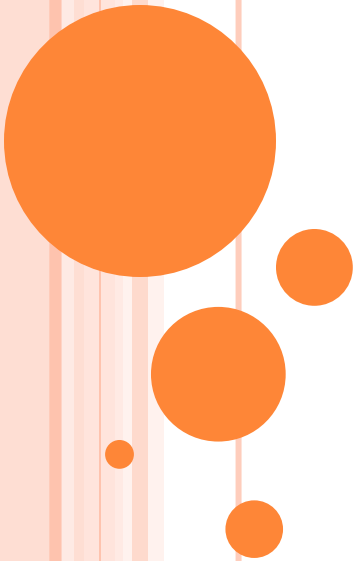

KIV/ZOS cvičení 10

L. Pešička, verze 2012

VYTVOŘTE **MUTEX** POMOCÍ **MONITORU**

```
#define ZAMCENO         1
#define ODEMCENO        0

monitor mutex {

void mylock()      { … dopište kód zde … }
void myunlock()    { … dopište kód zde … }

init() { … }
}
```

UKÁZKA: SEMAFOR POMOCÍ MONITORU

```
monitor monsem {
    int sem;
    condition c1;

    void P() {
       if sem > 0
         sem=sem-1;
       else
         wait(c1);
    }
```

```
    void V() {
        if not empty(c1)
           signal(c1);
        else
           sem=sem+1;
    }

    void monIni(int param)
      { sem = param; }

    init {sem = 1;}
    }
```

Nevýhoda uvedeného řešení

potřebujeme funkci **empty()**,
která zjistí, zda je fronta nad podmínkou prázdná

jak bychom se bez funkce empty obešli?

Semafor pomocí monitoru 2:

```
monitor m2 {

    int sem;
    condition c1;

    void P() {
      if (sem == 0)
         wait(c1);
      sem = sem-1;
    }

    void V() {
       signal(c1);
      sem = sem + 1;
    }
  …
  }
```

operace P() i V() zde vždy manipulují s proměnnou sem

Producent / konzument na 1 položku monitorem

```
monitor ProdKonz {
   condition c1, c2;
   int zaplneny = 0;          // 0 prázdný, 1 plný
   int  buffer_na_jedno;      // buffer na 1 položku

   void insert(int parx);     // vloží položku
   int remove();              // vyjme položku
  ...
}
```

napište funkce insert(), remove()

Možné řešení

```
void insert (int parx) {
    if ( zaplneny == 1)
            wait(c1);
    buffer = parx; zaplneny = 1; signal (c2);
}

int remove() {
    int pom;
    if (zaplneny == 0)
            wait(c2);
     pom = buffer; zaplneny = 0; signal(c1);
     return pom;
}
```

Ukázka

Producent Konzument semaforem v C

int sem_getvalue (&f, &sem)

- návratová hodnota: 0 při úspěchu
- parametrem je semafor f
- v sem je obsazena aktuální hodnota semaforu

Monitor v PTHREAD - C

Podmínková proměnná:

- pthread_cond_t

Vytváření, rušení:

- pthread_cond_init
- pthread_cond_t c1= PTHREAD_COND_INITIALIZER;
- pthread_cond_destroy

Čekání na podmínku:

- pthread_cond_wait
- pthread_cond_timedwait

Vzbuzení vlákna:

- pthread_cond_signal
- pthread_cond_broadcast

Použití monitoru

- 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);

Bariéra

- bariéra nastavená na N vláken
- vlákna volají barrier()
- N-1 vláken se zde zablokuje
- když přijde N-té vlákno, všichni najednou projdou bariérou

- použití: např. iterační výpočty

Ukázka tvorby bariéry - C

```
pthread_mutex_t bariera_zamek = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond_bariera  = PTHREAD_COND_INITIALIZER;
int bariera_cnt = 0;

void barrier()  {
    pthread_mutex_lock(&bariera_zamek);
    bariera_cnt++;
    if (bariera_cnt < N)
            pthread_cond_wait(&cond_bariera, &bariera_zamek)
    else {
            bariera_cnt = 0;
            pthread_cond_broadcast(&cond_bariera);
    }
    pthread_mutex_unlock(&bariera_zamek);
}
```

Bariéra - java

```
private synchronized void barrier() {
   cnt++;
   if ( cnt < N)
        try {  wait(); } catch (InterruptedException ex) {}
  else {
         cnt = 0;
         notifyAll();
   }
}
```

Monitor Java – podmínková proměnná

- balík java.util.concurrent.locks

```
Lock zamek = new ReentrantLock();
Condition podminka1 = zamek. newCondition();
Condition podminka2 = zamek. newCondition();
```

- Nad instancí třídy Condition lze pro uspání volat metody `await()`, `awaitUninterruptibly()`, `awaitNanos(long nanosTimeout)`
- Pro probuzení lze volat metody `signal()`, `signalAll()`

Ukázka monitor s podmínkou Java

```
try
{
      lock.lock();
      //kriticka sekce
      if (!podminka)

      podminka1.awaitUninterruptibly();
      else
            podminka1.signal();
}
finally
{
      lock.unlock();
}
```

Monitor - Java

- Ukázka příkladu se závozníkem

CAS - FILOZOFIE

- Compare and Swap

- Jiná filozofie – místo zamykání spoléháme, že nám pod rukama hodnotu proměnné nikdo nezmění
- S předpokládanou hodnotou provedeme výpočet
- Pokud nám risk vyšel a předpokládaná hodnota se nezměnila, můžeme dosadit vypočtenou hodnotu
- Pokud nevyšel, musíme výpočet opakovat pro novou předpokládanou hodnotu
- Není deterministické

CAS

- Poskytována procesorem

- Provede atomicky test hodnoty, pokud je shoda, nastaví na novou hodnotu
- Vždy vrátí původní hodnotu

```
atomic int CAS(int *pointer,
   int expectedValue, int newValue){
       int oldValue = *pointer;
      if (oldValue == expectedValue)
              *pointer = newValue;
       return oldValue;
}
```

CAS

int CAS (x, ocekavana, nova)

- provede atomicky:
- vrátí původní hodnotu x
- pokud x == ocekavana, nastavi x na nova

návratovou hodnout CAS je původní hodnota x

puvodni = CAS(x,ocekavana,nova)

- pokud puvodni == ocekavana
  - povedlo se, nepočítali jsme zbytečně
  - pokud ne, risk nevyšel, musíme počítat znovu

POUŽITÍ CASU (!)

znovu:

```
snimek = x;
nova = nejaky_vypocet(snimek);
puvodni = CAS(x, snimek, nova);
if puvodni == snimek
    printf(“”Uspěli jsme ”);
else
  goto znovu:
```

je zde nedeterminismus, nevíme, kolikrát budeme muset cyklus opakovat

CAS – UKÁZKY V C, JAVĚ

- c

```
ref = __sync_val_compare_and_swap(&criticalNum,
        myCritNum, myIncreaseCritNum);
```

- Java:

```
public synchronized int compareAndSwap(myInteger var, int
                                        expectedValue, int
  newValue) {
        int oldValue = var.getMyIntValue();
        if (oldValue == expectedValue)
        var.setMyIntValue(newValue);
        return oldValue;
}
```