

Synchronizace jádra a multiprocessing

problém synchronizace

příklad 1

proměnná **V** obsahuje počet volných prostředků, např. vyrovnávacích pamětí diskových bloků

proces **A** chce získat volný prostředek a čte **V**, přičemž nechť $V == 1$ a vlákno je přerušeno

proces **B** také čte **V**, přečte hodnotu **1** a sníží hodnotu **V** o **1**, tedy $V == 0$ a začne používat prostředek

proces **A** pokračuje, sníží **V** o **1**, tedy $V == -1$ a také začne používat prostředek

kdyby však proces **A** snížil **V** před přerušením, bylo by vše správné

správnost výsledku tedy závisí na plánování procesů, říkáme, že vznikla soutěž, (*race condition*)

správný výsledek by byl zaručen, kdyby čtení a snížení proměnné **V** o hodnotu **1** bylo nedělitelné, tedy běh procesu by nemohl být mezi čtením a snížením **V** přerušen

obecně je přístup ke společným proměnným bezpečný používáme-li atomické operace

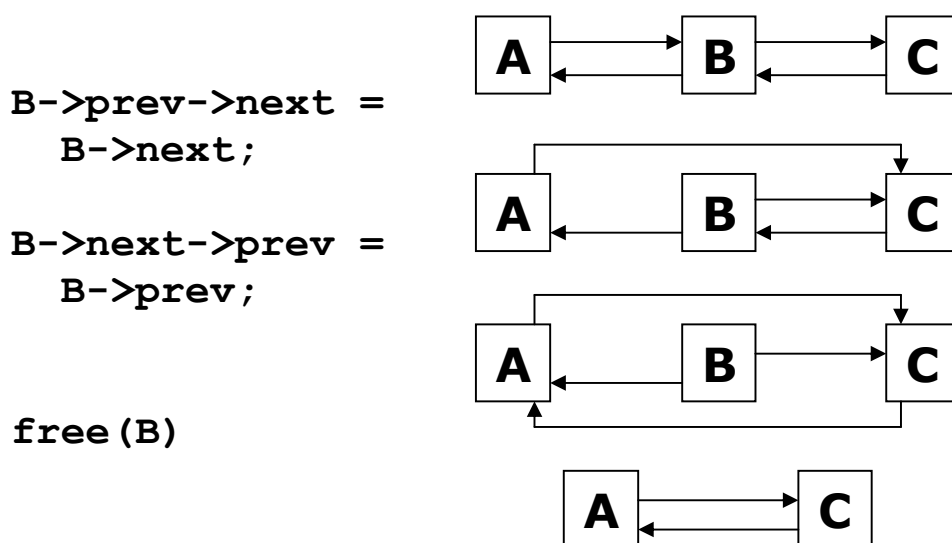
v našem příkladě můžeme (pro jednoprocessorový systém) použít obecné instrukce **dec**, která nastavuje příznak výsledku v registru příznaků

v případě přerušení procesu i bezprostředně po jejím vykonání je

registr příznaků uložen v HW kontextu o po obnovení běhu procesu, máme k dispozici příznak výsledku, je-li výsledek nezáporný, prostředek byl získán a další procesy naleznou správnou hodnotu **V**

příklad 2

deskriptory procesů jsou (cyklicky) obousměrně spojeny a odstranění deskriptoru procesu B ze seznamu potom znamená vykonat



je-li odstraňování B přerušeno po prvním příkazu, údajová struktura zůstane pro další procesy v nekonzistentním stavu

úsek kódu, který může být vykonáván současně nejvíce jedním procesem se nazývá kritická oblast, sekce (*critical region, section*), jinými slovy, každý proces, který začal vykonávat kritickou oblast, musí ji dokončit předtím než ji začne vykonávat jiný proces

jádro

kód jádra je vykonáván

- po volání systému
- při obsluze výjimek
- při obsluze přerušení

přitom se jednotlivá vykonávání jádra mohou přerušovat tak, jak bylo uvedeno v části výpočet v módu jádro

protože sdílejí datové struktury jádra způsobem uvedeným v příkladech, vzniká problém synchronizace výpočtů v jádře

metody synchronizace

při prokládaném vykonávání systémových volání a obsluh výjimek a přerušení musíme zabránit vzniku soutěže nad sdílenými daty

jádro používá tyto metody:

- nepreemptivnost procesů v módu jádro
- atomické operace
- zákaz přerušení
- zamykání

nepreemptivnost procesů v módu jádro

pokud je proces běžící v módu jádro, nemůže se vykonat preempce, tj. nemůže být nahrazen procesem s vyšší prioritou, pokud samotný proces neuvolní procesor

proces v módu jádro může být přerušen obsluhou výjimky nebo přerušení

obsluha přerušení nebo výjimky může být přerušena obsluhou přerušení

neblokující systémová volání jsou atomická vzhledem k ostatním systémovým voláním, mohou bezpečně přistupovat k datům jádra, ke kterým nepřistupují obslužné programy přerušení a výjimek

v blokujícím systémovém volání zanecháme data jádra před
přímým voláním funkce **schedule()** v konzistentním
stavu, po návratu zkontrolujeme jejich hodnoty

atomické operace

operace nad daty vykonaná jednou instrukcí jsou na HW
úrovni atomické

80x86

- instrukce, které přistupují k paměti nejvíce jednou jsou atomické
- čti-modifikuj-zapiš instrukce, např. **inc** nebo **dec**, které čtou data z paměti, modifikují je a aktualizovaná data zapíší zpátky do paměti jsou atomické, jestliže sběrnici mezi čtením a zápisem nezískal jiný procesor nenastane na jednoprocessorovém systému
- čti-modifikuj-zapiš instrukce, kterých instrukční kód má prefix **lock** jsou atomické i na multiprocessorových systémech, řídicí jednotka zamkne sběrnici dokud instrukce není dokončena

maskování přerušení

ne všechny operace nad daty můžeme vykonat atomickými operacemi

data jádra, nad kterými pracují obsluhy přerušení můžeme efektivně zabezpečit tím, že při práci s nimi zakážeme přerušení

maskovatelná přerušení jsou zakázána instrukcí **cli**, kterou vykonávají makra **__cli()** a **cli()** (pro jednoprocessorový systém jsou ekvivalentní)

přerušení jsou povolena instrukcí **sti** a odpovídající makra jsou **__sti()** a **sti()**

uvedené makra nastavují a nulují **IF** příznak registru **eflags**

jádro při vstupu do kritické oblasti nuluje příznak **IF**, na jejím konci ho však nemůžeme obecně přímo nastavit, protože vzhledem na možnost vnoření obsluh přerušení jádro neví jaký byl příznak **IF** před okamžitou obsluhou přerušení

uložíme obsah registru **eflag** makrem

__save_flags(), respektive **save_flags()** a jeho obsah obnovíme makrem **__restore_flags()** respektive **restore_flags()**

```
__save_flags(old) ;  
__cli() ;  
...  
__restore_flags(old) ;
```

příklady maker zakazujících a povolujících přerušení
(parametr lck je pro přerušení ignorován)

```
spin_lock_irqsave(lck, flags)  
    __save_flags(flags) ; __cli()
```

```
spin_unlock_irqrestore(lck, flags)  
    __restore_flags(flags)
```

```
write_lock_irqsave(lck, flags)  
    __save_flags(flags) ; __cli()
```

```
write_unlock_irqrestore(lck, flags)  
    __restore_flags(flags)
```

například funkce `add_wait_queue()` a
`remove_wait_queue()` chrání přístup k frontě
čekajících procesů pomocí funkcí
`write_lock_irqsave()` a
`write_unlock_irqrestore()`

kritické oblasti, vytvořené zákazem přerušení musí být krátké, protože když do nich jádro vstoupí je blokována jakákoli komunikace mezi V/V zařízeními a procesorem

metoda zákazu přerušení není použitelná, když proces se stane spícím, protože je možné, že čeká na událost, která bude oznámená přerušením

řešením v těchto situacích je zamykání

- jádrové semaforey
- kruhové blokování (*spin lock*)

jádrové semaforey (Linux)

když se jádro pokusí o přístup k prostředku, který je obsazen jiným procesem, odpovídající proces přejde do stavu spící a stane se připravený, když je prostředek uvolněn

jádrové semaforey jsou objekty typu **struct semaphore**, který má položky

count

uchovává celočíselnou hodnotu, je-li kladná prostředek je volný jinak je obsazen a absolutní hodnota udává počet čekajících požadavků jádra

wait

uchovává adresu fronty čekajících procesů

waking

zabezpečuje, aby jenom jeden proces po uvolnění prostředku mohl tento získat

položka **count** je dekrementována, když se proces pokouší získat prostředek a inkrementována, když proces uvolní prostředek

inicializace

MUTEX nastaví **count** na hodnotu **1**

MUTEX_LOCKED nastaví na hodnotu **0**

může být libovolné celé kladné číslo

operace **P()**, funkce **down()**

```
void down(struct semaphore * sem)
{
    /* začátek kritické sekce */
    --sem->count < 0;
    if (sem->count < 0) {
        /* konec kritické sekce */
        struct wait_queue wait = {běžící,
                                   NULL};

        běžící->state =
            TASK_UNINTERRUPTIBLE;
        add_wait_queue(&sem->wait,&wait);

        for (;;) {
            unsigned long flags;
            spin_lock_irqsave(
                &semaphore_wake_lock,flags);
            if (sem->waking > 0) {
                sem->waking--;
                break;
            }
            spin_unlock_irqrestore(
                &semaphore_wake_lock,flags);
            schedule();
            běžící->state =
                TASK_UNINTERRUPTIBLE;
        }
        spin_unlock_irqrestore(
            &semaphore_wake_lock,flags);
        běžící->state = BĚŽÍCÍ;
        remove_wait_queue(&sem->wait,
                           &wait);
    }
}
```

operace **V()** , funkce **up()**

```
void up(struct semaphore * sem)
{
    /* začátek kritické sekce */
    ++sem->count;
    if (sem->count <= 0) {
        /* konec kritické sekce */
        unsigned long flags;
        spin_lock_irqsave(
            &semaphore_wake_lock, flags);
        if (atomic_read(&sem->count) <= 0)
            sem->waking++;
        spin_unlock_irqrestore(
            &semaphore_wake_lock, flags);
        wake_up(&sem->wait);
    }
}
```

zabránění vzniku uváznutí

dva procesy **A** a **B** požadují dva prostředky chráněné
semafony **sem1** a **sem2**

```
MUTEX (sem1) ;  
MUTEX (sem2) ;
```

proces A:

```
down (sem1) ;  
...  
down (sem2) ;  
...
```

proces B:

```
down (sem2) ;  
...  
down (sem1) ;  
...
```

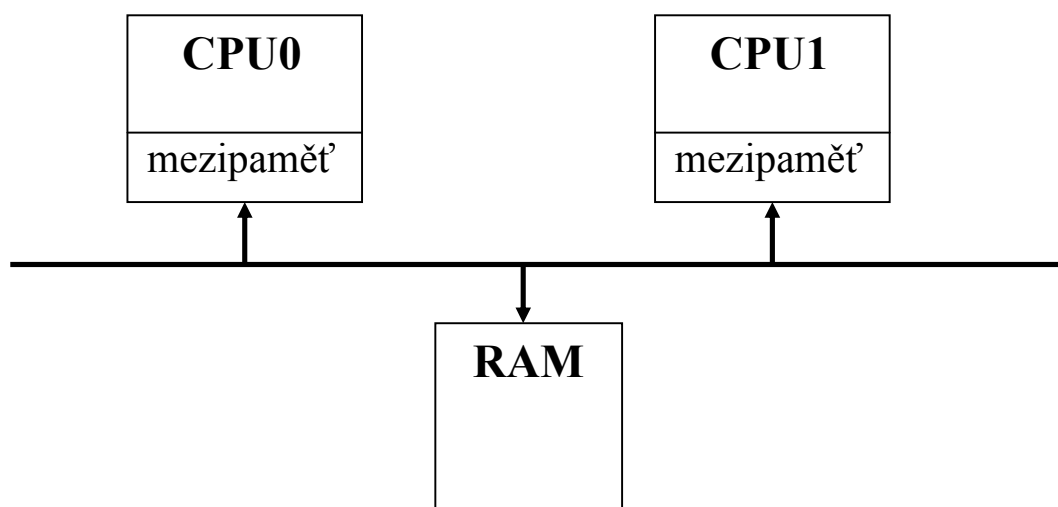
může vzniknout uváznutí

požadavky se vykonají v pořadí adres datových struktur
semaphore

Multiprocessing

v jednoprocessorových strojích má paralelizmus procesů tvar prokládání jejich činnosti

na víceprocesorových strojích můžou být činnosti procesů až do počtu procesorů vykonávány současně



SMP (*symmetrical multiprocessing*) architektura

procesory a sdílená hlavní paměť jsou připojeny ke společné sběrnici

synchronizace mezipaměti (*cache*) procesorů

když procesor modifikuje svou mezipaměť, musí kontrolovat jestli stejná data nejsou v mezipaměti jiného procesoru a když ano, musí je aktualizovat nebo zneplatnit

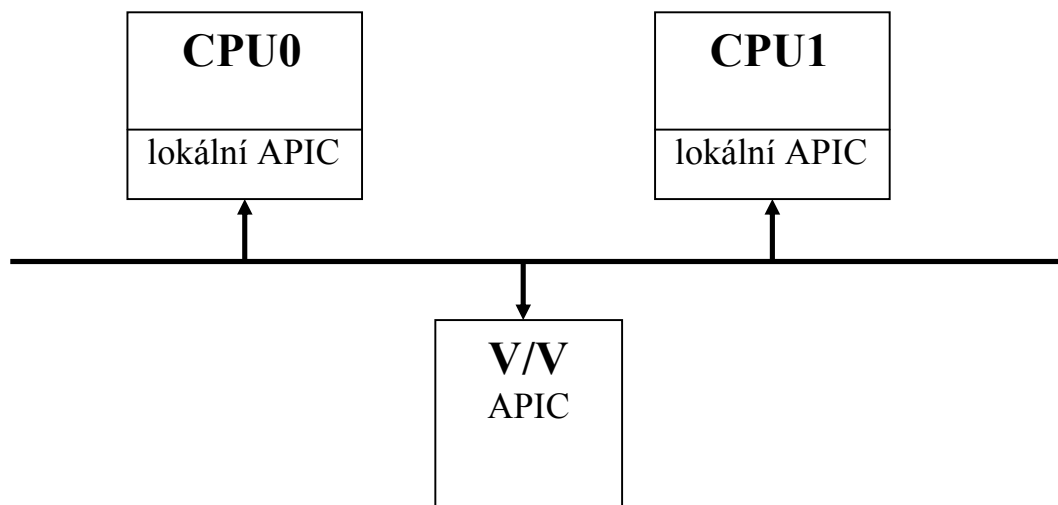
atomické operace

instrukce čti/modifikuj/zapiš na multiprocesorových systémech nejsou atomické, sběrnice se musí zamknout – **lock; dec**

distribuované zpracování přerušení

který procesor obslouží přerušení?

Intel – APIC (*Advanced Programmable Interrupt Controller*)



- přerušení může být směrováno na pevný APIC nebo na APIC procesoru vykonávající proces s nejnižší prioritou, APIC má programovatelný registr obsahující prioritu běžícího procesu

SMP jádro

tradiční řešení synchronizace

- neblokující systémová volání jsou atomická vzhledem k ostatním systémovým voláním
- pro synchronizaci prokládání obsluh přerušení při přístupu ke společným datům stačí maskovat přerušení

multiprocessorový systém

- na více procesorech můžou být současně vykonávána systémová volání
- na více procesorech můžou být současně obsluhována přerušení i když každý z nich maskuje přerušení

řešení: semaforey a kruhové blokování na chránění přístupu ke sdíleným datům, tj. na implementaci kritických sekcí

kruhové blokování

instrukce test-and-set zjistí hodnotu sdílené proměnné, 0 nebo 1, a nastaví ji na hodnotu 1

peudokód

```
spinlock_t TS(spinlock_t *lock) {  
    /* začátek kritické sekce */  
    spinlock_t loc = *lock;  
    *lock = 1;  
    return(loc);  
    /* konec kritické sekce */  
}
```

obvyklá interpretace

lock == 0, prostředek je volný, možno vstoupit do kritické oblasti, proces může pokračovat

lock == 1, prostředek je obsazen, do kritické oblasti není možno vstoupit, proces musí čekat

```
void spin_lock (spinlock_t *lock) {  
    while (TS(lock) != 0) /*zamčeno*/  
        ;                /*cykluj*/  
}
```

```
void spin_unlock (spinlock_t *lock) {  
    *lock = 0;  
}
```


instrukce odpovídající **TS** jsou typu čti-modifikuj-zapiš a sběrnice musí být během jejich vykonávání implicitně nebo explicitně zamčena, navíc pokud hodnota ***lock** byla v mezipaměti jiných procesorů, tyto se při zápisu musely aktualizovat nebo zneplatnit

čekání přepisuje přečtenou hodnotu 1 opět hodnotou 1

spin_lock můžeme zlepšit tím, že jestliže začneme čekat, instrukcí typu čti, která nevyžaduje zamykání sběrnice čekáme v cyklu, než nastane možnost vstoupit do kritické oblasti

```
void spin_lock (spinlock_t *lock) {  
    while (TS(lock) != 0) /*zamčeno*/  
        while(*lock != 0) /*cykluj*/  
            ;  
}
```

kruhové blokování je neefektivní na jednoprocesorových systémech, protože nemá kdo splnit podmínku, na kterou se čeká

je efektivní na multiprocesorových systémech, protože řešení se semaforem, vyžaduje přepnutí kontextu, což je nákladné

Intel x86 (Linux)

makro **spin_lock**, **slp** je adresa zámku

```
1: lock; btsl $0, slp
    jnc 3f
2: testb $1,slp
    jne 2b
    jmp 1b
3:
```

btsl atomicky kopíruje bit 0 z ***slp** do příznaku přenosu (*carry flag*) a potom bit 0 nastaví

makro **spin_unlock**

```
lock; btrl $0, slp
```

btrl atomicky nuluje bit 0 v ***slp**

spin_lock a **spin_unlock** můžou chránit jenom data jádra, ke kterým nikdy nepřistupují obslužné programy přerušení

pro taková data nutno použít makra s maskováním přerušení

příklady maker s kruhovým blokováním pro multiprocesorové systémy (pro jednoprocesorové systémy byla uvedena jenom s maskováním přerušení)

```
spin_lock_irqsave(slp, flags)
```

```
    __save_flags(flags);  
    __cli();  
    spin_lock(slp)
```

```
spin_unlock_irqrestore(slp, flags)
```

```
    spin_unlock(slp);  
    __restore_flags(flags)
```

kruhové blokování pro čtení/zápis **(*read/write spin locks*)**

přístup ke sdíleným datovým strukturám v jádře nemusí být nevyhnutně exklusivní

čtení je možné povolit současně několika výpočtům v jádře

je-li sdílená datová struktura zamčena pro čtení, můžou i další výpočty v jádře tuto strukturu číst, nemůžou však do ní zapisovat

zapisovat však může nejvíce jeden proces, který ji zamkne pro zápis a žádný další proces do ní nemůže zapisovat ani ji číst

možnost současného čtení zvyšuje paralelnost výpočtů jádra

oba zámky jsou realizovány kruhovým blokováním pro čtení/zápis

jsou implementovány 32 bitovým počítadlem ***rwlp**, reprezentující okamžitý počet výpočtů jádra, které čtou chráněnou datovou strukturu

bit v nejvyšším řádu slouží pro kruhové blokování zápisu, je nastaven modifikuje-li jádro datovou strukturu

makro **read_lock**

```
1: lock; incl rwlp
    jns 3f
    lock; decl rwlp
2: cmpl $0, rwlp
    js 2b
    jmp 1b
3:
```

po zvýšení o 1 kontrolujeme jestli zámek nemá zápornou hodnotu, nemá-li pokračujeme na návěští 3, má-li je zamčeno pro zápis, cykluje se dokud nejvyšší bit není 0 a přejde se zpátky na začátek

makro **read_unlock**

```
lock; decl rwlp
```

makro **write_lock**

```
1: lock; btsl $31, rwlp
    jc 2f
    testl $0x7fffffff, rwlp
    je 3f
    lock; btrl $31, rwlp
2: cmp $0, rwlp
    jne 2b
    jmp 1b
3:
```

atomicky se zjistí hodnota nejvyššího bitu a současně se nastaví

byla-li jeho hodnota 1, je zamčeno pro zápis a pokračuje se návěštím 2, kde čekáme dokud hodnota ***rwlp** nebude nula, kdy začneme od začátku

nebylo-li zamčeno pro zápis, zjišťuje se není-li čteno, kdy jsme získali zámek pro zápis, jinak se uvolní zámek pro zápis a čeká se dokud hodnota ***rwlp** nebude nula a opět začneme od začátku

makro **write_unlock**

```
lock; btrl $31, rwlp
```

příklady maker pro zápis s kruhovým blokováním na multiprocesorovém systému

```
write_lock_irqsave(rwlp, flags)
```

```
    __save_flags(flags);  
    __cli();  
    write_lock(rwlp)
```

```
write_unlock_irqrestore(rwlp, flags)
```

```
    write_unlock(rwlp);  
    __restore_flags(flags)
```