B4B35OSY: Operační systémy

Lekce 4. Plánování a synchronizace

Petr Štěpán
stepan@fel.cvut.cz

19. října, 2022

Outline

Obsah

Jak to bude fungovat?

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

volatile int a;
void *fce(void *n) {
   int i;
   for (i=0; i<10000; i++) {
      a+=1;
   }
   printf("a=%i\n", a);
   pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
   pthread_t tid1,tid2;
   a=0;
   pthread_create(&tid1, NULL, fce, NULL);
   pthread_create(&tid2, NULL, fce, NULL);
   pthread_join(tid1, NULL);
   pthread_join(tid2, NULL);
   return 0;
}
```

Co program vytiskne?

A - Dvakrát a=10000
B - a=20000 a=20000
C - a=X a=20000 , kde X je menší než 20000
D - Něco náhodného

Problém synchronizace

- Souběžný přístup ke sdíleným datům může způsobit jejich nekonzistenci
  - nutná koordinace procesů
- Synchronizace běhu procesů
  - Čekání na událost vyvolanou jiným procesem
- Komunikace mezi procesy (IPC = Inter-process Communication) – příští přednáška
  - Výměna informací (zpráv)
  - Způsob synchronizace, koordinace různých aktivit
- Sdílení prostředků – problém soupeření či souběhu (race condition)
  - Procesy používají a modifikují sdílená data
  - Operace zápisu musí být vzájemně výlučné
  - Operace zápisu musí být vzájemně výlučné s operacemi čtení
  - Operace čtení (bez modifikace) mohou být realizovány souběžně
  - Pro zabezpečení integrity dat se používají kritické sekce

Producent konzument

Ilustrační příklad

- Producent generuje data do vyrovnávací paměti s konečnou kapacitou (bounded-buffer problem) a konzument z této paměti data odebírá
- Zavedeme celočíselnou proměnnou count, která bude čítat platné položky v bufferu. Na počátku je count = 0
- Pokud je v poli místo, producent vloží položku do pole a inkrementuje count
- Pokud je v poli nějaká položka, konzument při jejím vyjmutí dekrementuje count

Producent a konsument

Sdílená data

```
#define BUF_SIZE = 20
typedef struct { /* data */ } item;
item buffer[BUF_SIZE];
int count = 0;
```

Producent

```
void producer() {
    int in = 0;
    item nextProduced;
    while (1) {
        /* Vygeneruj novou položku do
           proměnné nextProduced */
        while (count == BUF_SIZE);
          /* čekání nedělej nic */
        buffer[in] = nextProduced;
        in = (in + 1) % BUF_SIZE;
        count++;
    }
}
```

Je tu nějaký problém?

A - Není
B - Problém je proměnná buffer
C - Problém je proměnná count
D - Problémem jsou buffer i count

Konzument

```
void consumer() {
    int out = 0;
    item nextConsumed;
    while (1) {
        while (count == 0);
          /* čekání nedělej nic */
        nextConsumed =  buffer[out];
        out = (out + 1) % BUF_SIZE;
        count--;
        /*  Zpracuj položku z
          proměnné nextConsumed */
    }
}
```

Problém soupeření

- count ++ bude obvykle implementováno:
  - P_1: **count** → registr `mov count, %eax`
  - P_2: registr+1 → regist `add 1, %eax`
  - P_3: registr → **count** `mov %eax, count`
- count - - bude obvykle implementováno:
  - K_1: **count** → registr `mov count, %eax`
  - K_2: registr-1 → registr `sub 1, %eax`
  - K_3: registr → **count** `mov %eax, count`

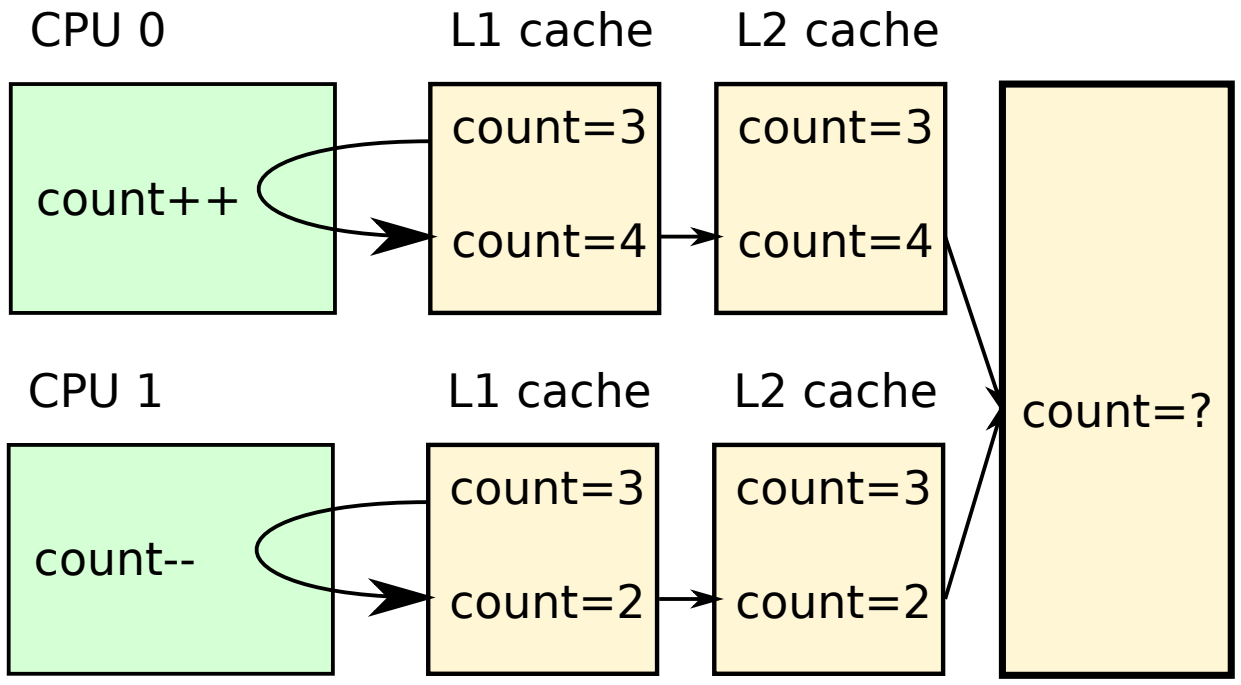
Může nastat následující paralelizace procesů konzument a producent:

akce	běží	akce	výsledek
P_1:	producent	**count** → registr	eax = 3
P_2:	producent	registr+1 → registr	eax = 4
K_1:	konzument	**count** → registr	eax = 3
K_2:	konzument	registr-1 → registr	eax = 2
K_3:	konzument	registr → **count**	count = 2
P_3:	producent	registr → **count**	count = 4

- Na konci může být count roven 2 nebo 4, ale správně je 3 (což se většinou podaří)
- Je to důsledkem nepředvídatelného prokládání procesů/vláken vlivem možné preempce

Problém soupeření – cache

- Problém soupeření je ještě složitější při vícejádrových procesorech
- Jedna proměnná je uložena na více místech cache úrovně L1, úrovně L2 a pouze jednom místě úrovně L3 a paměti RAM



- Výsledek zápisu je určen kdo přijde dříve a kdo později, ale pouze hodnota 2 nebo 4

Kritická sekce

- Problém lze formulovat obecně:
  - Jistý čas se proces zabývá svými obvyklými činnostmi a jistou část své aktivity věnuje sdíleným prostředkům.
  - Část kódu programu, kde se přistupuje ke sdílenému prostředku, se nazývá kritická sekce procesu vzhledem k tomuto sdílenému prostředku (nebo také sdružená s tímto prostředkem).
- Je potřeba zajistit, aby v kritické sekci sdružené s jistým prostředkem, se nacházel nejvýše jeden proces
  - Pokud se nám podaří zajistit, aby žádné dva procesy nebyly současně ve svých kritických sekcích sdružených s uvažovaným sdíleným prostředkem, pak je problém soupeření vyřešen.
- Modelové prostředí pro řešení problému kritické sekce
  - Předpokládá se, že každý z procesů běží nenulovou rychlostí
  - Řešení nesmí záviset na relativních rychlostech procesů

Požadavky na kritickou sekci

- Vzájemné vyloučení – podmínka bezpečnosti (Mutual Exclusion)
  - Pokud proces P_i je ve své kritické sekci, pak žádný další proces nesmí být ve své kritické sekci sdružené s týmž prostředkem
- Trvalost postupu – podmínka živosti (Progress)
  - Jestliže žádný proces neprovádí svoji kritickou sekci sdruženou s jistým zdrojem a existuje alespoň jeden proces, který si přeje vstoupit do kritické sekce sdružené se tímto zdrojem, pak výběr procesu, který do takové kritické sekce vstoupí, se nesmí odkládat nekonečně dlouho.
- Konečné čekání – podmínka spravedlivosti (Fairness)
  - Proces smí čekat na povolení vstupu do kritické sekce jen konečnou dobu.
  - Musí existovat omezení počtu, kolikrát může být povolen vstup do kritické sekce sdružené se jistým prostředkem jiným procesům než procesu požadujícímu vstup v době mezi vydáním žádosti a jejím uspokojením.

Řešení kritických sekcí

Základní struktura procesu s kritickou sekcí:

```
do {
    enter_cs();
       // critical section
    leave_cs ();
       // non-critical section
} while (TRUE);
```

Klíčem k řešení celého problému kritických sekcí je korektní implementace funkcí enter_cs() a leave_cs().

- Čistě softwarová řešení na aplikační úrovni
  - Algoritmy, jejichž správnost se nespoléhá na další podporu
  - Základní (a problematické) řešení s aktivním čekáním (busy waiting)
- Hardwarové řešení
  - Pomocí speciálních instrukcí CPU
  - Stále ještě s aktivním čekáním
- Softwarové řešení zprostředkované operačním systémem
  - Potřebné služby a datové struktury poskytuje OS (např. semafory)
  - Tím je umožněno pasivní čekání – proces nesoutěží o procesor
  - Podpora volání synchronizačních služeb v programovacích systémech/jazycích (např. monitory, zasílání zpráv)

Řešení na aplikační úrovni

Zaveďme proměnnou lock, jejíž hodnota určuje, zda je kritická sekce obsazená

```
while(TRUE) {
    while(lock!=0);
        /* čekej */
    lock = 1;
    critical_section();
    lock = 0;
    noncritical_section();
}
```

Je zde nějaký problém?

Je tu nějaký problém?

A - Není, bude to fungovat
B - Je problém se vzájemným vyloučením
C - Je problém s živostí, aby vlákno zbytečně nečekalo
D - Je problém se spravedlností, aby vlákna nepředbíhali

Řešení na aplikační úrovni

Zaveďme proměnnou lock, jejíž hodnota určuje, zda je kritická sekce obsazená

```
while(TRUE) {
    while(lock!=0);
        /* čekej */
    lock = 1;
    critical_section();
    lock = 0;
    noncritical_section();
}
```

Je zde nějaký problém?

Je tu nějaký problém?

A - Není, bude to fungovat
B - Je problém se vzájemným vyloučením
C - Je problém s živostí, aby vlákno zbytečně nečekalo
D - Je problém se spravedlností, aby vlákna nepředbíhali

Řešení (B) problém se vzájemným vyloučením

- **Je to úplně špatně!**
- Protože mezi otestováním proměnné lock a jejím nastavení je možné, že proběhne další otestování jiným vláknem.
- Neřeší tedy základní podmínku exkluzivity kritické sekce

Řešení na aplikační úrovni

Striktní střídání dvou procesů nebo vláken.

- Zaveďme proměnnou turn, jejíž hodnota určuje, který z procesů smí vstoupit do kritické sekce.
- Je-li `turn == 0`, do kritické sekce může P_0,
- je-li `turn == 1`, pak P_1.

P_0

```
while(TRUE) {
        while(turn!=0);
           /* čekej */
        critical_section();
        turn = 1;
        noncritical_section();
}
```

P_1

```
while(TRUE) {
        while(turn!=1);
           /* čekej */
        critical_section();
        turn = 0;
        noncritical_section();
}
```

Je zde nějaký problém?

A - Není, bude to fungovat
B - Je problém se vzájemným vyloučením
C - Je problém s živostí, aby vlákno zbytečně nečekalo
D - Je problém se spravedlností, aby vlákna nepředbíhali

Řešení na aplikační úrovni

Striktní střídání dvou procesů nebo vláken.

- Zaveďme proměnnou turn, jejíž hodnota určuje, který z procesů smí vstoupit do kritické sekce.
- Je-li `turn == 0`, do kritické sekce může P_0,
- je-li `turn == 1`, pak P_1.

P_0

```
while(TRUE) {
        while(turn!=0);
           /* čekej */
        critical_section();
        turn = 1;
        noncritical_section();
}
```

P_1

```
while(TRUE) {
        while(turn!=1);
           /* čekej */
        critical_section();
        turn = 0;
        noncritical_section();
}
```

Je zde nějaký problém?

- (C) je porušen požadavek Trvalosti postupu - podmínka živosti
- P_0 proběhne svojí kritickou sekcí velmi rychle, turn = 1 a oba procesy jsou v nekritických částech. P_0 je rychlý i ve své nekritické části a chce vstoupit do kritické sekce. Protože však turn == 1, bude čekat, přestože kritická sekce je volná.
- Navíc řešení nepřípustně závisí na rychlostech procesů

Petersonovo řešení

- Petersonovo řešení střídání dvou procesů nebo vláken
- Řešení pro dva procesy P_i (i = 0, 1) – dvě globální proměnné:
  - `int turn;`
    - Proměnná turn udává, který z procesů je na řadě při přístupu do kritické sekce
  - `boolean interest[2];`
    - V poli interest procesy indikují svůj zájem vstoupit do kritické sekce; (interest[i]==TRUE) znamená, že P_i tuto potřebu má
    - Prvky pole interest nejsou sdílenými proměnnými.

```
j = 1 - i;
interest[i] = TRUE;
turn = j;
while (interest[j] && turn == j) ;        /* čekání */
/*  KRITICKÁ SEKCE                    */
interest[i] = FALSE;
/*  NEKRITICKÁ ČÁST PROCESU           */
```

- Náš proces bude čekat jen pokud druhý proces je na řadě a současně má zájem do kritické sekce vstoupit
- Všechna řešení na aplikační úrovni obsahují aktivní čekání, nebo používají funkci sleep/usleep

Petersonovo řešení

```
int a;
volatile int turn;
volatile int interest[2];

void *fce(void *n) {
  int i;
  int j=*(int*)n;
  for (i=0; i<1000000; i++) {
    interest[j]=1;
    __sync_synchronize();     /* memory barrier */
    turn = (1-j);
    while (interest[1-j]==1 && turn==(1-j));  /* repeat and wait */
    a+=1;
    interest[j]=0;
  }
  printf("a=%i\n", a);
  pthread_exit(NULL);
}
```

Memory barrier

- Většina moderních CPU umí měnit pořadí dvou po sobě jdoucích instrukcí kvůli zrychlení přístupu do paměti.
- Pro Petersonovo řešení je pořadí zápisu do proměnných `turn` a `interest` klíčové
- `__sync_synchronize` memory barrier pro překladač gcc (visual studio má funkci `MemoryBarrier`)
- memory barrier umožní i synchronizaci cache pamětí

```
interest[i] = TRUE;
__sync_synchronize(); /* memory barrier */
turn = j;
while (interest[j] && turn == j) ;        /* čekání */
```

- Nyní je všechno v pořádku a řešení funguje

Petersonovo řešení

Obecné řešení pro N procesů

- je již daleko více komplikovanější, tím je náchylnější k implementační chybě
- proměnné `level` charakterizují, kdo čeká na kritickou sekci
- proces, který dospěje až do nejvyšší úrovně (level), tak získá kritickou sekci

```
int level[N]
int last_to_enter[N-1]
for (l=0; l<N-1; l++)
  level[i] = l
  last_to_enter[l] = i
  while (last_to_enter[l] == i and exists k != i; level[k] >= l)
    wait;
```

- kontrukce exist k je zkratka za

```
set = False
for (k=0; k<N; k++) {
  if (k!=i && level[k] >= l)
    set = True;
```

HW podpora

- Využití zamykací proměnné je rozumné, avšak je nutná atomicita
- Jednoprocesorové systémy mohou vypnout přerušení, při vypnutém přerušení nemůže dojít k preempci
  - Nelze použít na aplikační úrovni (vypnutí přerušení je privilegovaná akce)
  - Nelze jednoduše použít pro víceprocesorové systémy
- Moderní systémy nabízejí speciální nedělitelné (atomické) instrukce
  - Tyto instrukce mezi paměťovými cykly „nepustí“ sběrnici pro jiný procesor
  - Instrukce TestAndSet atomicky přečte obsah adresované buňky a bezprostředně poté změní její obsah (tas – MC68k, tsl – Intel)
  - Instrukce Swap (xchg) atomicky prohodí obsah registru procesoru a adresované buňky
  - Např. IA32/64 (I586+) nabízí i další atomické instrukce
  - Prefix „LOCK“ pro celou řadu instrukcí typu read-modify-write (např. ADD, AND, … s cílovým operandem v paměti)

HW podpora

- tas např. Motorola 68000

```
enter_cs:   tas  lock          ; nastav lock na 1 a otestuj starou hodnotu
            jnz enter_cs       ; byla stará hodnota nenulová?
            ret

leave_cs:   mov $0, lock       ; vynuluj lock pro uvolnění kritické sekce
            ret
```

- xchg – IA32

```
enter_cs:   mov $1, %eax       ; připrav hodnotu 1 pro výměnu
            xchg  lock, %eax   ; eax obsahuje nyní starou hodnotu
            jnz enter_cs       ; byla stará hodnota nenulová
            ret

leave_cs:   mov $0, lock       ; vynuluj lock pro uvolnění kritické sekce
            ret
```

Xchg řešení

```
volatile int a;
volatile int turn;
void *fce(void *n) {
  int i, tmp;
  for (i=0; i<1000000; i++) {
    tmp=1;
    asm volatile ("xchg%z0  %2, %0;"
     : "=g" (turn), "=r" (tmp): "1" (tmp) : );
    while (tmp!=0) {
      asm volatile ("xchg%z0  %2, %0;"
       : "=g" (turn), "=r" (tmp): "1" (tmp) : );
    }
    a+=1;
    turn=0;
  }
  printf("a=%i\n", a);
  pthread_exit(NULL);
}
```

Je zde nějaký problém?

A - Není, bude to fungovat
B - Je problém se vzájemným vyloučením
C - Je problém s živostí, aby vlákno zbytečně nečekalo
D - Je problém se spravedlností, aby vlákna nepředbíhali

Semafor

- Obecný synchronizační nástroj (Edsger Dijkstra, NL, [1930–2002])
- Systémem spravovaný objekt
- Základní vlastností je celočíselná proměnná (obecný semafor, nebo také čítající semafor)
- Dvě standardní atomické operace nad semaforem
  - sem_wait(&S) [někdy nazývaná lock(), acquire() nebo down()]
  - sem_post(&S) [někdy nazývaná unlock(), release() nebo up()]

```
sem_wait(S) {
    while (S <= 0);
    S--;
}
```

```
sem_post(S) {
    S++;
    // Čeká-li jiný proces před
    // semaforem, pusť ho dál
}
```

- Tato sémantika stále obsahuje aktivní čekání
- Skutečná implementace však aktivní čekání obchází tím, že spolupracuje s plánovačem CPU, což umožňuje blokovat a reaktivovat procesy (vlákna)

Implementace semaforů

Struktura semaforu

```
typedef struct {
    int value;                  // „Hodnota" semaforu
    struct process *list;               // Fronta procesů stojících „před semaforem"
} sem_t;
```

Operace nad semaforem jsou pak implementovány jako nedělitelné s touto sémantikou

```
void sem_wait(sem_t *S) {
    S->value= S->value - 1;
    if (S->value < 0)           // Je-li třeba, zablokuj volající proces a zařaď ho
        block(S->list);         //   do fronty před semaforem (S.list)
}

void sem_post(sem_t *S) {
    S->value= S->value + 1
    if (S->value <= 0) {
        if (S->list != NULL) {      // Je-li fronta neprázdná
            //   vyjmi proces P z čela fronty
            wakeup(P);          //    a probuď P
        }
    }
}
```

Implementace semaforů

- Záporná hodnota S.value udává, kolik procesů „stojí“ před semaforem
- Fronty před semaforem:
  - Většinou FIFO bez uvažování priorit procesů, jinak vzniká problém se stárnutím
  - Systémy reálného času (RTOS) většinou prioritu uvažují
- Operace wait(S) a post(S) musí být vykonány atomicky
- OS na jednom procesoru nemá problém, OS rozhoduje o přepnutí procesu
- OS na více jádrech:
  - Jádro musí používat atomické instrukce či jiný odpovídající hardwarový mechanismus na synchronizaci skutečného paralelizmu
  - Instrukce xchg, tas, či prefix lock musí umět zamknout sběrnici proti přístupu jiných jader, či zamknout a aktualizovat cache systémem cache snooping

Mutex

- Mutex – speciální rychlejší semafor, hodnoty pouze 0,1 – binární semafor
- Implementace musí zaručit:
- Operace lock() (odpovídá funkci wait() u semaforu) a unlock() (odpovídá funkci post()) musí být atomické stejně jako u semaforů
- Aktivní čekání není plně eliminováno, je ale přesunuto z aplikační úrovně (kde mohou být kritické sekce dlouhé) do úrovně jádra OS pro implementaci atomicity operací se semafory
- Mutex definuje koncept “vlastníka mutexu” a díky tomu jej lze například zamykat rekurzivně z jednoho vlákna nebo lze implementovat mechanismus pro zabránění uváznutí.

Užití:

```
void *fce(void *n) {
  int i;
  for (i=0; i<100000; i++) {
    pthread_mutex_lock(&mutex);
    a+=1;
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(NULL);
}
```

Synchronizace bez aktivního čekání

- Aktivní čekání mrhá strojovým časem
- Může způsobit i nefunkčnost při rozdílných prioritách procesů
- Např. vysokoprioritní producent zaplní pole, začne aktivně čekat a nedovolí konzumentovi odebrat položku (samozřejmě to závisí na metodě plánování procesů a na to navazující dynamicky se měnící priority)
- Blokování pomocí systémových atomických primitiv
  - suspend() místo aktivního čekání – proces se zablokuje
  - wakeup(process) probuzení spolupracujícího procesu při opouštění kritické sekce

Synchronizace bez aktivního čekání

```
void producer() {
  while (1) {
    pthread_mutex_lock(&mutex);
    if (count == BUFFER_SIZE) {
      pthread_mutex_unlock(&mutex);
      suspend();    // Je-li pole plné, zablokuj se
      pthread_mutex_lock(&mutex);
    }
    buffer[in]=nextProduced; in=(in+1)%BUFFER_SIZE;
    count++ ;
    pthread_mutex_unlock(&mutex);
    if  (count == 1)
      wakeup(consumer); // probuď konzumenta
} }
void consumer() {
  while (1) {
    pthread_mutex_lock(&mutex);
    if (count == 0) {
      pthread_mutex_unlock(&mutex);
      suspend(); // Je-li pole prázdné, zablokuj se
      pthread_mutex_lock(&mutex);
    }
    nextConsumed= buffer[out]; out=(out+1)%BUFFER_SIZE;
    count-- ;
    pthread_mutex_unlock(&mutex);
    if (count == BUFFER_SIZE-1) // probuď producenta
       wakeup(producer);
} }
```

Je možné vyřešit konzumenta a producenta bez aktivního čekání pouze s mutexy? Je uvedené řešení funkční?

A - Není, bude to fungovat
B - Je problém s kritickými sekcemi
C - Je problém s přístupem k bufferu
D - Program se může zaseknout, obě vlákna budou suspendována

Problém s čekáním

- Z minulého kvízu platí D
- Předešlý kód není řešením – zůstalo konkurenční soupeření mezi přístupem ke count a uspáním:
  - Konzument přečetl count == 0, zavolá unlock a než zavolá suspend(), je mu odňat procesor
  - Producent získá mutex, vloží do pole položku, zvýší count == 1, načež se pokusí se probudit konzumenta, který ale ještě nespí!
  - Po znovuspuštění se konzument domnívá, že pole je prázdné a volá suspend()
  - Po čase producent zaplní pole a rovněž zavolá suspend() – spí oba!
  - Příčinou této situace je ztráta budícího signálu
- Lepší řešení:
  - Jedině OS umí uspat a vzbudit procesy – semafory, nebo mutexy a podmínkové proměnné

Producent – konzument

Tři semafory

- **mutex** s iniciální hodnotou 1 – pro vzájemné vyloučení při přístupu do sdílené paměti
- **used** – počet položek v poli – inicializován na hodnotu 0
- **free** – počet volných položek – inicializován na hodnotu BUF_SIZE

```
void producer() {
    while (1) {     /* Vygeneruj položku do proměnné nextProduced */
        sem_wait(&free);
        sem_wait(&mutex);
        buffer [in] = nextProduced;  in = (in + 1) % BUF_SZ;
        sem_post(&mutex);
        sem_post(&used);
    }
}
void consumer() {
    while (1) {
        sem_wait(&used);
        sem_wait(&mutex);
        nextConsumed =  buffer[out];  out = (out + 1) % BUF_SZ;
        sem_post(&mutex);
        sem_post(&free);
        /*  Zpracuj položku z proměnné nextConsumed */
    }
}
```

Producent – konzument jen s mutexy

Pro korektní uspání potřebujeme **podmínkové proměnné**

- čekání na podmínku se provádí:
  - funkcí `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
  - časově omezené čekání funkcí `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);`
- při čekání na podmínku se spojí podmínka s mutexem, který se čekáním uvolní
- uspání vlákna a uvolnění mutexu se děje uvnitř jádra OS jako nedělitelná dvojice operací

Producent – konzument jen s mutexy

- probuzení vlákna čekajícího na podmínku se provede:
  - funkcí `int pthread_cond_signal(pthread_cond_t *cond);`
  - probuzení všech vláken čekajících na konkrétní podmínku se provede funkcí `int pthread_cond_broadcast(pthread_cond_t *cond);`
- při probuzení vlákna, čekajícího na podmínku se opět mutex obsadí
  - tedy při probuzení všech vláken, každé vlákno čeká na možnost uzamknout si mutex pro sebe
- čekání na podmínku musí být vždy uvnitř kritické sekce mutexu
- probuzení cizího vlákna nastavením podmínky, by mělo být také uvnitř kritické sekce mutexu, abychom zabránili možnosti souběhu.

Podmínkové proměnné - simulace semaforů

```
void sem_init(struct sem *s, int val) {
  s->cnt = val;
  pthread_mutex_init(&s->mutex, NULL);
  pthread_cond_init(&s->cond, NULL);
}

void sem_wait(struct sem *s) {
  pthread_mutex_lock( &s->mutex );
  while (s->cnt<=0) {
      pthread_cond_wait(&s->cond, &s->mutex);
  }
  s->cnt--;
  pthread_mutex_unlock( &s->mutex );
}

void sem_post(struct sem *s) {
  pthread_mutex_lock( &s->mutex );
  s->cnt++;
  pthread_cond_signal( &s->cond );
  pthread_mutex_unlock( &s->mutex );
}
```

```
void *prod(void *n) {
  int i;
  for (i=0; i<30; i++) {
    sem_wait(&empty);
    pthread_mutex_lock(&buf_mutex);
    buf[wr_ptr]=i;
    wr_ptr++; wr_ptr%=10;
    pthread_mutex_unlock(&buf_mutex);
    sem_post(&full);
  }
  return NULL;
}

void *cons(void *n) {
  int i;
  for (i=0; i<30; i++) {
    sem_wait(&full);
    pthread_mutex_lock(&buf_mutex);
    rd_ptr++; rd_ptr%=10;
    pthread_mutex_unlock(&buf_mutex);
    sem_post(&empty);
  }
  return NULL;
}
```

Podmínkové proměnné

```
void *producer(void *i) {
   char str[256], *s;
   int wr_ptr=0;

   while ((s=fgets(str, 250, stdin))!=NULL) {
     pthread_mutex_lock(&mutex);
     while (glob_free<=0) {
       printf("Wait full\n");
       pthread_cond_wait(&full, &mutex);
     }
     memcpy(global[wr_ptr], s, 256);
     glob_free--;
     pthread_cond_signal(&empty);
     pthread_mutex_unlock(&mutex);

     wr_ptr=(wr_ptr+1)%8;
  }
  glob_end=1;
  pthread_mutex_lock(&mutex);
  pthread_cond_signal(&empty);
  pthread_mutex_unlock(&mutex);
  return NULL;
}
```

```
void *consumer(void *i) {
  int rd_ptr=0;
  while (!glob_end) {
    pthread_mutex_lock(&mutex);
    while(glob_free>=8 && !glob_end) {
      printf("Wait empty\n");
      pthread_cond_wait(&empty, &mutex);
    }
    if (glob_free<8) {
      printf("Zadano: %s\n", global[rd_ptr]);
      glob_free++;
    }
    pthread_cond_signal(&full);
    pthread_mutex_unlock(&mutex);
    rd_ptr=(rd_ptr+1)%8;
    sleep(1);
  }
  return NULL;
}
```

Čtenáři a písaři

- Úloha: Několik procesů přistupuje ke společným datům
- Některé procesy data jen čtou – čtenáři
- Jiné procesy potřebují data zapisovat – písaři
- Souběžné operace čtení mohou čtenou strukturu sdílet – libovolný počet čtenářů může jeden a tentýž zdroj číst současně
- Operace zápisu musí být exklusivní, vzájemně vyloučená s jakoukoli jinou operací (zápisovou i čtecí)
  - v jednom okamžiku smí daný zdroj modifikovat nejvýše jeden písař
  - Jestliže písař modifikuje zdroj, nesmí ho současně číst žádný čtenář
- Dva možné přístupy
  - Přednost čtenářů
    - Žádný čtenář nebude muset čekat, pokud sdílený zdroj nebude obsazen písařem. Jinak řečeno: Kterýkoliv čtenář čeká pouze na opuštění kritické sekce písařem.
    - Písaři mohou stárnout
  - Přednost písařů
    - Jakmile je některý písař připraven vstoupit do kritické sekce, čeká jen na její uvolnění (čtenářem nebo písařem). Jinak řečeno: Připravený písař předbíhá všechny připravené čtenáře.
    - Čtenáři mohou stárnout

Priorita čtenářů

- Sdílená data
  - semaphore wrt, readcountmutex;
  - int readcount
- Inicializace
  - wrt = 1; readcountmutex = 1; readcount = 0;

Čtenář:

```
sem_wait(readcountmutex);
readcount++;
if (readcount==1) sem_wait(wrt);
sem_post(readcountmutex);

   // čtení sdíleného zdroje

sem_wait(readcountmutex);
readcount--;
if (readcount==0) sem_post(wrt);
sem_post(readcountmutex);
```

Písař:

```
sem_wait(wrt);
   // písař modifikuje zdroj
sem_post(wrt);
```

Priorita písařů

- Sdílená data
  - semaphore wrt, rdr, readcountmutex, writecountmutex;
  - int readcount, writecount;
- Inicializace
  - wrt = 1; rdr = 1; readcountmutex = 1; writecountmutex = 1;
  - readcount = 0; writecount = 0;

Písař:

```
sem_wait(writecountmutex);
writecount++;
if (writecount==1) sem_wait(rdr);
sem_post(writecountmutex);
sem_wait(wrt);
   // písař modifikuje zdroj
sem_post(wrt);
sem_wait(writecountmutex);
writecount--;
if (writecount==0) sem_post(rdr);
sem_post(writecountmutex);
```

Čtenář:

```
sem_wait(rdr);
sem_wait(readcountmutex);
readcount++;
if (readcount == 1) sem_wait(wrt);
sem_post(readcountmutex);
sem_post(rdr);
   // čtení sdíleného zdroje
sem_wait(readcountmutex);
readcount--;
if (readcount == 0) sem_post(wrt);
sem_post(readcountmutex);
```

Monitor

- Monitor je synchronizační nástroj vyšší úrovně
- Umožňuje bezpečné sdílení libovolného datového typu
- Na rozdíl od semaforů, monitor explicitně definuje která data jsou daným monitorem chráněna
- Monitor je jazykový konstrukt v jazycích „pro paralelní zpracování“
- Podporován např. v Concurrent Pascal, Modula-3, C#, Java, ...
- V Javě může každý objekt fungovat jako monitor (viz metoda Object.wait() a klíčové slovo synchronized)
- Procedury definované jako monitorové procedury se vždy vzájemně vylučují

```
monitor monitor_name {
    int i;                      //  Deklarace sdílených proměnných
    void p1(...) { ... }        //  Deklarace monitorových procedur
    void p2(...) { ... }
    {
        // inicializační kód
    }
}
```

Synchronizace v Javě

- Java používá pro synchronizaci Monitor
- Uživatel si může nadefinovat semafor následovně:

```
public class CountingSemaphore {
  private int signals = 1;

  public synchronized void sem_wait() throws InterruptedException{
    while(this.signals <= 0) wait();
    this.signals--;
  }

  public synchronized void sem_post() {
    this.signals++;
    this.notify();
  }
}
```

Případně lze použít i efektivnější java.util.concurrent.Semaphore

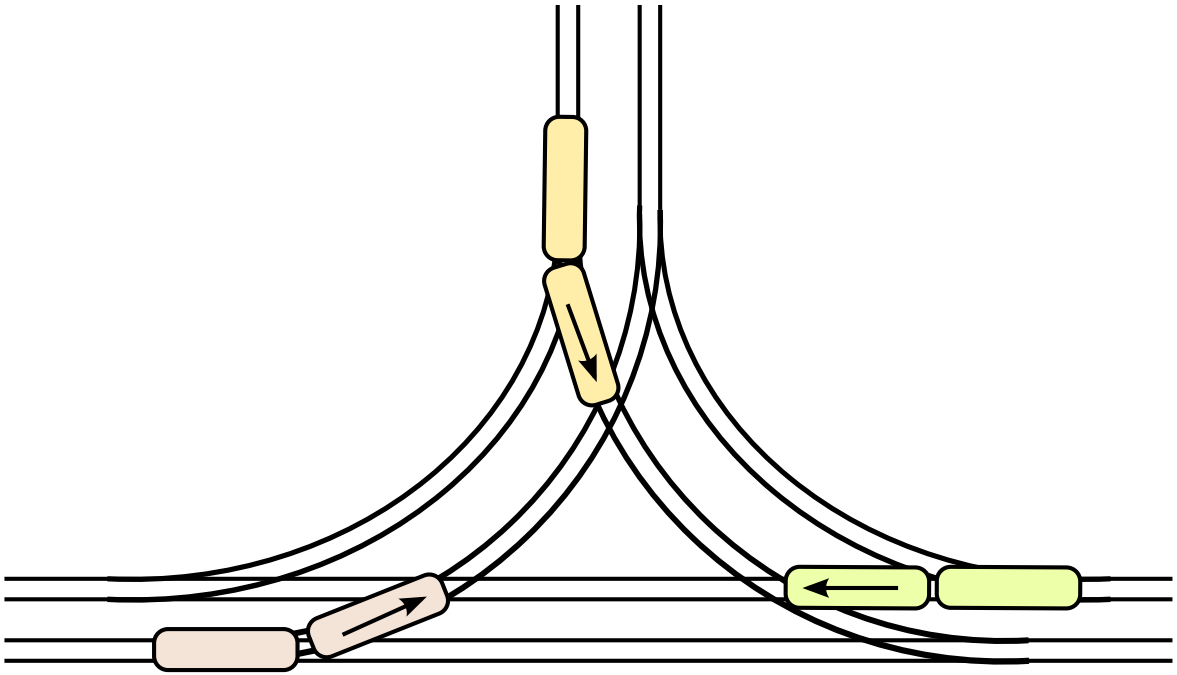
Spin-lock

- Spin-lock je obecný (čítající) semafor, který používá aktivní čekání místo blokování
- Blokování a přepínání mezi procesy či vlákny by bylo časově mnohem náročnější než ztráta strojového času spojená s krátkodobým aktivním čekáním
- Používá se ve víceprocesorových systémech pro implementaci krátkých kritických sekcí
- Typicky uvnitř jádra
- Např. při obsluze přerušení, kde není možné blokování (přerušení není součástí žádného procesu, jedná se o hardwarový koncept)
- Další použití je pro krátké kritické sekce, např. zajištění atomicity operací se semafory (ale to se většinou řeší efektivnějšími atomickými instrukcemi)
- Užito např. v multiprocesorových Windows 2k/XP/7 i Linuxu

Obsah

Deadlock v životě

- Karlovo náměstí – tramvaje většinou jedou jen dopředu
- Zdroje jsou křížení tramvajových kolejí
- Aby tramvaj projela, musí použít dva zdroje na své cestě

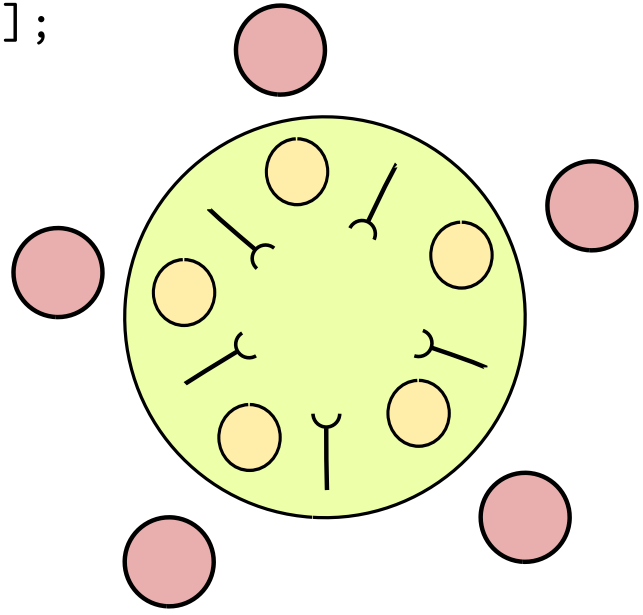


- “It takes money to make money” – anglické přísloví
- K získání kvalitního zaměstnání je potřeba kvalitní praxe, kvalitní praxi získáte pouze v kvalitním zaměstnání

Večeřící filozofové

Sdílená data

```
/* Inicializace */
  semaphore chopStick[ ] = new Semaphore[5];
  for(i=0; i<5; i++) chopStick[i] = 1;
/* Implementace filozofa i: */
do  {
    chopStick[i].wait;
    chopStick[(i+1) % 5].wait;
    eating();          // Teď jí
    chopStick[i].signal;
    chopStick[(i+1) % 5].signal;
    thinking();        //  A teď přemýšlí
} while (TRUE) ;
```



- Možné ochrany proti uváznutí
  - Zrušení symetrie úlohy
  - Jeden filozof bude levák a ostatní praváci (levák zvedá vidličky opačně)
  - Jídlo se n filozofům podává v jídelně s n+1 židlemi
  - Filozof smí uchopit vidličku jen, když jsou obě volné a uchopí obě najednou
  - Příklad obecnějšího řešení – tzv. skupinového zamykání prostředků

Časově závislé chyby

- Příklad časově závislé chyby
- Vlákna V_1 a V_2 spolupracují za použití mutexů A a B

Vlákno V_1

```
pthread_mutex_lock(&mutex_A);
pthread_mutex_lock(&mutex_B);
a+=b;
pthread_mutex_unlock(&mutex_B);
pthread_mutex_unlock(&mutex_A);
```

Vlákno V_2

```
pthread_mutex_lock(&mutex_B);
pthread_mutex_lock(&mutex_A);
b+=a;
pthread_mutex_unlock(&mutex_A);
pthread_mutex_unlock(&mutex_B);
```

Co se tedy stane?

A - Program vždy skončí v deadlocku, nikdy nedoběhne
B - Program někdy skončí, někdy nedoběhne
C - Program vždy skončí
D - Nelze určit podle uvedené části

Časově závislé chyby

- Správná odpověď je (B).
- Deadlock nastane pouze v situaci, že vlákno V_1 získá mutex A a vlákno V_2 paralelně získá mutex B dřív než V_1.
- Pokud tato speciální situace nenastane, tak program v pořádku skončí.
- Nebezpečnost takových chyb je v tom, že v praxi vznikají jen zřídkakdy za náhodné souhry okolností.
- Jsou tudíž fakticky neodladitelné, program se zasekne a není jasné z jakého důvodu.

Coffmanovy podmínky

Coffman formuloval čtyři podmínky, které musí platit současně, aby uváznutí mohlo vzniknout

1. Vzájemné vyloučení, Mutual Exclusion
   - sdílený zdroj může v jednom okamžiku používat nejvýše jeden proces
2. Postupné uplatňování požadavků, Hold and Wait
   - proces vlastnící alespoň jeden zdroj potřebuje další, ale ten je vlastněn jiným procesem, v důsledku čehož bude čekat na jeho uvolnění
3. Nepřipouští se odnímání zdrojů, No preemption
   - zdroj může uvolnit pouze proces, který ho vlastní, a to dobrovolně, když již zdroj nepotřebuje
4. Zacyklení požadavků, Circular wait
   - Existuje posloupnost čekajících procesů $P_0, P_1, \ldots, P_k, P_0$ takových, že P_0 čeká na uvolnění zdroje drženého P_1, P_1 čeká na uvolnění zdroje drženého $P_2, \ldots, P_{k-1}$ čeká na uvolnění zdroje drženého P_k, a P_k čeká na uvolnění zdroje drženého P_0.
   - V případě jednoinstančních zdrojů splnění této podmínky značí, že k uváznutí již došlo.

Co dělat?

Existují čtyři přístupy

- Zcela ignorovat hrozbu uváznutí
  - Pštrosí algoritmus – strč hlavu do písku a předstírej, že se nic neděje
  - Používá většina současných OS včetně většiny implementací UNIXů
  - Linux se snaží o prevenci deadlocku uvnitř jádra, neovlivňuje ale použití deadlocků v uživatelských programech
- Prevence uváznutí
  - Pokusit se přijmout taková opatření, aby při splnění opatření bylo uváznutí nemožné
  - Ale pozor! Pokud pravidlo nedodržíme, může k deadlocku dojít.
- Vyhýbání se uváznutí
  - Zajistit, že k uváznutí nikdy nedojde
  - Prostředek se nepřidělí, pokud by hrozilo uváznutí
  - hrozí stárnutí
- Detekce uváznutí a následná obnova
  - Uváznutí se připustí, detekuje se jeho vznik a zajistí se obnova stavu před uváznutím

Prevence uváznutí

- Narušení některé Coffmanovy podmínky
  - Eliminace potřeby vzájemného vyloučení
    - Nepoužívat sdílené zdroje, virtualizace (spooling) periferií
    - Mnoho činností však sdílení nezbytně potřebuje ke své funkci
  - Eliminace postupného uplatňování požadavků
    - Proces, který požaduje nějaký zdroj, nesmí dosud žádný zdroj vlastnit
    - Všechny prostředky, které bude kdy potřebovat, musí získat naráz
    - Nová specifikace mutexů v C++

Prevence uváznutí

- Narušení některé Coffmanovy podmínky
  - Připustit násilné odnímání přidělených zdrojů (preempce zdrojů)
    - Procesu žádajícímu o další zdroj je dosud vlastněný prostředek odňat
    - To může být velmi riskantní – zdroj byl již modifikován
    - Proces je reaktivován, až když jsou všechny potřebné prostředky volné
    - Metoda inkrementálního zjišťování požadavků na zdroje – nízká průchodnost
    - Metody prevence uváznutí aplikované za běhu způsobí výrazný pokles průchodnosti systému
  - Zabránit cyklu zdrojů
    - Vzniku cyklu se brání tak, že zdroje jsou očíslovány a procesy je smějí alokovat pouze ve vzrůstajícím pořadí čísel zdrojů
    - Nerealistické – zdroje vznikají a zanikají dynamicky
    - Často ale stačí uvažovat třídy zdrojů (LOCKDEP v jádře Linuxu – podobné jako alg. vyhýbání se uváznutí dále)

Vyhýbání se uváznutí

- Základní problém:
  - Systém musí mít dostatečné apriorní informace o požadavcích procesů na zdroje
- Nejčastěji se požaduje, aby každý proces udal maxima počtu prostředků každého typu, které bude za svého běhu požadovat
- Algoritmus:
  - Dynamicky se zjišťuje, zda stav subsystému přidělování zdrojů zaručuje, že se procesy v žádném případě nedostanou do cyklu
  - Stav systému přidělování zdrojů je popsán
    - Počtem dostupných a přidělených zdrojů každého typu a
    - Maximem očekávaných žádostí procesů
  - Stav může být bezpečný nebo nebezpečný

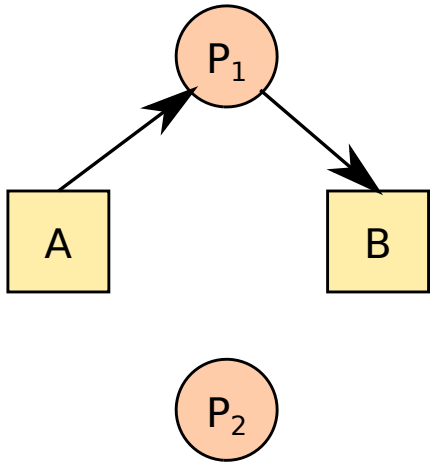
Vyhýbání se uváznutí

- Systém je v bezpečném stavu, existuje-li „bezpečná posloupnost procesů“
  - Posloupnost procesů P_0, P_1, ... , P_n je bezpečná, pokud požadavky každého P_i lze uspokojit právě volnými zdroji a zdroji vlastněnými všemi P_k , $k < i$
    - Pokud nejsou zdroje požadované procesem P_i volné, pak P_i bude čekat dokud se všechny P_k nedokončí a nevrátí přidělené zdroje
    - Když P_{i-1} skončí, jeho zdroje může získat P_i, proběhnout a jím vrácené zdroje může získat P_{i+1}, atd.
- Je-li systém v bezpečném stavu (safe state) k uváznutí nemůže dojít. Ve stavu, který není bezpečný (unsafe state), přechod do uváznutí hrozí
- Vyhýbání se uváznutí znamená:
  - Plánovat procesy tak, aby systém byl stále v bezpečném stavu
  - Nespouštět procesy, které by systém z bezpečného stavu mohly vyvést
  - Nedopustit potenciálně nebezpečné přidělení prostředku

Model uváznutí

RAG – Resource allocation graph

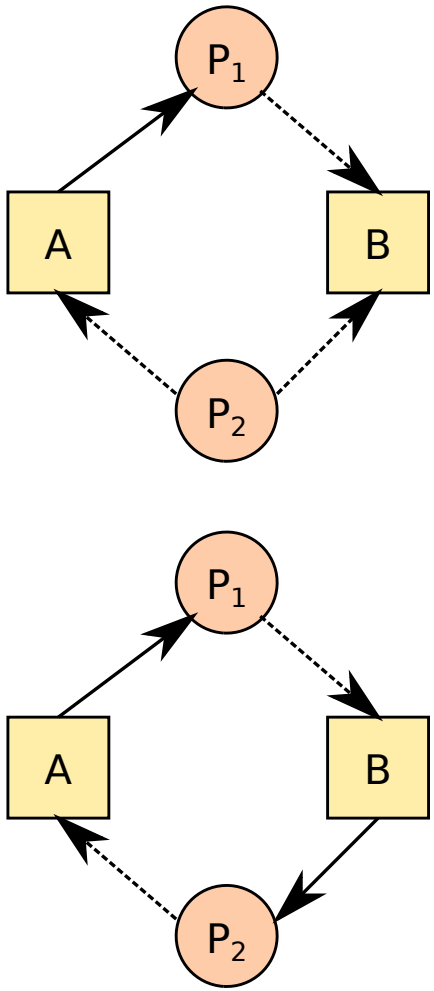
- Typy prostředků (zdrojů) R_1, R_2, ... R_m
  - např. datové struktury, V/V zařízení, ...
- Každý typ prostředku R_i má W_i instancí
  - např. máme 4 síťové karty a 2 CD mechaniky
  - často W_i = 1 tzv. jednoinstanční prostředky – např. mutex
- Každý proces používá potřebné zdroje podle schématu
  - žádost – request, wait
  - používání prostředku po konečnou dobu (kritická sekce)
  - uvolnění (navrácení) – release, signal
- žádost o zdroj značí hrana od procesu P_i ke zdroji R_j
- přidělený zdroj značí hrana od zdroje R_j k procesu P_i



Vyhýbání uváznutí – jednoinstanční zdroje

Potřebujeme znát budoucnost:

- Do RAG se zavede „nároková hrana“
  - Nároková hrana $P_i \rightarrow R_j$ značí, že někdy v budoucnu bude proces P_i požadovat zdroj R_j
  - V RAG hrana vede stejným směrem jako požadavek na přidělení, avšak kreslí se čárkovaně
- Nároková hrana se v okamžiku vzniku žádosti o přidělení převede na požadavkovou hranu
  - Když proces zdroj získá, požadavková hrana se změní na hranu přidělení
  - Když proces zdroj vrátí, hrana přidělení se změní na požadavkovou hranu
- Převod požadavkové hrany v hranu přidělení nesmí v RAG vytvořit cyklus (včetně uvažování nárokových hran)
  - LOCKDEP v Linuxu (systém běží cca $10\times$ pomaleji)



Bankéřský algoritmus

- Chování odpovědného bankéře:
  - Klienti žádají o půjčky do určitého limitu
  - Bankéř ví, že ne všichni klienti budou svůj limit čerpat současně a že bude půjčovat klientům prostředky postupně
  - Všichni klienti v jistém okamžiku svého limitu dosáhnou, avšak nikoliv současně
  - Po dosažení přislíbeného limitu klient svůj dluh v konečném čase vrátí
  - Příklad:
    - Ačkoliv bankéř ví, že všichni klienti budou dohromady potřebovat 22 jednotek a na celou transakci má jen 10 jednotek, je možné uspokojit postupně všechny klienty

Bankéřský algoritmus

- Zákazníci přicházející do banky pro úvěr předem deklarují maximální výši, kterou si budou kdy chtít půjčit
- Úvěry v konečném čase splácí
- Bankéř úvěr neposkytne, pokud si není jist, že bude moci uspokojit všechny zákazníky (vždy alespoň jednomu zákazníku bude moci půjčit všechny peníze a zákazník je pak vrátí)
- Analogie
  - Zákazník = proces
  - Úvěr = přidělovaný prostředek
- Vlastnosti
  - Procesy musí deklarovat své potřeby předem
  - Proces požadující přidělení může být zablokován
  - Proces vrátí všechny přidělené zdroje v konečném čase
- Nikdy nedojde k uváznutí
  - Proces bude spuštěn, jen pokud bude možno uspokojit všechny jeho požadavky
  - Sub-optimální pesimistická strategie
  - Předpokládá se nejhorší případ

Bankéřský algoritmus

Datové struktury

- n ... počet procesů
- m ... počet typů zdrojů
- Vektor $available[m]$
  - $available[j] = k$ značí, že je k instancí zdroje typu R_j je volných
- Matice $max[n, m]$
  - povinná deklarace procesů maximálních požadavků
  - $max[i, j] = k$ znamená, že proces P_i bude během své činnosti požadovat až k instancí zdroje typu R_j
- Matice $allocated[n, m]$
  - $allocated[i, j] = k$ značí, že v daném okamžiku má proces P_i přiděleno k instancí zdroje typu R_j
- Matice $needed[n, m]$ ($needed[i, j] = max[i, j] - allocated[i, j]$)
  - $needed[i, j] = k$ říká, že v daném okamžiku procesu P_i chybí ještě k instancí zdroje typu R_j

Bankéřský algoritmus

Test bezpečnosti stavu

1. Inicializace
   - $work[m]$ a $finish[n]$ jsou pracovní vektory
   - Inicializujeme $work = available; finish[i] = false; i = 1, ..., n$
2. Najdi i, pro které platí $(finish[i] = false) \wedge (needed[i] \leq work[i])$
3. Pokud takové i neexistuje, jdi na krok 6
4. Simuluj dokončení procesu i
   - $work[i] = work[i] + allocated[i]; finish[i] = true;$
5. Pokračuj krokem 2
6. Pokud platí $finish[i] = true$ pro všechna i, pak stav systému je bezpečný

Bankéřský algoritmus

- Proces P_i formuje vektor request:
- $request[j] = k$ znamená, že proces P_i žádá o k instancí zdroje typu R_j
- if($request[j] \geq needed[i,j]$) proces nárokuje víc než bylo maximum na začátku;
- if($request[j] \geq available[j]$) zatím nelze splnit, je nutné počkat na uvolnění zdrojů, proces se uspí;
- Jinak otestuj přípustnost požadavku simulováním přidělení prostředku a pak ověříme bezpečnost stavu:
  - $available[j] = available[j] - request[j]$;
  - $allocated[i,j] = allocated[i,j] + request[j]$;
  - $needed[i,j] = needed[i,j] - request[j]$;
  - Spusť test bezpečnosti stavu
  - Je-li bezpečný, přiděl požadované zdroje
  - Není-li stav bezpečný, pak vrať úpravy „Akce 3“ a zablokuj proces P_i, neboť přidělení prostředků by způsobilo nebezpečí uváznutím

Bankéřský algoritmus – příklad

Test bezpečnosti stavu:

Prostředky na začátku:

A	B	C
10	6	6

Maximum:

	A	B	C
P_1	8	4	4
P_2	2	1	4
P_3	6	3	3
P_4	5	4	3

Alokace:

	A	B	C
P_1	3	1	1
P_2	1	0	1
P_3	2	1	0
P_4	1	3	1

Požadavek = Maximum – Alokace:

	A	B	C
P_1	5	3	3
P_2	1	1	3
P_3	4	2	3
P_4	4	1	2

Dostupné prostředky

A	B	C
3	1	3

Hledáme proces, který by mohl být dokončen, požadavek $\leq$ dostupné prostředky, pouze P_2.
Po dokončení tohoto procesu, proces uvolní svoje prostředky:

A	B	C
4	1	4

Opět hledáme proces, který by mohl být dokončen, pouze P_4.
Po dokončení tohoto procesu, proces uvolní svoje prostředky:

A	B	C
5	4	5

Opět hledáme proces, který by mohl být dokončen, nyní lze dokončit P_1 a P_3. Po dokončení procesu např. P_1, proces uvolní svoje prostředky:

A	B	C
8	5	6

Nyní lze dokončit P_3, po dokončení jsou dostupné prostředky stejné jako na začátku:

A	B	C
10	6	6

Všechny procesy skončily, stav je bezpečný.

Bankéřský algoritmus – příklad

Proces P_3 žádá o 2 zdroje A:

Prostředky na začátku:

A	B	C
10	6	6

Maximum:

	A	B	C
P_1	8	4	4
P_2	2	1	4
P_3	6	3	3
P_4	5	4	3

Alokace po simulovaném přidělení prostředků:

	A	B	C
P_1	3	1	1
P_2	1	0	1
P_3	4	1	0
P_4	1	3	1

Požadavek = Maximum − Alokace

	A	B	C
P_1	5	3	3
P_2	1	1	3
P_3	2	2	3
P_4	4	1	2

Dostupné prostředky:

A	B	C
1	1	3

Hledáme proces, který by mohl být dokončen, požadavek $\leq$ dostupné prostředky, pouze P_2.

Po dokončení tohoto procesu, proces uvolní svoje prostředky:

A	B	C
2	1	4

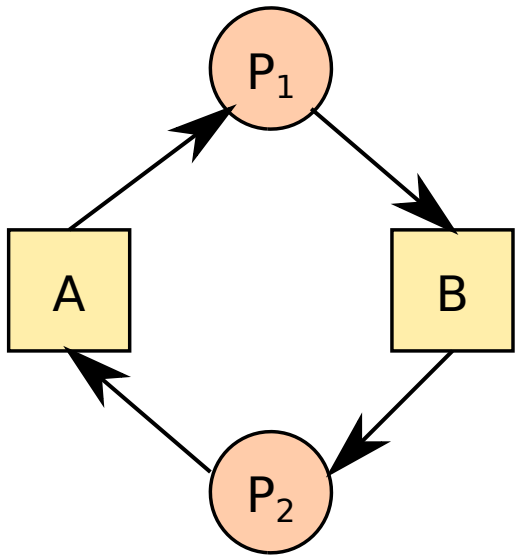
- Nyní nelze dokončit žádný proces, neboť dostupné prostředky nestačí pro dokončení žádného procesu (procesu P_1 chybí 3 zdroje A, 2 zdroje B; procesu P_3 chybí 1 zdroj B, procesu P_4 chybí 2 zdroje A).
- Stav není bezpečný – pokud by všechny procesy chtěli své maximum a teprve potom uvolnili zdroje, pak nastane deadlock.
  - O chování procesů nic nevíme, ale abychom se vyhnuli deadlocku musíme předpokládat nejhorší případ

Detekce uváznutí

- Strategie připouští vznik uváznutí:
- Uváznutí je třeba detekovat
- Vznikne-li uváznutí, aplikuje se plán obnovy systému
- Aplikuje se zejména v databázových systémech, kde je obnova dotazu běžná

Detekce uváznutí s RAG

- Případ jednoinstančního zdroje daného typu
  - Udržuje se čekací graf – uzly jsou procesy
  - Periodicky se provádí algoritmus hledající cykly
  - Algoritmus pro detekci cyklu v grafu má složitost $O(n^2)$, kde n je počet hran v grafu



Detekce uváznutí

- Případ více instancí zdrojů daného typu
  - n ... počet procesů
  - m ... počet typů zdrojů
  - Vektor $available[m]$
    - $available[j] = k$ značí, že je k instancí zdroje typu R_j je volných
  - Matice $allocated[n, m]$
    - $allocated[i, j] = k$ značí, že v daném okamžiku má proces P_i přiděleno k instancí zdroje typu R_j
  - Matice $request[n, m]$
    - Indikuje okamžité požadavky každého procesu:
    - $request[i, j] = k$ znamená, že proces P_i požaduje dalších k instancí zdroje typu R_j

Detekce uváznutí pro více-instanční problémy

Obdoba bankéřského algoritmu (nevíme budoucnost – neznáme maximum):

- *work*$[m]$ a *finish*$[n]$ jsou pracovní vektory
  1. Inicializujeme *work* $=$ *available*; *finish*$[i]$ $=$ *false*; $i = 1, ..., n$
  2. Najdi *i*, pro které platí $(finish[i] = false) and (request[i] \leq work[i])$
  3. Pokud takové *i* neexistuje, jdi na krok 6
  4. Simuluj dokončení procesu *i*
     - *work*$[i] +=$ *allocated*$[i]$;
     - *finish*$[i]$ $=$ *true*;
  5. Pokračuj krokem 2
  6. Pokud platí
     - *finish*$[i]$ $=$ *false* pro některé *i*, pak v systému došlo k uváznutí.
     - Součástí cyklů ve WG jsou procesy P_i, kde *finish*$[i]$ $==$ *false*
- Algoritmus má složitost $O(m \cdot n^2)$, *m* a *n* mohou být veliká a algoritmus časově značně náročný

Detekce uváznutí

Kdy a jak často algoritmus vyvolávat? (Detekce je drahá)

- Jak často bude uváznutí vznikat?
- Kterých procesů se uváznutí týká a kolik jich „likvidovat“?
  - Minimálně jeden v každém disjunktním cyklu v RAG
  - Násilné ukončení všech uvázlých procesů – velmi tvrdé a nákladné
  - Násilně se ukončují dotčené procesy, dokud cyklus nezmizí
    - Jak volit pořadí ukončování
    - Jak dlouho už proces běžel a kolik mu zbývá do dokončení
    - Je to proces interaktivní nebo dávkový (dávku lze snáze restartovat)
    - Cena zdrojů, které proces použil
    - Výběr oběti podle minimalizace ceny
    - Nebezpečí stárnutí
    - některý proces bude stále vybírán jako oběť

Uváznutí – shrnutí

- Metody popsané jako „prevence uváznutí“ jsou velmi restriktivní
  - ne vzájemnému vyloučení, ne postupnému uplatňování požadavků, preempce prostředků
- Metody „vyhýbání se uváznutí“ nemají dost apriorních informací
  - zdroje dynamicky vznikají a zanikají (např. úseky souborů)
- Detekce uváznutí a následná obnova
  - jsou vesměs velmi drahé – vyžadují restartování aplikací
- Smutný závěr
  - Problém uváznutí je v obecném případě efektivně neřešitelný
  - Existuje však řada algoritmů pro speciální situace – zejména používané v databázových systémech
    - Transakce vědí, jaké tabulky budou používat
  - Praktickým řešením jsou distribuované systémy
  - Minimalizuje se počet sdílených prostředků
  - Nutnost zabývat se uváznutím v uživatelských paralelních a distribuovaných aplikacích

Obsah

Meziprocesní komunikace

Přehled meziprocesní komunikace

Název	**Anglicky**	**Standard**
Signál	Signal	POSIX
Roura	Pipe	POSIX
Pojmenovaná roura	Named pipe	POSIX
Soubor mapovaný do paměti	Memory-mapped file	POSIX
Sdílená paměť	Shared memory	System V IPC
Semafor	Semaphore	System V IPC
Zasílání zpráv	Message passing	System V IPC
Soket	Socket	Networking

Signály

- seznámili jste se již na cvičeních
- zaslání jednoduché zprávy (nastavení 1 bitu), která je definována číslem signálu
- příjemcem signálu je pouze proces, odesílatel je buď proces, nebo jádro OS
- obsluha signálů:
  - struct sigaction – sa_handler, či sa_sigaction
  - funkce sigaction – připojení funkce k obsluze signálu
- signál se zpracovává asynchronně (nezávisle) na přijímajícím procesu
  - dojde k přepnutí kontextu a spustí se připojená funkce
  - POZOR na kritické sekce se sdílenými proměnnými

Signály

Použití signálů:

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>

int volatile zalohuj = 0;

void handler(int num) {
  zalohuj = 1;
}

int main() {
  pid_t child = fork();
  if (child == 0) {
    int prace = 30;
    struct sigaction action;
    memset(&action, 0, sizeof(action));
    action.sa_handler = handler;
    if (sigaction(SIGUSR1, &action, NULL)!=0)
      { return 2; }
    while (prace-->0) {
      printf("PRACUJI\n");
      sleep(1);
      if (zalohuj) {
        printf("Ukladam mezivysledek\n");
        zalohuj = 0;
      }
    }
  } else {
    int status;
    while (!waitpid(child, &status, WNOHANG)) {
      sleep(5);
      kill(child, SIGUSR1);
    }
  }
  return 0;
}
```

Roury

- seznámili jste se již na cvičeních
- zaslání dat mezi procesy systémem FIFO
- vlastně simulovaný neexistující soubor
- může být více příjemců i více zapisujících procesů – vznikají ale problémy stejné jako při psaní a čtení více procesů ze souborů
- použití roury:
  - `int pipe(int [2])` – rodič vytvoří rouru
  - všichni potomci i rodič může do roury zapisovat i číst
  - standardně rouru zavřou všichni, kdo ji nepoužívají a slouží k přesunu dat mezi dvěmi procesy

Roury

Použití roury:

```
int main() {
  int pipef[2];
  if (pipe(pipef)!=0) {return 2;}

  pid_t child = fork();
  if (child ==  0) {
    dup2(pipef[1],1);
    close(pipef[0]);
    close(pipef[1]);
    for (int i=0; i<10000; i++) {
       printf("Data %d\n", i);
       fflush(stdout);
    }
  } else { // rodič
    char line[256];
    dup2(pipef[0], 0);
    close(pipef[0]);
    close(pipef[1]);
    while (fgets(line, sizeof(line), stdin))
    {
      printf("Prijata data: %s\n", line);
    }
    wait(&status1);
  }
  return 0;
}
```

Pojmenované roury

- stejný princip – zaslání data mezi procesy systémem FIFO
- může být více příjemců i více zapisujících procesů – vznikají ale problémy stejné jako při psaní a čtení více procesů ze souborů
- oproti normální rouře ji mohou používat libovolné procesy
- použití pojmenované roury:
  - `mkfifo` – vytvoření roury z příkazové řádky
  - `int mknod(const char *, mode_t, dev_t)` – vytvoří pojmenovanou rouru programem, pokud mode využije S_IFIFO
  - všechny procesy pak mohou rouru využít jako ”standardní”soubor

Pojmenované roury

Použití roury: Producent

```
int main() {
  char line[1000];
  mknod("/tmp/myfifo", S_IFIFO | 0660, 0);
  int fd = open("/tmp/myfifo", O_WRONLY, 0);
  for (int i=0; i<100000; i++) {
    sprintf(line, "Data %i\n", i);
    write(fd, line, strlen(line));
  }
  return 0;
}
```

Konzument

```
int main() {
  char line[1024];
  int fd = open("/tmp/myfifo", O_RDONLY, 0), rd;
  while ((rd=read(fd, line, 1000))>0) {
    line[rd]=0;
    printf("Prijata data: %i %s\n", rd, line);
  }
  return 0;
}
```

Sdílený soubor mapovaný do paměti

- data lze přenášet mezi procesy pomocí souborů
  - jeden proces zapisuje do souboru, druhý čte ze souboru
- použití normálních souborů je pomalejší, i když je zápis bufferován v paměti
- rychlejší varianta je soubor mapovaný do paměti
- vybraný úsek souboru je mapován přímo do paměti procesu:
  - `mmap(void *addr, size_t delka, int proto, int typ, int fd, off_t posunuti)`
  - vrací adresu, kterou lze použít pro zápis/čtení přímo do/z paměti
  - soubor musí mít alespoň délku, kterou mapujeme do paměti
  - synchronizace je složitější, nejlépe za použití jiného mechanismu
  - Mapování souboru */dev/zero* = alokace paměti

Soubor mapovaný do paměti

```c
int main() {
  char *shared_mem, buf[256];
  int fd = open("/tmp/mapedfile",
                O_RDWR | O_CREAT, 0660);
  printf("Open file %i\n", fd);
  for (int i=0; i<1000; i++) {
    sprintf(buf, "Data %03i\n", i);
    buf[9]=0;
    write(fd, buf, 10);
  }
  shared_mem = mmap(NULL, 10000,
     PROT_READ | PROT_WRITE,
     MAP_SHARED, fd, 0);
  printf("mmap %p\n", shared_mem);
  close(fd);
  return 0;
}
```

```c
int main() {
  char *shared_mem;
  int fd = open("/tmp/mapedfile",
                O_RDWR | O_CREAT, 0660);
  shared_mem = mmap(NULL, 10000,
     PROT_READ | PROT_WRITE,
     MAP_SHARED, fd, 0);
  for (int i=0; i<1000; i++) {
    printf("Ulozeno: %s\n", shared_mem);
    shared_mem+=10;
  }
  close(fd);
  return 0;
}
```

Sdílená paměť

- velmi podobné, jako soubory mapované do paměti
- soubor je pouze virtuální a nezapisuje se na disk
- po vypnutí počítače se data ve sdílené paměti ztratí
- použití je velmi podobné:
  - shm_open – vytvoří virtuální soubor, nebo připojí k existujícímu podle jména
  - mmap(void *addr, size_t délka, int proto, int typ, int fd, off_t posunutí)
    - vrací adresu, kterou lze použít pro zápis/čtení přímo z paměti
  - ftruncate nebo write, kvůli vytvoření místa ve virtuálním souboru
  - synchronizace je složitější, nejlépe za použití jiného mechanismu

Sdílená paměť – příklad

```c
int main() {
  char *shared_mem;
  int fd = shm_open("pamet",
     O_RDWR | O_CREAT | O_TRUNC, 0660);
  ftruncate(fd, 10000);
  shared_mem = mmap(NULL, 10000, PROT_READ
     | PROT_WRITE, MAP_SHARED, fd, 0);
  for (int i=0; i<1000; i++) {
    sprintf(shared_mem, "Data %03i\n", i);
    shared_mem[9]=0;
    shared_mem += 10;
  }
  close(fd);
  shm_unlink("pamet");
  return 0;
}
```

```c
int main() {
  char *shared_mem;
  int fd = shm_open("pamet",
      O_RDWR | O_CREAT, 0660);
  shared_mem = mmap(NULL, 10000, PROT_READ
      | PROT_WRITE, MAP_SHARED, fd, 0);
  close(fd);
  for (int i=0; i<1000; i++) {
    printf("Ulozeno: %s\n", shared_mem);
    shared_mem+=10;
  }
  return 0;
}
```

Pojmenovaný semafor

- podobně jako pojmenovaná roura je možné k němu přistoupit z nového procesu
- semafor se připojuje k již existujícímu souboru
  - pouze identifikace, nic do souboru neukládá
- podobně jako semafory pro vlákna, umožňuje implementovat kritickou sekci, nebo počítat
- použití semaforu:
  - ftok – vytvoří identifikátor (klíč) podle jména souboru
  - semget – připojí/vytvoří semafor ke klíči
  - semctl – nastaví hodnotu semaforu
  - semop – provede operaci (odečtení, nebo přičtení)

Semaphore System V a sdílená paměť

```c
int main() {
  key_t s_key;
  union semun {
    int val;
    struct semid_ds *buf;
    ushort array [1];
  } sem_attr;
  struct sembuf asem;
  int buffer_count_sem, spool_signal_sem;
  char *shared_mem;

  /* key identifikuje semafor  */
  if ((s_key = ftok ("/tmp/free", 'a')) == -1)
  { perror ("ftok"); exit (1); }
  if ((buffer_count_sem = semget (s_key, 1,
   0660 | IPC_CREAT)) == -1)
   { perror ("semget"); exit (1); }
  sem_attr.val = 10; // nastav na velikost bufferu
  if (semctl (buffer_count_sem, 0, SETVAL, sem_attr)
    == -1) { perror (" semctl SETVAL "); exit (1); }
  /* key druheho semaforu  */
  if ((s_key = ftok ("/tmp/data", 'a')) == -1) {
    perror ("ftok"); exit (1);
  }
  if ((spool_signal_sem = semget (s_key, 1,
   0660 | IPC_CREAT)) == -1) {
    perror ("semget"); exit (1);
  }
  sem_attr.val = 0;  // inicializace na 0
  if (semctl (spool_signal_sem, 0,
    SETVAL, sem_attr) == -1)
    { perror (" semctl SETVAL "); exit (1); }
  int fd = shm_open("pamet", O_RDWR
   | O_CREAT | O_TRUNC, 0660);
  ftruncate(fd, 1000);
  shared_mem = mmap(NULL, 1000, PROT_READ
   | PROT_WRITE, MAP_SHARED, fd, 0);
  close(fd);
  asem.sem_num = 0;
  asem.sem_flg = 0;
  for (int i=0; i<50; i++) {
    asem.sem_op = -1;
    if (semop (buffer_count_sem, &asem, 1) == -1)
    { perror ("semop: buffer_count_sem"); exit (1); }
    sprintf(shared_mem, "Data %03i\n", i);
    printf( "Data %03i %p\n", i, shared_mem);
    shared_mem[9]=0;
    shared_mem += 10;
    if (i%10==9) {
      shared_mem-=100;
    }
    asem.sem_op = 1;
    if (semop (spool_signal_sem, &asem, 1) == -1)
    { perror ("semop: spool_signal_sem"); exit (1); }
  }
  return 0;
}
```

Semaphore System V a sdílená paměť

```
int main() {
  key_t s_key;
  union semun {
    int val;
    struct semid_ds *buf;
    ushort array [1];
  } sem_attr;
  struct sembuf asem;
  int buffer_count_sem, spool_signal_sem;

  /* pouzij stejny semafor jako producent  */
  if ((s_key = ftok ("/tmp/free", 'a')) == -1) {
    perror ("ftok"); exit (1);
  }
  if ((buffer_count_sem = semget (s_key, 1,
   0660 | IPC_CREAT)) == -1) {
    perror ("semget"); exit (1); }

  /* pouzij stejny semafor jako producent  */
  if ((s_key = ftok ("/tmp/data", 'a')) == -1) {
    perror ("ftok"); exit (1);
  }
  if ((spool_signal_sem = semget (s_key, 1,
   0660 | IPC_CREAT)) == -1) {
    perror ("semget"); exit (1); }
```

```
  char *shared_mem;
  int fd = shm_open("pamet", O_RDWR |
     O_CREAT, 0660);
  ftruncate(fd, 1000);
  shared_mem = mmap(NULL, 1000, PROT_READ
   | PROT_WRITE, MAP_SHARED, fd, 0);
  close(fd);
  asem.sem_num = 0;
  asem.sem_flg = 0;
  for (int i=0; i<50; i++) {
    asem.sem_op = -1;
    if (semop (spool_signal_sem, &asem, 1) == -1) {
      perror ("semop: buffer_count_sem"); exit (1);
    }
    printf("Ulozeno: %s, %p\n", shared_mem, shared_mem);
    shared_mem += 10;
    if (i%10==9) {
      shared_mem-=100;
      sleep(1);
    }
    asem.sem_op = 1;
    if (semop (buffer_count_sem, &asem, 1) == -1) {
      perror ("semop: spool_signal_sem"); exit (1);
    }
  }
  close(fd);
  return 0;
}
```

Fronta zpráv

- zprávy jsou zasílány a vyzvedávány do/z fronty zpráv identifikovaných libovolným souborem
- podobně jako pojmenovaná roura a semafor je možné k němu přistoupit z nového procesu
- zprávy mají povinně typ, podle kterého je možné vybírat z fronty zpráv pouze zprávy zadaného typu
- použití fronty zpráv:
  - msgget – vytvoří virtuální frontu zpráv, nebo připojí k existující frontě podle jména souboru zadaného jeho klíčem
    - nutné vytvořit si vlastní strukturu zpráv, která jako první obsahuje long – typ zprávy
  - msgsnd – zaslání zprávy, pozor délka zprávy je délka struktury zmenšená o velikost long – typ zprávy
  - msgrcv – přijmutí zprávy zadaného typu
  - msgctl – odstranění fronty zpráv

Fronty zpráv

```c
struct my_msg {
  long mtype;
  int len;
  char txt[10];
};
int main() {
  key_t s_key;
  int msg_id;
  struct my_msg msg;
  if ((s_key = ftok ("/tmp", 'a')) == -1)
  { perror ("ftok"); exit (1); }
  if ((msg_id = msgget(s_key, 0660 | IPC_CREAT))
    == -1) { perror ("msgget"); exit (1); }
  for (int i=0; i<50; i++) {
    msg.mtype=1;
    msg.len = 10;
    sprintf(msg.txt, "Data %03i\n", i);
    msg.txt[9]=0;
    if (msgsnd(msg_id, &msg, sizeof(msg)-sizeof(long),0)
      == -1) { perror ("msgsnd"); exit (1); }
  }
  if (msgrcv(msg_id, &msg, sizeof(msg)-sizeof(long), 2, 0)
     == -1) { perror ("msgrcv"); exit (1); }
  printf("Prijato: %s\n", msg.txt);
  if (msgctl(msg_id, IPC_RMID, 0) == -1)
  { perror ("msgctl"); exit (1); }
  return 0;
}
```

```c
int main() {
  key_t s_key;
  int msg_id;
  struct my_msg msg;

  if ((s_key = ftok ("/tmp", 'a')) == -1) {
    perror ("ftok"); exit (1);
  }
  if ((msg_id = msgget(s_key, 0660 | IPC_CREAT))
    == -1) { perror ("msgget"); exit (1); }
  for (int i=0; i<50; i++) {
    if (msgrcv(msg_id, &msg, sizeof(msg)-sizeof(long), 1, 0)
     == -1) { perror ("msgrcv"); exit (1); }
    printf("Prijato: %s\n", msg.txt);
  }
  msg.mtype=2;
  msg.len = 10;
  sprintf(msg.txt, "Koncime\n");
  msg.txt[9]=0;
  if (msgsnd(msg_id, &msg, sizeof(msg)-sizeof(long), 0)
   == -1) { perror ("msgsnd"); exit (1); }
  return 0;
}
```

Bod z přednášky

Pokud Váš program obsahuje deadlock pak

A - program vždy skončí nekonečným čekáním

B - program někdy pracuje správně, někdy skončí nekonečným čekáním

C - program vždy skončí, někdy ale s chybným výsledkem

D - program vždy skončí se správným výsledkem