# OS – Konkurence procesů a IPC

Tomáš Hudec

Tomas.Hudec@upce.cz

http://fei-as.upceucebny.cz/usr/hudec/vyuka/os/

# Konkurence procesů (vláken) a problémy současného běhu

- prostředky poskytované systémem jsou sdílené
  - paměť, CPU, soubory, …
  - procesy o prostředky soupeří – konkurence
  - vznikají problémy souběhu (race conditions)
    - provádění procesu může ovlivnit ostatní procesy
    - některé prostředky nelze (v jednom okamžiku) sdílet
    - přidělení prostředku procesu může omezit ostatní procesy
- při sdílení je třeba komunikace procesů (vláken)
  - synchronizace

# Problémy souběhu

- vzájemné vylučování – mutual exclusion
  - v každém okamžiku smí mít přístup ke sdílenému prostředku pouze jeden proces
- synchronizace
  - proces čeká na dokončení operace jiného procesu
- stav uváznutí – deadlock, livelock
  - procesy vzájemně čekají na uvolnění prostředků
- vyhladovění – starvation
  - nekončící čekání procesu na získání prostředku

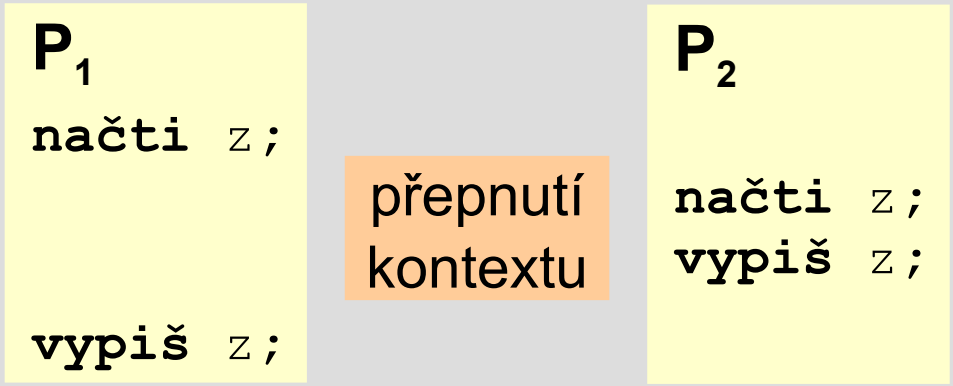
# Přístup ke sdíleným prostředkům

- bez řízení přístupu ke sdíleným prostředkům:
  - může dojít k porušení konzistence dat
  - výsledek akcí procesů může záviset na pořadí, v jakém dostanou procesy přidělen procesor
  - tomu je třeba zabránit
- přístup ke sdíleným prostředkům je třeba řídit
  - zajištění integrity – zavedení kritické sekce
    - v jednom okamžiku smí sdílená data měnit jediný proces
  - OS: nástroje IPC (Inter-Process Communication)

# Příklad se sdílenou proměnnou

- procesy $P_1$ a $P_2$ sdílejí znakovou proměnnou *z*
- provádějí stejnou funkci **`echo`**

```
echo() {
  načti z;
  vypiš z;
}
```




- procesy mohou být přerušeny kdykoliv
  - přeruší-li OS proces $P_1$ po provedení **`načti z`**,
  - pak proces $P_1$ vypíše znak načtený procesem $P_2$!

# Příklad s tiskárnou

- procesy $P_1$ a $P_2$ potřebují tisknout
- provádějí operaci **`tiskni`**

```
tiskni(tiskárna t) {
   tisk(t, řádek1);
   tisk(t, řádek2);
}
```

**$P_1$**

```
tisk ř1;


tisk ř2;
```

přepnutí kontextu

**$P_2$**

```
tisk ř1;
tisk ř2;
```

- procesy mohou být přerušeny kdykoliv
  - přepne-li OS z procesu $P_1$ mezi tiskem řádků na $P_2$,
  - pak tiskárna vytiskne mix řádků obou procesů

# Kritická sekce (KS)

- část kódu, kde proces manipuluje se sdíleným prostředkem a současná manipulace jiným procesem by vedla k problému (nekonzistenci)
- provádění tohoto kódu musí být vzájemně výlučné
  - v každém okamžiku smí být v kritické sekci (pro daný prostředek) pouze jediný proces
  - proces musí žádat o povolení vstupu do kritické sekce

# Struktura programu s KS

- vstupní sekce (entry section)
  - implementace povolení vstupu do KS
- kritická sekce (critical section)
  - manipulace se sdílenými prostředky
- výstupní sekce (exit section)
  - implementace uvolnění KS pro jiné procesy
- zbytková sekce (remainder section)
  - zbytek kódu procesu

# Předpoklady řešení vstupu do KS

- procesy se provádějí nenulovou rychlostí
- žádné předpoklady o relativní rychlosti procesů
- uvažujeme i víceprocesorové systémy
  - předpoklad: paměťové místo smí v jednom okamžiku zpřístupnit vždy pouze jediný procesor
- žádné předpoklady o prokládaném provádění
  - procesy se nemusí pravidelně střídat v běhu
- stačí specifikovat vstupní a výstupní sekci

# Požadované vlastnosti řešení KS

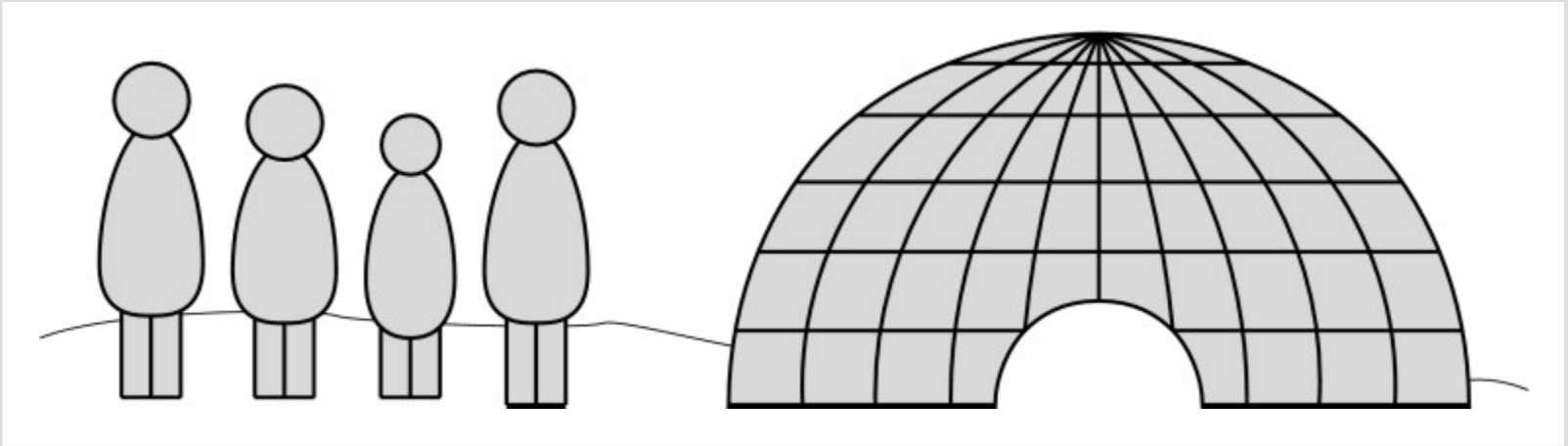
- vzájemné vylučování (mutual exclusion)
  - vždy jediný proces v KS
- pokrok v přidělování (progress)
  - na rozhodování o vstupu do volné KS se mohou podílet výhradně procesy, které nejsou ve ZS
  - toto rozhodnutí musí padnout v konečném čase
  - volná KS ⇒ požadavku musí být vyhověno
- omezené čekání (bounded waiting)
  - mezi požadavkem na vstup do KS a vyhověním smí do KS vstoupit pouze omezený počet jiných procesů

# Typy řešení KS

- SW řešení
  - použití algoritmu pro vstupní a výstupní sekci
- HW řešení
  - využití speciálních instrukcí procesoru
- řešení OS
  - nabízí programátorovi prostředky pro řešení KS (datové typy a funkce)
- řešení programovacího jazyka
  - konkurenční / souběžné programování

# SW řešení KS: aktivní čekání

Příklad:



- svého šamana (kritickou sekci) může v daném čase navštívit pouze jediný eskymák (proces)
- iglú má malý vchod, takže dovnitř může vstoupit vždy jen jeden jeden eskymák, aby si přečetl jméno napsané na tabuli
- je-li na tabuli napsané jeho jméno, může k šamanovi
- je-li na tabuli napsané jiné jméno, iglú opustí a čeká
- čas od času eskymák opět vstoupí do iglú podívat se na tabuli

# SW řešení KS

- budou ukázány potenciální algoritmy snažící se o řízení vstupu do KS
- nejprve budou předpokládány pouze dva procesy
  - označení procesů: $P_0$ a $P_1$
- poté se správné řešení zobecní pro *n* procesů
  - označení: $P_i$ aktuální proces a $P_j$ ostatní procesy

# SW řešení – algoritmus 1

- sdílená proměnná **`locked`**
  - udává obsazenost KS
  - inicializovaná na 0
- proces $P_i$ čeká, dokud je KS obsazena (**`locked ≠ 0`**)
  - jakmile je KS volná, nastaví se **`locked = 1`**
- pokud oba procesy současně zjistí volnou KS, oba nastaví obsazeno a vstoupí do ní
  - **požadavek vzájemného vylučování není splněn!**

**proces $P_i$**

```
repeat
  while
    (locked != 0);
  locked = 1;
  KS;
  locked = 0;
  ZS;
forever
```

# SW řešení – algoritmus 2

- sdílená proměnná **turn**
  - inicializovaná na 0 nebo 1
- kritická sekce (KS) procesu $P_i$ se provádí, jestliže **turn = i**
- proces $P_i$ aktivně čeká (busy waiting), když je proces $P_j$ v kritické sekci
  - **požadavek vzájemného vylučování je splněn**
- **požadavek pokroku není splněn!**
  - vyžadována alternace kritických sekcí

**proces $P_i$**

```
repeat
   while (turn != i);
   KS;
   turn = j;
   ZS;
forever
```

# SW řešení – algoritmus 2 – test

- předpokládejme, že $P_0$ má dlouhou zbytkovou sekci (ZS) a $P_1$ ji má krátkou
- pokud `turn = 0`, vstoupí $P_0$ do KS a pak provádí dlouhou ZS (`turn = 1`)
- zatím $P_1$ vstoupí do KS a provede pak krátkou ZS (`turn = 0`) a hned se znovu pokusí vstoupit do KS – **požadavek je ale odmítnut!**
- **proces $P_1$ musí čekat, dokud $P_0$ nedokončí ZS**

**proces $P_i$**

```
repeat
   while (turn != i);
   KS;
   turn = j;
   ZS;
forever
```

# SW řešení – algoritmus 3

- sdílená proměnná **flag**
  - pro každý proces: **flag[i]**
- požadavek vstupu do KS: **flag[i] = true**
- **požadavek vzájemného vylučování je splněn**
- **požadavek pokroku není splněn!**
  - po sekvenci: $P_0$: **flag[0] = true;**
    $P_1$: **flag[1] = true;** – **DEADLOCK**

**proces $P_i$**

```
repeat
   flag[i] = true;
   while (flag[j]);
   KS;
   flag[i] = false;
   ZS;
forever
```

# SW řešení – Petersonův algoritmus

- inicializace
  - `flag[i] = false`, i = 0..1
  - `turn = 0` nebo `1`
- signalizace připravenosti ke vstupu do KS nastavením `flag[i] = true`

**proces $P_i$**

```
repeat
   flag[i] = true;
   turn = j;
   while (flag[j]
         && turn == j);
   KS;
   flag[i] = false;
   ZS;
forever
```

- pokud se oba procesy pokusí vstoupit do KS současně, pouze jeden bude mít potřebnou hodnotu proměnné `turn`

# SW řešení – Peterson – analýza

- vzájemné vylučování splněno
  - není možné, aby $P_0$ a $P_1$ byly oba současně v KS
    - nemůže nastat **`turn == i`** a **`flag[i] == true`** pro každý $P_i$
- pokrok a omezenost čekání
  - pokud $P_j$ nepožaduje vstup do KS, je **`flag[j] == false`** a $P_i$ tedy může vstoupit do KS
  - $P_i$ nemůže vstoupit do KS dokud $P_j$ požaduje KS (**`flag[j]`**) a současně je $P_j$ na řadě (**`turn == j`**)

**proces $P_i$**

```
repeat
   flag[i] = true;
   turn = j;
   while (flag[j]
        && turn == j);
   KS;
   flag[i] = false;
   ZS;
forever
```

# SW řešení KS pro *n* procesů

- Leslie Lamport's bakery algorithm
  - každý proces dostane před vstupem do KS číslo
  - držitel nejmenšího čísla smí vstoupit do KS
  - dostanou-li $P_i$ a $P_j$ stejná čísla, přednost má $P_{min(i, j)}$
  - ve výstupní sekci nastaví proces přidělené číslo na 0
- poznámky k zápisu:
  - $(a, b) < (c, d)$, když $a < c$ nebo když $a = c$ a $b < d$
  - $max(a_0, \ldots, a_k)$ je takové $b >= a_i$ pro $i = 0, \ldots, k$

# SW řešení KS pro *n* procesů

- sdílená data
  - `bool choosing[n];` – proces vybírá číslo
    - inicializace všech na **`false`**
  - `int number[n];` – přidělené číslo číslo
    - inicializace všech na nulu
- korektnost algoritmu závisí na faktu
  - je-li $P_i$ v KS a $P_k$ si právě vybral své číslo, pak $(number_i, i) < (number_k, k)$

# SW řešení KS – algoritmus bakery

**proces $P_i$**

```
repeat
  choosing[i] = true;
  number[i] = max(number[0]..number[n-1]) + 1;
  choosing[i] = false;
  for (j = 0; j < n; ++j) {
    while (choosing[j]);          // kvůli souběhu výpočtu max
    while (number[j] != 0 &&      // Pj požaduje KS
      (number[j],j) < (number[i],i)); // Pj má přednost
  }
  KS;
  number[i] = 0;
  ZS;
forever
```

# Vliv chyb procesu

- splněním všech tří kritérií (vzájemné vylučování, pokrok, omezené čekání) je řešení odolné vůči chybám ve zbytkové sekci (ZS)
  - chyba ve ZS je totéž, co neomezeně dlouhá ZS
- řešení však **nemůže zajistit odolnost vůči chybám v KS**
  - pokud proces $P_i$ havaruje v KS,
  - pro ostatní procesy je stále v KS
  - a ty se do ní nedostanou

# Aktivní čekání

- označuje se též jako spin lock
- procesy čekající na vstup do KS spotřebovávají zbytečně a neproduktivně čas procesoru
  - zvláště pokud je KS dlouhá
  - efektivnější by bylo čekající procesy blokovat
- vhodné pouze tehdy, když je KS velmi krátká
  - ve srovnání s délkou časového kvanta běhu na CPU

# HW řešení – výchozí předpoklady

- procesy se provádějí v procesoru kontinuálně, dokud nevyvolají službu OS nebo nejsou přerušeny
- k přerušení procesu může dojít pouze na hranicích instrukcí
  - mezi dokončením jedné instrukce a zahájením další
- přístup k paměti je obvykle výlučný
  - důležité zejména pro systémy SMP

# HW řešení – zákaz přerušení (1)

- proces běží, dokud nezavolá službu OS nebo není přerušen
  - plánovač využívá přerušení
  - zákaz přerušení způsobí, že proces nemůže být přerušen, tudíž žádný jiný proces nemůže vstoupit do KS

**proces $P_i$**

```
repeat
  disable irqs;
  KS;
  enable irqs;
  ZS;
forever
```

- vzájemné vylučování je zajištěno
  pouze na jednoprocesorových systémech
  - **na systémech SMP není zaručeno!**

# HW řešení – zákaz přerušení (2)

- zvyšuje latenci systému
  - během KS nemůže být obsloužena žádná událost – vadí zejména v multimediálních a RT systémech

**proces $P_i$**

```
repeat
  disable irqs;
  KS;
  enable irqs;
  ZS;
forever
```

- v KS se nesmí volat služba OS
  - jádro OS by mohlo aktivovat plánovač, jenž by mohl přepnout na jiný proces vstupující do KS
- zákaz přerušení je privilegovaná instrukce
- závěr: **obecně nevhodné řešení**

# HW řešení – speciální instrukce

- přístup k paměťovému místu je obvykle výlučný
- lze tedy navrhnout instrukci, která atomicky provede dvě akce s jedním paměťovým místem
  - čtení a zápis jako nedělitelná operace
- provedení instrukce nelze přerušit
  - context switch probíhá pouze na hranicích instrukcí
  - zaručí nám tak vzájemné vylučování, a to i na **víceprocesorových systémech**
  - ostatní požadavky je třeba řešit algoritmicky

# HW řešení – instrukce test-and-set

- jediná instrukce procesoru přečte příznak a současně ho nastaví
  - byl-li příznak již nastaven, nové nastavení nic nemění
    - KS je obsazena
  - nebyl-li příznak nastaven, proces smí vstoupit do KS
    - instrukce je nepřerušitelná, proto jiný proces vyhodnotí touto instrukcí příznak správně
- lze využít pro více různých KS

**instrukce test-and-set**

```
int testAndSet(int *lck)
{
  if (*lck == 0) {
    *lck = 1;
    return 0; // KS
  } else {
    return 1; // čekání
  }
}
```

# HW řešení – nevýhody test-and-set

- při čtení příznaku se používá aktivní čekání (AČ)
- může dojít k vyhladovění
  - soupeří-li několik procesů o vstup do KS po jejím uvolnění, dostane se do ní první, který provede instrukci, ostatní mohou vyhladovět
- může vzniknout deadlock
  - proces s nízkou prioritou je přerušen v KS
  - proces s vyšší prioritou požaduje vstup do KS (AČ)
  - nikdy se jí nedočká, protože proces s nižší prioritou nedostane šanci ji opustit

# HW řešení KS – test-and-set

- sdílená prom. **`locked`**
  - inicializovaná na 0
  - vstupní sekce využívá instrukci test-and-set
  - ve výstupní sekci stačí přiřazovací příkaz

**inicializace**

```
int locked = 0;
```

**proces $P_i$**

```
repeat
  while
    (testAndSet(&locked));
  KS;
  locked = 0;
  ZS;
forever
```

- první proces nastaviv **`locked`**, vstoupí do KS
  - ostatní aktivně čekají prováděním test-and-set
  - provádění je vzájemně výlučné i na SMP

# HW řešení KS – instrukce xchg

- Intel x86 – instrukce xchg
  - vymění obsah dvou proměnných
- sdílená proměnná **locked**
  - inicializace na 0
- lokální prom. **s** pro každý $P_i$
  - ve vstupní sekci nastavena na 1
  - v aktivním čekání **s** vyměňuje hodnotu s **locked**
  - je-li **locked** = 0, nastaví se na 1 a končí čekání
    - proces vstupuje do KS

**proces $P_i$**

```
int s;
repeat
  s = 1;
  while (s)
    xchg(s,locked);
  KS;
  locked = 0;
  ZS;
forever
```

# Systémy SMP a SW řešení KS

- je třeba uvažovat
  - cache na každém procesoru
    - typ cache z hlediska zápisu změn
    - write through
    - write back (behind)
  - umístění a vyrovnávání (caching) sdílené paměti
  - způsob zápisu do paměti jednotlivými procesory
    - řazení zápisu pro procesor
    - řazení zápisu různými procesory a viditelnost změn

# Systémy SMP a cache – příklad

- uvažujte následující sekvenci na systému SMP
  - inicializace je: A = 0, B = 0
  - procesor 1 zapíše hodnotu 1 na adresu A
  - procesor 1 zapíše hodnotu 1 na adresu B
  - procesor 2 čeká na změnu hodnoty na adrese B
    - aktivní čekání dokud je B nulové
  - procesor 2 přečte hodnotu z adresy A
- Jakou hodnotu procesor 2 přečte?

# Systémy SMP a cache – příklad

**procesor 1**

```
mov [A],1     # zápis do prom. A
mov [B],1     # zápis do prom. B
```

**procesor 2**

```
loop:
cmp [B],0     # porovnání hodnot
jz loop       # rovnost ⇒ skok
mov R,[A]     # čtení prom. A
```

**datová cache procesoru 1**

```
A: 1
B: 1
```

**datová cache procesoru 2**

```
B: 1   # aktualizovaná hodnota
A: 1   # aktualizovaná hodnota
```

**systémová paměť**

```
A: 1          # změněná proměnná A
B: 1          # změněná proměnná B
```

Co se uloží do registru R?

# Systémy SMP a cache – Peterson

**procesor 1 – proces 0**

```
mov [flag],1     # true do flag[0]
mov [turn],1     # zápis 1 do turn
cmp [flag+1],0   # je-li flag[1] false,
jz KS            # skok do KS
```

**procesor 2 – proces 1**

```
mov [flag+1],1   # true do flag[1]
mov [turn],0     # zápis 0 do turn
cmp [flag],0     # je-li flag[0] false,
jz KS            # skok do KS
```

**datová cache procesoru 1**

```
flag[0]: 1
flag[1]: 0
turn: 1
```

**datová cache procesoru 2**

```
flag[0]: 0
flag[1]: 1
turn: 0
```

**systémová paměť je pomalejší, tudíž zápis se neprojeví ihned**

```
flag[0]: 0
flag[1]: 0
turn: 0
```

Důsledek: **oba procesy jsou v KS!**

# Systémy SMP a zápis do paměti

- řazení zápisu do paměti – write ordering
  - dřívější zápis do paměti procesorem bude vidět **před** pozdějším zápisem
  - **vyžaduje write through cache**
- sekvenční konzistence – sequential consistency
  - pokud procesor 1 zapíše na adresu A
  - **před** zápisem procesoru 2 na adresu B,
  - pak nová hodnota na A musí být viditelná **všemi** procesory **před** změnou na B
  - **vyžaduje nepoužívat cache pro sdílená data!**

# Systémy SMP – důsledky

- SW algoritmy řešící vstup do KS vyžadují
  - write ordering (změny CPU-cache → RAM)
  - sequential consistency (změny RAM → CPU-cache)
  - víceprocesorové systémy toto **nemusejí zaručovat**
    - superskalární CPU s pipeline – dopad na rychlost
- **čistě SW řešení bychom se měli vyvarovat**
- HW instrukce test-and-set na SMP způsobí
  - zápis změn z cache do hlavní paměti
  - aktualizaci cache na všech procesorech

# Řešení KS pomocí OS – semafor

- synchronizační nástroj
  - prostředek OS
- nevyžaduje aktivní čekání
  - dokud je KS obsazená
    - čekající proces je blokován a
    - zařazen do fronty procesů čekajících na uvolnění KS
  - po uvolnění KS je z fronty vybrán další proces

# Řešení OS – definice semaforu

- semafor je obecně datová struktura
  - celočíselný čítač
  - fronta procesů
  - atomické operace init, wait a signal

**semafor**

```
typedef struct {
  int count;
  fifo_t *q;
} sem_t;

void sem_init(sem_t *s, int v);
void sem_wait(sem_t *s);
void sem_post(sem_t *s);
```

- k čítači se přistupuje výhradně pomocí operací
- operace musí být provedeny **atomicky**
  - operaci smí provádět vždy jen jediný proces

# Řešení OS – operace semaforu

- init inicializuje čítač
- wait snižuje čítač
  - je-li záporný
    - zařadí volající vlákno do fronty a blokuje je
- signal zvyšuje čítač
  - je-li fronta neprázdná
    - vybere vlákno z fronty
    - zařadí je do seznamu připravených vláken

**semafor – implementace**

```
void sem_init(sem_t *s, int v){
  s->count = v;
}
void sem_wait(sem_t *s) {
  s->count--;
  if (s->count < 0) {
    fifo_put(s->q, self);
    block calling thread self;
  }
}
void sem_post(sem_t *s) {
  s->count++;
  if ((t = fifo_get(s->q)))
    activate thread t;
}
```

# Řešení OS – semafor – pozorování

- čítač je třeba inicializovat nezápornou hodnotou
- nezáporný čítač udává počet procesů, jež smějí **bez blokování** zavolat **wait**
- absolutní hodnota záporného čítače udává, **kolik procesů čeká ve frontě** na semafor
- atomicita oper. zaručuje **vzájemné vylučování**
  - programové bloky operací se semaforem jsou vlastně také **kritické sekce**
    - OS musí tyto KS v operacích ošetřit

# Řešení OS – semafor – KS operací

- kritické sekce v implementaci operací semaforu jsou velmi krátké
  - typicky asi deset instrukcí
- vhodné ošetření KS
  - zákaz přerušení
    - pouze jednoprocesorové systémy
  - spin lock užitím HW nebo SW metody
    - HW spin lock je na víceprocesorových systémech nutností

# Použití semaforu – řešení KS

- sdílený semafor **s** se inicializuje na počet procesů smějících vstoupit do KS
  - typicky na jeden
- vstupní sekce volá wait
- výstupní sekce volá signal
- obecně lze povolit neblokující volání wait pro *n* procesů
  - inicializujeme-li semafor na *n*

**jeden z procesů $P_i$**

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

**proces $P_i$**

```
repeat
   sem_wait(&s);
   KS;
   sem_post(&s);
   ZS;
forever
```

# Použití semaforu – synchronizace

- příkaz **S2** v procesu $P_1$ musí být proveden až **po provedení S1** v $P_0$
- čítač semaforu **sync** inicializujeme na 0
- proces $P_1$ před provedením **S2** počká na signál od $P_0$
- provede-li se nejprve signal, pak wait neblokuje

**inicializace**

```
sem_init(&sync, 0);
```

**proces $P_0$**

```
vypočti x;   // S1
sem_post(&sync);
```

**proces $P_1$**

```
sem_wait(&sync);
použij x;    // S2
```

# Binární semafor

- místo čítače je použita Booleovská proměnná
  - inicializace na false
- obvykle se označuje pojmem mutex
  - odvozeno od využití pro vzájemné vylučování – **mutual exclusion**
  - nelze jej (samostatně) využít pro synchronizaci
- typicky se operace wait a signal označují termíny lock a unlock

**binární semafor**

```
typedef struct {
  bool locked;
  fifo_t *q;
} mutex_t;
```

# Binární semafor – implementace

- operace lock (wait) a unlock (signal) nepočítají počty procesů
  - lock je neblokující pouze pro jeden proces
  - unlock odemkne mutex, je-li fronta procesů prázdná

**mutex – implementace**

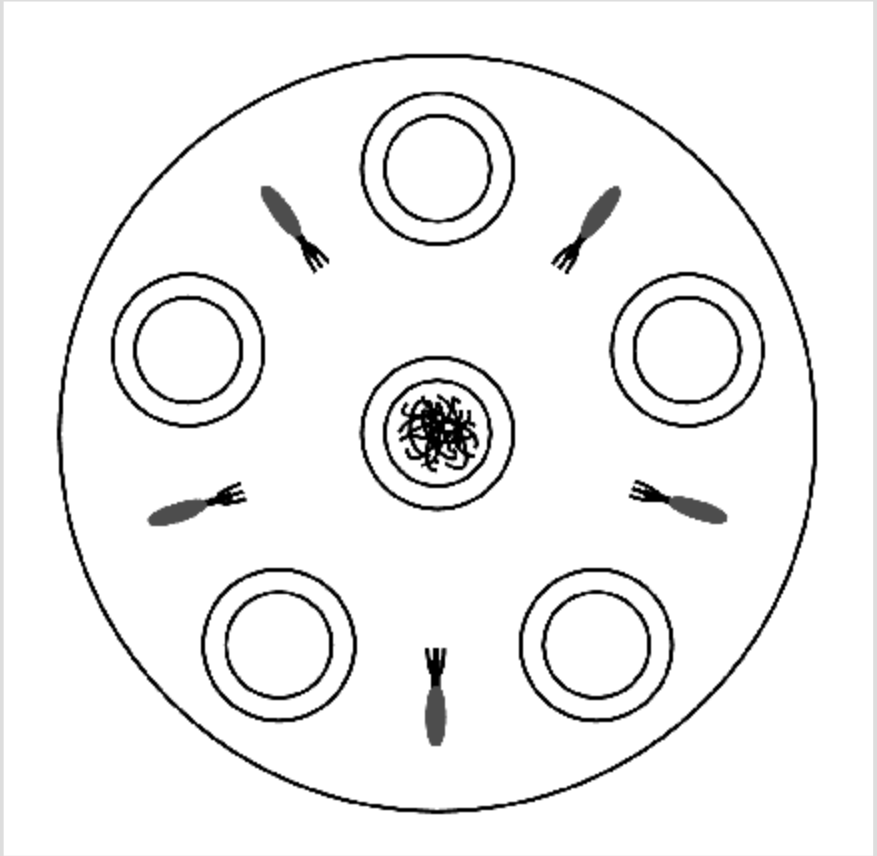
```
void mutex_lock(mutex_t *m) {
  if (m->locked) {
    fifo_put(m->q, self);
    block calling thread self;
  } else {
    m->locked = true;
  }
}
void mutex_unlock(mutex_t *m) {
  if ((t = fifo_get(m->q))) {
    activate thread t;
  } else {
    m->locked = false;
  }
}
```

# Semafor – hodnocení

- semafor je výkonný nástroj pro řešení
  - vzájemného vylučování
  - synchronizace procesů
- protože jsou operace wait a signal volány z různých procesů, může být obtížné plně porozumět jejich působení
  - použití musí být korektní u **všech** procesů
  - jediné chybné použití může způsobit problémy s celou skupinou procesů

# Problém obědvajících filosofů

- klasický synchronizační problém
- u stolu sedí pět filosofů
  - každý buď přemýšlí,
  - nebo jí
- při jídle každý potřebuje dvě vidličky
- k dispozici je pouze pět vidliček



# Problém filosofů – návrh řešení

- každý filosof odpovídá jednomu procesu
- jedení – kritická sekce
- přemýšlení – zbytková sekce
- vidličky – sdílené prostředky
  - pro výlučný přístup k vidličkám je třeba pro každou použít semafor

**inicializace**

```
sem_t fork[n]; // n = 5
for (i = 0; i < n; ++i)
  sem_init(&fork[i], 1);
```

**proces $P_i$**

```
repeat
  think;   // ZS
  sem_wait(&fork[i]);
  sem_wait(&fork[(i+1)%n]);
  eat;     // KS
  sem_post(&fork[(i+1)%n]);
  sem_post(&fork[i]);
forever
```

# Problém filosofů – analýza návrhu

- každý filosof čeká na vidličku po své levé i pravé ruce, dokud se neuvolní
- avšak může nastat následující situace:
  - všichni se ve stejný okamžik rozhodnou najíst
  - každý zvedne vidličku po své levé ruce
  - všichni čekají na uvolnění vidličky po své pravé ruce

    ⇒ DEADLOCK

**proces $P_i$**

```
repeat
   think;   // ZS
   sem_wait(&fork[i]);
   sem_wait(&fork[(i+1)%n]);
   eat;     // KS
   sem_post(&fork[(i+1)%n]);
   sem_post(&fork[i]);
forever
```

# Problém filosofů – řešení

- je třeba dovolit zvedat vidličky nejvýše čtyřem (n − 1) filosofům
- pak vždy aspoň jeden filosof může jíst
  - ostatní musejí čekat (tři s vidličkou v ruce, jeden bez vidličky)

**inicializace**

```
sem_t waiter;
sem_init(&waiter, n - 1);
```

**proces $P_i$**

```
repeat
  think;   // ZS
  sem_wait(&waiter);
  sem_wait(&fork[i]);
  sem_wait(&fork[(i+1)%n]);
  eat;     // KS
  sem_post(&fork[(i+1)%n]);
  sem_post(&fork[i]);
  sem_post(&waiter);
forever
```

- pro omezení zvedání vidliček lze použít semafor

# Problém filosofů – analýza řešení

**inicializace**

```
for (i = 0; i < n; ++i) sem_init(&fork[i], 1); // vidličky
sem_init(&waiter, n - 1);  // n = 5, pouze čtyři smějí zvedat vidličku
```

| proces $P_i$ | $P_0$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ |
|---|---|---|---|---|---|
| `think;  // ZS` | ZS | ZS | ZS | ZS | ZS |
| `sem_wait(&waiter);` | R | R | R | R | $B_w$ |
| `sem_wait(&fork[i]);` | R | R | R | R | R |
| `sem_wait(&fork[(i+1)%n]);` | $B_{f_1}$ | $B_{f_2}$ | $B_{f_3}$ | R | $B_{f_0}$ |
| `eat;  // KS` | KS | KS | KS | KS | KS |
| `sem_post(&fork[(i+1)%n]);` | R | R | R | R | |
| `sem_post(&fork[i]);` | R | R | R | R | |
| `sem_post(&waiter);` | | | | R | |

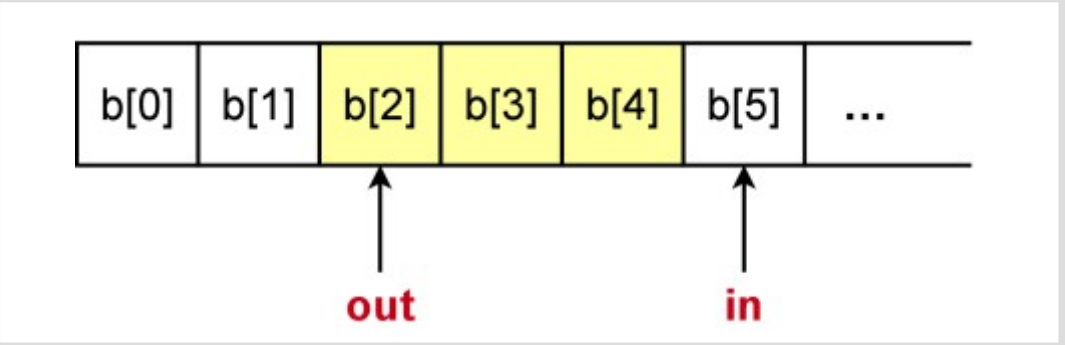
| $f_0$ | $f_1$ | $f_2$ | $f_3$ | $f_4$ | w |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 4 |
| 0 | 0 | 0 | 0 | | −1 |
| | −1 | −1 | −1 | 0 | |
| | | | 0 | 1 | 0 |
| −1 | | 0 | 1 | 0 | |
| | 0 | 1 | | | |
| 0 | 1 | | | | |

# Problém producenta a konzumenta

- dva typy procesů
  - producenti – produkují výrobky (data)
  - konzumenti – spotřebovávají data
- pro lepší efektivitu je třeba zavést mezisklad
  - vyrovnávací paměť na vyprodukovaná data
  - nevzniknou tak zbytečné čekací doby
- pro řízení přístupu do skladu lze použít semafory
  - kapacita skladu – pro počet výrobků na skladě
  - možnost vstupu do skladu

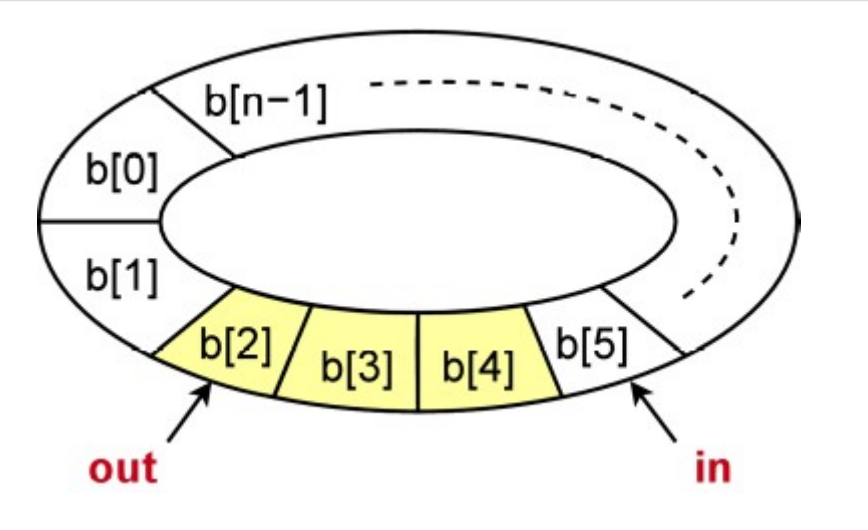
# Nekonečná kapacita skladu

- vyrovnávací paměť (buffer) – sklad
  - pole s nekonečnou kapacitou
- pole musí mít nezávislý přístup pro čtení a zápis
  - dva indexy pole
    - index prvního volného místa pro zápis – in
    - index prvního obsazeného místa pro čtení – out



# Kruhový buffer

- v praxi je kapacita vyrovnávací paměti omezená
  - pole je propojeno do kruhu – kruhový buffer
    - po poslední položce následuje zase první
  - je třeba hlídat kapacitu skladu (bufferu)
    - po zaplnění musejí producenti čekat



# Producent / konzument – funkce

**Producent**

```
repeat
   item = produce_item();
   while (count == n);
   buffer[in] = item;
   in = (in + 1) % n;
   count++;
forever
```

**Konzument**

```
repeat
   while (count == 0);
   item = buffer[out];
   out = (out + 1) % n;
   count--;
   consume_item(item);
forever
```

- producentů i konzumentů může být několik
- je třeba zajistit
  - vzájemné vylučování při práci s bufferem a indexy
  - hlídání obsazenosti bufferu (stavy prázdný a plný)

# Producent / konzument – řešení

## Inicializace

```
sem_init(&item_count, 0);        // počet položek v bufferu
sem_init(&mutex, 1);             // vzájemné vylučování
// neúplné řešení
```

## Producent

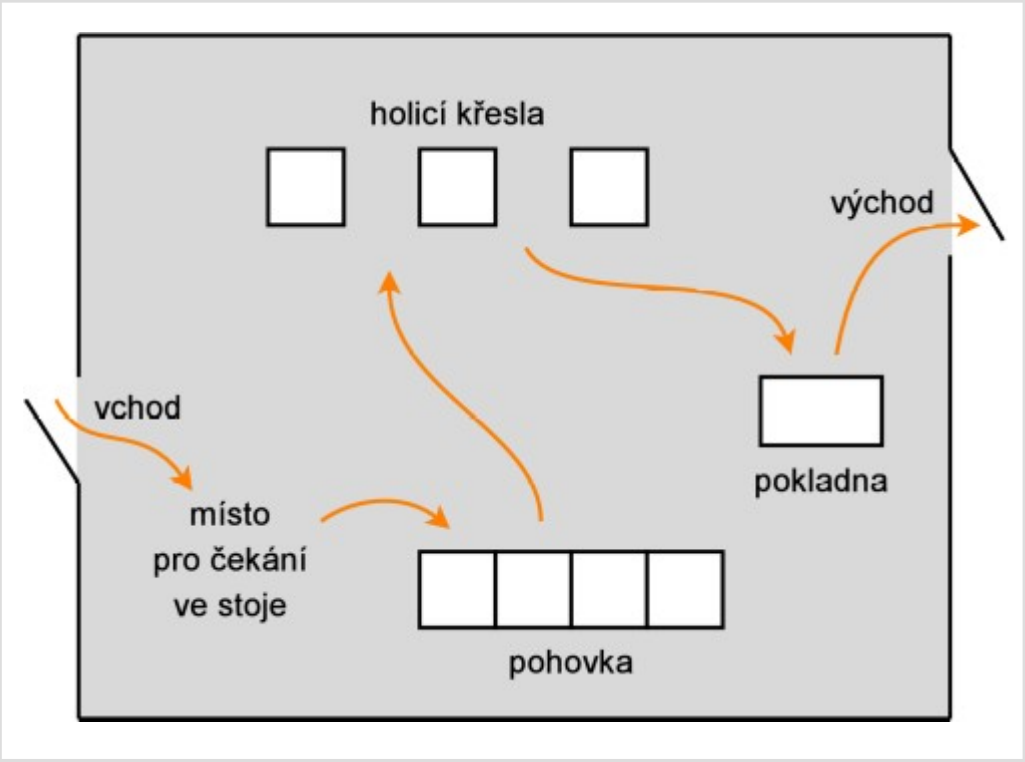
```
repeat
   item = produce_item();
   sem_wait(&mutex);
   buffer[in] = item;
   in = (in + 1) % n;
   sem_post(&mutex);
   sem_post(&item_count);
forever
```

## Konzument

```
repeat
   sem_wait(&item_count);
   sem_wait(&mutex);
   item = buffer[out];
   out = (out + 1) % n;
   sem_post(&mutex);
   consume_item(item);
forever
```

# Barbershop Problem

- u holiče jsou jen 3 křesla
  - a tedy jen 3 holiči
- dovnitř holičství se vejde nejvýše 20 zákazníků
  - další musí čekat venku nebo odejít
- na pohovce je místo pro 4 čekající zákazníky
  - ostatní musí čekat ve stoje
- pokladna je pouze jedna
  - inkasovat může v jednom okamžiku pouze jediný holič



# Potřebujete ostříhat / oholit?

- když je holič volný, přesune se nejdéle čekající zákazník z pohovky na křeslo
- nejdéle stojící zákazník pak usedne na pohovku
- platit může v daném okamžiku vždy pouze jediný zákazník
- holiči obsluhují zákazníky
  - pokud nečeká žádný zákazník, holiči spí

# Co je nutné sledovat?

- vzájemné vylučování
  - kapacitu holičství, pohovky, křesel a pokladen
- synchronizaci
  - umístění právě jednoho zákazníka do křesla
  - přítomnost zákazníka v křesle (chci oholit)
  - dokončení holení
  - opuštění křesla a přechod k pokladně
  - zaplacení a vydání účtenky

# Barbershop Problem – inicializace

## Barbershop Problem – init

### limity

```
sem_init(max_capacity, 20);     // místnost
sem_init(sofa, 4);              // pohovka
sem_init(chair, 3);             // křesla
```

### synchronizace

```
sem_init(cust_ready, 0);        // zákazník je na křesle
for (i = 0; i < chairs; ++i)
  sem_init(finished[i], 0);     // holič dostříhal
sem_init(payment, 0);           // zákazník zaplatil
sem_init(receipt, 0);           // holič vydal účtenku
```

# Barbershop Problem – algoritmus

**Zákazník**

```
sem_wait(max_capacity);
enter_barbershop();
sem_wait(sofa);
sit_on_sofa();
sem_wait(chair);
get_up();
sem_post(sofa);
ch = sit_in_chair();
sem_post(cust_ready);
sem_wait(finished[ch]);
leave_chair();
sem_post(chair);
pay();
sem_post(payment);
sem_wait(receipt);
exit_shop();
sem_post(max_capacity);
```

**Holič**

```
repeat
  sem_wait(cust_ready);
  ch = go_to_chair();
  cut_hair();
  sem_post(finished[ch]);
  go_to_cash_register();
  sem_wait(payment);
  accept_pay();
  sem_post(receipt);
forever
```

# IPC dle OS UNIX System V

- UNIX System V – AT&T, Bell Labs, 1983
  - přímý následník původního UNIXu z roku 1969
  - je základem mnoha implementací
    - AIX (IBM), Solaris (Sun Microsystems) nebo HP-UX (HP)
- meziprocesová komunikace – svipc(7), ftok(3)
  - prostředky OS zpřístupněné systémovými voláními
    - fronty zpráv – msgget(2), msgctl(2), msgop(2)
    - sady semaforů – semget(2), semctl(2), semop(2)
    - segmenty sdílené paměti – shmget(2), shmctl(2), shmop(2)
  - XSI (X/Open System Interface) rozšíření POSIX.1

# IPC dle norem POSIX.1-2001

- standardizace meziprocesové komunikace
- implementuje nověji a s lepším designem IPC
  - posixová volání jsou knihovní funkce, opírají se však o systémová volání
  - sdílené prostředky
    - fronty zpráv – mq_overview(7)
    - sady semaforů – sem_overview(7)
    - sdílená paměť – shm_open(3), mmap(2)
  - obvykle se s nimi pracuje obdobně jako se soubory
    - open, close, unlink, práva

# Systémy vyhovující POSIXu

- certifikované
  - AIX (IBM), HP-UX (HP), IRIX (SGI), OS X od verze 10.5 (Apple), Solaris (Sun, Oracle), Tru64 UNIX (DEC, Compaq, HP), UnixWare (AT&T, Novell, SCO, Xinuos), QNX Neutrino (BlackBerry)
- vyhovující, kompatibilní a většinově kompatibilní
  - Linux, Android
  - BSD: FreeBSD, NetBSD, OpenBSD
  - aj.: VxWorks, MINIX, Darwin (jádro OS X a iOS), …

# POSIX a MS Windows

- rozšíření, POSIX-compliant IDE
  - Cygwin, MinGW
- Microsoft POSIX subsystem
  - volitelné rozšíření pro Windows do verze Win 2000
  - pouze POSIX.1-1990, bez vláken a socketů
- Interix, později Windows Services for UNIX, později Subsystem for UNIX-based Applications
  - pro Windows Server 2003 R2 a pozdější
  - zavrženo od Win 8, vypuštěno od 2012 R2 a 8.1

# Posixové semafory

- hlavičkový soubor semaphore.h(7)

```
#include <semaphore.h>
gcc … -lrt … # připojit knihovnu real-time (librt.so)
```

- čítací semafor s nezáporným čítačem
  - POSIX dovoluje u volání sem_getvalue(3) vracet i zápornou hodnotu – délku fronty
- existují dva typy semaforů – sem_overview(7)
  - pojmenované – jméno je tvaru **/name**
  - nepojmenované – jsou jen v paměti

# Posixové semafory – typy

- s pojmenovanými semafory se pracuje podobně jako se soubory
  - otevření (open), zavření (close), odstranění (unlink)
  - procesy mohou sdílet semafor stejného jména
  - operace wait a post (signal)
- nepojmenované semafory lze alokovat
  - v paměti procesu – sdílení mezi vlákny
  - ve sdílené paměti – sdílení mezi procesy
    - musí nasdílet pomocí shmget(2) nebo shm_open(3)

# Semafory POSIX – inicializace

- inicializace – sem_init(3)

```
int sem_init(sem_t *sem, int pshared,
  unsigned int value);
```

  - inicializuje čítač semaforu **sem** na hodnotu value
  - **pshared** udává typ sdílení
    - nulová hodnota – v rámci procesu mezi vlákny
    - nenulová hodnota – mezi procesy
      - semafor pak musí být ve sdílené paměti
  - vrací: 0 = úspěch, −1 = chyba, **errno** udává chybu

# Semafory POSIX – wait a post

- čekání – sem_wait(3)

```
int sem_wait(sem_t *sem);
```

  - snižuje čítač semaforu **sem**
    - blokuje proces, pokud by se snižovalo do záporu
  - vrací: 0 = úspěch, −1 = chyba, **errno** udává chybu

- signalizace – sem_post(3)

```
int sem_post(sem_t *sem);
```

  - zvyšuje čítač, probudí jeden čekající proces
  - vrací: 0 = úspěch, −1 = chyba, **errno** udává chybu

# Semafory POSIX – linuxová implementace wait a post

**posixová implementace v Linuxu – nezáporným čítač + futex**

```
int sem_wait(sem_t *s) {
    if (atomic dec_if_positive(s->count))       // je-li čítač kladný,
      return 0;                                 // atomicky jej sníží o jedna a ukončí se
    atomic s->nwaiters++;    // jinak zvýší počet čekajících vláken
    while (1) { // futex = fast userspace mutex, řadí vlákna do prioritní fronty
      futex_wait(s->count, 0, …);     // dokud je čítač nulový, blokuje
      if (atomic dec_if_positive(s->count))     // kladný čítač sníží
        break;                                  // a cyklus končí
    }    // snížilo-li čítač na nulu mezitím jiné vlákno, cyklus blokování pokračuje
    atomic s->nwaiters--;    // počet čekajících vláken je snížen
}
int sem_post(sem_t *s) {
    atomic s->count++;          // atomicky zvýší hodnotu čítače o jedna
    if (s->nwaiters > 0)        // existuje-li u semaforu blokované vlákno,
      futex_wake(s->count, 1, …);     // první se odblokuje (dle priority)
}    // probuzení prvního z fronty nezaručuje jeho naplánování před jiným
```

# Semafory POSIX – linuxový futex

- futex – fast userspace mutex
  - operace implementuje knihovna v userspace
  - volání jádra jen při potřebě blokovat a probouzet
  - operace wait – parametry: proměnná, hodnota, …
    - zablokuje volající vlákno, má-li proměnná danou hodnotu
    - blokované vlákno je dle své priority vloženo do fronty
  - operace wake – parametry: proměnná, počet, …
    - probudí daný počet vláken čekajících na dané proměnné
    - přestože jsou vlákna vybírána z fronty, není zaručeno jejich naplánování před ostatními nad stejnou proměnnou

# Semafory POSIX – linuxová implementace wait a post (2)

## vlákno 1

```
sem_wait(&s):
  s->count--;
  return;
// provádí KS



// konec KS
sem_post(&s):
  s->count++;
  futex_wake(…);
  // probudí vlákno 2
```

## vlákno 2

```
sem_wait(&s):
  // čítač je nulový
  futex_wait(…);
  // blokuje
  // je první ve frontě



  // vlákno je připraveno



  // čítač je nulový
  // vlákno opět blokuje
```

## vlákno 3

```
// vlákno běží
sem_wait(&s):
  s->count--;
  return;
// provádí KS
```

# Semafory POSIX – get, destroy

- zjištění hodnoty čítače – sem_getvalue(3)

```
int sem_getvalue(sem_t *sem, int *sval);
```

  - vloží hodnotu čítače na adresu **sval**
  - vrací: 0 = úspěch, −1 = chyba, **errno** udává chybu
- uvolnění prostředků – sem_destroy(3)

```
int sem_destroy(sem_t *sem);
```

  - uvolní prostředky inicializovaného semaforu
  - vrací: 0 = úspěch, −1 = chyba, **errno** udává chybu

# Semafory POSIX – omezené wait

- omezené čekání – sem_timedwait(3)

```
int sem_timedwait(sem_t *sem,
  const struct timespec *abs_timeout);
```

  - blokuje než vyprší timeout
  - vrací: 0 = OK, **ETIMEDOUT** při nutnosti čekat nadále

- pokus o čekání – sem_trywait(3)

```
int sem_trywait(sem_t *sem);
```

  - neblokující – vrací **EAGAIN** při nutnosti čekat

# Semafory POSIX – otevření

- otevření pojmenovaného sem. – sem_open(3)

```
sem_t *sem_open(const char *name, int *oflag);
```

  - otevře sdílený semafor jména **name** (tvar: **/jméno**)
  - obsahuje-li **oflag** **O_CREAT**, semafor se vytvoří (neexistuje-li) a je nutné zadat další dva argumenty

```
sem_t *sem_open(const char *name, int *oflag,
  mode_t mode, unsigned int value);
```

    - **mode**: práva (**S_IRWXU**, **S_IRUSR**, …), respektuje umask
    - **value**: inicializace čítače semaforu
  - vrací: semafor nebo **SEM_FAILED** při chybě, **errno**
  - **#include** **<fcntl.h>** (oflag) a **<sys/stat.h>** (mode)

# Semafory POSIX – zavření, zrušení

- zavření semaforu – sem_close(3)

```
int sem_close(sem_t *sem);
```

  - zavře pojmenovaný semafor
  - vrací: 0 = úspěch, −1 = chyba, **errno** udává chybu

- uvolnění prostředků – sem_unlink(3)

```
int sem_unlink(const char *name);
```

  - odstraní jméno semaforu
    - prostředky semaforu jsou uvolněny, až všechny procesy semafor zavřou
  - vrací: 0 = úspěch, −1 = chyba, **errno** udává chybu

# Semafory System V – IPC

- alokace semaforů – semget(2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

  - vrátí identifikátor sady semaforů podle klíče **key**, při chybě vrací −1 (a nastaví **errno**)
  - nová sada **nsems** semaforů je vytvořena, pokud
    - **key** = **IPC_PRIVATE** (lepší název by byl **IPC_NEW**) nebo
    - **key** ≠ **IPC_PRIVATE**, sada sdružená s klíčem **key** neexistuje a **semflg** obsahuje **IPC_CREAT**

# Semafory System V – alokace

- alokace semaforů (pokračování) – semget(2)

```
int semget(key_t key, int nsems, int semflg);
```

  - semafory **nejsou** inicializovány (POSIX.1-2001)
  - struktura **semid_ds** je naplněna – viz semctl(2)
    - vlastník a skupina podle volajícího procesu, práva a počet semaforů v sadě podle parametrů, čas modifikace
  - práva lze specifikovat dolními 9 bity v **semflg**
    - bity **x** (execute) systém nevyužívá; práva – viz též stat(2)
  - je-li dáno **IPC_CREAT** a **IPC_EXCL** a semafor již existuje, vrací chybu (**EEXIST**)

# Semafory System V – operace

- operace se sadou semaforů – semop(2)

```
int semop(int semid, struct sembuf *sops,
  unsigned nsops);
```

  - provede **nsops** operací se semafory s id **semid**
    - operace jsou v poli **sops**, jehož položky jsou:

```
struct sembuf {
  unsigned short semnum;  // číslo semaforu (čísluje se od 0)
  short sem_op;  // číslo, o které se změní čítač, nebo 0 (čekej)
  short sem_flg; // příznaky: SEM_UNDO, IPC_NOWAIT
}
```

    - **SEM_UNDO** – při ukončení procesu se zruší změna
  - **sem_op** je obvykle −1 (wait) nebo (+1) (signal)
    - hodnota **sem_op** 0 znamená čekání na nulu

# Semafory System V – ovládání

- ovládání sady semaforů – semctl(2)

```
int semctl(int semid, int semnum, int cmd, …);
```

  - vykoná příkaz **cmd** na semaforu **semnum** v **semid**
    - **IPC_STAT** (zjištění info), **IPC_SET** (nastavení práv), **IPC_RMID** (okamžité odstranění sady semaforů), **SETVAL**, **SETALL** (nastavení čítače/ů semaforu/ů), …
  - podle **cmd** je třeba i čtvrtý argument
    - typ je nutné deklarovat v programu!

```
union semunion {
  int val;                  // hodnota semaforu (pro SETVAL)
  struct semid_ds *buf;     // buffer pro IPC_STAT a IPC_SET
  unsigned short *array;    // pole pro SETALL a GETALL
}
```

# Semafory System V – příklad

## použití semaforu System V

### definice union

```
typedef union {
  int val;
  struct semid_ds *buf;
  unsigned short *array;
} semunion_t;
```

### deklarace proměnných

```
int sID;
semunion_t sdata;
key_t sKey = 1357;
// lokálně pro každé vlákno:
struct sembuf sops;
```

### inicializace

```
sID = semget(sKey, 1,
  IPC_CREAT|0666); // umask platí
sdata.val = 1;
semctl(sID, 0, SETVAL, sdata);
```

### wait a signal

```
sops.sem_num = 0;
sops.sem_flg = SEM_UNDO;
sops.sem_op = -1; // wait
semop(sID, &sops, 1);
sops.sem_op = +1; // signal
semop(sID, &sops, 1);
```

### odstranění

```
semctl(sID, 0, IPC_RMID);
```

# Semafory – Win32 API

- CreateSemaphore() – alokace / otevření semaforu
- OpenSemaphore() – otevření pojmenovaného semaforu
- WaitForSingleObject() – operace wait
- ReleaseSemaphore() – operace signal
- CloseHandle() – uvolnění prostředků

# Posixová vlákna a mutexy

- mutex je zámek, který zaručuje
  - atomicitu a vzájemné vylučování
    - lze zamknout pouze jediným vláknem
  - neaktivní čekání
    - vlákno je při pokusu zamknout zamčený mutex blokováno
- tři typy mutexů (dva jsou rozšiřující)
  - fast mutex – lze zamknout pouze jednou
  - recursive mutex – lze zamknout „na více západů“
  - error checking mutex – zamykání zamčeného selže

# Mutex – inicializace

- deklarace a inicializace – pthread_mutex_init(3)

```
pthread_mutex_t a_mutex =
  PTHREAD_MUTEX_INITIALIZER;
```

  - fast mutex – lze zamknout jen jednou
  - zamčení jedním vláknem podruhé = DEADLOCK

```
pthread_mutex_t a_mutex =
  PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
```

  - recursive mutex – lze zamknout **jedním** vláknem vícekrát
  - jiná vlákna jsou při zamykání (již zamčeného mutexu) vždy blokována
  - musí se zamykajícím vláknem vícekrát odemknout

# Mutex – zamčení

- zamčení – pthread_mutex_lock(3)

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

  - zamkne mutex; je-li už zamčený, blokuje vlákno
  - vrací: 0 = úspěch, nenulová hodnota = číslo chyby
- pokus o zamčení – pthread_mutex_trylock(3)

```
int pthread_mutex_trylock(pthread_mutex_t *m);
```

  - zamkne mutex; je-li už zamčený, neblokuje
    - rekurzivní mutex se podaří vlastnícímu vláknu zamknout
  - vrací: 0 = úspěch, nenulová hodnota = číslo chyby

# Mutex – odemčení

- odemčení – pthread_mutex_unlock(3)

```
int pthread_mutex_unlock(pthread_mutex_t *m);
```

  - odemkne mutex zamčený tímto vláknem
    - nelze odemknout vláknem nevlastnícím zámek
    - čeká-li jiné vlákno na odemčení, je odblokováno
    - výběr vlákna (čeká-li jich více) závisí na plánovací strategii
  - vrací: 0 = úspěch, nenulová hodnota = číslo chyby

# Mutex – zrušení, inicializace

- zrušení – pthread_mutex_destroy(3)

```
int pthread_mutex_destroy(pthread_mutex_t *m);
```

  - zruší mutex – stane se neplatným, neinicializovaným
    - zamčený mutex nelze zrušit
  - mutex lze znovu inicializovat

```
int pthread_mutex_init(pthread_mutex_t *mutex,
  const pthread_mutexattr_t *restrict attr);
```

  - necháme-li **attr** **NULL**, budou atributy implicitní
  - vrací: 0 = úspěch, nenulová hodnota = číslo chyby

# Mutex – řešení kritické sekce

**inicializace**

```
pthread_mutex_t a_mutex = PTHREAD_MUTEX_INITIALIZER;
int rc;
```

**použití**

```
rc = pthread_mutex_lock(&a_mutex);      // wait
if (rc) {
    perror("pthread_mutex_lock");
    pthread_exit(NULL);
}
KS; // kritická sekce
rc = pthread_mutex_unlock(&a_mutex);    // signal
if (rc) {
    perror("pthread_mutex_unlock");
    pthread_exit(NULL);
}
ZS; // zbytková sekce
```

# Mutexy – Win32 API

- CreateMutex(), CreateMutexEx() – alokace / otevření mutexu
- OpenMutex() – otevření pojmenovaného mutexu
- WaitForSingleObject() – operace lock
- ReleaseMutex() – operace unlock
- CloseHandle() – uvolnění prostředků

# Posixová vlákna a podmínky

- podmínková proměnná
  - slouží k synchronizaci vláken
  - umožňuje vláknu neaktivně čekat na událost
  - nezaručuje exkluzivitu přístupu
    - je třeba k ní přistupovat pomocí mutexu
  - na událost může čekat i několik vláken
  - událost oznamuje některé vlákno signálem
    - signál může probudit jediné vlákno
    - lze poslat i všesměrový signál a probudit všechna vlákna
    - nečeká-li žádné vlákno, je signál ztracen

# Podmínky – inicializace

- deklarace a inicializace – pthread_cond_init(3)

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

  - inicializuje podmínkovou proměnnou (při překladu)
  - pro inicializaci v době běhu (run-time), je třeba volat

```
int pthread_cond_init(pthread_cond_t *restrict
  cond, const pthread_condattr_t *restrict attr);
```

  - necháme-li **attr** **NULL**, budou atributy implicitní
  - vrací: 0 = úspěch, nenulová hodnota = číslo chyby

# Podmínky – signalizace události

- signalizace události – pthread_cond_signal(3)

```
int pthread_cond_signal(pthread_cond_t *cond);
```

  - signalizuje událost, probudí jedno čekající vlákno
    - POSIX.1 explicitně stanovuje „**alespoň jedno**“
      - SMP: není efektivní ošetřovat krajní možnost probuzení více vláken
    - nečeká-li žádné vlákno, je signál ztracen

```
int pthread_cond_broadcast(pthread_cond_t
  *cond);
```

  - signalizuje událost, probudí všechna čekající vlákna
  - vrací: 0 = úspěch, nenulová hodnota = číslo chyby

# Podmínky – čekání na událost

- čekání na událost – pthread_cond_wait(3)

```
int pthread_cond_wait(pthread_cond_t *restrict
  cond, pthread_mutex_t *restrict mutex);
```

  - blokuje vlákno, dokud nedostane signál
  - před čekáním je třeba zamknout **mutex**

```
int pthread_cond_timedwait(pthread_cond_t
  *restrict cond, pthread_mutex_t *restrict
  mutex, const struct timespec *restrict abst);
```

  - časově omezené čekání – **abst** je absolutní čas
  - vrací: 0 = úspěch, nenulová hodnota = číslo chyby
    - v případě chyby zůstane **mutex** zamčený

# Podmínkové proměnné – příklad

## inicializace

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
bool done = false;                          // příznak dokončení
```

## vlákno 1 – čeká na událost (např. vypočtení hodnoty *x*)

```
pthread_mutex_lock(&mutex);                 // zamčení mutexu
if (!done)                                  // kontrola příznaku: není-li hotovo,
  pthread_cond_wait(&cond, &mutex);         // čekání na signál
pthread_mutex_unlock(&mutex);               // odemčení mutexu
use(x);                                     // použití sdílené proměnné x
```

## vlákno 2 – signalizuje událost (hodnota *x* je už platná)

```
compute(x);                                 // nastavení hodnoty proměnné x
pthread_mutex_lock(&mutex);                 // zamčení mutexu
done = true;                                // nastavení příznaku dokončení
pthread_cond_signal(&cond);                 // signalizace dokončení
pthread_mutex_unlock(&mutex);               // odemčení mutexu
```

# Podmínkové proměnné – implementace čekání

## vlákno 1 – čeká na událost

```
pthread_mutex_lock(&mutex);       // 1. zamčení mutexu
if (!done)                        // 2. kontrola příznaku: není-li hotovo,
  pthread_cond_wait(&cond, &mutex);       // 3. čekání na signál
pthread_mutex_unlock(&mutex);     // 13. odemčení mutexu
```

## vlákno 2 – signalizuje událost

```
pthread_mutex_lock(&mutex);       // 4. blokace, 7. zamčení mutexu
done = true;                      // 8. nastavení příznaku dokončení
pthread_cond_signal(&cond);       // 9. signalizace dokončení
pthread_mutex_unlock(&mutex);     // 11. odemčení mutexu
```

## knihovna – implementace čekání na signál

```
pthread_cond_wait(…_cond_t *cond, …_mutex_t *mutex) {
  pthread_mutex_unlock(&mutex);// 5. odemčení mutexu
  // 6. blokování volajícího vlákna – bude probuzeno signálem
  pthread_mutex_lock(&mutex);   // 10. blokace, 12. zamčení mutexu
```

# Synchronizace – Win32 API

- CreateEvent() – alokace / otevření objektu
- OpenEvent() – otevření pojmenovaného objektu
- SetEvent(), ResetEvent() – signalizace události, zrušení signalizace události
  - na rozdíl od pthread_cond_signal(3) se signál, na který žádné vlákno nečeká, neztratí
- WaitForSingleObject() – čekání na událost
- CloseHandle() – uvolnění prostředků

# Posixová vlákna a bariéry

- objekt bariéra pro synchronizaci vláken
  - umožňuje vláknu neaktivně čekat na ostatní vlákna
  - jakmile k bariéře dospěje daný počet vláken, bariéra propustí všechna vlákna – paralelní běh
  - definováno normou POSIX.1-2001 a Single UNIX Specification, Version 3
  - nutno definovat jeden z následujících symbolů **před** všemi direktivami `#include`

```
#define _XOPEN_SOURCE 600 // obvykle tento
#define _POSIX_C_SOURCE 200112L
```

# Bariéry – inicializace

- deklarace a inicializace – pthread_barrier_init(3)

```
pthread_barrier_t a_barrier;
int pthread_barrier_init(
  pthread_barrier_t *restrict barrier,
  const pthread_barrierattr_t *restrict attr,
  unsigned count);
```

  - **attr** nastavuje sdílení mezi procesy, smí být **NULL**
  - hodnota **count** udává, kolik vláken musí zavolat funkci pthread_barrier_wait(3) pro návrat z ní
  - vrací: 0 = úspěch, nenulová hodnota = číslo chyby

# Bariéry – inicializace a zrušení atributů

- deklarace, inicializace a zrušení atributů bariéry

```
pthread_barrierattr_t attr;
int pthread_barrierattr_init(
  pthread_barrierattr_t *attr);
```

  - inicializuje **attr** výchozími hodnotami

```
int pthread_barrierattr_destroy(
  pthread_barrierattr_t *attr);
```

  - zruší atributy (dealokuje)
  - vrací: 0 = úspěch, nenulová hodnota = číslo chyby

# Bariéry – sdílení mezi procesy

- nastavení a zjištění sdílení mezi procesy

```
int pthread_barrierattr_setpshared(
  pthread_barrierattr_t *attr,
  int pshared);

int pthread_barrierattr_getpshared(
  const pthread_barrierattr_t *restrict attr,
  int *restrict pshared);
```

  - hodnota **pshared**: **PTHREAD_PROCESS_SHARED**
    - bariéra musí být ve sdílené paměti mezi procesy
  - nesdíleno: **PTHREAD_PROCESS_PRIVATE**

# Bariéry – čekání na bariéře

- čekání na bariéře – pthread_barrier_wait(3)

```
int pthread_barrier_wait(
  pthread_barrier_t *barrier);
```

  - blokuje vlákno, dokud tuto funkci nezavolá počet vláken stanovený inicializací pthread_barrier_init(3)
  - pak jsou všechna vlákna odblokována současně
  - vrací při úspěchu pro jedno (nespecifikované) vlákno hodnotu **PTHREAD_BARRIER_SERIAL_THREAD** a pro ostatní nulu; nenulová hodnota = číslo chyby
    - bariéra je znovu připravena pro použití na daný počet

# Bariéry – zrušení

- uvolnění prostředků – pthread_barrier_destroy(3)

```
int pthread_barrier_destroy(
  pthread_barrier_t *barrier);
```

  - odstraní bariéru a uvolní s ní sdružené prostředky
  - vrací: 0 = úspěch, nenulová hodnota = číslo chyby

# Bariéry – příklad

**inicializace**

```
#define _XOPEN_SOURCE 600                    // pro přenositelnost
#include <pthread.h>
pthread_barrier_t a_barrier;                 // deklarace
if (pthread_barrier_init(&a_barrier, NULL, 5)) {
    perror("barrier init");
    exit(EXIT_FAILURE);
}
```

**použití** (v pěti vláknech)

```
switch (pthread_barrier_wait(&a_barrier)) {
  case PTHREAD_BARRIER_SERIAL_THREAD:  // jedno z vláken
    (void) pthread_barrier_destroy(&a_barrier);
  case 0: break;                       // ostatní vlákna
  default: perror("barrier wait"); exit(EXIT_FAILURE);
}
```

# Bariéry – Win32 API

- InitializeSynchronizationBarrier() – inicializace
- EnterSynchronizationBarrier() – čekání
  - návratová hodnota: 1 vlákno TRUE, ostatní FALSE
- DeleteSynchronizationBarrier() – zrušení

# Předávání zpráv

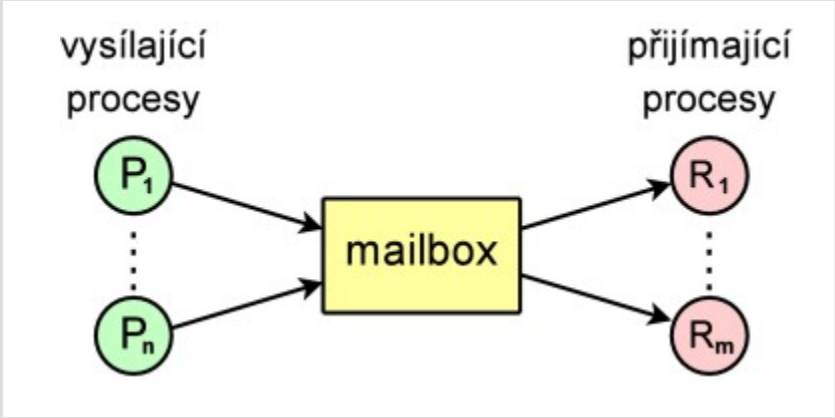
- komunikační prostředek OS pro procesy
- je nutné vzájemné vylučování
  - dochází k výměně informací
  - zajišťuje OS
- systémová volání
  - send(cíl, zpráva)
  - receive(zdroj, zpráva)

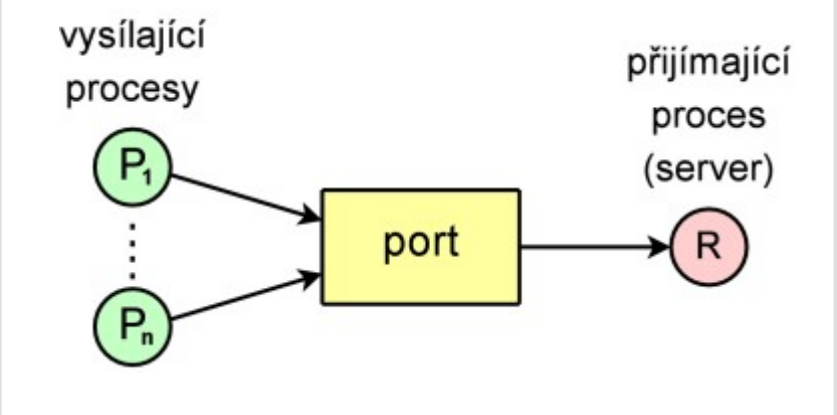
# Předávání zpráv – adresace

- adresace cíle může být pro send příp. i receive
  - **přímá** – adresujeme cílový proces
    - s případnou možností určovat obecné adresy
  - **nepřímá** – adresujeme frontu zpráv (mailbox)
    - různé procesy pak mohou zprávy vyzvedávat
    - implementace může umožňovat také vybírání jen určitých typů zpráv
- receive může dostat parametrem hodnotu, pomocí které potvrdí přijetí
  - umožňuje zjistit doručení při neblokujícím send

# Předávání zpráv – mailbox a port

- mailbox
  - je vlastněn párem vysílač / přijímač
  - může být sdílen více vysílači / přijímači
- port
  - svázán s jediným přijímačem – serverem
  - model klient / server



# Předávání zpráv – (ne)blokování

- možné implementace blokování send a receive
  - **neblokující** send
    - nečeká se na doručení (vyzvednutí zprávy adresátem)
    - může blokovat při zaplnění fronty
  - **blokující** send
    - čeká se na vyzvednutí zprávy adresátem
  - **neblokující** receive
    - není-li dostupná žádná zpráva, vrací receive chybu
  - **blokující** receive
    - adresát je blokován, dokud není dostupná zpráva
- blokující send a blokující receive – **rendezvous**

# Předávání zpráv – (ne)blokování, typická implementace

- typická implementace blokování send a receive pro fronty zpráv s omezenou velikostí
  - **neblokující send**
    - blokuje pouze při zaplnění fronty zpráv
  - **blokující receive**
    - příjemce je blokován, není-li dostupná žádná zpráva
  - neblokující varianty lze nastavit parametrem, volání pak místo blokování vrací chybu

# Předávání zpráv – řešení KS

- sdílená fronta **mutex** se inicializuje zasláním jedné zprávy
- vstupní sekce volá **blokující** receive
- výstupní sekce volá **neblokující** send

**jeden z procesů $P_i$**

```
send(mutex, "go");
```

**proces $P_i$**

```
repeat
   receive(mutex, &msg);
   KS;
   send(mutex, msg);
   ZS;
forever
```

- první proces, který provede receive, se dostane do KS, ostatní jsou blokovány

# Předávání zpráv – synchronizace

- příkaz **S2** v procesu $P_1$ musí být proveden až **po provedení S1** v $P_0$
- vyprázdníme mailbox sync
- $P_1$ před provedením **S2** volá **blokující receive**
- $P_0$ po provedení **S1** volá **neblokující send**
- provede-li se nejprve send, receive neblokuje

**inicializace**

```
// vyprázdnění fronty zpráv
```

**proces $P_0$**

```
vypočti x;  // S1
send(sync, msg);
```

**proces $P_1$**

```
receive(sync, &msg);
použij x;   // S2
```

# Producent / konzument – fronta zpráv s omezenou kapacitou

- do fronty zpráv **storage** zasílají producenti položky
  - jedná se o sklad (buffer)
- kapacita skladu je daná maximální velikostí fronty
  - zaplní-li producenti sklad, bude send blokovat

**Producent**

```
repeat
   msg = produce_item();
   send(storage, msg);
forever
```

**Konzument**

```
repeat
   receive(storage, &msg);
   consume_item(msg);
forever
```

- konzumenti vybírají položky voláním **receive**
- funguje pro více producentů i konzumentů

# Zprávy System V – IPC

- alokace fronty – msgget(2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

  - vrátí identifikátor fronty zpráv podle klíče **key**, při chybě vrací −1 (a nastaví **errno**)
  - nová fronta zpráv je vytvořena, pokud
    - **key** = **IPC_PRIVATE** (lepší název by byl **IPC_NEW**) nebo
    - **key** ≠ **IPC_PRIVATE**, fronta sdružená s klíčem **key** neexistuje a **msgflg** obsahuje **IPC_CREAT**

# Zprávy System V – alokace

- alokace (pokračování) – msgget(2)

```
int msgget(key_t key, int msgflg);
```

  - struktura **msqid_ds** je naplněna – viz msgctl(2)
    - vlastník a skupina podle volajícího procesu, práva dle **msgflg**, čas modifikace na aktuální, maximální velikost fronty na **MSGMNB** a zbytek parametrů je vynulován
  - práva lze specifikovat dolními 9 bity v **msgflg**
  - je-li dáno **IPC_CREAT** a **IPC_EXCL** a fronta existuje, vrací chybu (**EEXIST**)

# Zprávy System V – zaslání zprávy

- poslání zprávy do fronty zpráv – msgop(2)

```
void *msgsnd(int msqid, const void *msgp,
  size_t msgsz, int msgflg);
```

  - zkopíruje zprávu do fronty s id **msqid**
    - **msgp** je ukazatel na strukturu, v níž **mtext** je pole o velikosti **msgsz** – dovolena je i nulová velikost

```
struct msgbuf {
  long mtype;          // typ zprávy; musí být > 0
  char mtext[1];       // obsah zprávy
}
```

  - je-li ve frontě dost místa, je volání neblokující
  - není-li místo, blokuje dle **IPC_NOWAIT** v **msgflg**

# Zprávy System V – přijetí zprávy

- přijetí zprávy z fronty zpráv – msgop(2)

```
ssize_t msgrcv(int msqid, void *msgp,
  size_t msgsz, long msgtyp, int msgflg);
```

  - zkopíruje zprávu z fronty **msqid** na adresu **msgp**
  - **msgsz** udává maximální velikost položky **mtext**
    - je-li zpráva větší, rozhoduje **MSG_NOERROR** v **msgflg** – buď se zpráva zkrátí (část je ztracena) nebo vrací chybu
  - **msgtyp** určuje, které zprávy se mají vybírat
    - 0 – první ve frontě, > 0 – první daného nebo jiného typu (**MSG_EXCEPT** v **msgflg**), < 0 – první nejmenší typ hodnoty ≤ |**msgtyp**|

# Zprávy System V – ovládání fronty

- ovládání front zpráv – msgctl(2)

```
int msgctl(int msqid, int *cmd,
  struct msqid_ds *buf);
```

  - vykoná příkaz **cmd** na frontě zpráv
    - **IPC_SET** – nastavení velikosti fronty a oprávnění
    - **IPC_RMID** – okamžité odstranění fronty
    - **IPC_STAT** – zjištění atributů (naplní strukturu **msqid_ds**)
  - datová struktura **msqid_ds** obsahuje
    - oprávnění (vlastník, skupina, práva), časy (poslední zaslání, přijetí, změna), počet zpráv ve frontě, maximální velikost fronty v bajtech, PID (poslední zaslání, přijetí)

# Zprávy System V – data fronty

- datová struktura **`msqid_ds`** – msgctl(2)

```
struct msqid_ds {
  struct ipc_perm  msg_perm;     // oprávnění (vlastnictví a práva)
  time_t           msg_stime;    // čas posledního msgsnd(2)
  time_t           msg_rtime;    // čas posledního msgrcv(2)
  time_t           msg_ctime;    // čas posledního změny
  msgqnum_t        msg_qnum;     // počet zpráv ve frontě
  msglen_t         msg_qbytes;   // maximální velikost fronty
  pid_t            msg_lspid;    // PID posledního msgsnd(2)
  pid_t            msg_lrpid;    // PID posledního msgrcv(2)
};
struct ipc_perm {
  key_t            __key;        // klíč předaný msgget(2)
  uid_t uid, gid, cuid, cgid;    // UID/GID vlastníka a tvůrce
  unsigned short   mode;         // práva
  unsigned short   __seq;        // pořadové číslo
};
```

# Posixové fronty zpráv

- fronty zpráv – mq_overview(7)

```
#include <mqueue.h>
```

  - knihovna librt – nutnost linkování

```
gcc … -lrt …
```

  - lepší design než původní IPC UNIX System V
  - nemusejí být implementovány všude
    - Linux od verze jádra 2.6.6 (2004), glibc od verze 2.3.4
- Linux
  - rozhraní proc: /proc/sys/fs/mqueue/
  - lze připojit i virtuální FS typu mqueue

# Zprávy POSIX – vytvoření fronty

- vytvoření nebo otevření fronty – mq_open(3)

```
mqd_t mq_open(const char *name, int oflag);
```

  - vytvoří / otevře posixovou frontu zpráv – jako open(2)
    - jméno fronty je tvaru **/jméno**
    - **oflag** – způsob otevření (**O_CREAT**, **O_RDWR**, …)
  - obsahuje-li **oflag** **O_CREAT**, přidat 2 parametry

```
mqd_t mq_open(const char *name, int oflag,
  mode_t mode, struct mq_attr *attr);
```

    - **mode** – přístupová práva (**S_IRWXU**, **S_IRUSR**, …) – respektuje se nastavení umask(2)
    - **attr** může být **NULL** (výchozí hodnoty), viz mq_getattr(3)
  - vrací **queue descriptor** nebo **(mqd_t) -1** při chybě

# Zprávy POSIX – atributy

- zjištění / nastavení atributů – mq_getattr(3)

```
mqd_t mq_getattr(mqd_t mqdes,
  struct mq_attr *attr);
mqd_t mq_setattr(mqd_t mqdes, struct mq_attr
  *newattr, struct mq_attr *oldattr);
struct mq_attr {
  long mq_flags;     // 0 nebo O_NONBLOCK
  long mq_maxmsg;    // max. počet zpráv ve frontě (10)
  long mq_msgsize;   // max. velikost zprávy v bajtech (8 KiB)
  long mq_curmsgs;   // aktuální počet zpráv ve frontě
}
```

  - **mq_setattr** smí modifikovat pouze **mq_flags**
  - vrací: 0 = úspěch, −1 = chyba, **errno** udává chybu

# Zprávy POSIX – uzavření, odpojení

- uzavření fronty – mq_close(3)

```
mqd_t mq_close(msq_t mqdes);
```

  - uzavře posixovou frontu zpráv deskriptoru **mqdes**
  - ruší případnou registraci procesu na upozornění
- uvolnění prostředků – mq_unlink(3)

```
mqd_t mq_unlink(const char *name);
```

  - podobné unlink(2) – odstraní jméno fronty
    - prostředky fronty jsou uvolněny, až všechny procesy frontu uzavřou
  - vrací: 0 = úspěch, −1 = chyba, **errno** udává chybu

# Zprávy POSIX – zaslání

- zaslání zprávy – mq_send(3)

```
mqd_t mq_send(mqd_t mqdes, const char *msg_ptr,
  size_t msg_len, unsigned msg_prio);
```

  - vloží do fronty **mqdes** zprávu na adrese **msg_ptr** délky 0 ≤ **msg_len** ≤ **mq_msgsize** (atribut fronty)
  - řazení do fronty je podle priority **msg_prio**
    - vyšší hodnota = vyšší priorita
  - volání je blokující, pokud je fronta plná a **mq_flags** (atribut fronty) neobsahuje **O_NONBLOCK**
  - vrací: 0 = úspěch, −1 = chyba, **errno** udává chybu

# Zprávy POSIX – přijetí

- přijetí zprávy – mq_receive(3)

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
  size_t msg_len, unsigned *msg_prio);
```

  - zkopíruje z fronty **mqdes** zprávu na adresu **msg_ptr** max. délky **msg_len** ≥ **mq_msgsize** (atribut fronty)
  - není-li **msg_prio** **NULL**, je vrácena i priorita
  - volání je blokující, pokud je fronta prázdná a **mq_flags** (atribut fronty) neobsahuje **O_NONBLOCK**
  - vrací: délku zprávy, −1 = chyba, **errno** udává chybu

# Zprávy POSIX – operace s limitem

- zaslání / přijetí s limitem – mq_timedsend(3)

```
mqd_t mq_timedsend(mqd_t mqdes, const char
  *msg_ptr, size_t msg_len, unsigned msg_prio,
  const struct timespec *abs_timeout);
ssize_t mq_timedreceive(mqd_t mqdes, char
  *msg_ptr, size_t msg_len, unsigned *msg_prio,
  const struct timespec *abs_timeout);
```

  - volání blokuje nejvýše po dobu podle **abs_timeout**
    - timeout je absolutní čas od Epochy (doba od 1. 1. 1970)

```
struct timespec {
  time_t    tv_sec;          // sekundy
  long      tv_nsec;         // nanosekundy
};
```

# Zprávy POSIX – upozornění (1)

- upozornění na neprázdnou frontu – mq_notify(3)

```
mqd_t mq_notify(mqd_t mqdes,
  const struct sigevent *notification);

struct sigevent {
  int sigev_notify;           // metoda upozornění
  int sigev_signo;            // upozorňující signál
  union sigval sigev_value;   // data předaná při upozornění
  void (*sigev_notify_function) (union sigval);
                              // funkce pro upozornění vláknem
  void *sigev_notify_attributes; // atributy vláknové funkce
};

union sigval {                // hodnota předaná při upozornění
  int sigval_int;
  void *sigval_ptr;
};
```

# Zprávy POSIX – upozornění (2)

- upozornění na neprázdnou frontu – mq_notify(3)

```
mqd_t mq_notify(mqd_t mqdes,
  const struct sigevent *notification);
```

  - upozorní proces přijde-li zpráva do prázdné fronty
    - registrace je po upozornění zrušena
  - **sigev_notify** nastavuje typ upozornění
    - **SIGEV_NONE** – pouze registrace procesu, bez upozornění
    - **SIGEV_SIGNAL** – upozornění signálem **sigev_signo**
    - **SIGEV_THREAD** – upozornění vytvořením vlákna
  - je-li **notification** **NULL**, ruší se registrace procesu
  - pouze jediný proces se smí registrovat

# Monitory (řešení problémů souběhu)

- koncept: B. Hansen 1972, C. A. R. Hoare 1974
- konstrukce ve vyšším programovacím jazyce pro stejné služby jako semafory (synchronizace a vzájemné vylučování), snadněji ovladatelné
- vyskytují se v řadě jazyků pro souběžné programování (concurrent programming)
  - Concurrent Pascal, Modula-3, uC++, Java, C#
    - implementace (Java, C#) se liší od původního konceptu
- mohou být implementovány pomocí semaforů

# Monitor (nástroj pro řešení problémů souběhu)

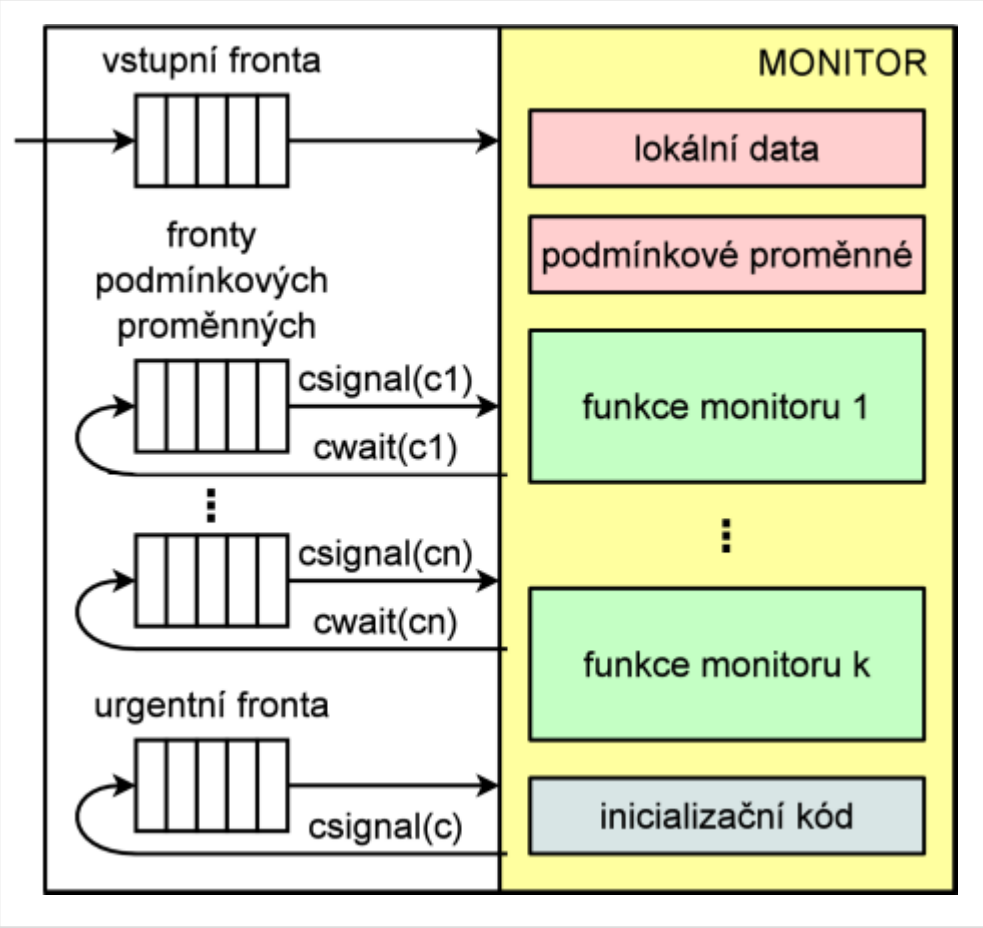
- SW modul (podobný objektu / třídě)
  - lokální proměnné – sdílená data
    - tato data nejsou viditelná vně monitoru
  - funkce zpřístupňující lokální data
  - inicializační část
- **v monitoru** (jeho funkci) **smí být v daném okamžiku pouze jediné vlákno**
  - monitor tak zajišťuje vzájemné vylučování
  - synchronizaci lze zajistit podmínkovými proměnnými

# Monitor – podmínkové proměnné

- synchronizační nástroj monitoru
- podmínkové proměnné jsou lokální v monitoru a dostupné pouze pomocí funkcí monitoru
- lze je měnit pouze dvěma funkcemi monitoru
  - cwait(cv) – blokuje vlákno, dokud není zavoláno:
  - csignal(cv) – obnoví provádění vlákna blokovaného podmínkovou proměnnou cv
    - je-li takových vláken více, vybere se jedno z nich
    - **není-li žádné takové vlákno, neprovede se nic**

# Monitor (obrázek)

- čekající vlákna jsou ve frontě vstupní nebo podmínkové
- provedením cwait(c) se vlákno zařadí do podmínkové fronty
- csignal(c) aktivuje jedno čekající vlákno z podmínkové fronty
  - csignal blokuje (není-li to poslední příkaz ve funkci)



# Monitor a Java

- implementace se odlišuje, je třeba úprav:
  - lokální proměnné je třeba deklarovat jako privátní
  - všechny metody je třeba deklarovat `synchronized`
    - Java umožňuje způsobit výjimku (přerušení) v KS!
    - ošetření výjimky KS – uvést data do konzistentního stavu
  - existuje jediná anonymní podmínková proměnná ovládaná pomocí wait() a notify() či notifyAll()
  - vstupní ani podmínková „fronta“ není FIFO
- HANSEN, Per Brinch: Java's Insecure Parallelism. In: *ACM SIGPLAN Notices* [online]. 1999, č. 34, 38–45. [cit. 2012-11-12]. ISSN 0362-1340. Odkaz: <PBHansenParalelism.pdf>.

# Monitor a Java 5 (2004)

- přidány nástroje pro souběžné programování
  - semafor, bariéra, synchronizační nástroje
  - monitor s podmínkovými proměnnými lze napodobit pomocí zámku

```
final Lock monitor = new ReentrantLock();
final Condition c1 = monitor.newCondition();
final Condition c2 = monitor.newCondition();
monitor.lock();
try { // kritická sekce, může generovat výjimku přerušení
    c1.await();      // čekání na podmínku
    c2.signal();     // signalizace, probudí čekající vlákno
}
finally { monitor.unlock(); }
```

# Monitor a C# (.NET Framework)

- implementace se odlišuje
  - zámek (KS) je použit na (privátní) objekt
  - použito příkazem `lock(obj) { … }`
    - vstup do KS: Monitor.Enter(obj) před blokem / v bloku `try`
    - výstup z KS: Monitor.Exit(obj) v bloku `finally`
  - čekání a signalizace, jediná podmínková proměnná: Monitor.Wait() a Monitor.Pulse() či Monitor.PulseAll()
    - nečeká-li žádné vlákno, signál je ztracen
  - podmínková fronta nemá přednost před tou vstupní
    - do KS může vstoupit vlákno a změnit signalizovaný stav

# Monitor a C# – rozdíly ve verzích a možné problémy – C# 3.0

- implementace `lock(obj) { body; }`
  - C# 3.0

```
var temp = obj;
Monitor.Enter(temp);
// instrukce no-op + výjimka zde = neuvolněný zámek → DEADLOCK
try { body; }  // critical section
finally { Monitor.Exit(temp); }
```

  - korektnost závisí na (ne)optimalizaci kompilátoru
    - vloží-li kompilátor před blok try instrukci no-op, může během ní dojít k vyvolání výjimky (thread abort exception) a zámek zůstane zamčený
    - důsledek: **možný DEADLOCK**

# Monitor a C# – rozdíly ve verzích a možné problémy – C# 4.0

- implementace `lock(obj) { body; }`
  - C# 4.0

```
bool lockWasTaken = false;
var temp = obj;
try { Monitor.Enter(temp, ref lockWasTaken); body; }
finally { if (lockWasTaken) Monitor.Exit(temp); }
```

  - lepší řešení zámku?
    - nekonzistentní stav v body + výjimka (přerušení) → uvolnění zámku = zpřístupnění nekonzistentních dat!
    - ošetření výjimky KS – uvést data do konzistentního stavu
    - Lippert, Eric: *Locks and exceptions do not mix* [online]. 2009 [cit. 2015-11-25]. Odkaz: <Eric Lippert's Blog>.

# Producent / konzument – monitor

## Producent

```
repeat
  item = produce_item();
  PCmon.append(item);
forever
```

## Konzument

```
repeat
  PCmon.take(&item);
  consume_item(item);
forever
```

## Monitor: proměnné, init

```
monitor PCmon {
  item_t buffer[BUF_SIZE];
  cond not_full, not_empty;
  int in=0, out=0, count=0;
  void append(item_t item) {
    while (count == BUF_SIZE)
      cwait(not_full);
    buffer[in] = item;
    in = (in+1)%BUF_SIZE;
    count++;
    csignal(not_empty);
  }
  void take(item_t *item) {
    while (count == 0)
      cwait(not_empty);
    *item = buffer[out];
    out = (out+1)%BUF_SIZE;
    count--;
    csignal(not_full);
  }
}
```

# Producent / konzument – monitor – while vs. if

- stačilo by použití **`if`** místo **`while`**?
  - po probuzení signálem je monitor volný jak pro probuzené vlákno, tak i pro jiná vlákna ve vstupní frontě
  - záleží na implementaci přednosti podmínkové a vstupní fronty

```
void append(item_t item) {
  while (count == BUF_SIZE)
    cwait(not_full);
  buffer[in] = item;
  in = (in+1)%BUF_SIZE;
  count++;
  csignal(not_empty);
}
void take(item_t *item) {
  while (count == 0)
    cwait(not_empty);
  *item = buffer[out];
  out = (out+1)%BUF_SIZE;
  count--;
  csignal(not_full);
}
```

# Komunikace procesů – roura

- roura (pipe) – POSIX.1
  - jednosměrný komunikační nástroj pro dva procesy
  - FIFO
    - nezávislé ukazatele pozice pro čtení a zápis
    - pozice lze měnit pouze čtením/zápisem
    - co bylo přečteno se odstraní
  - jeden proces zapisuje, druhý čte
    - dokud není roura otevřena druhou stranou, operace na ní blokují

# Roura – vytvoření

- vytvoření roury – pipe(2)

```
#include <unistd.h>
int pipe(int filedes[2]);
```

  - vytvoří pár deskriptorů propojených rourou
    - jeden pro čtení: **`filedes[0]`**
    - jeden pro zápis: **`filedes[1]`**
  - vrací: 0 = úspěch, −1 = chyba, **`errno`** udává chybu

# Komunikace procesů – socket

- socket (někdy též soket) – POSIX.1
  - obousměrný komunikační nástroj pro dva procesy
  - domény (rodiny) socketů
    - UNIX domain – lokální
    - IPv4 / IPv6 – síťový nad IP
    - další síťové: IPX, X.25, AX.25, ATM, Appletalk, …
  - typy socketů:
    - STREAM – proudový spojovaný (nad IP odpovídá TCP)
    - DGRAM – datagramový nespojovaný (UDP)
    - a jiné: sekvenční (SEQPACKET), surový (RAW), …

# Socket – vytvoření páru propojených deskriptorů

- vytvoření páru deskriptorů – socketpair(3)

```
#include <sys/types.h>
#include <sys/socket.h>
int socketpair(int d, int type, int protocol,
  int sv[2]);
```

  - vytvoří pár deskriptorů **sv[]** propojených socketem
    - oba pro obousměrnou komunikaci: čtení i zápis
    - doména **d**: **AF_UNIX**, **AF_INET**, **AF_INET6** aj.
    - **type**: **SOCK_STREAM**, **SOCK_DGRAM** aj.
    - **protocol**: **0** znamená výchozí pro daný typ socketu
  - vrací: 0 = úspěch, −1 = chyba, **errno** udává chybu

# Socket – klient

- klientský proces a socket
  - překlad adresy – getaddrinfo(3) / gethostbyname(3)
  - alokace socketu – socket(2)
  - svázání socketu s lokálním portem – bind(2)
    - pouze volitelně, jinak OS přiřadí port automaticky
  - navázání spojení (proudový socket) – connect(2)
    - pro datagramový socket nastaví jen výchozí adresu cíle
  - komunikace – write(2) / send(2), read(2) / recv(2)
    - pro datagramový sendto(2) a recvfrom(2)
  - zavření – close(2)

# Socket – server

- serverový proces a socket
  - nastavení lokální adresy a portu – getaddrinfo(3)
  - alokace socketu – socket(2)
  - svázání socketu s lokálním portem – bind(2)
  - zahájení naslouchání – listen(2)
  - přijetí spojení (pouze proudový socket) – accept(2)
    - vytvoří nový socket pro komunikaci s klientem
  - komunikace – read(2) / recv(2), write(2) / send(2)
    - pro datagramový recvfrom(2) a sendto(2)
  - zavření – close(2)

# Prostředky komunikace procesů

- prostředky komunikace
  - soubor, databáze
    - pomalé, náhodný přístup, současný přístup je třeba řídit
  - roura
    - proudový přístup (FIFO), jednosměrná komunikace
  - socket
    - proudový přístup (FIFO), obousměrná síťová komunikace
  - fronty zpráv
    - exkluzivní přístup (FIFO), (jednosměrná) komunikace
  - sdílená paměť
    - nejrychlejší, náhodný přístup, současný přístup nutno řídit