5. Implementace vzájemného vyloučení

úrovně abstrakce

synchronizační nástroje	zámky, semafory, monitory, zasílání zpráv – na úrovni programovacích jazyků
synchronizační prostředky	čtení/zápis, zakázání přerušení, speciální instrukce – nutné pro implementaci nástrojů

5.1 Jednoprocesorové systémy

Jednoprocesorový systém - úsek kódu je atomický, pokud nemůže dojít k **přepnutí kontextu** (multiprogramování) nebo **přerušení (**souběžný běh ovladače a procesů).

Vzájemné vyloučení mezi:

1) procesy/vlákny jádra a obsluhou přerušení
2) procesy/vlákny v jádře
3) uživatelskými procesy

ad 1) Vzájemné vyloučení proti obsluze přerušení

Jediný způsob je zakázaní přerušení a tím spuštění paralelně vykonávaného kódu.

Příklad: jádro Unixu (1973)

```
int level = spl4(); /* zakázání přerušení<=4 */
```

.... kritická sekce jádra pracující se stejnými daty jako podprogram obsluhy přerušení úrovně 1-4...

```
splx(level);     /* obnovení předchozí úrovně */
```

Problém IA32 - komunikace s řadičem přerušení trvá dlouho!
Pesimistické řešení - předpokládá, že přerušení nastane vždy
Optimistické řešení – neblokuje přerušení v *spl()*, pouze zaznamená úroveň blokování. Pokud přijde přerušení před provedením *splx()*, zablokuje danou úroveň a nastaví příznak čekajícího přerušení, obsluha přerušení se provede při *splx()*

ad 2) Vzájemné vyloučení v rámci jádra

Kdy se přepíná kontext v jádře (jednoprocesorovém)?

- synchronně (zahájení čekání, spuštění jiného procesu) – není problém, datové struktury jsou v konzistentním stavu,
- asynchronně, na externí událost (přerušení od časovače, apod.) – pokud zabráníme, je kód jádra nepreemptivní (nemůže se spustit jiný proces, dokud se předchozí synchronně nezastaví).

a) **Zakázat přerušení** - blokuje přepínání kontextu, ale i V/V, nelze použít na delší úseky kódu (ale nutné pro ad 1)!

b) **Zakázat preemptivní přepínání kontextu v jádře** (Unix).
Kontext se může přepnout pouze:

- synchronně (explicitně, zahájení čekání),
- při ukončení služby jádra před návratem do uživatelského režimu (explicitně),
- po přerušení uživatelského režimu před návratem zpět do uživatelského režimu (jádro neběží, není problém).

Kontext se nesmí se přepnout v obsluze přerušení, ani při návratu z přerušení zpět do systémového režimu!

Jádro = jedna velká kritická sekce

Nevýhoda - delší reakce na události na úrovni procesů (ne RT) - než se spustí potřebný proces (řádově až sekundy).

c) **Vzájemné vyloučení zamykáním datových struktur** a povolení preemptivního přepínání kontextu uvnitř jádra (také pro dlouhodobé vyloučení v klasickém Unixu):

Implementace binárního semaforu (zámku) v jednoprocesorovém systému:

```
struct LOCK {
    volatile sig_atomic_t value;   /* zamčení */
    queue_t queue;      /* fronta procesů */
};

void lock(struct LOCK *lk)
{
    zakázání přerušení
    while (lk->value != 0) {
        insert(&lk->queue, current_pcb);
        switch();       /* plánovač */
    }
    lk->value = 1;
    povolení přerušení
}

void unlock(struct LOCK *lk)
{
    zakázání přerušení
    if (!empty(&lk->queue)) {
        insert(ready, get(&lk->queue));
    }
    lk->value = 0;
    povolení přerušení
}
```

Kontrolní otázka: proč je zde zakázání přerušení?

ad 3) Vzájemné vyloučení mezi uživatelskými procesy

Problém – v uživatelském režimu nelze zakázat přepnutí kontextu, musí se řešit jako pro víceprocesorové systémy

Pozn.: vzájemné vyloučení u vláken je buď případ ad 2) (pokud jsou vlákna implementována na úrovni uživatelské, umíme na úrovni uživatelské zakázat přepínání kontextu mezi vlákny) nebo ad 3) (implementace N:N nebo N:M).

5.2 Víceprocesorové systémy

Vždy nutná synchronizace, nelze obejít (zakázání přerušení nestačí pro zamezení přepnutí kontextu – ostatní procesory běží současně a mohou také provádět kód jádra).

Rozlišení:

1) *spin_lock* krátkodobé vyloučení s aktivním čekáním
2) *mutex*, *lock* dlouhodobé s pozastavením procesu

ad 1) Krátkodobé vzájemné vyloučení

Může střežit pouze kritické sekce, které jsou **krátké, neblokující a bez preempce** (proces/vlákno nesmí být pozastaveno, nesmí na nic zahájit čekání). Aktivní čekání je v tomto případě přijatelné, protože pak může být kritická sekce obsazena pouze procesem běžícím na jiném procesoru a ten ji brzy uvolní. Pozastavení procesu by bylo náročnější než krátké aktivní čekání (a vyžadovalo by opět vzájemné vyloučení). Krátkodobé vyloučení je nutné pro implementaci synchronizačních nástrojů (mutex, semafor, atd.).

a) Implementace pouze čtením/zápisem

- elegantní algoritmy pouze pro malý počet procesů, složitost
- dostupnost lepších speciálních atomických instrukcí

b) Speciální atomické instrukce

Nutná atomická instrukce nedělitelného **čtení a zápisu** (RMW) do paměti (musí korektně fungovat ve víceprocesorovém systému se sdílenou pamětí!)

Instrukce Test&Set (X86-32>=386 LOCK BTS, ...)

```
bool atomic_flag_test_and_set(atomic_flag *ptr)
atomic {
    bool temp = *ptr;
    *ptr = true;
    return temp;
}
```

Implementace krátkodobého vzájemného vyloučení:

```
struct SLOCK {
    atomic_flag lock;  /* zámek, 0/1 */
};

void spin_lock(struct SLOCK *lk)
{
    zakázání přerušení        /* kdy je třeba? */
    while (atomic_flag_test_and_set(&lk->lock)) ;
}

void spin_unlock(struct SLOCK *lk)
{
    atomic_flag_clear(&lk->lock); /* je SEQ_CST*/
    /* nebo lk->lock = 0; MFENCE; */
    povolení přerušení
}
```

Vlastnosti:

- bezpečný, nedochází k uváznutí, blokování
- stárnutí (starving) – je možné předbíhání

Kontrolní otázka: proč je zde zakázání přerušení?

Podobné instrukce Compare&Swap, Swap (XCHG), apod.
Architektury RISC nemají instrukce typu Mem-op-Mem, jak nahradit?

LoadLinked&StoreConditional (RISC)

```
int load_linked(int *m)
{
    global_hw_flag =  m;
    return *m;
}
/* jakýkoli zápis *m global_lock vynuluje */

int store_conditional(int *m, int v)
atomic {
    if (global_hw_flag == m) { /*nikdo nezapsal*/
        *m = v;
        global_hw_flag = 0;
        return 1;       /* úspěšně zapsáno */
    } else return 0;    /* někdo jiný zapsal */
}
```

Implementace vzájemného vyloučení pomocí LL/SC:

```
void spin_lock(struct SLOCK *lk)
{
    do {
        while (load_linked(&lk->lock) == 1) {
            while (lk->lock == 1); /* nutné! */
        }
    } while (store_conditional(&lk->lock, 1)==0);
    MFENCE;     /* konzistence paměti */
}

void spin_unlock(struct SLOCK *lk)
{
    lk->lock = 0;
    MFENCE;     /* konzistence paměti */
}
```

Alpha LDQ_L/STQ_C, MIPS LL/SC, PowerPC LWARX/STWCX (Load Word and Reserve Indexed)

Problémy implementace vzájemného vyloučení

1. cyklus *test&set* (*ll/sc*) stále čte a zapisuje společnou paměť:
   - zatěžuje společnou paměťovou sběrnici
   - atomická instrukce je většinou zároveň paměťová bariéra
   - hyperthreading (SMT) – aktivní čekání blokuje 2. logický procesor na jednom fyzickém jádru CPU
2. stárnutí

Řešení 1:

- při čekání zámek pouze číst (data budou v cache, při zápisu jiným procesorem bude invalidována a pouze pak půjde transakce po společné paměťové sběrnici) – nelze pro HT

```
while (test_and_set(&lk->lock)) {
    while (lk->lock != 0) ;
}
```

- řešení pro HT (Intel P4 Prescott – MONITOR a MWAIT):

```
while (test_and_set(&lk->lock)) {
    MONITOR(&lk->lock);/* monitoruje zápis */
/* MWAIT=HALT logického procesoru dokud
    nebude zapsáno na lk->lock */
    if (lk->lock) MWAIT();
}
```

- lineární backoff (NUMA)

```
while (test_and_set(&lk->lock)) {
    wait = 25;          /* lokální proměnná */
    while (--wait >= 0);
}
```

- exponenciální backoff (NUMA)

```
waiting = 1;
while (test_and_set(&lk->lock)) {
    if (waiting < waiting_limit) waiting*=2;
    wait = waiting;
    while (--wait >= 0);
}
```

Řešení 2:

- **Ticket lock** – vyžaduje atomický *fetch&increment*

```
struct TLOCK {
    atomic_int next;    /* příští t. */
    atomic_int now;     /* aktuální t. */
};

void spin_lock(struct SLOCK *lk)
{
    int my;
    my = atomic_fetch_add(&lk->next, 1); /* my
ticket */
    while (atomic_load(&lk->now) != my) /* spin
wait */;
}

void spin_unlock(struct SLOCK *lk)
{
    atomic_fetch_add(&lk->now, 1);
}
```

Řešení komunikace 1 i stárnutí 2:

- **Anderson Queue lock** – O(pn), atomický *fetch&increment*
- **MCS (Mellor-Crumney, Scott)** – O(p+n), *swap+cmp&swap*

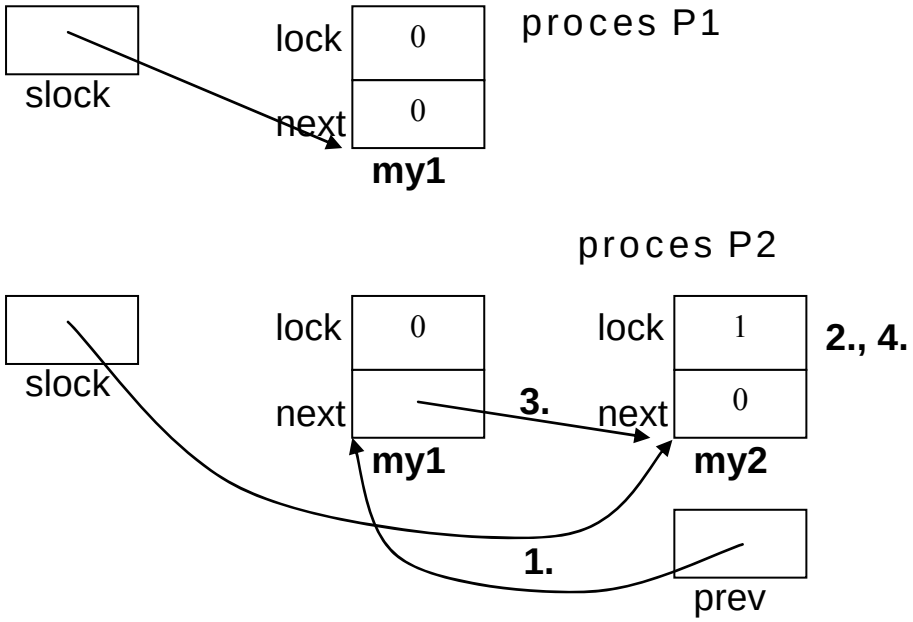
Atomický swap (Intel XCHG):

```
int swap(int *v, int new)
atomic {
    int old = *v; *v = new; return old;
}
```

instrukce Swap(mem, reg);

Atomický compare and swap (Intel CMPXCHGB, od 486)

```
int compare_and_swap(int *v, int old, int new)
atomic {
    if (*v == old) { *v = new; return 1; }
    else return 0;
}
```




```
typedef struct QNODE {
    volatile sig_atomic_t lock;    /* zámek */
    volatile struct QNODE *next;   /* další
čekající */
} qnode_t;
/* zámek je reprezentován hlavičkou fronty */
static volatile qnode_t *slock;
/* položka do seznamu, lokální pro každý proces*/
auto qnode_t mynode;

/*              zámek        položka do seznamu */
void spin_lock(qnode_t **lk, qnode_t *my)
{
    qnode_t *prev;
    my->next = NULL;
    zakázání přerušení
    if ((prev = swap(lk, my)) != NULL) {  1.
        my->lock = 1;                     2.
        prev->next = my;                  3.
        while (my->lock == 1) ;           4.
    }
}
```

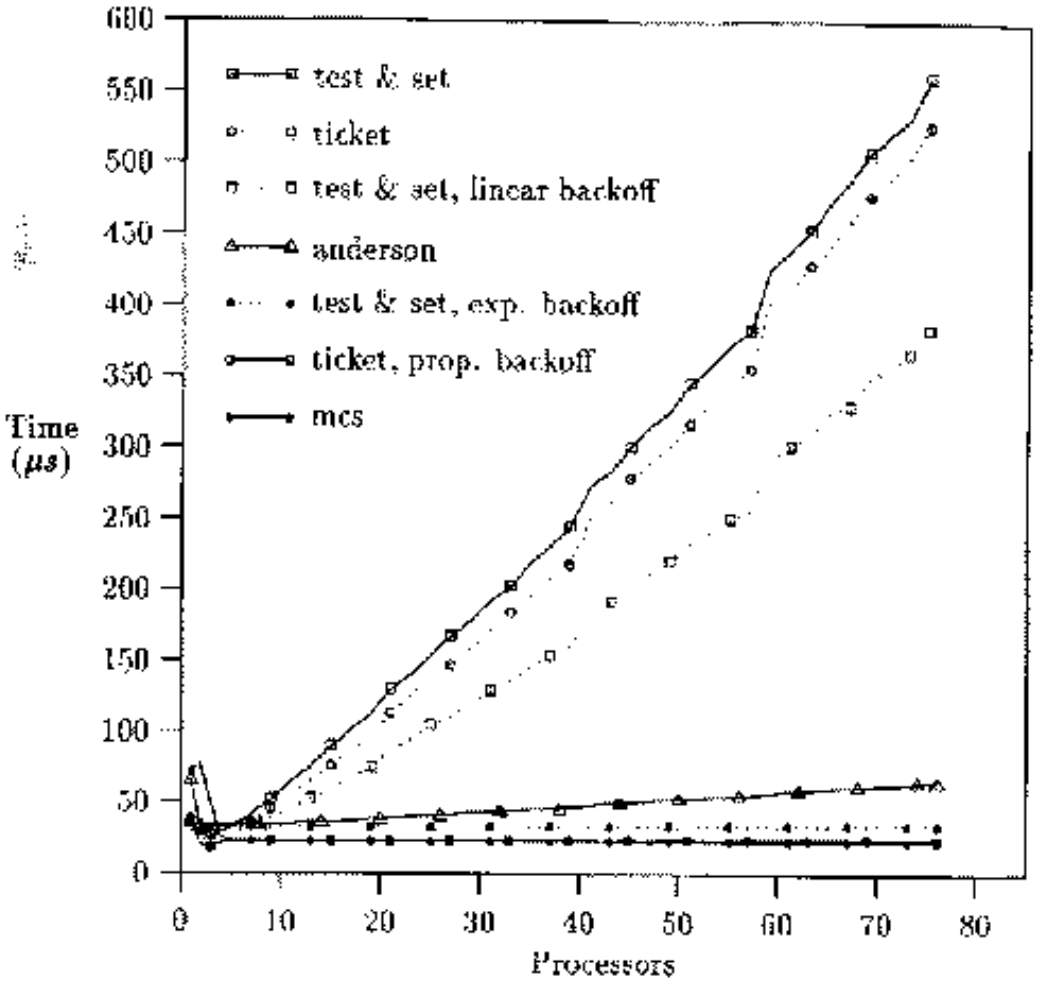
```
void spin_unlock(qnode_t **lk, qnode_t *my)
{
    if (my->next == NULL) { /*na konci (lk==my)*/
        if (compare_and_swap(lk, my, NULL)) {
            povolení přerušení
            return;
        }
    /* vstup je mezi krokem 1. a 3.,
       počkáme až provede krok 3. */
        while (my->next == NULL) ;
    }
    my->next->lock = 0;
}
```

Použití:

```
...
spin_lock(&slock, &mynode);
KS;
spin_unlock(&slock, &mynode);
```

Porovnání časové náročnosti atomické operace a zamykání pro případ sdíleného čítače (counter++ 10 mil. krát):

1. Nesynchronizováno (nefunkční) - 0,03 s.
2. Atomický fetch&add – 0,35 s.
3. Petersonův algoritmus – 2,4 s.
4. Ticket lock – 1,3 s.
5. Posix mutex (test&set s pasivním čekáním) – 2,2 s.
6. SYSV IPC sem – 25 s.



Lock-free (Wait-free) programming

Co když je kritická sekce krátká = pár instrukcí?
Režie zamykání je v tomto případě větší než provedení pár instrukcí.

Problémy:

- Musíme znát paměťové modely a správně synchronizovat obsah paměti - **zámek to udělá za nás**, každý lock/unlock je operace acquire/release z hlediska konzistence paměti.
- Nemáme k dispozici dostatečně silné atomické instrukce, sekvence atomických instrukcí není sama o sobě jako celek automaticky atomická (viz příklady vadných algoritmů vzájemného vyloučení)!

ad 2) Dlouhodobé vyloučení (binární zámek)

```
struct LOCK {
    struct SLOCK lock;      /* zámek kr. sekce */
    int value;              /* příznak zamčení */
    queue_t queue;          /* fronta procesů */
};

void lock(struct LOCK *lk)
{
    zakázání přerušení  /* přepínání kontextu */
    while (1) {
        while (test_and_set(&lk->lock)) {
            while (lk->lock != 0) ;
        }
        if (lk->value == 0) break;
        append(&lk->queue, current_pcb);
        lk->lock = 0;
        switch();  /* pozastavení procesu */
    }
    lk->value = 1;
    lk->lock = 0;
    povolení přerušení
}

void unlock(struct LOCK *lk)
{
    zakázání přerušení
    while (test_and_set(&lk->lock));
        while (lk->lock != 0) ;
    }
    if (!empty(&lk->queue)) {  /* while */
        append(ready, get(&lk->queue));
    }
    lk->value = 0;     /* důležité pořadí */
    lk->lock = 0;
    povolení přerušení
}
```

ad 3) Vzájemné vyloučení v uživatelském režimu

Implementace službou jádra je pomalá, má příliš velkou režii v případě, kdy je kritická sekce volná (99% případů), vždy se musí volat jádro.
Problém implementace binárního zámku v uživatelském procesu tak, aby režie v obvyklém případě (zámek je volný) byla minimální - kombinovaný přístup. Atomickými instrukcemi lze na úrovni uživatelské bezpečně testovat a nastavit příznak, co ale v případě, kdy je nastaven?

- **Nelze aktivně čekat** (není zaručeno, že proces v kritické sekci nebude odstaven od procesoru preemptivním přepnutím kontextu, tím je i krátká kritická sekce efektivně dlouhodobá!). Aktivní čekání v jiných procesech by jen prodloužovalo dobu nedostupnosti (nebyl by prováděn proces v kritické sekci).
- **Nelze pozastavit** (*sleep()*, apod.), protože mezi tím může jiný proces příznak zámku vynulovat (test a zahájení čekání není dohromady atomickou operací!),

Nutná kombinace jednoduchého zámku na úrovni uživatelské s podporou pro pasivní čekání na změnu hodnoty zámku na úrovni jádra (vláknové knihovny) v případě, že nelze zámek zamknout:

```
while (test_and_set(&lk->lock)) { // zámek zamčen
    kernel_wait(&lk->lock);    // pasivní čekání
}
```

- Problém – co když zhavaruje proces, který má zamčené semafory, u kterých nedošlo ke konfliktu (nikdo na ně nečeká, takže o nich jádro neví).

Příklad:
Linux futex (fast mutex):
lk->flag = 0 volno
1 obsazeno, nikdo nečeká
2 obsazeno, čeká alespoň jeden proces na uvolnění

```
void lock(struct LOCK *lk)
{
    int old;
    /* pokud je volno, nastaví na 1 a konec */
    if ((old=compare_and_swap(&lk->flag,0,1))) {
        do {
            if (old == 2 ||  /* už někdo čeká */
            /* nikdo nečeká, nastavíme - nelze
             použít přiřazení, protože mezitím
             mohl být zámek odemčen! */
             compare_and_swap(&lk->flag, 1, 2)) {
                /* čeká až nebude rovno 2 */
                futex_wait(&lk->flag, 2);
        /* čekáme, dokud nebude po probuzení
            volno - nevíme kolik čeká, proto 2 */
        } while ((old =
            compare_and_swap(&lk->flag, 0, 2)));
    }
}

void unlock(struct LOCK *lk)
{
    /* pokud 1, pak zapíše 0 a konec */
    if (compare_and_swap(&lk->flag, 1, 0) == 2) {
        lk->flag = 0;
        /* odblokuje 1 proces čekající na flag */
        futex_wake(&lk->flag, 1);
    }
}
```

Podobné řešení je obvykle součástí podpory synchronizace v POSIX vláknech.