7. Další synchronizační nástroje

Motivace:

```
lock(sem);
if (nenineco) {
     /* nenastal potřebný stav, je nutné počkat, až nastane */
a)   wait();              /* nelze, zůstal by zamčený sem */

b)   unlock(sem);         /* nelze, uvolnili jsme kritickou sekci,
     wait();                 než uděláme wait, stav se může změnit  */
     /* !?! co s tím? */
}
unlock(sem);
```

semafory – koordinační podmínka (ano/ne nebo počet) spojena se synchronizací, nelze oddělit podmínku od pozastavení, vyžaduje kooperaci procesů při vytváření kritických sekcí

monitory - oddělení operací se sdílenými proměnnými a pozastavení, vynucení ochrany sdílených proměnných

7.1 Monitor - abstraktní datový typ (Hoare, Hansen, 1974)

1. Sdílené proměnné dostupné pouze operacemi monitoru.
2. Provádění operací jednoho monitoru je vzájemně vyloučené.
3. Inicializace monitoru nastaví sdílené proměnné.

```
TYPE T = monitor
           var shared: int; /* nedostupné vně */

           procedure increment;   /* dostupné */
           begin
               shared := shared + 1;
           end;

         begin                  /* inicializace */
           shared := 0;
         end;
VAR  m: T;
begin
    ...
    m.increment;           /* použití monitoru */
```

Monitor zajistí vzájemné vyloučení operací nad monitorem = všechny operace monitoru jsou atomické.

4. Pozastavení - nelze čekat aktivně uvnitř monitoru! Proč?

condition - fronta čekajících procesů

```
VAR c: condition;    /* pouze uvnitř monitoru */
```

Operace:

`c.wait;`	pozastavení procesu, vzdání se monitoru
`c.signal;`	odblokování prvního čekajícího (pokud je), ten získá opět výlučný přístup k monitoru

Problém: operace se sdílenou proměnnou musí být uvnitř monitoru → po *c.signal* jsou dva procesy v monitoru!

Řešení: někdo musí být pozastaven

1. Blokující signalizace podmínky
2. Neblokující signalizace podmínky

1. Hoare, Hansen - proces P provádějící *c.signal* je pozastaven a pokračuje odblokovaný proces Q za *c.wait*. Až Q opustí monitor nebo začne zase čekat, pokračuje nejprve P a pak teprve ostatní procesy čekající na vstup do monitoru. Signalizace stavu je kooperativní, ten kdo stav změnil, efektivně předá monitor probuzenému čekajícímu procesu. Pokud nikdo nečeká, je signalizace prázdnou operací.

Příklad: Implementace číselného semaforu 1.typem monitorů

```
TYPE SEMA = monitor
            var value: int;
                queue: condition;

            procedure down;
            begin
                if value <= 0 then queue.wait;
                value := value - 1;
            end;

            procedure up;
            begin
                value := value + 1;
                if value == 1 then queue.signal;
            end;
            ...
    end;
var semaphore: SEMA;
```

Pozn.: *queue.signal* by mohlo být voláno vždy (pokud nikdo nečeká, není koho probudit).

2. Lampson, Redell (Mesa, 1980) - operace *c.notify* pouze odblokuje pozastavený proces, monitor zůstane dále stejnému procesu, teprve po opuštění monitoru se může odblokovaný proces dostat do monitoru a pokračovat za *c.wait*, soutěží ovšem s procesy vstupujícími do monitoru normálně - nelze zaručit splnění testované podmínky v odblokovaném procesu! Proto je třeba vždy testovat podmínku čekání po probuzení znovu!

```
procedure down;
begin
    while value <= 0 do queue.wait;
    value := value - 1;
end;
```

Další důvod pro opakování testu podmínky po probuzení v praxi je možnost skončení čekání měkkou chybou (errno = EAGAIN, apod.).

7.2 Synchronizace vláken POSIX 1003.1c

Proměnná typu *pthread_cond_t* - *condition* (čekání na stav), vždy ve spojení se vzájemným vyloučením = tvoří **monitor**

Inicializace:

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

nebo

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Čekání na stav:

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

Atomická operace uvolnění *mutex* a zahájení čekání na *cond*. Po probuzení bude *mutex* opět zamčen. Vlákno může být probuzeno i jindy než při signalizaci splnění podmínky. Mutex musí vždy střežit proměnné, které jsou testovány v podmínce cyklu volání *pthread_cond_wait()*. Použití bez zamčeného mutexu je nesmysl! Použití bez opakovaného testování podmínky v cyklu je také špatně!

Příklad:

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int val = 0;        /* sdílená proměnná */
...
pthread_mutex_lock(&mutex);
while (val < MIN) { /* podmínka splněna? */
   pthread_cond_wait(&cond, &mutex);/*ne, čekat*/
}
...   /* zpracuj suma */
val -= neco;
pthread_mutex_unlock(&mutex);
```

Musí být podmínka testována opakovaně v cyklu? Proč?

Signalizace podmínky:

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

Pokud čeká nějaké vlákno na *cond*, je odblokováno (případně všechny právě čekající pro *broadcast*). Monitor je dále střežen mutexem, takže odblokované vlákno čeká, až bude volný (až se podaří znovu zamknout mutex, který je v *pthread_cond_wait*). Pokud nikdo nečeká, je signalizace **prázdnou** operací. Pokud čekají všechna vlákna na stejný stav (podmínku) a stačí probudit jedno, je vhodnější *signal*, jinak *broadcast*.

Použití signalizace:

```
pthread_mutex_lock(&mutex);
val += neco;        /* změna stavu podmínky */
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

Musí být signalizace chráněna pomocí *mutex_lock()*? A proč?

Zrušení podmínky

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

Uvolní případnou přidělenou paměť a systémové prostředky (obvykle prázdná operace, ale nemusí být).

Dále viz Příklady na použití *pthread_cond* u přednášek.

Implementace monitorů pomocí semaforů

1. Vzájemné vyloučení — `mutex mutex`
2. Vstup do monitoru — `lock(mutex)`
3. Výstup z monitoru — `unlock(mutex)`

Jak implementovat *wait* a *signal*?

Pokus č. 1:

```
semaphore c_wait;   init(c_wait, 0);    /* blokující */
c.wait              down(c_wait);

c.signal            up(c_wait);
```

Proč nelze? *(c.wait zablokuje monitor)*

Pokus č. 2:

```
c.wait              unlock(mutex);
                    down(c_wait);
                    lock(mutex);

c.signal            up(c_wait);
```

Proč nelze?

Rozdíl mezi semafory a *condition*:

- obecný semafor si pamatuje historii - počet *down()* a *up()*
- *condition* ne! *c.signal* na prázdnou frontu je prázdná operace

```
c.signal    if (value(c_wait) > 0) up(c_wait);
```

Proč takto ne?

- Není definováno získání hodnoty semaforu!
- Operace testování podmínky a *up()* není atomická - může přijít po testu na hodnotu (protože je mimo monitor!) → ztracený signál!

3. *c.signal* `semaphore block`
   - musí být aktivován čekající `up(c_wait)`
   - musí být pozastaven volající `down(block)`

4. Opuštění monitoru
   - aktivovat proces pozastavený *c.signal* nebo proces čekající na vstup `up(block)` / `unlock(mutex)`

5. *c.wait* `semaphore c_wait`
   - aktivovat proces pozastavený *c.signal* nebo proces čekající na vstup `up(block)` / `unlock(mutex)`
   - volající musí být pozastaven `down(c_wait)`

Semafory

```
void sema_down()
{
    lock(mutex);

    if (podmínka) {
        ++c_waiting;
        if (blocked > 0) up(block);
        else unlock(mutex);
        down(c_wait);       /* obecný semafor! */
        --c_waiting;
    }

    if (blocked > 0) up(block);
    else unlock(mutex);
}

...
    ++blocked;
    if (c_waiting > 0) {
        up(c_wait);
        down(block);
    }
    --blocked;
```

Monitor

```
procedure down;

begin

  if (podmínka)
     c.wait;

end;

                    c.signal;
```

POSIX 1003.1c - Semafory volitelné, implementace semaforů pomocí mutexu a condition je ale triviální:

```
typedef struct {
  pthread_cond_t cond;   /* pro čekání */
  pthread_mutex_t mutex; /* vzájemné vyloučení*/
  int value;             /* hodnota semaforu */
} sema_t;

void sema_init(sema_t *sema, int value)
{
    pthread_mutex_init(&sema->mutex, NULL);
    pthread_cond_init(&sema->cond, NULL);
    assert(value >= 0);
    sema->value = value;
}

void sema_down(sema_t *sema)
{
    pthread_mutex_lock(&sema->mutex);
    while (sema->value <= 0) {
        pthread_cond_wait(&sema->cond,
                                    &sema->mutex);
    }
    --sema->value;
    pthread_mutex_unlock(&sema->mutex);
}

void sema_up(sema_t *sema)
{
    pthread_mutex_lock(&sema->mutex);
    if (sema->value++ == 0)
        pthread_cond_signal(&sema->cond);
    pthread_mutex_unlock(&sema->mutex);
}
```

Závěr: Implementace mutexu/condition pomocí semaforů je podstatně komplikovanější (viz monitory). Mutex a condition je tedy praktičtější než semafor!

7.3 C11

Standard jazyka C z roku 2011 převzal pod jiným rozhraním synchronizační prostředky POSIX vláken. Mutex je typu *mtx_plain* nebo *mtx_timed*, navíc může být *mtx_recursive*.

```
#include <threads.h>
#include <stdatomic.h>
typedef struct {
  cnd_t cond;       /* pro čekání */
  mtx_t mutex;      /* vzájemné vyloučení*/
  atomic_int value;    /* hodnota semaforu */
} sema_t;
// inicializace
void sema_init(sema_t *sema)
{
    mtx_init(&sema->mutex, mtx_plain);
    cnd_init(&sema->cond);
    atomic_init(&sema->value, 0);
}

void sema_down(sema_t *sema)
{
    int pval;
    while ((pval = atomic_load(&sema->value))>0){
        int newval = pval-1;
        // pokud je hodnota>0, atomicky zmenšíme
        if (atomic_compare_exchange_strong(&sema-
>value, &newval, pval)) return;
    }
    mtx_lock(&sema->mutex);
    while (1) do {
        pval = atomic_load(&sema->value);
        newval = pval - 1;
        if (pval > 0 &&
            atomic_compare_exchange_strong(&sema-
>value, &newval, pval)) break;
        cnd_wait(&sema->cond,  &sema->mutex);
```

```
    }
    mtx_unlock(&sema->mutex);
}

void sema_up(sema_t *sema)
{
    if (atomic_fetch_add(&sema->value, 1) > 0)
        // pokud byla hodnota>0, nikdo nečeká
        return;
    mtx_lock(&sema->mutex);
    cnd_signal(&sema->cond);
    mtx_unlock(&sema->mutex);
}
```

Výhoda: může být implementováno v MS VS
Nevýhoda: nejsou atributy, implementace omezenější

7.4 C++11

```
class mutex {  // recursive_mutex, timed_mutex
public:
    mutex();
    ~mutex();
    void lock();
    bool try_lock();
    void unlock();
};

template <class Mutex>
class unique_lock {    // pouze zabaluje mutex
    unique_lock(); // konstruktor automat. zamkne
    ~unique_lock();// destruktor automat. odemkne
    void lock();   // explicitní operace
    void unlock();
};

class condition_variable {
public:
    condition_variable();
    ~condition_variable();
    void notify_one();
    void notify_all();
    void wait(unique_lock<mutex>& lock);
};
```

Příklad:

```
std::mutex lock;
class xxx {
    void m() {
        std::unique_lock<std::mutex> lk(lock);
        // zamčeno
    }   // automaticky odemčeno
};
```

7.5 Java

Jazyk Java je standardně paralelním programovacím jazykem.

Vlákno – třída odvozená od standardní třídy *Thread*:

```
public class SimpleThread extends Thread {
    public void run() {    /* tělo vlákna */
            ...
    }                      /* ukončení vlákna */
}
```

Vytvoření a spuštění vlákna – vytvoření instance třídy a vyvolání metody *start()*:

```
    SimpleThread p = new SimpleThread();
    p.start();
```

Čekání na ukončení vlákna – metoda *join()*:

```
    p.join();
```

Vzájemné vyloučení – synchronizovaná metoda (vše je skrytě odvozeno od *Object*).

Čekání – metody *wait()*, *notify()*, *notifyAll()* třídy *Object*:

```
public class SharedVariable { /* monitor */
    private int value;
    public synchronized int get()
    {   /* vzájemné vyloučení přístupu k instanci SharedVariable */
        while (value <= 0) try {
            wait();      /* pozastavení, uvolnění přístupu */
        } catch (InterruptedException e) { ... }
        ...
    }
    public synchronized int set(int val)
    {
        value = 1;
        notifyAll(); /* odblokování pozastavených vláken */
    }
}
```

Vlastnosti jako u monitoru, ale pouze jeden implicitní *condition*!

7.6 Semafory mezi procesy - SYSV IPC

SYSV IPC – Inter-Process Communication z Unix System V.2
Single Unix Specification v. 2 – součást

- Prostředky jsou identifikovány celými čísly. Tato čísla mají roli jmen, podobně jako u souborů jména souborů.
- Výsledkem operace otevření/vytvoření je interní deskriptor prostředku pro další operace (analogie deskriptorů souborů).
- Každý prostředek má majitele a majitelskou skupinu (stejně jako soubory v systému souborů).
- Majitel prostředku může provádět některé privilegované operace, například zrušit prostředek nebo změnit jeho majitelství.
- Přístup k prostředku je řízen přístupovými právy *read* a *write*.

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

Jeden prostředek obsahuje pole semaforů, semafory v poli jsou indexovány od 0. Počet semaforů v prostředku je dán parametrem *nsems*. Počáteční stav semaforů je nedefinovaný. Přístupová práva jsou dána parametrem *flag*.

Řídicí operace:

```
int semctl(int id, int semnum, int cmd, ...);
```

Číslo semaforu *semnum* je v intervalu 0 až *nsems*-1. V operacích, které pracují s prostředkem jako celkem, je ignorováno.
Parametr *cmd* určuje operaci:

`SETVAL` nastavení hodnoty semaforu *semnum*, poslední parametr je typu *int* a obsahuje nastavovanou hodnotu,
`SETALL` nastaví hodnoty všech semaforů, poslední parametr musí být pole prvků typu *unsigned short*,
`IPC_RMID` zrušení prostředku.

Operace se semafory:

```
struct sembuf {
    short sem_num; /* číslo semaforu v poli */
    short sem_op;  /* popis operace */
    short sem_flg; /* příznaky pro provádění */
};
int semop(int id, struct sembuf *sops, size_t
nsops);
```

Jedním voláním lze provést více operací. Počet operací je zadán parametrem *nsops*, typ operace je určen hodnotou *sem_op*:

sem_op < *0* *operace DOWN(S,* -sem_op*). Pokud nastane čekání, je ukončeno při změně hodnoty semaforu (pak je opakován pokus o zamčení), zrušení semaforu (návrat s chybou EIDRM) nebo při zpracování signálu (návrat s chybou EINTR). Čekání může být potlačeno příznakem IPC_NOWAIT, výsledkem je –1 a chyba EAGAIN.*

sem_op > *0* *operace UP(S,* sem_op*). Hodnota semaforu je zvětšena o* sem_op*. Pokud čekají nějaké procesy na uvolnění semaforu, pak jsou odblokovány a v nedefinovaném pořadí se znovu pokusí zamknout semafor (zmenšit jeho hodnotu).*

sem_op = *0* *Čekání na zamčení semaforu (na nulovou hodnotu). Pokud má semafor nulovou hodnotu, je volání ukončeno.*

Pokud je v jednom volání *semop()* zadáno více operací, pak je zaručena atomičnost pouze jednotlivých operací, ne celého volání funkce *semop()*. Pokud je použito IPC_NOWAIT, měl by vždy být zadán flag SEM_UNDO, který zajistí při neúspěšnosti celé operace navrácení změněných semaforů do původního stavu.

7.7 Podmíněné kritické sekce

- Hoare, 1972, Brinch-Hansen, 1973
- vymezení kritické sekce a sdílených proměnných

resource r::var1, var2, ...;
region r **when** C **do** KS; C je podmínka strážící KS

```
resource sem::value: integer;

procedure down;
begin
    region sem when value > 0 do value:=value-1;
end;

procedure up;
begin
    region sem do value:=value+1;
end;
```

Condition = **region** r **when** C; /* prázdná KS */
Vzájemné vyloučení = **region** r **do** KS; /* prázdná podmínka */
Owicki, Gries (1976): **await** C **then** KS **end**

Implementace pomocí semaforů

```
lock(mutex);                    /* binární/obecný */
++waiting;
while (!C) {
    unlock(mutex);
    down(block);                /* obecný */
    lock(mutex);
}
--waiting;
KS;
for (i = 0; i < waiting; i++) up(block);
unlock(mutex);
```

7.8 Bariéra (rendezvous)

čekání na dosažení stejného místa v N procesech:

barrier(b);

Typická synchronizace u paralelních algoritmů

Implementace pomocí semaforů:

- čítač s počátečním stavem N
- proces v bariéře dekrementuje čítač a pokud není nula, čeká na obecný blokující semafor
- až přijde poslední, čítač je nula, musí odblokovat všechny čekající (N-1 krát operace *up()*)

```
void barrier_init(int n)
{
    count = n;
}

void barrier_wait()    /* špatné řešení */
{
    lock(m);           /* mutex */
    if (--count > 0) {
        waiting++;
        unlock(m);
        down(b);       /* obecný semafor */
    } else {
        for (i = 1; i < waiting; i++) up(b);
        count = n;     /* reinicializace b. */
        waiting = 0;
        unlock(m);
    }
}
```

Problém:

- blokovaný proces nemusí stihnout provést *down(b)*, další odblokuje bariéru cyklem *up()* a bude tam o jednu signalizaci více, takže vstup do následující bariéry nebude blokující!

Řešení - dva čítače a dva blokující semafory, jednou použít jeden pár, pak druhý pár (nebo jeden pro vstup, druhý pro výstup)

```
semaphore m, b[2];
int count, waiting[2];

void barrier_init(int n)
{
    count = start = n;
    init(m, 1);
    init(b[0], 0); init(b[1], 0);
}
```

```
void barrier_wait()
{
    int side, i;
    lock(m);
    if (count > 0) { --count; side = 1; }
    else { ++count; side = 0; }
    if (count) {
        unlock(m);
        down(b[side]);
    } else {
        if (side) count = -start;
        else count = start;
        for (i = 0; i<start-1; i++) up(b[side]);
        unlock(m);
    }
}
```

Problém:

- neefektivní řešení, nutná efektivnější synchronizace na nižší úrovni (stromová synchronizace, po párech)

7.9 OpenMP (http://openmp.org)

Preprocesor pro jazyk C, C++ a Fortran a knihovna. Určeno pro zápis paralelních algoritmů, ne pro obecné paralelní programování. Podporuje paralelní provádění bloků, sekcí a cyklů *for*, kritické sekce a bariéry.

```
#include <stdio.h>

int main(int argc, char **argv)
{
   int thrid;
   int counter = 0;
   // následující blok bude paralelní
   // implicitně podle počtu procesorů
   #pragma omp parallel private(thrid)
   {
      thrid = omp_get_thread_num();
      printf("Thread %d\n", thrid);
      // následující příkaz je kritickou sekcí
      #pragma omp critical
      counter++;
      // čekání na průchod bariérou
      #pragma omp barrier
      if (thrid == 0) {
         printf("%d threads running\n",
                     omp_get_num_threads());
      }
   }  // implicitní join
   printf("Counter is %d\n", counter);
   return 0;
}
```

Podpora MS Visual Studio, gcc od verze 4.2:

```
gcc -fopenmp -O2 -o omp omp.c
```

7.10 RWLOCK

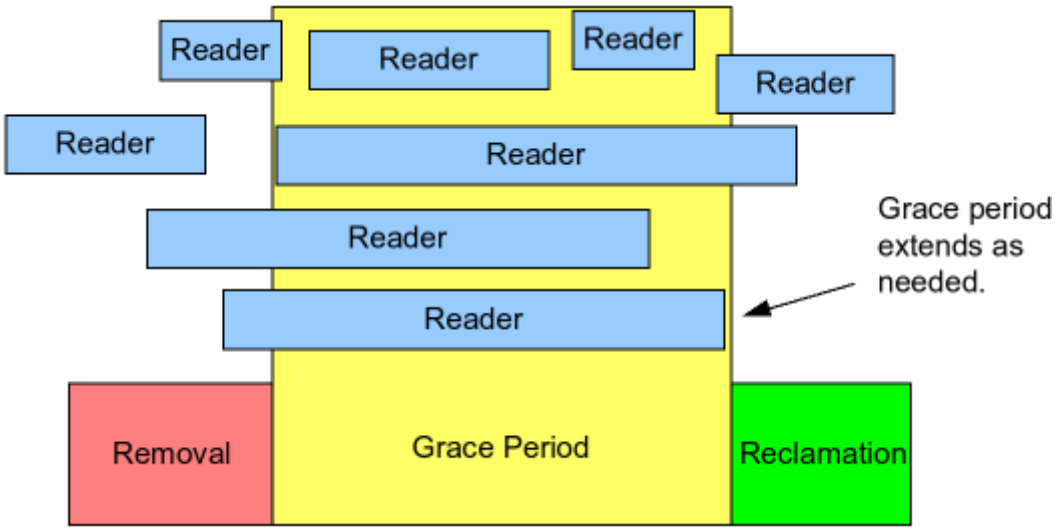
Nástroj pro synchronizaci typu čtenáři/písaři. Umožňuje paralelní přístup ke sdílenému prostředku bez modifikace a výlučný přístup pro modifikaci. Řešení inverze priority je obvykle pouze pro písaře (protože čtenáři se nezaznamenávají). Navíc mohou být doplněny operace přechodu ze sdíleného čtecího přístupu na výlučný (upgrade) a opačně (downgrade).

rwlock_t (Linux)	***rwlock (FreeBSD)***
*read_lock(rwlock_t *lck);*	*rw_wlock(struct rwlock *rw);*
*read_unlock(rwlock_t *lck);*	*rw_runlock(struct rwlock *rw);*
*write_lock(rwlock_t *lck);*	*rw_wlock(struct rwlock *rw);*
*write_unlock(rwlock_t *lck);*	*rw_wunlock(struct rwlock *rw);*
	*rw_try_upgrade(struct rwlock *rw);*
	*rw_downgrade(struct rwlock *rw);*

Při častém použití nastávají obvyklé problémy blokujících synchronizačních prostředků (uváznutí, latence) – v tomto případě bývá často řešen nástrojem typu RCU.

RCU (Read-Copy-Update)

Přístup čtenářů je neblokující, pouze oznamují začátek a konec přístupu.
Písaři musí:
1. vytvořit kopii datové struktury a nad ní udělat změny,
2. vyměnit ukazatel na novou kopii datové struktury,
3. počkat, až všichni čtenáři, kteří začali před kopií, skončí,
4. uvolnit původní datovou strukturu.



Zjednodušující předpoklady pro implementaci v jádře:

1. Písaři jsou navzájem vyloučení zamykáním a
2. datová struktura musí být modifikovatelná atomicky během rozpracovaného přístupu čtení - pak není třeba dělat kopii.
3. Při získání čtecího přístupu nesmí být proces pozastaven ani odstaven od procesoru - pak stačí počkat, až všechny ostatní procesory ve víceprocesorovém systému provedou přepnutí kontextu (tím už musela jakákoli čtecí operace skončit).

```
rcu_read_lock(void);          /*  Linux 2.6 */
/* čtení sdíleného seznamu */
p = head.next;        /* RCU safe list */
while (rcu_dereference(p) != head) {
  ...
  p = p->next;
}
rcu_read_unlock(void);
```

rcu_read_lock() musí u preemptivního jádra zablokovat preempci, *rcu_read_unlock()* odblokovat, u nepreemtivního jádra to mohou být prázdné operace.

/* operace písaře – modifikace seznamu */

```
rcu_assign_pointer(head, newhead);
list_del_rcu(entry);
list_add_rcu(&head, entry);
list_replace_rcu(old, new);
```

/* synchronizace zápisu */

```
synchronize_rcu();
/* pak lze uvolnit změněnou nebo zrušenou
položku */
kfree(entry);
```

synchronize_rcu() pozastaví provádění písaře, dokud nejsou uvolněny všechny právě zahájené čtecí sekce.

7.11 Klasické signály ISO C

Signály = standard ISO jazyka C, původně z Unixu

Typy signálů:

- *chybové* (dle PDP) = SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGFPE, SIGBUS a SIGSEGV, *implicitně ukončení procesu*
- *uživatelské* = SIGHUP, SIGINT, SIGQUIT, SIGKILL, SIGTERM, SIGUSR1 a SIGUSR2, *ukončení procesu* (SIGKILL nelze ignorovat).
- *systémové* = SIGCHLD, SIGSYS, SIGPIPE a SIGALRM, *ukončení procesu s výjimkou SIGCHLD*.

Zpracování:

- *zaslání* signálu procesu = jádro systému nebo *kill(),*
- pokud je proces pozastaven ve službě jádře, pak může zaslaný signál přerušit čekání (ale nemusí), havarijně ukončit volání systému (výsledkem bude -1 a *errno* = EINTR),
- pokud signál nemůže přerušit čekání (některá čekání jsou nepřerušitelná), pak je příchod signálu pouze zaznamenán a zpracování odloženo až na ukončení čekání nebo návrat z volání jádra systému,
- pokud proces čeká na přidělení procesoru nebo běží, pak je příchod signálu zaznamenán a zpracování odloženo až do chvíle, kdy se vrací ze systémové fáze do uživatelské,
- pokud má proces zaznamenány signály čekající na zpracování, pak při přechodu ze systémové fáze do uživatelské proběhne *zpracování* čekajících signálů, což může být:

a) implicitní zpracování - u většiny signálů způsobí *ukončení* procesu.

b) *ignorování* – signál je zahozen,

c) zpracování *ošetřující* funkcí v uživatelském programu:

```
void func(int signo)
{
    ....
}
```

Ošetření musí být předem nastaveno std. funkcí *signal()*:

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

func = SIG_DFL, SIG_IGN nebo adresa funkce

Průběh ošetření signálu vlastní funkcí *func()*:

1. Před vyvoláním funkce jádro obnoví zpracování signálu SIG_DFL. Pokud mají být další příchozí signály ošetřeny, musí funkce *func()* obnovit nastavení zpracování signálu.
2. Na zásobník je uložen stav všech registrů procesu, pomocný kód obnovující stav všech registrů a návratová adresa směřující do pomocného kódu. Při návratu procesu z jádra pokračuje proces prováděním funkce zpracování signálu.
3. Pokud tato funkce skončí normálně příkazem *return*, vrátí se do pomocného kódu uloženého na zásobník a ten obnoví všechny registry a vrátí proces do místa, kdy byl přerušen nebo odkud volal jádro systému.
4. Norma ISO jazyka C dovoluje ukončit ošetřující funkci také voláním standardních funkcí *abort()*, *exit()* nebo *longjmp()*.
5. Ostatní funkce standardní knihovny nesmí být použity (s výjimkou *signal()*).

Problémy:

- nebezpečí ukončení procesu, přestože si proces ošetřuje signál vlastní funkcí (díky 1),
- po dobu provádění ošetřující funkce není zablokováno doručování a zpracování signálů stejného typu, takže ošetřující funkce může být vyvolána několikanásobně,
- nereentrantnost standardní knihovny,
- nutnost ošetřit všechna volání standardní knihovny (a systému) testem na chybový návrat a *errno*==EINTR a opakovat volání,
- nelze bezpečně čekat na příchod signálu:

```
if (sig == 0) pause();
```

Proč není bezpečné? (viz lednice v kap. 3)

7.12 Signály standardu POSIX 1003.1

```
#include <signal.h>
int sigaction(int sig,
    const struct sigaction *act,
    struct sigaction *oact);
```

Nastavení zpracování signálu *sig* na *act*, je vráceno předchozí nastavení v *oact*.

```
struct sigaction {
 void (*sa_handler)(int); /* ošetřující funkce*/
 sigset_t sa_mask;
        /* signály blokované při zpracování */
 int sa_flags;     /* příznaky zpracování */
 void (*sa_sigaction)(int, siginfo_t *, void *);
 /* ošetřující funkce (XPG4E, POSIX 1003.1b) */
};
```

- Po dobu provádění ošetřující funkce je automaticky blokováno zpracování téhož signálu, lze zadat, které další mají být taky blokovány.
- Nastavení zpracování signálu je ponecháno (nevrací se na SIG_DFL).
- Real-time signály POSIX 1003.1b - fronta signálů, zachován počet zaslání (klasické - jen jeden příznak na každý signál)
- SA_RESTART – automatický restart přerušených volání jádra
- SA_SIGINFO – použít sa_sigaction

Operace s množinou signálů:

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

Bezpečné čekání na příchod signálu

Součástí stavu každého procesu (resp. vlákna) je množina *blokovaných signálů*. Zpracování signálů z této množiny je blokováno trvale, lze nastavit funkcí *sigprocmask()* (proces), resp. *pthread_sigmask()* (vlákno):

```
#include <signal.h>
int sigprocmask(int how,
    const sigset_t *newmask, sigset_t *oldmask);
int pthread_sigmask(int how,
    const sigset_t *newmask, sigset_t *oldmask);
```

set - blokované signály, *oset* - předcházející nastavení, *how*:
SIG_SETMASK nastavit podle množiny newmask,
SIG_BLOCK přidat newmask k blokovaným signálům,
SIG_UNBLOCK vyjmout newmask z blokovaných signálů.

Pro klasické a jednovláknové procesy je určen *sigprocmask()*, pro vícevláknové *pthread_sigmask()*. Nastavení *sigprocmask()* platí pro celý proces a dědí se při *fork()*, nastavení *pthread_sigmask()* platí pro dané vlákno a dědí se při spuštění nového vlákna.

Nyní lze bezpečně zablokovat příchod signálu a otestovat, zda byl signál doručen a ošetřen, ale potřebujeme ještě atomicky povolit příchod signálu a zahájit čekání. K tomu je doplněna funkce *sigsuspend()*, resp. *sigwait()*:

```
#include <signal.h>
int sigsuspend(const sigset_t *sigs);
int sigwait(const sigset_t *sigm, int *sig);
```

Funkce *sigsuspend()* atomicky nastaví množinu blokovaných signálů pro volající vlákno na *sigs* a pozastaví jej. Pozastavení je ukončeno příchodem signálu a jeho ošetřením. Po zpracování

signálu ošetřující funkcí obnoví původní množinu blokovaných signálů a vrátí se.
Funkce *sigwait()* čeká na příchod některého signálu z množiny *sigm*, signál odebere (ošetřující funkce není vyvolána), skončí úspěšně (výsledek je 0) a vrátí číslo signálu v návratovém parametru *sig*.

Příklad: Čekání na příchod signálu

Pro spolehlivé čekání je třeba nejprve zablokovat signál, na který se má čekat funkcí, otestovat, zda mezitím nepřišel a teprve pak zahájit čekání na příchod funkcí *sigsuspend()*:

```
/* celočíselný typ s atomickým přístupem */
volatile sig_atomic_t sig;

/* ošetření signálu jen nastaví příznak */
void sigfunc(int signo)
{
    sig = 1;
}

struct sigaction sa;
sigset_t sb;
...
/* maska pro zablokování zvoleného signálu */
sigemptyset(&sb);
sigaddset(&sb, SIGUSR1);
/* nyní je zpracování signálu zablokováno */
sigprocmask(SIG_BLOCK, &sb, NULL);

/* nastavení ošetření signálu */
sa.sa_handler = sigfunc;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
if (sigaction(SIGUSR1, &sa, NULL) == -1) {
    perror("sigaction"); return(1);
}
/* dokud nepřijde čekáme s povoleným zpracováním
*/
while (sig == 0 && sigsuspend(&sa.sa_mask) == -1
    && errno == EINTR) /* cyklit */;

/* signál přišel (nebo chyba) */
```

Jak správně ošetřit EINTR v programu:

```
while (fgets(line, MAXLINE, fin) == NULL &&
    ferror(fin) && errno == EINTR) clearerr(fin);
```

je třeba doplnit u všech volání systému (i nepřímých)!

7.13 Synchronizační prostředky WIN32

1. kritická sekce (mutex) - v rámci jednoho procesu

Inicializace:

```
CRITICAL_SECTION mutex;

InitializeCriticalSection(&mutex);
```

Vstup do kritické sekce (lze rekurzivně):

```
EnterCriticalSection(&mutex);
```

Výstup:

```
LeaveCriticalSection(&mutex);
```

Zrušení:

```
DeleteCriticalSection(&mutex);
```

2. Binární semafor

Vytvoření:

```
HANDLE CreateMutex(                    OpenMutex()
    LPSECURITY_ATTRIBUTES mutexattr,
    BOOL  InitialOwner,                zamčení při vytvoření
    LPCTSTR  Name                      jméno
   );
```

- Pokud je *Name* rovno NULL, vytvoří se lokální objekt.
- Jinak je jméno je globální v rámci systému, nesmí obsahovat \.

Čekání na objekt – zamčení semaforu, rekurzivní, FIFO:

```
DWORD WaitForSingleObject(
    HANDLE handle,                     deskriptor
    DWORD Timeout                      limit v ms
);
```

Limit INFINITE = nekonečný časový limit

Obecně lze čekat na vstup, zprávu o změně stavu souborů, událost, mutex, proces, vlákno a semafor

Uvolnění objektu - pouze vlákno, které zamklo:

```
BOOL ReleaseMutex(HANDLE handle);
```

3. Obecný semafor

Vytvoření:

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES sematttr,
    LONG  InitialCount,          počáteční hodnota
    LONG  MaximumCount,          maximální hodnota
    LPCTSTR  Name                jméno
);
```

Zamčení - snížení hodnoty semaforu:
WaitForSingleObject() čeká na nabytí nenulové hodnoty a pak dekrementuje

Uvolnění - zvýšení hodnoty semaforu, bez omezení:

```
BOOL ReleaseSemaphore(
    HANDLE handle,
    LONG ReleaseCount,           o kolik
    LPLONG PreviousCount         předcházející hodnota
);
```

4. Událost

Vytvoření:

```
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES eventattr,
    BOOL ManualReset,            nerušit událost při wait
    BOOL InitialState,           počáteční stav
    LPCTSTR Name                 jméno
);
```

Čekání na událost:

WaitForSingleObject()

Zaslání události:

```
BOOL SetEvent(HANDLE handle);
```

- pro manual-reset = při signalizaci probudí všechny čekající a signál zůstává nastaven
- pro auto-reset = při signalizaci probudí jednoho čekajícího, pokud není žádný, zůstane čekat na převzetí

```
BOOL PulseEvent(HANDLE handle);
```

- pokud nikdo nečeká - nic
- manual-reset = zaslání události s odblokování všech čekajících a zrušení události
- auto-reset = odblokování jednoho čekajícího

Zrušení signalizace (pro ManualReset):

```
BOOL ResetEvent(HANDLE handle);
```

```
DWORD WaitForMultipleObjects(
    DWORD  Objects,    počet
    HANDLE *handles,   pole deskriptorů
    BOOL   WaitAll,    čekat na všechny
    DWORD  Timeout     časový limit v ms
);
```

Při čekání na všechny je stav objektů při čekání odložen na splnění podmínky u všech čekajících.

Problém: neexistuje atomická kombinace mutexu a čekání.
After years of repeatedly seeing Win32 implementations of condition variables posted in newsgroups like comp.programming.threads it became apparent that many Win32 implementations are either incorrect or contain subtle problems that can lead to starvation, unfairness, or race conditions.

Příklad „řešení“ pomocí Event: (nesprávný)

```
/* auto-reset event */
cond = CreateEvent(NULL, FALSE, FALSE, NULL);
InitializeCriticalSection(&mutex);
...
/* pthread_mutex_lock */
EnterCriticalSection(&mutex);

/* pthread_cond_wait(cond, mutex); */
LeaveCriticalSection(&mutex);
WaitForSingleObject(cond, INFINITE);
EnterCriticalSection(&mutex);
...
/* pthread_cond_broadcast(cond) */
PulseEvent(cond);
```

Chyba: signál se může ztratit, pokud nestihne po opuštění kritické sekce zavolat *WaitForSingleObject()*

Microsoft navíc nyní *PulseEvent()* oficiálně nedoporučuje:
" This function is unreliable and should not be used. A thread waiting on a synchronization object can be momentarily removed from the wait state by a kernel-mode APC, and then returned to the wait state after the APC is complete. If the call to PulseEvent occurs during the time when the thread has been removed from the wait state, the thread will not be released because PulseEvent releases only those threads that are waiting at the moment it is called. Therefore, PulseEvent is unreliable and should not be used by new applications."

Problém – při odblokování pomocí *SetEvent()* je třeba pro *cond_broadcast()* použít manual-reset a pak máme problém, jak bezpečně propustit jen právě čekající.

Řešení 1: spolehlivé jedině s použitím manual-reset, *SetEvent()* a alokací Event (nebo Semaphore) pro každé čekající vlákno (velké množství prostředků).

Řešení 2: co obecný semafor?

```
c_wait = 0;
cond = CreateSemaphore(NULL, 0, INT_MAX, NULL);
InitializeCriticalSection(&mutex);
...
/* pthread_mutex_lock */
EnterCriticalSection(&mutex);

/* pthread_cond_wait(cond, mutex); */
c_wait++;
LeaveCriticalSection(&mutex);
WaitForSingleObject(cond, INFINITE);
EnterCriticalSection(&mutex);
c_wait--;
...
EnterCriticalSection(&mutex);
...
/* pthread_cond_broadcast(cond) */
if (c_wait) {
    RelaseSemaphore(cond, c_wait, &pval);
}
LeaveCriticalSection(&mutex);
```

Problém: řešení není ekvivalentní POSIX – je možné předbíhání (počet čekajících je inkrementován, ale není zahájeno čekání, nyní je inkrementován semafor, pak někdo jiný může projít přes *cond_wait()*) a signalizace musí být v kritické sekci střežené mutexem (správné řešení viz pthreads-win32).

Řešení 3: Od Vista jsou doplněny CONDITION_VARIABLE (ve spojení s CriticalSection nebo SlimRWlock).

7.14 Zasílání zpráv

synchronní - odesílatel čeká na přijetí zprávy příjemcem (CSP)
asynchronní - bez čekání, proces může pokračovat

Adresace:

explicitní (přímá):
send(p, msg), *receive(q, msg)* - *p*, *q* jsou procesy
implicitní:
send(msg), *receive(msg)* - komukoli, od kohokoli
nepřímá:
send(m, msg), *receive(m, msg)* - *m* je schránka, port

Problémy implementace:
- priorita zpráv a výběr podle priority
- velikost zpráv a efektivní kopie mezi adresovými prostory
- vyrovnávací paměť (0 = rendezvous)

XPG4/SVR3 - *msgsnd()*, *msgrcv()*
POSIX 1003.1b - *mq_send()*, *mq_receive()*
sockets - *sendmsg()*, *recvmsg()*
STREAMS - *putmsg()*, *getmsg()*
MPI (Message Passing Interface) - *MPI_Send()*, *MPI_Recv()*

Příklad: vzájemné vyloučení pomocí asynchronního zasílání

```
/* init: zaslání zprávy do schránky lock */
send(lock, msg);

while (1) {
    receive(lock, msg);
    KS;
    send(lock, msg);
    ...
}
```

1. Zasílání zpráv

P1 P2

```
send(addr, port, msg);
                          receive(addr, port, msg);
                          send(addr, port, reply);
receive(addr, port, reply);
```

+ jednoduché
- volný formát zpráv
- různé implementace
- zpracování chyb
- závislost na komunikačním mechanismu

2. Vzdálené volání podprogramů (RPC - Remote Procedure Call)

Klasické zasílání zpráv - jednosměrné
Volání podprogramů - zaslání parametrů a převzetí výsledku

P1 P2

```
res=func(arg1,…, argn);   int func(arg1, …, argn)
                          {
                                …
                                return res;
                          }
```

Implementace (zabalení, client/server stubs):

```
int func(arg1,..., argn)        while (1) {
{
  zabalení parametrů
  send(port, parametry);        receive(port, parametry);
                                rozbalení parametrů
                                res = func(arg1,...argn);
                                zabalení výsledků
                                send(rport, results);
  receive(rport, params);
  rozbalení výsledků
  return res;
```

+ přijatelná složitost (IDL, XDR)
+ definovaný formát zpráv (IDL)
+ síťová reprezentace dat (XDR)
+ automatické generování pomocného kódu
+ automatické ošetření chyb
+ nezávislé na komunikačním protokolu
- omezená množina typů
- statický procedurální charakter

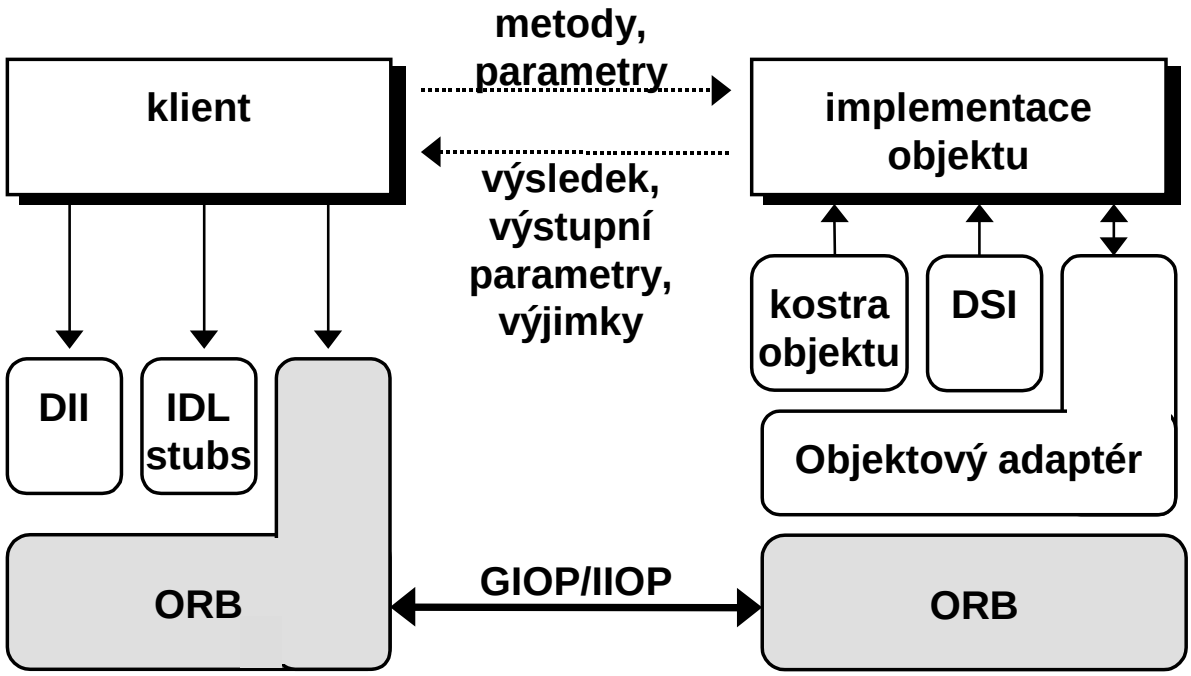
3. Komunikace objektů (CORBA, COM)

P1

```
res=obj->func(arg1,…, argn);
```

P2

```
OBJECT obj;
```



ORB

- zajišťuje předání požadavku klienta implementaci objektu
- volání služeb a implementace služeb je transparentní
- přizpůsobení implementačnímu jazyku zajišťují spojky

IDL stubs

- vygenerováno kompilátorem jazyka IDL
- propojení uživatelského kódu na ORB

- značná složitost (IDL, XDR, uživatelské typy)
+ definovaný formát zpráv (IDL)
+ síťová reprezentace dat (XDR)
+ automatické ošetření chyb
+ uživatelské typy (jednoduché, složené)
+ dynamický charakter
+ komunikují ne programy, ale objekty

Přehled synchronizačních nástrojů

	čekání	**uvolnění**	**komutativní**	**inverze prio**
spin_lock	aktivní	kdokoli	ne	ne
mutex	pasivní	majitel pro rekurzivní	ne	ano
binární semafor	pasivní	kdokoli	ne	ano
číselný semafor	pasivní	kdokoli	ano	ne
condition	pasivní	kdokoli	ne	ne
signál	pasivní	kdokoli	ne	ne
zpráva	pasivní	kdokoli	ano	
událost (win32)	pasivní	kdokoli	ano	ne

podmínka čekání (blokování):

binární semafor	while (true) do wait
obecný semafor	while (zero) do wait
monitor	while (C) do wait
signál	while (not signal) do wait