čekání)

---

**spin-lock** – vhodný, pokud je čekání **krátké** a procesy běží paralelně

není vhodné pro použití v **aplikacích**

- Nevíme jak dlouho budeme čekat, např. na vstup od uživatele

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, ale 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:

```
V1:  
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

---

Stejné 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, že život je pestrý a jsou různé varianty mutexů:  
reentrantní mutex, futex, ...

# Reentrantní mutex

---

```
var m : Mutex // A non-recursive mutex, initially unlocked.  
  
function lock_and_call(i : Integer)  
    m.lock()  
    callback(i)  
    m.unlock()  
  
function callback(i : Integer)  
    if i > 0  
        lock_and_call(i - 1)
```

Nereentrantní mutex - způsobí deadlock při lock\_and\_call (1)

Reentrantní (rekurzivní) mutex – doběhne v pořádku

Příklad – zdroj wikipedia

# 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.

**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

# Futex – dokumentace v Linuxu:

## man 2 futex , man 7 futex

```
FUTEX(2)                                Linux Programmer's Manual                                FUTEX(2)

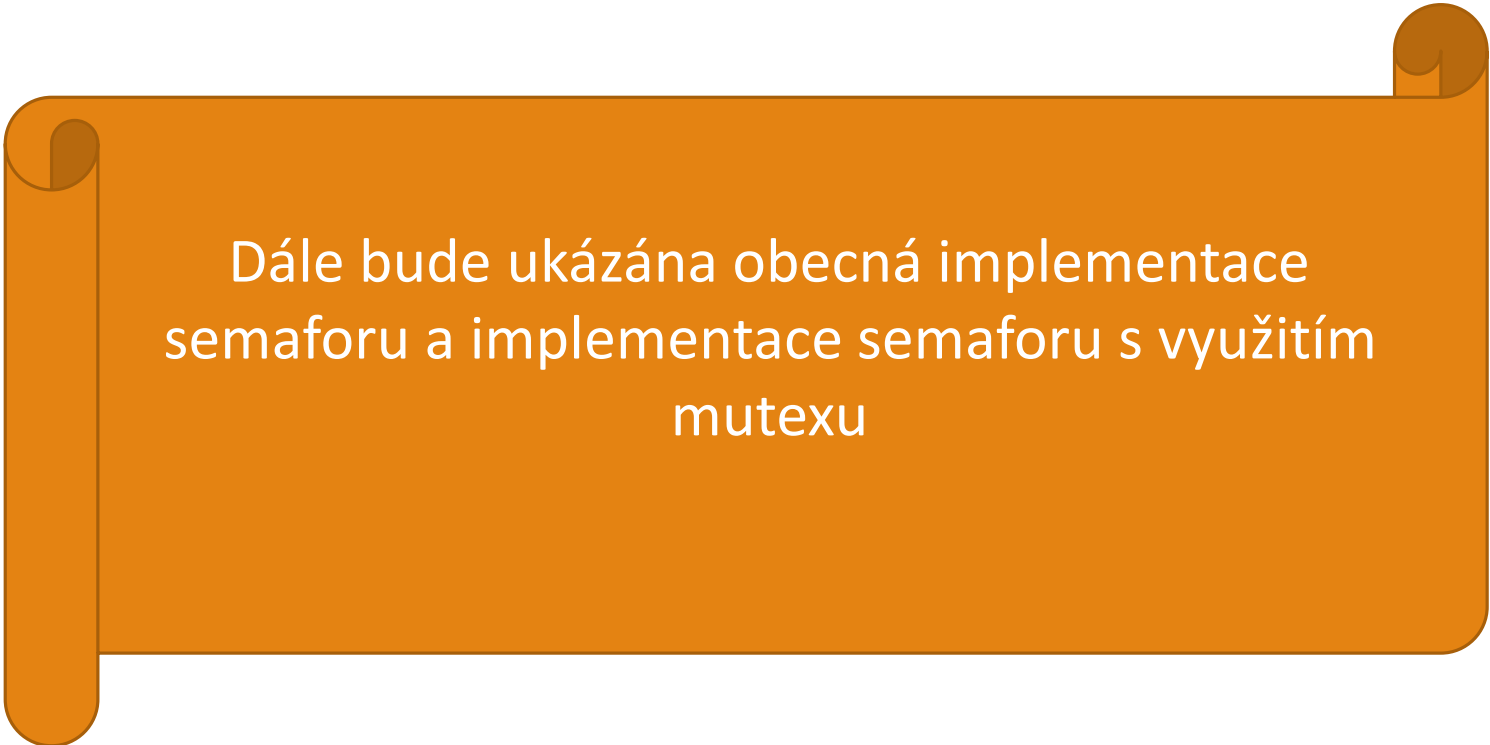
NAME
    futex - fast user-space locking

SYNOPSIS
    #include <linux/futex.h>
    #include <sys/time.h>

    int futex(int *uaddr, int op, int val, const struct timespec *timeout,
              int *uaddr2, int val3);
    Note: There is no glibc wrapper for this system call; see NOTES.

DESCRIPTION
    The futex() system call provides a method for a program to wait for a
    value at a given address to change, and a method to wake up anyone
    waiting on a particular address (while the addresses for the same mem-
    ory in separate processes may not be equal, the kernel maps them inter-
    nally so the same memory mapped in different locations will correspond
    for futex() calls). This system call is typically used to implement
    the contended case of a lock in shared memory, as described in
    futex(7).
```

---



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

---

A large orange scroll graphic with a dark orange border and rounded corners. The scroll is unrolled, showing a white rectangular area in the center. The text "1. ukázka" is written in white on this area. The scroll has a small dark orange tab on the left side and a small dark orange loop on the right side.

1. ukázka

# 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 blokovaných 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);          // blokový -> příprav.  
        }  
}
```

Důležité !!

## 2. Ukázka

Víme, že implementace P() a V() musí být **atomická**, tj.  
Když někdo volá P() a druhý chce také zavolat P(), tak ten druhý musí  
**počkat**

Jak to zařídit?

**Třeba tak, že využijeme mutex**

Všimněte si:

**TSL -> mutex -> semafor**

**TSL -> aktivní čekání**

# 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

---

semafony v **jádře OS**

- přístup pomocí služeb systému

semafony ve **sdílené paměti**

# Popis implementace

---

```
type semaphore = record  
  m: mutex;           // mutex pro přístup k semaforu  
  c: integer;         // hodnota semaforu  
  L: seznam procesu   // fronta blokováných  
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. je-li 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 při ochraně kritické sekce:

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



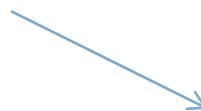
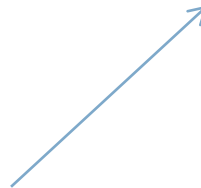
# Semafor

## obecný koncept, programovací jazyk

---

### obecný koncept

semafor  
 $s = 0, 1, 2, \dots$   
 $P(), V()$



### Java

```
java.util.concurrent  
Semaphore (int ..)  
• acquire() <-> P()  
• release() <-> V()  
• tryAcquire()
```

### C

```
#include<semaphore.h>  
sem.wait() <-> P()  
sem.post() <-> V()
```

# 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>

# Java semaforey (vybrané operace)

---

dokumentace:

<http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html>

java.util.concurrent.Semaphore

funkce	popis
public Semaphore(int permits)	Vytvoří semafor inicializovaný na hodnotu permits
acquire()	Operace P() nad semaforem
release()	Operace V() nad semaforem
tryAcquire()	Neblokující pokus o P()

# 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();
```

# Semaforey - C

## (Posixové semaforey)

---

```
#include <semaphore.h>
```

```
sem_t s;
```

Funkce	popis
<code>sem_init(&amp;s, 0, 1);</code>	Inicializuje semafor na hodnotu <b>1</b> prostřední hodnota říká: <b>0</b> – semafor mezi vlákny <b>1</b> – semafor mezi procesy
<code>sem_wait(&amp;s);</code>	Operace P() nad semaforem
<code>sem_post(&amp;s);</code>	Operace V() nad semaforem
<code>sem_destroy(&amp;s);</code>	Zrušení semaforu

# 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);
```

---

# Příklad – semaforey v C



```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
```

```
sem_t s;           /* semafor */
int x = 0;         /* sdilena promenna */
```

```
int main(void)
{
```

```
    pthread_t a, b;    /* id vlaken */
    void *pocitej();    /* funkce vlakna */
```

```
    /* inicializace semaforu s pocatecni hodnotou 1 */
```

```
    /* viz man sem_init */
```

```
    if (sem_init(&s, 0, 1) < 0) {
```

```
        perror("sem_init");
```

```
        exit(1);
```

```
    }
```

Příklad – I.

inicializace  
semaforu

```
/* vytvoreni vlaken */
```

```
    if (pthread_create(&a, NULL, pocitej, NULL) != 0) {  
        exit(1);  
    }  
  
    if (pthread_create(&b, NULL, pocitej, NULL) != 0) {  
        exit(1);  
    }
```

Příklad – II.

vytvoření  
vláken

čekání na  
dokončení  
vláken

```
/* cekame na dokonceni vlaken */
```

```
pthread_join(a, NULL);  
pthread_join(b, NULL);
```

---

```
/* zruseni semaforu */  
    sem_destroy(&s);  
    printf("Vysledna hodnota x: %d\n ", x);  
    return 0;  
}
```

Příklad – III.

Zrušení  
semaforu

```
/* funkce vlákna */  
/* obě vytvořená vlákna budou vykonávat tuto funkci */  
/* přístup ke sdílené proměnné x představuje kritickou sekci */
```

```
void *pocitej()  
{  
    int i;  
  
    for (i=0; i<50; i++) {  
        sem_wait(&s); /* operace P(s) */  
        x++;          /* kritická sekce */  
        sem_post(&s); /* operace V(s) */  
    }  
    return NULL;  
}
```

Příklad – IV.  
funkce vlákna

# System V semaforey

---

pro doplnění

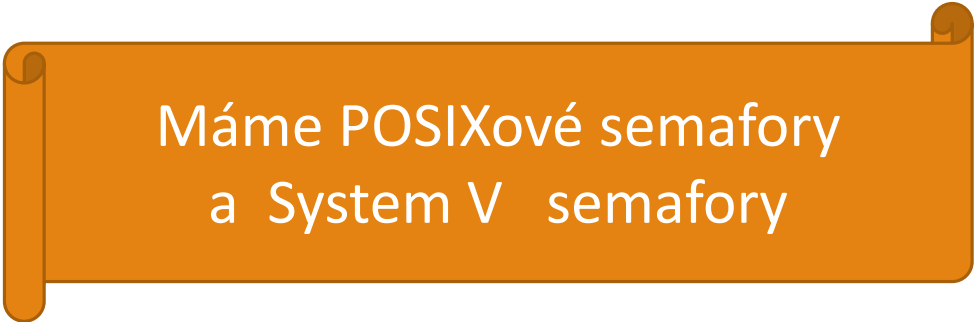
alokují se, používají, ruší podobně jako sdílená paměť

vytváří se celé **pole** semaforů

`semget()`

`semctl()`

`semop()`

An orange callout box with a folded paper effect, containing text about POSIX and System V semaphores.

Máme POSIXové semaforey  
a System V semaforey

# Rozdíl POSIX vs. System V semaforey

---

<http://stackoverflow.com/questions/368322/differences-between-system-v-and-posix-semaphores>

pro naše účely většinou použijeme POSIXové semaforey

- One marked difference between the System V and POSIX semaphore implementations is that in System V you can control how much the semaphore count can be increased or decreased; whereas in POSIX, the semaphore count is increased and decreased by 1.
- POSIX semaphores do not allow manipulation of semaphore permissions, whereas System V semaphores allow you to change the permissions of semaphores to a subset of the original permission.
- Initialization and creation of semaphores is atomic (from the user's perspective) in POSIX semaphores.
- From a usage perspective, System V semaphores are clumsy, while POSIX semaphores are straight-forward
- The scalability of POSIX semaphores (using unnamed semaphores) is much higher than System V semaphores. In a user/client scenario, where each user creates her own instances of a server, it would be better to use POSIX semaphores.
- System V semaphores, when creating a semaphore object, creates an array of semaphores whereas POSIX semaphores create just one. Because of this feature, semaphore creation (memory footprint-wise) is costlier in System V semaphores when compared to POSIX semaphores.
- It has been said that POSIX semaphore performance is better than System V-based semaphores.
- POSIX semaphores provide a mechanism for process-wide semaphores rather than system-wide semaphores. So, if a developer forgets to close the semaphore, on process exit the semaphore is cleaned up. In simple terms, POSIX semaphores provide a mechanism for non-persistent semaphores.

---

Two major problems with POSIX shared/named semaphores used in separate processes (not threads): POSIX semaphores provide no mechanism to wake a waiting process when a different process dies while holding a semaphore lock. This lack of cleanup can lead to zombie semaphores which will cause any other or subsequent process that tries to use them to deadlock. There is also no POSIX way of listing the semaphores in the OS to attempt to identify and clean them up. The POSIX section on SysV IPC does specify the `ipcs` and `ipcrm` tools to list and manipulate global SysV IPC resources. No such tools or even mechanisms are specified for POSIX IPC, though on Linux these resources can often be found under `/shm`. This means that a KILL signal to the wrong process at the wrong time can deadlock an entire system of interacting processes until reboot.



# Mutexy - C

---

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

```
pthread_mutex_t m;
```

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

```
// kritická sekce
```

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

# C Mutex

---

funkce	popis
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;	Inicializace mutexu Implicitně je odemčený
pthread_mutex_ <b>destroy</b> (&m)	Zrušení mutexu
pthread_mutex_ <b>lock</b> (&m)	Pokusí se zamknout mutex. Pokud je mutex již zamčený, je volající vlákno zablokováno.
pthread_mutex_ <b>unlock</b> (&m)	Odemkne mutex
pthread_mutex_ <b>trylock</b> (&m)	Pokusí se zamknout mutex. Pokud je mutex již zamčený, vrátí se okamžitě s kódem EBUSY

# C Mutex - příklad

---

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

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
int x;
```

```
void inkrementuj() {
```

```
    pthread_mutex_lock (&mutex);
```

```
    x++;  /* kritická sekce */
```

```
    pthread_mutex_unlock (&mutex);
```

```
}
```

# Ukázky programů

---



























Courseware KIV/ZOS

=> Cvičení

=> Materiály ke cvičení

=> C, Java příklady

## C, Java příklady

	<a href="#">cobegin/coend</a> (265KB)	
	<a href="#">graf_paralel</a> (357KB)	
	<a href="#">graf_fork</a> (338KB)	
	<a href="#">Semafor</a> (594KB)	
	<a href="#">fork_prikklady</a> (2KB)	
	<a href="#">fork_IPC</a> (4KB)	
	<a href="#">pthreads-semafor</a> (1KB)	
	<a href="#">Příklady synchronizace</a> (38KB)	
	<a href="#">Příklady synchronizace 2</a> (6KB)	
	<a href="#">ProducentKonzument (!)</a> (13KB)	
	<a href="#">prikklady_vlakna.zip</a> (15KB)	
	<a href="#">monitorJavanew</a> (7KB)	
	<a href="#">pr_zavoznik</a> (15KB)	

# Monitory

---

Synchronizační mechanismus

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

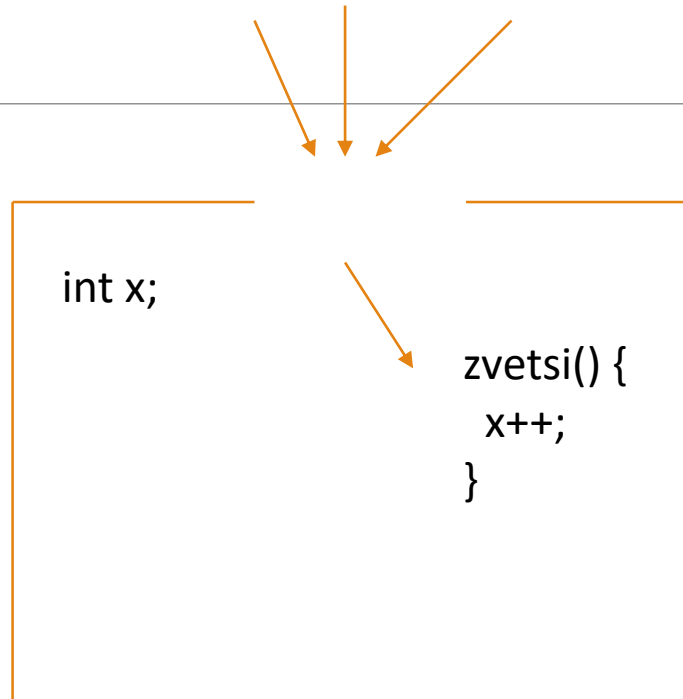
# Monitor

---

V monitoru může být **v jednu chvíli**  
**AKTIVNÍ** pouze jeden proces !!

Ostatní procesy jsou při pokusu o vstup do monitoru  
**pozastaveny**

Procesy (vlákna)  
čekající na vstup do monitoru



Pouze jeden aktivní  
proces (vlákno) uvnitř  
monitoru

Může provést bez obav  
`x++`, protože žádný  
souběh nad `x` nehrozí

Až opustí monitor, může  
dovnitř další čekající  
proces (vlákno)



# 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**

# Monitor

---

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;
```

```
int x;
```

```
void inc_x();
```

```
{
```

```
    x=x+1;
```

```
};
```

```
int get_x();
```

```
{
```

```
    return x;
```

```
}
```

```
begin
```

```
    x=0
```

```
end;
```

```
// příklad – vzájemné vyloučení
```

```
{ zvětší x }
```

```
{ vrací x }
```

```
{ 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)`



# Operace nad podmínkami

---

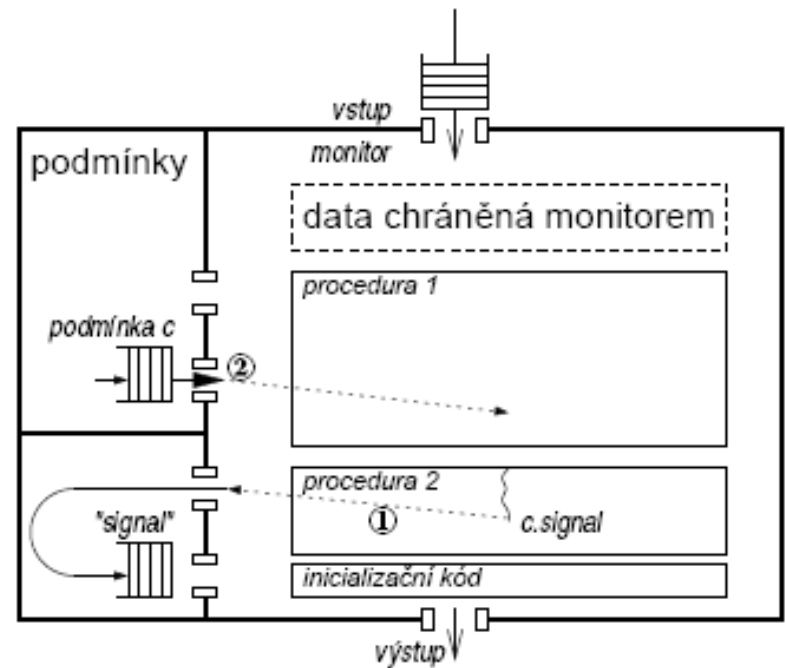
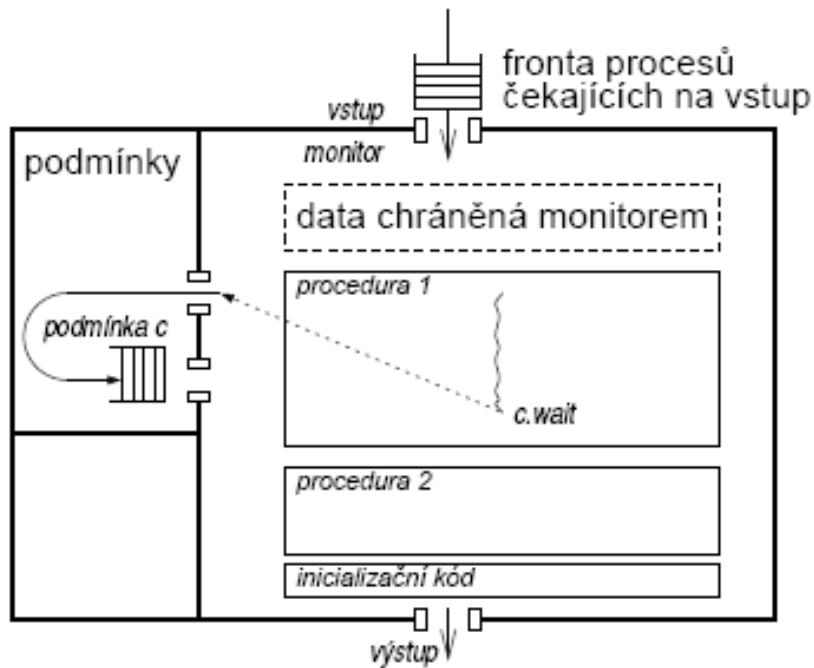
## C.signal

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

# 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 = monitor

# 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 jmeno {  
    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 (Hoare, Hansen, [Java](#)):

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):

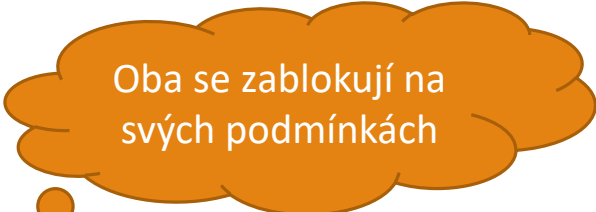
Pokud se proces pozastaví,  
protože proměnná **B** byla **false**,  
**nemůže** počítat s tím, že po vzbuzení bude **B** rovna **true**.

# 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 – po vzbuzení znovu otestovat(x od Hoarovských)

Místo if not B do c.wait použít:

**while** not B do **c.wait**; !!!! PAMATOVAT !!!!

# Pamatuj (!!)

---

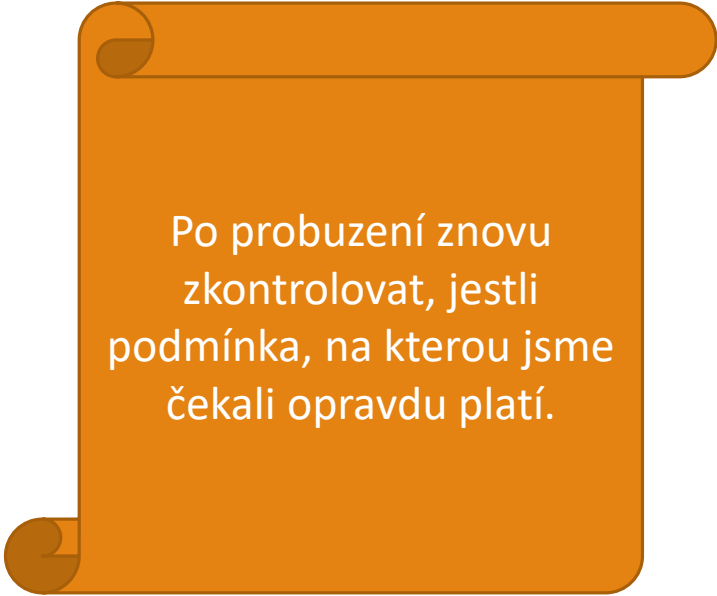
Pokud nevíme, jakou sémantiku monitor používá, je nejlépe místo `if` používat preventivně `while`.

Tedy místo:

`if not B do c.wait`

Používat:

`while not B do c.wait;`



Po probuzení znovu  
zkontrolovat, jestli  
podmínka, na kterou jsme  
čekali opravdu platí.

# 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**

# 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

## 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 wait(f);
```

```
{paměť je plná, čekám }
```

```
  enter_item;
```

```
{ vlož položku do bufferu }
```

```
  i=i+1;
```

```
  if i==1 signal(e);
```

```
{ první položka => vzbudím konz. }
```

```
end;
```

---

```
procedure remove;
```

```
begin
```

```
if i==0  wait(e);      { pamět je prázdná => čekám }
```

```
remove_item;          { vyjmi položku z bufferu }
```

```
i=i-1;
```

```
if i== (N-1)  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
        produkuje 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  $\Rightarrow$  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 zablokovaný
    V(u);        //že volal signal? Ano – pustíme ho
else            // jinak pustíme další
    V(m);        // proces do monitoru ze vstupu
```

# Implementace volání c.wait()

---

```
wcnt [i] = wcnt [i] + 1;
```

```
if ucnt > 0 then           // někdo bude pokračovat
```

```
    V(u);                  // blokováný 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
```

# Monitory v programovacím jazyce

---

## C

- mutex + podmínková proměnná = monitor (pthread.h)

## Java

- mutex + podmínková proměnná = monitor (java.util.concurrent)
- Synchronizované metody

Java má dvojí způsob monitorů

# Použití monitoru v C

---

```
pthread_mutex_t zamek = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_cond_t c1 = PTHREAD_COND_INITIALIZER;
```

```
pthread_mutex_lock( &zamek);
```

- `pthread_cond_wait ( &c1, &zamek );`
- `pthread_cond_signal ( &c1 );`

```
pthread_mutex_unlock( &zamek);
```