# KIV/ZOS

Cvičení 7, 2024

L. Pešička

# UKÁZKA FS S I-UZLY

- Courseware ZOS – Samostatná práce – Ukázka 2023-24

- Jednoduchá ukázka vytvoření filesystému
- Nejde o „torzo“ semestrálky
- Ukázka pracuje s i-uzly
- Vše je kódováno „natvrdo“ jen pro účel ukázky
- Poděkování A. Vrbovi za tvorbu programu

- Lze přeložit na eryx.zcu.cz pomocí
  gcc –o example main.c

# UKÁZKA – II.

- Vytvoříme filesystém a uložíme do něj soubor soubor.txt
- **./example myfs.dat format soubor.txt**
- Adresář bude obsahovat dvě položky
  - Odkaz na nadřazený adresář
  - Položku pro soubor.txt

- V příkladu je ukázáno nespojité uložení soubor.txt
- Přečteme si informace uložené v našem filesystému
- **./example myfs.dat read**

# UKÁZKA – VÝPIS – III.

eryx3> ./example myfs.dat read

- signature: == ext ==
- volume descriptor: popis bla bla bla
- disk size: **800** (480 + 32 *10 = 800)
- cluster size: **32** (do jednoho bloku se vejde 32 bytů)
- cluster count: **10**
- bitmapi_start_address: 60
- bitmap_start_address: 70
- inode_start_address: 80
- data_start_address: 480

# UKÁZKA - BITMAPA I-UZLŮ

- Inode bitmapa:

11000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

Inode mapa nám říká, že první dva i-uzly jsou obsazené

- První obsahuje info o hlavním adresáři
- Druhý obsahuje info o souboru soubor.txt

# UKÁZKA - BITMAPA DATOVÝCH BLOKŮ

- Data bitmapa:

11010100000000000000000000000000000000000000000000000000000000000000000000000000000000

Cluster 0 slouží k uložení **hlavního adresáře**
Cluster 1 slouží k uložení první části **souboru**
Cluster 2 je volný
Cluster 3 slouží k uložení druhé části **souboru**
Cluster 4 je volný
Cluster 5 slouží k uložení třetí části **souboru**

(pozn. první blok je dále číslován jako nultý)

# NAČTENÉ I-NODY

- Nactene inody:
- id: 0, isDirectory: **1**, references: 1, file_size: 32, direct1: 0, direct2: 0, direct3: 0
- id: 1, isDirectory: 0, references: 1, file_size: 74, direct1: 1, direct2: 3, direct3: 5

- I-uzel s číslem 0 obsahuje **hlavní adresář**, zabírá 1 cluster s číslem 0
- I-uzel s číslem 1 představuje obyčejný soubor velikosti 74 bytů, přímé odkazy na clustery jsou 1,3,5

# OBSAH HLAVNÍHO ADRESÁŘE

- Obsah root slozky:
- name: '.', inode_id: 0
- name: 'soubor.txt', inode_id: 1

- Poznámka – většinou potřebujeme uložit také .. – ukazuje na nadřazený adresář (v kořeni sám na sebe)

# OBSAH SOUBORU SOUBOR.TXT

- name: 'soubor.txt', inode_id: 1

- data z clusteru 1: 1111111111222222222222222222333
- data z clusteru 3: 33333333333333333444444444444444
- data z clusteru 5: 444444444

- Cluster má zde velikost 32 znaků
- Soubor má 74 bytů – 2 celé clustery a kus dalšího

# SPARSE SOUBOR NTFS (DOPLNĚNÍ K FS)

- VCN – virtuální číslo clusteru 0,1,2,3,…
  - Číslo clusteru od začátku daného souboru
- LCN – logické číslo clusteru 101, 102, 103, ..
  - Adresa diskového bloku, který obsahuje příslušná data
- Př.: soubor se skládá z 102 bloků (0..101), prvních pět bloků obsahuje data, pak je spoustu nul a až zase od bloku 100 jsou dva obsazené bloky

| VCN | LCN | Počet bloků |
|---|---|---|
| 0 | 105 | 3 |
| 3 | 120 | 2 |
| 100 | 180 | 2 |

# SPARSE SOUBOR

- "řídký" soubor
- diskové bloky obsahující data:
  - 105, 106, 107, 120, 121, .....180, 181
- soubor bude tvořen bloky:
  - 0, 1, 2, ... 100, 101
- uloženy budou jen informace o:
  - 0, 1, 2, 3, 4 , 100, 101

# NAKRESLETE GRAF, ZAPIŠTE POMOCÍ COBEGIN/COEND

Příklady:

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

Platí běžné precedence operátorů

Každý operátor představuje určitý proces

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

```
cobegin
begin
 cobegin
   x1 = a+b  ||  x2 = c-d
 coend
   x3 = x1 * x2
end
||
begin
 cobegin
   x4 = e/f  ||   x5 = g-h
 coend
   x6 = x4 * x5
end
coend
x7 = x3 - x6
```

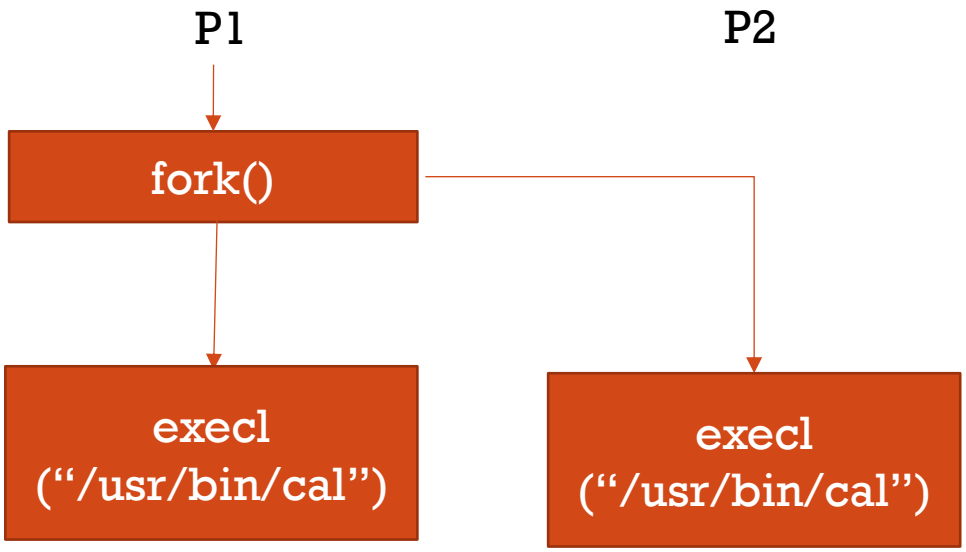
# CO BUDE VÝSTUPEM (F81.C)?

```
#include <stdio.h>
#include <unistd.h>

int i;

for (i=0; i<10; i++) {
  fork();
  execl("/usr/bin/cal","cal",NULL)
}
printf(''Fakt?'');
```

# GRAF PROCESU




Výsledkem bude: 2x vypíše kalendář

# PROCES UNIXU

- Proces ID, proces group ID, user ID, group ID
- Prostředí ( proměnné prostředí, viz. příkaz set)
- Pracovní adresář ( open("ahoj.txt", …) – kde ho bude hledat )
- Instrukce programu
- Registry
- Zásobník (stack)
- Halda (heap)
- Popisovače souborů (file descriptors)
- Signal actions ( defaultní akce, ignore, vlastní obsluha)
- Shared libraries ( např. libc používají programy)
- IPC (fronty zpráv, roury, semafory, 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

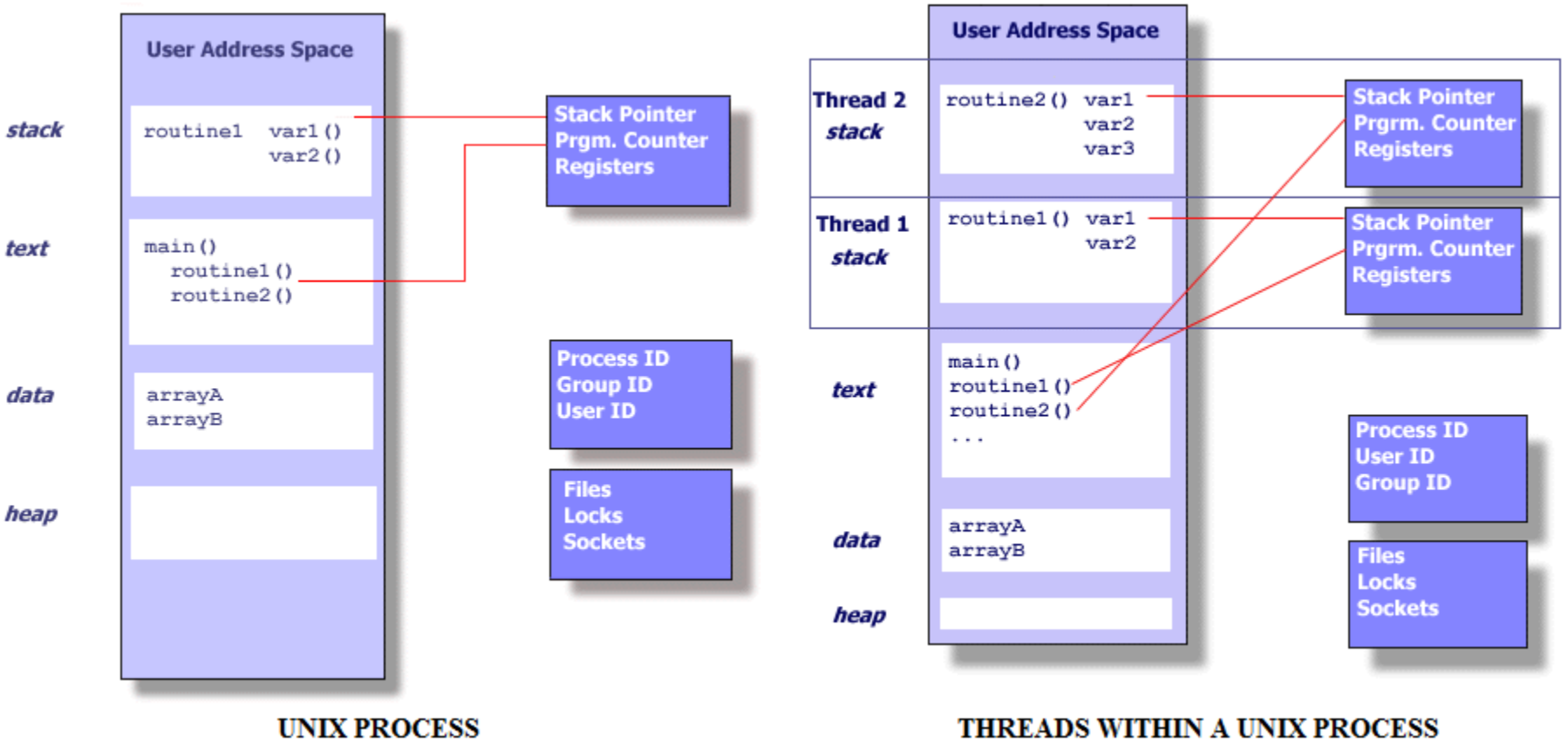
# VLÁKNO MÁ VLASTNÍ:

- **Zásobník (stack pointer)**
- **Registry**
- Plánovací vlastnosti (policy, priority)
- Množina blokovaných signálů
- Data specifická pro vlákno

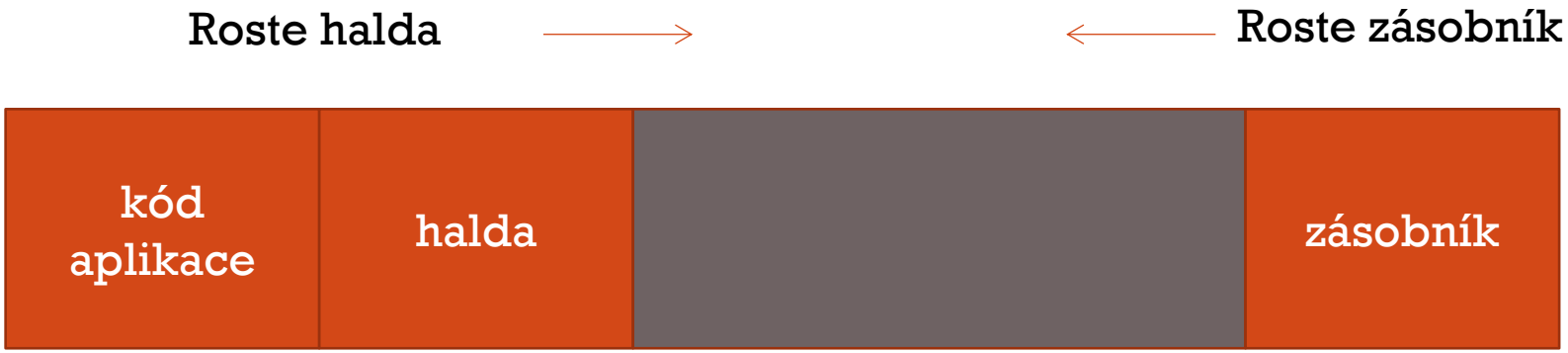
====

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

# Vlákno – každé vlastní zásobník, vlastní registry



# ROZDĚLENÍ PAMĚTI PRO PROCES




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

# ZÁSOBNÍK PRO VLÁKNO

- Při vytvoření vlákna můžeme specifikovat velikost zásobníku

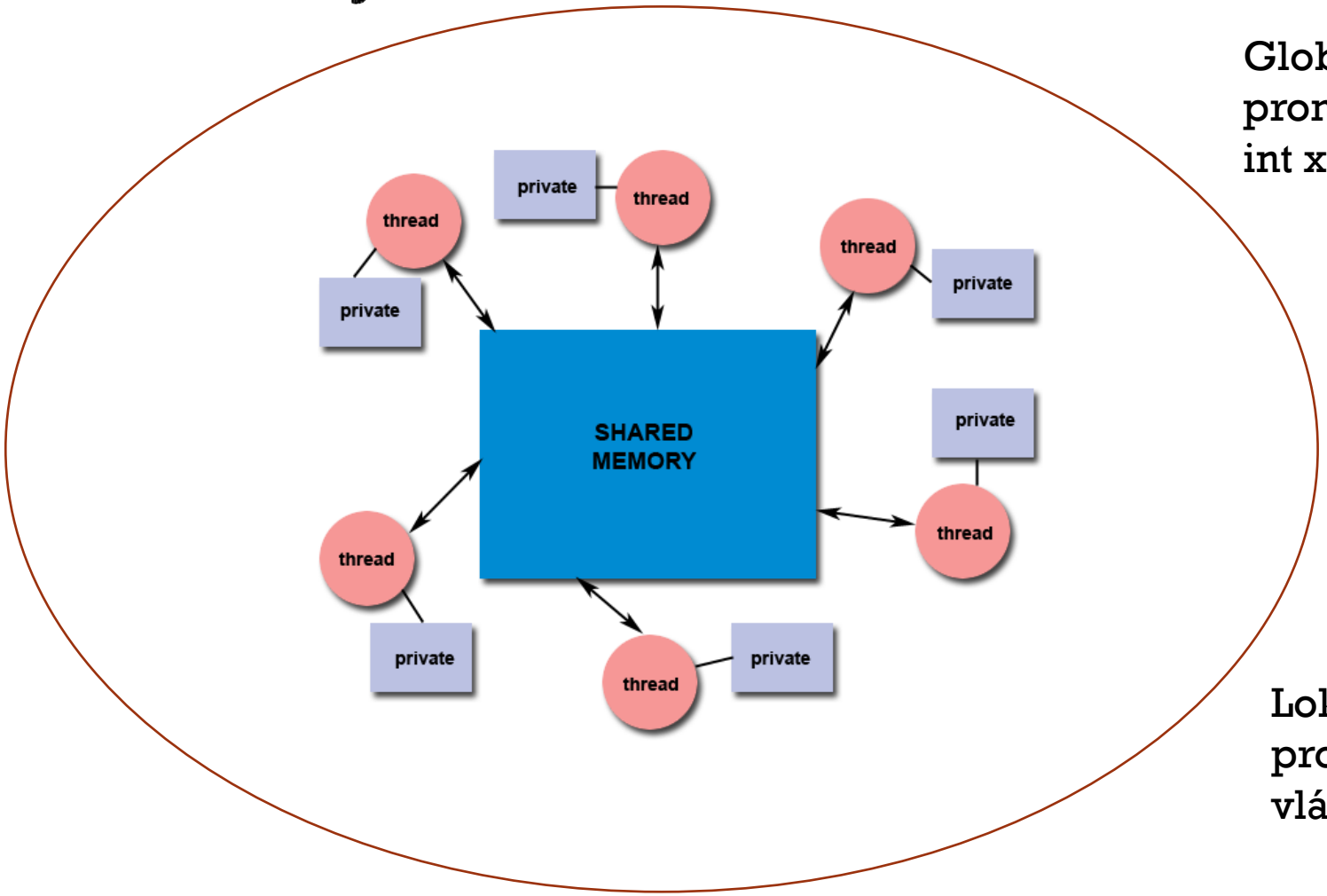
- Je potřeba celkem šetřit..
  Při max. velikost 8MB * 512 vláken = 4 GB

# 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)

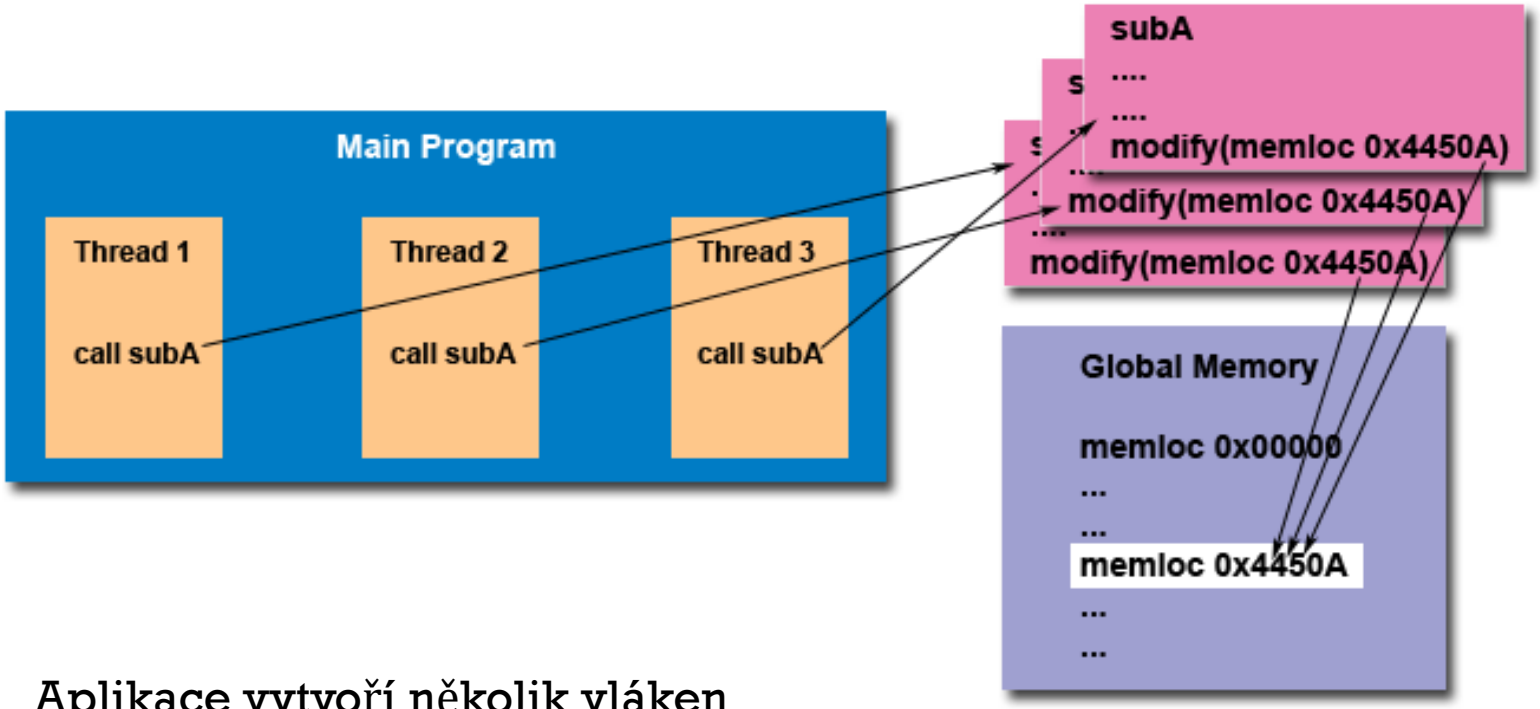
# GLOBÁLNÍ A PRIVÁTNÍ PAMĚŤ VLÁKNA UVNITŘ JEDNOHO PROCESU




Globální proměnné
int x;

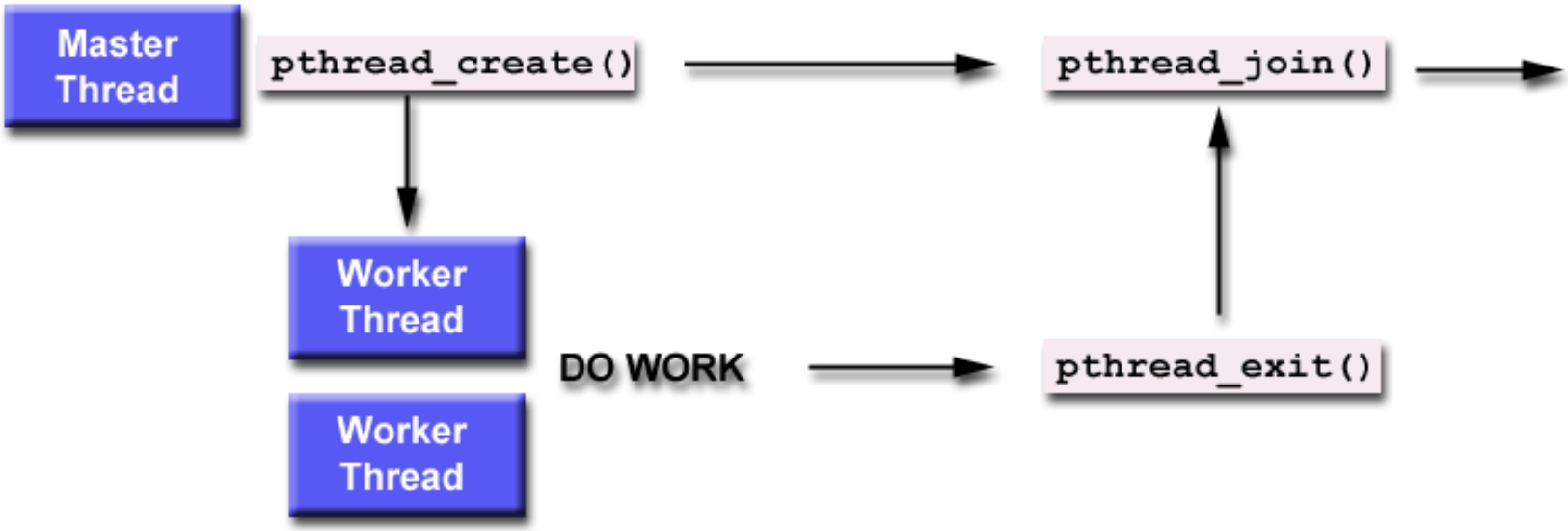
Lokální proměnné vláken

# 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




- Master vytvoří vlákna
- Vlákna (workers) vykonají práci
- Master čeká na dokončení těchto 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

- wget http://home.zcu.cz/~pesicka/zos/sem1.c

- Další příklady
  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
   wget http://home.zcu.cz/~pesicka/zos/sem1.c

2. Přeložte gcc –l phtread –o sem1 sem1.c
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. K čemu slouží usleep v kritické sekci?

# POJMENOVANÝ SEMAFOR (SEM2.C)

- 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

# 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();**

Funkce v pořadí: (Teorie OS, C, Java)

# VLÁKNA: OŠETŘENÍ KRITICKÉ SEKCE – POKRAČOVÁNÍ

**Monitor**

- POSIX: mutex, pthread_cond_t (podmínková proměnná)
  **mutex + podmínková proměnná = monitor**
  JAVA: synchronized metoda; zámek + podmínka
  JAVA: java.util.concurrent – podobně jako v C
- 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

# 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**
  - Slouží pro ošetření KS, jednodušší než semafor
  - Obvyklý výklad:
    Vlákno, které mutex zamklo jej musí i odemknout

# OBECNÝ POPIS

- Definice (sem: datové struktury, operace)
- Použití (sem: ošetření KS, synchronizace – štafetový kolík, producent / konzument)
- 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**
  - Celočíselná proměnná – 0, 1, 2, 3, …
- Fronta procesů (vláken) blokovaných 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, …

# 1. POUŽITÍ SEMAFORU OŠETŘENÍ KRITICKÉ SEKCE

Sdílené proměnné:

*int x, y;*

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

*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
```

# 2. POUŽITÍ SEMAFORU - 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.

# ŘEŠENÍ

Proces P1:

```
Print(“Ahoj ”)
V(s1);
P(s3);
Print(“je”)
V(s2);
```

Proces P2:

```
P(s1);
Print(“dnes  ”)
V(s3);
P(s4);
Print(“krásně.”)
```

Proces P3:

```
P(s2);
Print(“venku  ”)
V(s4);
```

Semaphore

s1 = 0
s2 = 0
s3 = 0
s4 = 0

# 3. POUŽITÍ SEMAFORU PRODUCENT – KONZUMENT

```
semaphore … = … ;
semaphore … = … ;
semaphore … = … ;
Buffer velikosti N;

producent() {  while(1) { … }  }
konzument() { while(1) { … } }

cobegin
  producent() || konzument()
coend
```

Dopište kód producenta a konzumenta:

- Vloz_poloz_do bufferu()
- Vyber_poloz_z bufferu()
- P()
- V()
- Produkuj_polozku()
- Konzumuj_polozku()

```
semaphore
   e = N;
   f = 0;
   m = 1; // přístup k bufferu chápeme jako KS

producent() {
  while(1) {
     produkuj_polozku();
     P(e);
        P(m); vlozdobuf(); V(m);
     V(f);

}
```

```
konzument() {
  while(1) {
     P(f);
     P(m); vyberzbuf(); V(m);
     V(e);
     konzumuj_polozku();
}
}
```