

# KIV/ZOS

Cvičení 8, 2014

L. Pešička

# NAKRESLETE GRAF, ZAPIŠTE POMOCÍ COBEGIN/COEND

Tři příklady:

- $(a+b) * (c-d) - (e/f)*(g-h)$
- $a+b+c$
- $a+b+c+d$

Platí běžné precedence operátorů

Každý operátor představuje určitý proces



# KOLIKRÁT SE VYPÍŠE TEXT?

```
int i;
```

```
for (i=0; i<3; i++) {  
    fork();  
    execl("/usr/bin/cal","cal",NULL)  
}  
printf("text");
```



# KOLIKRÁT SE VYPÍŠE AHOJ? NAKRESLETE STROM PROCESŮ

```
fork();  
printf("ahoj");  
fork();  
printf("ahoj");  
if ( fork() == 0)  
    printf("ahoj");
```



# VLÁKNA

Tutoriál popisující vlákna:

<https://computing.llnl.gov/tutorials/pthreads/>

Využit jako zdroj pro některé z následujících slidů  
(text, obrázky)



# PROCES UNIXU

## – OBSAHUJE INFORMACE:

- Proces ID, proces group ID, user ID, group ID
- Prostředí
- Pracovní adresář
- Instrukce programu
- Registry
- Zásobník (stack)
- Halda (heap)
- Popisovače souborů (file descriptors)
- Signal actions
- Shared libraries
- IPC (fronty zpráv, roury, semaforey, sdílená paměť)



# PROCESS GROUP, SESSION

## Process group

- Kolekce jednoho či více procesů
- Pro řízení distribuce signálů
- Signál pro procesní skupinu je distribuován každému členu skupiny

## ○ Sessions

- Procesní skupiny se grupují do sessions
- Process group nemohou migrovat z jedné session do jiné
- Proces může vytvořit novou process group patřící ke stejné session jako on



# VLÁKNO MÁ VLASTNÍ:

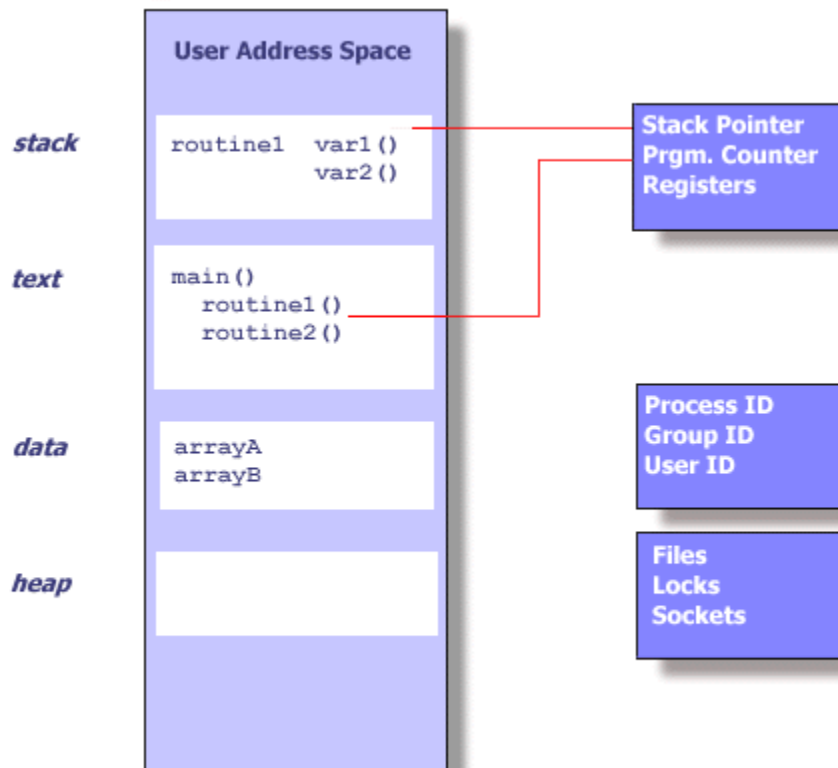
- Zásobník (stack pointer)
- Registry
- Plánovací vlastnosti (policy, priority)
- Množina pending a blokováných signálů
- Data specifická pro vlákno

====

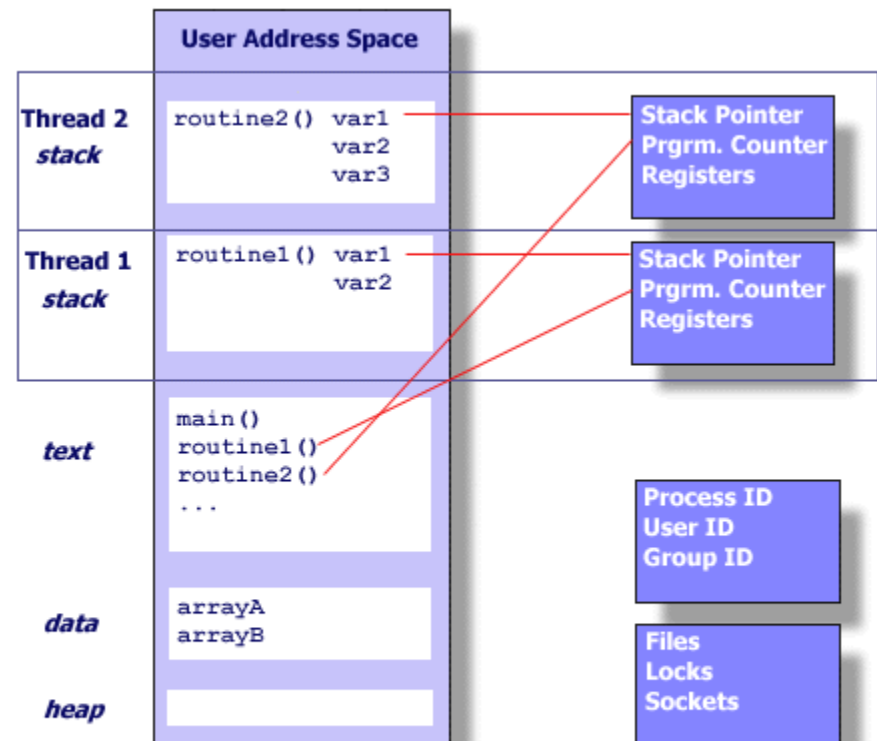
- Všechna vlákna uvnitř procesu sdílejí stejný adresní prostor
- Mezivláknová komunikace je efektivnější a snadnější než meziprocesová







**UNIX PROCESS**



**THREADS WITHIN A UNIX PROCESS**



# ROZDĚLENÍ PAMĚTI PRO PROCES

Roste halda



Roste zásobník



Máme-li více vláken => více zásobníků, limit velikosti zásobníku



Vytvořené vlákno



# ZÁSOBNÍK PRO VLÁKNO

- Při vytvoření vlákna můžeme specifikovat velikost zásobníku
- Je potřeba celkem šetřit..  
Při max. velikost  $8\text{MB} * 512 \text{ vláken} = 4 \text{ GB}$

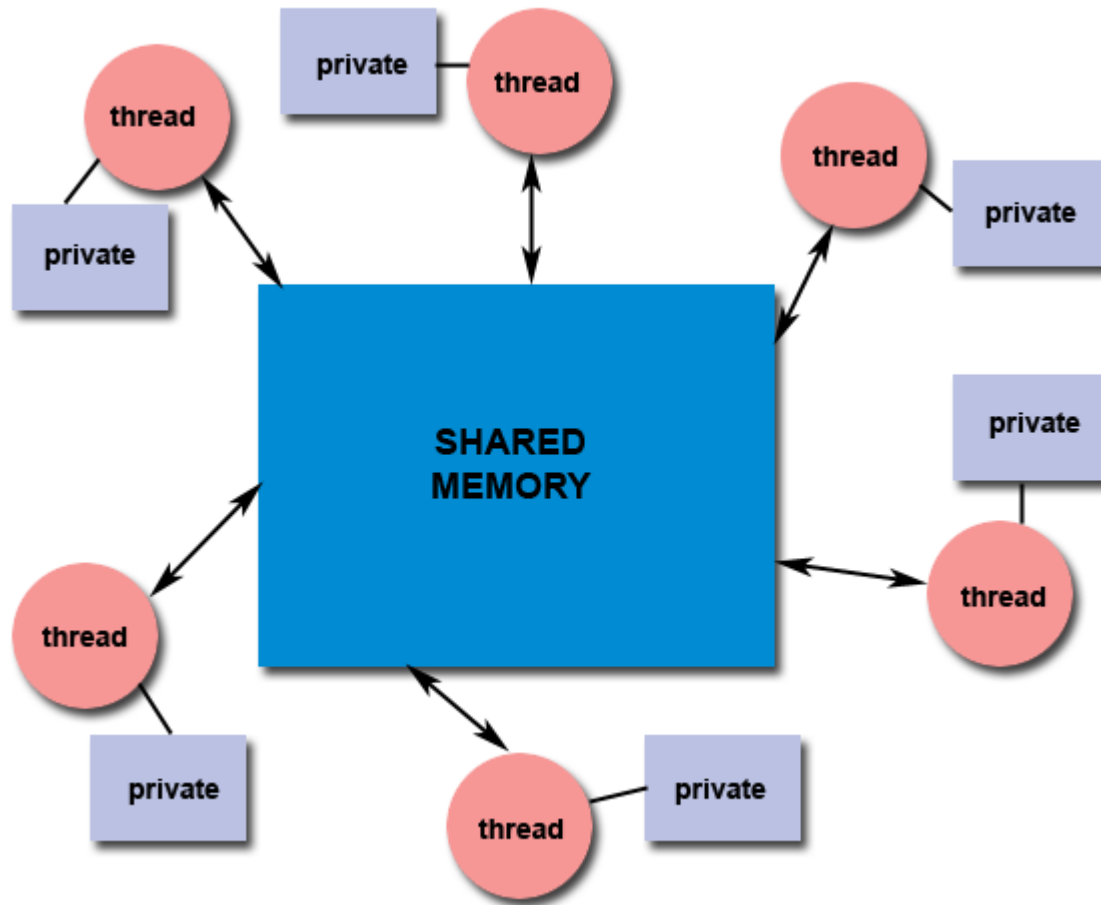


# PTHREADS

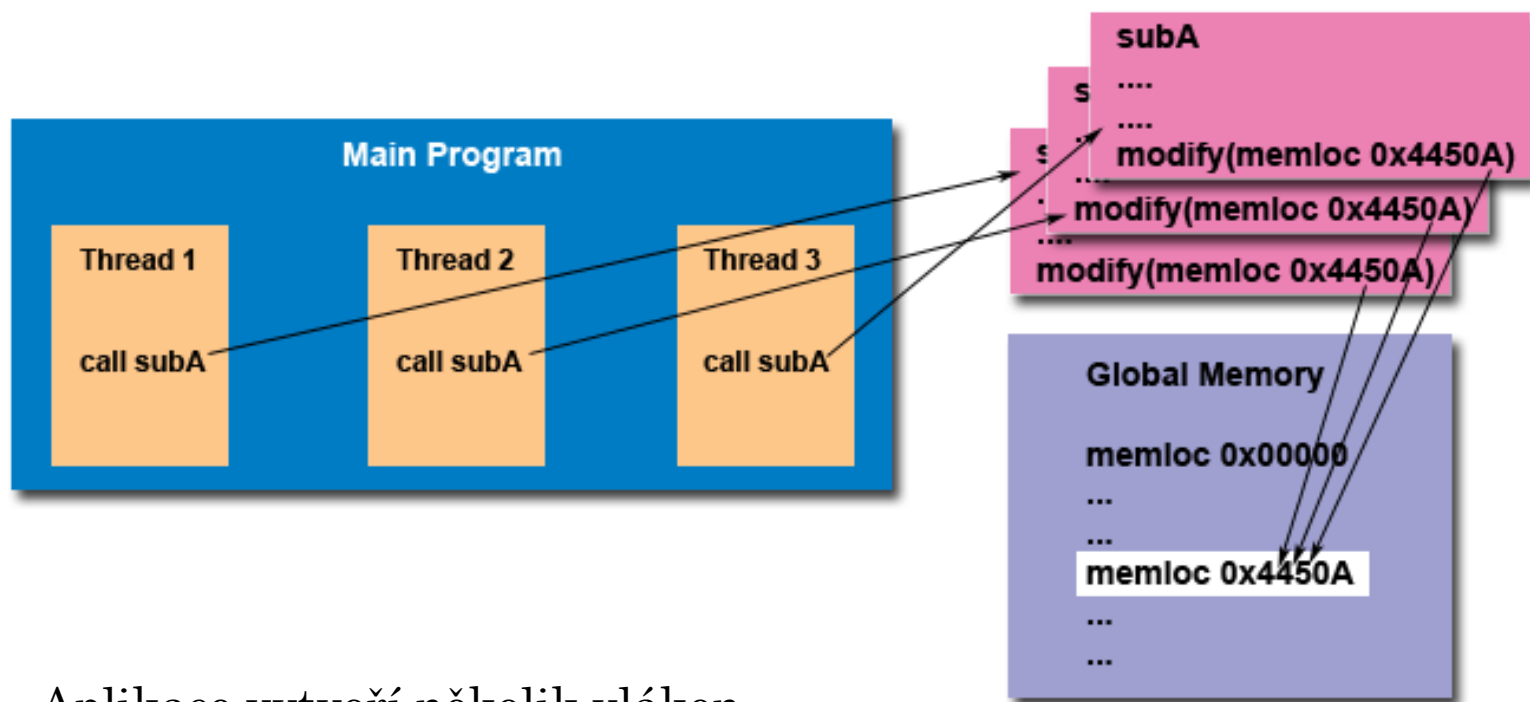
- Rozhraní specifikované IEEE POSIX 1003.1c (1995)
  - Implementace splňující tento standard:  
POSIX threads , pthreads
  - Popis v [pthread.h](#)
- 
1. **Management** vláken (create, detach, join)
  2. **Mutexy** (create, destroy, lock, unlock)
  3. **Podmínkové proměnné** (create, destroy, wait, signal)
  4. **Synchronizace** (read-write locks, bariéry)



# GLOBÁLNÍ A PRIVÁTNÍ PAMĚŤ VLÁKNA



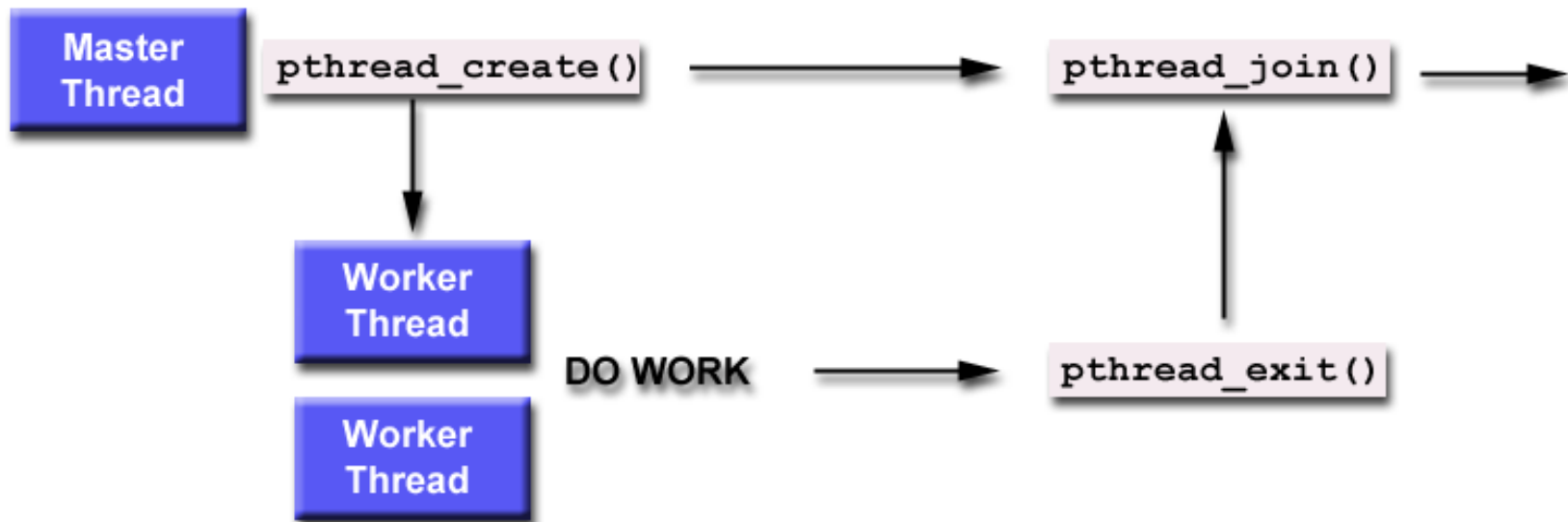
# VLÁKNOVÁ BEZPEČNOST (THREAD-SAFE)



Aplikace vytvoří několik vláken  
Každé vlákno vyvolá stejnou rutinu  
Tato rutina modifikuje společná globální  
data – pokud nemá synchronizační  
mechanismy, není thread-safe



# ČEKÁNÍ NA DOKONČENÍ VLÁKEN



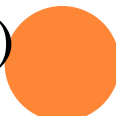
# MOŽNOSTI UKOČENÍ VLÁKNA

- Vlákno dokončí „proceduru vlákna“
- Vlákno kdykoliv zavolá `pthread_exit()`
- Vlákno je zrušené jiným přes `pthread_cancel()`
- PROCES zavolá `exec()` nebo `exit()`
- Pokud `main()` skončí první bez explicitního volání `pthread_exit()`





# VLÁKNA: VYTVOŘENÍ VLÁKNA

- **#include <pthread.h>** .. vlákna pthread
  - **pthread\_t a, b;** .. id vláken a,b
  - **pthread\_create(&a, NULL, pocitej, NULL)**
    - **a** – id vytvořeného vlákna
    - **NULL** – atributy vlákna (man pthread\_attr\_init)
    - **pocitej** – funkce vlákna
    - **NULL** – argument předaný funkci pocitej
    - **Návratová hodnota** – 0 když se vlákno podařilo vytvořit
  - **pthread\_join(a, NULL);**
    - Čeká na dokončení vlákna s id **a**
    - Vlákno musí být v joinable state (ne detach, viz atributy)
    - **NULL** – místo null lze číst návrat. hodnotu
- 

# PŘEDÁNÍ PARAMETRU VLÁKNU

```
void *print_message_function( void *ptr );
```

```
    // hlavička funkce vlákna
```

```
pthread_t thread1, thread2;
```

```
char *message1 = "Thread 1";
```

```
char *message2 = "Thread 2";
```

```
int iret1, iret2;
```

```
iret1 = pthread_create( &thread1, NULL,  
print_message_function, (void*) message1);
```

```
    iret2 = pthread_create( &thread2, NULL,  
print_message_function, (void*) message2);
```



# FUNKCE VLÁKNA – ZPRACOVÁNÍ PARAMETRU

```
void *print_message_function( void *ptr )  
{  
    char *message;  
    message = (char *) ptr;  
    printf("%s \n", message);  
}
```



# DALŠÍ UKÁZKA PŘEDÁNÍ PARAMETRU VLÁKNU

**//vytvareni vlaken**

```
for (i = 0; i < THREAD_COUNT; i++) {  
    thID = malloc(sizeof(int));  
    *thID = i + 1;  
    pthread_create(&threads[i], NULL, thread,  
thID);  
}
```

**// funkce vlakna**

```
void *thread(void * args) {  
    printf("Jsem vlakno %d\n", *((int *) args) );  
}
```



# PŘÍKLADY

- Příklad

Courseware – ZOS – Cvičení – C, Java příklady:  
pthreads-semafor



# VLÁKNA: OŠETŘENÍ KS SEMAFOREM(!!)

## Ošetření KS semaforem:

- `#include <semaphore.h>` .. využijeme semafor
- `sem_t s;` .. typ semafor
- `sem_init(&s, 0, 1);` .. inicializace semaforu na hodnotu **1** !!
  
- `sem_wait(&s);` .. operace P(s);
- KS .. kritická sekce
- `sem_post(&s);` .. operace V(s);



# INICIALIZACE SEMAFORU

```
sem_init(&s, 0, 1);
```

Semafor s

Počáteční hodnota 1

0 ... semafor sdílený vlákny jednoho procesu  
1 ... semafor sdílený mezi procesy, měl by být v  
regionu sdílené paměti



# CVIČENÍ

1. Stáhněte si z Courseware **pthread-semaphor**
2. Přeložte a ověřte správnou funkcionalitu
3. Zkuste změnit ošetření kritické sekce, tak abyste vyvolali **deadlock** – 2 způsoby:
  - a) modifikací počáteční hodnoty
  - b) pomocí operací P(), V()
4. Jaké výsledky budete dostávat, když P() a V() úplně vynecháte? V čem je problém?
5. Jak byste „prodloužili“ kritickou sekci, aby se chyba dříve projevila?





# POJMENOVANÝ SEMAFOR

- místo inicializace **sem\_init** se otevírá **sem\_open**

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

```
int main() {
```

```
    sem_t *sem1;
```

```
    sem1 = sem_open("/mujsem1", O_CREAT, 0777, 10);
```

```
    ...
```

```
    sem_wait(), sem_trywait(), sem_post(), sem_getvalue()
```

```
    ...
```

```
    sem_close(sem1);
```

```
    sem_unlink("/mujsem1");
```

Další informace: [http://linux.die.net/man/7/sem\\_overview](http://linux.die.net/man/7/sem_overview)



# SEMAFOR - JAVA

- `import java.util.concurrent.Semaphore;`
- `Semaphore sem = new Semaphore(1);`
- `sem.acquire();`
- *.. kritická sekce ..*
- `sem.release();`



# VLÁKNA: OŠETŘENÍ KRITICKÉ SEKCE – PŘEHLED SYNCHRONIZAČNÍCH PRIMITIV

## **Atomické operace (nutná podpora hardware)**

- TSL (test and set lock) + spinlock, CAS (compare and swap)

## **Zámky (lock)**

- POSIX (c, c++) :pthread\_mutex;  
JAVA: java.util.concurrent.locks.Lock
- Implementace: flag (zamčeno, odemčeno), fronta čekajících vláken
- Funkce:
  - Vstup: pthread\_mutex\_lock(lock), lock.lock
  - Opuštění: pthread\_mutex\_unlock(lock), lock.unlock



# VLÁKNA: OŠETŘENÍ KRITICKÉ SEKCE – POKRAČOVÁNÍ

## Semafor

- POSIX (c,c++): `sem_t`;  
JAVA: `java.util.concurrent.Semaphore`
- Implementace: čítač, fronta vláken
- Standardní operace:
  - Vstup do semaforu:  
**P(sem), sem\_wait(sem), sem.acquire()**
  - Opuštění semaforu:  
**V(sem), sem\_post(sem), sem.release();**



# VLÁKNA: OŠETŘENÍ KRITICKÉ SEKCE – POKRAČOVÁNÍ

## Monitor

- POSIX: mutex, pthread\_cond\_t (podmínková proměnná)  
JAVA: synchronized metoda; zámek + podmínka
- Implementace:  
zámek, podmínková proměnná, fronta vláken
- Standardní operace:
  - Vstup do kritické sekce, případné uspání nad podmínkovou proměnnou (wait), případné vzbuzení nad podmínkovou proměnnou(notify, signal), opuštění kritické sekce



# CVIČENÍ - ŘEŠENÍ

- Prodloužení kritické sekce:
  - deklarujeme: `int pom;`
  - `pom = x;`
  - `usleep(200);`
  - `pom = pom + 1;`
  - `x=pom;`
- Odstraňte semaforey, jaký výsledek bude?
- Vraťte semaforey a operaci `V()` nahradte operací `P()`.. Ověřte, že dojde k zablokování procesu.



# ÚLOHA K ÚVAZE

Dvě vlákna pracují nad společnou proměnnou  $x$

Počáteční hodnota proměnné  $x$  je 0

Obě 100x provedou  $x++$  bez ošetření KS.

Správný výsledek je 200.

Jaký je nejhorší možný výsledek?



proces 1:	R1	x	R2		proces 2:
LD R, x	0	0	-		
-----					
	0	0	0	99x:LD R, x	
	0	0	1	INC R	
	0	1	1	LD x, R	
-----					
INC R	1	99	-		
LD x, R	1	1	-		
-----					
	-	1	1	LD R, x	
-----					
99x:LD R, x	1	1	1		
INC R	2	1	1		
LD x, R	2	2	1		
-----					
	100	2		INC R	
	2	2		LD x, R	





# SEMAFORY, BINÁRNÍ SEMAFORY, MUTEXY

## ○ **Obecný semafor**

- Nabývá hodnot 0, 1, 2, 3, ...
- Pro vzájemné vyloučení i synchronizaci

## ○ **Binární semafor**

- Nabývá hodnot 0, 1
- Pro vzájemné vyloučení

## ○ **Mutex**

- Speciální případ binárního semaforu
- Obvyklý výklad:  
Vlákno, které mutex zamklo jej musí i odemknout



# OBEČNÝ POPIS

- Definice (sem: datové struktury, operace)
- Použití (sem: ošetření KS, synchronizace, ...)
- Implementace

*U každého synchronizačního primitiva vždy uvažujte, jak daný mechanismus definovat, uveďte příklad jeho použití a návrh, jak by šel tento mechanismus implementovat s využitím jiných primitiv.*



# SEMAFOR

- Hodnota semaforu  $s$ 
  - Celočíslná proměnná
- Fronta procesů (vláken) blokováných nad semaforem
  - Zpočátku samozřejmě prázdná
- Operace nad semaforem
  - $P()$  – blokující operace
  - $V()$
  - Inicializace semaforu
- Před použitím musíme semafor inicializovat na vhodnou počáteční hodnotu – velmi důležité
  - Ošetření KS: 1
  - Synchronizace: různá, např. 0, 10, ...



# OŠETŘENÍ KRITICKÉ SEKCE

Sdílené proměnné:

*int x, y;*

Představují **různé kritické sekce**, tedy 2  
semafony:

*semaphore sx = 1;* -- správně zvolit poč. hodnotu

*semaphore sy = 1;*

Ošetření kritické sekce:

*P(sx);* // vstup do kritické sekce

***x = x - 5;*** // **kritická sekce** – i více příkazů

*V(sx);* // výstup z kritické sekce



# SYNCHRONIZACE

Proces P1:

Print("Ahoj ")

Print("je")

Proces P2:

Print("dnes ")

Print("krásně.")

Proces P3:

Print("venku ")

P1,P2, P3 běží paralelně.  
Ošetřete SEMAFORY,  
aby vždy byla vypsána  
správná věta:

Ahoj dnes je venku krásně.

