Zamykání v kernelu

- Tomáš Pop tomas.pop at seznam.cz
- Připraveno na seminář linux kernel na MFF UK

Zamykání v kernelu

- Dnešní kernel je multitaskový (samozřejmě)
- Potřeba zamykat
  - konkurence
  - reentrance
- Výsledek procesu bez pořádného zamykání se nazývá „Race condition“
- Problémy preemptivního plánování a SMP (symmetric multiprocessing) oproti UP (uni-processor)

Zamykání v kernelu

- Příklad race condition
-

Instance 1	Instance 2
read very_important_count (5)	
	read very_important_count (5)
add 1 (6)	
	add 1 (6)
write very_important_count (6)	
	write very_important_count (6)

Zamykání v kernelu

- Jak tomu předejít??
  - používání atomických operací
    - atomic_t v;
    - atomic_set(&v, 5); /* v = 5 (atomically) */
    - atomic_add(3, &v); /* v = v + 3 (atomically) */
    - atomic_dec(&v); /* v = v - 1 (atomically) */
    - printf("This will print 7: %d\n", atomic_read(&v));
  - Používání synchronizačních primitiv
  - Lockless data acces

Zamykání v kernelu

- Synchronizační primitiva - základní
  - Spinlocks
  - Semaphores
  - Mutexes
- Ostatní
  - Reader/Writer Locks
  - Big-Reader Locks
  - The Big Kernel Lock
  - Completion variables
  - Memory Barriers

Zamykání v kernelu

- Spinlock
  - include/asm/spinlock.h a include/linux/spinlock.h
  - zámek vlastněný jedním procesem
  - aktivní čekání
  - není rekursivní (deadlocky, když to zkusíte)
  - základní užití (i na SMP)

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;
unsigned long flags;
spin_lock_irqsave(&mr_lock, flags);
/* critical section ... */
spin_unlock_irqrestore(&mr_lock, flags);
```

Zamykání v kernelu

- Spinlock
  - Totéž jako na předchozím slidu na UP

```
unsigned long flags;
save_flags(flags);
cli();  //zakazat interrupty
/* critical section ... */
restore_flags(flags);
```

Zamykání v kernelu

- Spinlock
  - Totéž na UP, ale nesměly být před tím zakázané interrupty (nebudou obnoveny)

```
spinlock_t mr_lock =
SPIN_LOCK_UNLOCKED;
spin_lock_irq(&mr_lock);
/* critical section ... */
spin_unlock_irq(&mr_lock);
```

Zamykání v kernelu

- Spinlock
  - Pokud víme, že jsme v user-context kernel code (e.g., a system call)
    - spin_lock()
    - spin_unlock()
  - Bottom halves spin locky (BH je starý název pro softirq)
    - pokud máme kód vně softirq a může být použit i uvnitř softirq
    - spin_lock_bh()
    - spin_unlock_bh()

Zamykání v kernelu

- Spinlock
  - Mohou se použít kdekoliv
  - Aktivní čekání
    - neměly být okolo delších sekcí
    - nic co má spinlock by nemělo zavolat sleep
  - Na zamykání delších sekcí jsou lepší semafory a mutexy
  - Na multiprocesorech jsou někdy jediným řešením

Zamykání v kernelu

- Semaphory
  - include/asm/semaphore.h
  - struct semaphore, obsahuje wait queue a count
  - sema_init(), inicializace
  - použití
    - down_interruptible(), dekrementuje count
      - if (nová hodnota < 0), proces je zařazen do wait queue
      - pokud přijde signál, je vráceno EINTR, a semafor není zamčen
    - up()
      - pokud je nová hodnota větší než 0, jsou probuzeny čekající proc

Zamykání v kernelu

- Semaphory
  - použití
    - down(), podobné jako down_interruptible
      - dává volajícího do spánku, kdy ignoruje signály
    - down_trylock()
  - příklad

```
struct semaphore mr_sem;
sema_init(&mr_sem, 1);      /* usage count is 1 */
if (down_interruptible(&mr_sem))
  /* semaphore not acquired; received a signal ... */
/* critical region (semaphore acquired) ... */
up(&mr_sem);
```

Zamykání v kernelu

- Mutexy
  - nejsou implementovány jako semafory (dříve byly)
  - jsou o něco menší (asi o 4 byty na některch architekturách), lepší pro cachování na procesoru
  - Hlavní rysy
    - Jen jeden proces může držet zámek
    - jen vlastník ho může odemknout
    - několikanásobné odemčení je zakázáno
    - task nemůže skončit, pokud má mutex
    - mutexes nemohou být použity v irq contexts

Zamykání v kernelu

- Mutexy
  - Použití
    - DEFINE_MUTEX(name);
    - mutex_init(mutex);
    - void mutex_lock(struct mutex *lock);
    - int mutex_lock_interruptible(struct mutex *lock);
    - int mutex_trylock(struct mutex *lock);
    - void mutex_unlock(struct mutex *lock);
    - int mutex_is_locked(struct mutex *lock);

Zamykání v kernelu

- Completion Variables
  - Používané podobně jako semafory
  - Jeden nebo více procesů čeká na dokončení nějaké práce na completion variable, když je práce dodělána, jsou probuzeni
  - je reprezentován completion type, linux/completion.h
  - používá je třeba vfork() na vzbuzení otce
  - ????

Zamykání v kernelu

- Reader/Writer Locks
  - **semafory a spinlocky poskytují RW formu**
  - **klasicky, může být mnoho čtenářů, ale jen jeden zapisovatel**

```
rwlock_t mr_rwlock = RW_LOCK_UNLOCKED;
read_lock(&mr_rwlock);
/* critical section (read only) ... */
read_unlock(&mr_rwlock);
write_lock(&mr_rwlock);
/* critical section (read and write) ... */
write_unlock(&mr_rwlock);
```

```
struct rw_semaphore mr_rwsem;
init_rwsem(&mr_rwsem);
down_read(&mr_rwsem);
/* critical region (read only) ... */
up_read(&mr_rwsem);
down_write(&mr_rwsem);
/* critical region (read and write) ... */
up_write(&mr_rwsem);
```

Zamykání v kernelu

- Big-Reader Locks
  - include/linux/brlock.h
  - jsou velice rychlé pro synchronizaci readerů
  - jsou velice pomalé pro synchronizaci writerů
  -

```
br_read_lock(BR_MR_LOCK);
/* critical region (read only) ... */
br_read_unlock(BR_MR_LOCK);
```

Zamykání v kernelu

- The Big Kernel Lock
  - původně z kernelu 2.0, jako jediný SMP lock
  - mělo by se mu co nejvíc vyhýbat, v kernelu 2.6 by měl být jen minimálně
  - je to rekursivní spinlock
  - proces v něm může usnout nebo zavolat scheduler

```
lock_kernel();
/* critical region ... */
unlock_kernel();
```

Zamykání v kernelu

- Preemption control
  - Od 2.5 je kernel preemptivní, s tím může někdy nastat problém
  - mohou se použít
    - preempt_disable()
    - preempt_enable()

Zamykání v kernelu

- Jak a kdy co používat
  - Podle aktuálního kernelu (! ale vše by mělo chodit všem, takže vše by se mělo psát korektně)
    - kernely se liší podle toho jak byly kompilovány
    - CONFIG_SMP # pro viceprocesory
    - CONFIG_PREEMPT # kompilovat kernel jako preemptivni
      - pokud jsou obě vypnuté, nesjou potřeba spinlocky
      - pokud je zapnuté jen CONFIG_PREEMPT spinlocky jen zabraňují preempci
  - Podle toho, odkud můžeme k datům přistoupit
    - z user-contextu (syscally), z timerů, softirq, hardirq,taskletů

Zamykání v kernelu

- Jak a kdy co používat
  - Pokud se k datům přistupuje jen ze syscallů, měli by být použity mutexy nebo semafory
    - používat down_interruptible(), aby nebyl problém se signály
  - Pokud datová struktura může být přistupována ze softirq handlerů a syscalů
    - syscall může být přerušen softirq
    - kritická sekce může být přistupována z jiného CPU
    - použít spin_lock_bh(), zakáže softirq a zamkne zámek

Zamykání v kernelu

- Jak a kdy co používat
  - Pokud datová struktura může být přistupována z handlerů timerů a syscallů
    - timery jsou volán ze sotirq
    - použít spin_lock_bh(), zakáže softirq a zamkne zámek
  - Pokud datová struktura může být přistupována ze taskletů (dynamický softirq, vždy jeho handler běží jen na jednom CPU) a syscallů
    - tasklet je volán ze sotirq
    - použít spin_lock_bh(), zakáže softirq a zamkne zámek

Zamykání v kernelu

- Jak a kdy co používat
  - stejný tasklet/tasklet
    - nemůže nastat ani na multiprocesoru
    - použít spin_lock_bh(), zakáže softirq a zamkne zámek
  - tasklet/jiný tasklet, tasklet/timer
    - spin_lock(), není potřeba spin_lock_bh(), protože jsme v taskletu a nic jiného na tomhle procesoru nepoběží ???
  - stejné softirq/softirq
    - spin_lock(), spin_unlock()
  - různé softirq/softirq
    - spin_lock(), spin_unlock

Zamykání v kernelu

- Jak a kdy co používat
  - hardirq a softirq,
    - spin_lock_irq(), zakáže interrupty na daném CPU
      - na UP to vpodstatě jen použije local_irq_disable()
    - spin_lock_irqsave(), spin_unlock_irqrestore()
  - hardirq a hardirq
    - spin_lock_irqsave()

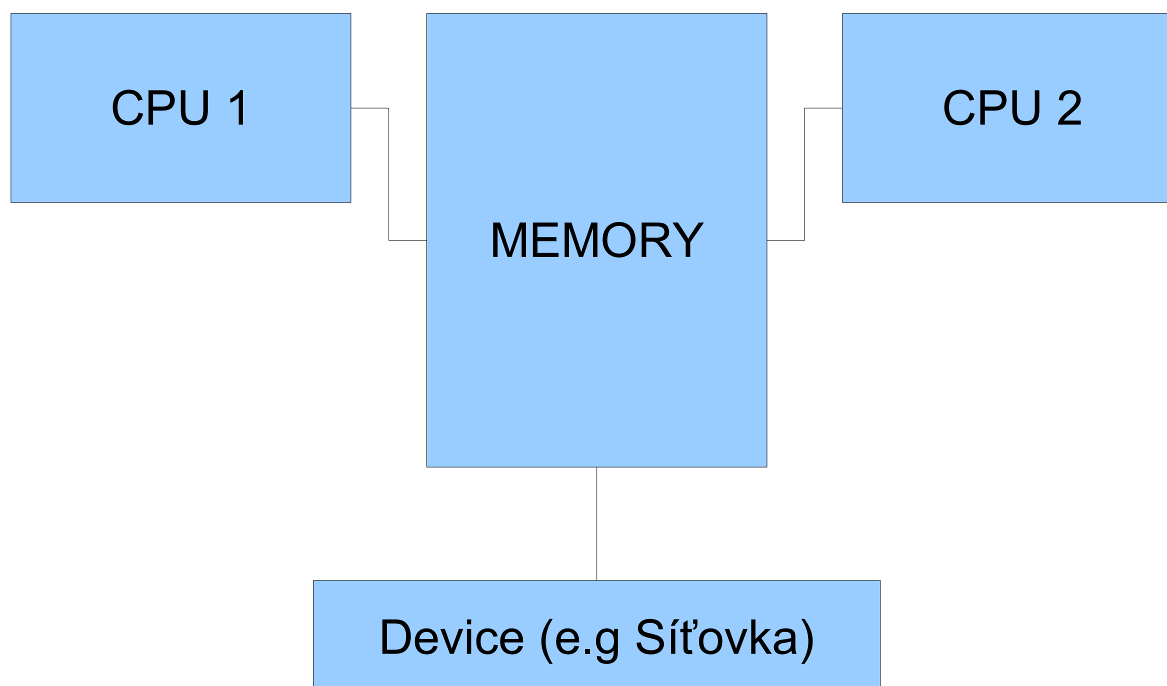
Zamykání v kernelu

- Shrnutí (Minimální použitelné synchronizační primitivum pro kombinaci X Y)

	IRQ Handler A	IRQ Handler B	Softirq A	Softirq B	Tasklet A	Tasklet B	Timer A	Timer B	User Context A	User Context B
IRQ Handler A	None									
IRQ Handler B	spin_lock_irqsave()	None								
Softirq A	spin_lock_irq()	spin_lock_irq()	spin_lock()							
Softirq B	spin_lock_irq()	spin_lock_irq()	spin_lock()	spin_lock()						
Tasklet A	spin_lock_irq()	spin_lock_irq()	spin_lock()	spin_lock()	None					
Tasklet B	spin_lock_irq()	spin_lock_irq()	spin_lock()	spin_lock()	spin_lock()	None				
Timer A	spin_lock_irq()	spin_lock_irq()	spin_lock()	spin_lock()	spin_lock()	spin_lock()	None			
Timer B	spin_lock_irq()	spin_lock_irq()	spin_lock()	spin_lock()	spin_lock()	spin_lock()	spin_lock()	None		
User Context A	spin_lock_irq()	spin_lock_irq()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	None	
User Context B	spin_lock_irq()	spin_lock_irq()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	spin_lock_bh()	down_interruptible	None

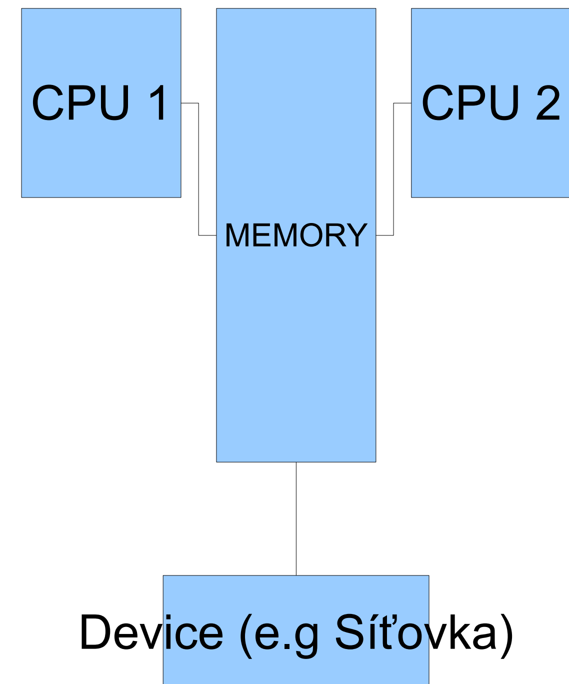
Zamykání v kernelu

- Memory barriers
  - Jde o synchronizaci přístupu do paměti



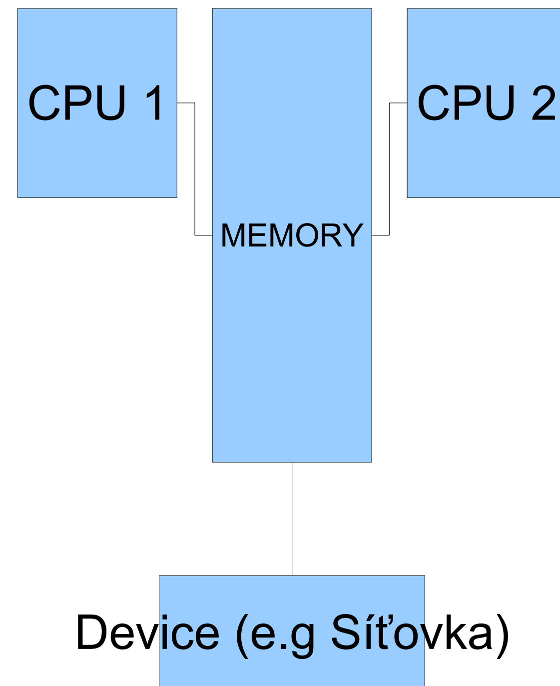
Zamykání v kernelu

- Memory barriers
  - Jde o synchronizaci přístupu do paměti
    - jeden CPU změní data druhému
      - CPU 1 CPU
      - ================
      - { A == 1; B == 2 }
      - A = 3; x = A;
      - B = 4; y = B;
    - Výsledky, které jsou možné
      - x == 1, y == 2
      - x == 1, y == 4
      - x == 3, y == 2
      - x == 3, y == 4



Zamykání v kernelu

- Memory barriers
  - Jde o synchronizaci přístupu do paměti
    - Přístup na device přes data a adress registr
      - *A = 5; // nastavit adres registr na 5
      - x = *D; // do x prescist data na adrese 5 (v data registru)
    - Výsledek
      - STORE *A = 5, x = LOAD *D
      - x = LOAD *D, STORE *A = 5



Zamykání v kernelu

- Memory barriers
  - cíl : garantovat pořadí přístupů do paměti
  - Požadavky, které se kladou na systém, aby MB fungovali:
    - Na každém CPU jdou Memory access za sebou jak mají
    - Některé další předpoklady
      - Nezávislé data accessy mohou být přeuspožádany libovolně
      - X = *A; Y = *(A + 4); může dopadnout libovolnou z následujících variant
        - X = LOAD *A; Y = LOAD *(A + 4);
        - Y = LOAD *(A + 4); X = LOAD *A;
        - {X, Y} = LOAD {*A, *(A + 4) };
  - http://kerneltrap.org/node/6431

Zamykání v kernelu

- Memory barriers
  - Typy barrier
    - Write (or store) memory barriers
      - všechny STORE operace před barierou budou provedeny (budou viditelné) dříve než všechny STORE za barrierou
    - Data dependency barrier
      - Pro případy typu X = &Y; Z = *X;
      - zajišťují, že to, co je na pozici kam ukazuje X bude updatovano, než to zkusíme číst.
    - Read (or load) memory barrier
      - DDB a a garance, ze všechny LOAD před barrierou budou vidět dřív než ty za barrierou
    - General memory barriers
      - vše před bude vidět dřív než vše za barrierou

Zamykání v kernelu

- Memory barriers
  - Co od nich nejde čekat
    - Nic negarantuje, že memory accessy před barierou budou dokončené ve chvíli dokončení MB
    - Nic negarantuje, že MB na jednom CPU bude mít přímý vliv na data na jiném CPU (viz příklad později)
    - Neexistuje garance, že NĚCO JINÉHO NEŽ PROCESOR přerovná přístupy do paměti (jiný hardware)

Zamykání v kernelu

- Memory barriers
  - Příklad
    - CPU 1 CPU 2
    - =================
    - { A == 1, B == 2, C = 3, P == &A, Q == &C }
    - B = 4;
    - <write barrier>
    - P = &B
    - Q = P;
    - D = *Q;
      - co když CPU 2 uvidí zápis do B později než zápis do P?

Zamykání v kernelu

- Memory barriers
  - Příklad
    - CPU 1 CPU 2
    - ================
    - { A == 1, B == 2, C = 3, P == &A, Q == &C }
    - B = 4;
    - <write barrier>
    - P = &B
    - Q = P;
    - D = *Q;
      - co když CPU 2 uvidí zápis do B později než zápis do P?
        - (Q == &A) implies (D == 1)
        - (Q == &B) implies (D == 4)
      - Ale taky (to pry realne je, třeba na Alphach)
        - (Q == &B) and (D == 2)

Zamykání v kernelu

- Memory barriers
  - Řešení
  -
    - CPU 1 CPU 2
    - \+ =============== ===============
    - \+ { M[0] == 1, M[1] == 2, M[3] = 3, P == 0, Q == 3 }
    - \+ M[1] = 4;
    - \+ <write barrier>
    - \+ P = 1
    - \+ Q = P;
    - \+ <data dependency barrier>
    - \+ D = M[Q];
  - Obecně bariery se párují (W/DD, W/R, ALL/ALL)

Zamykání v kernelu

- Memory barriers
  - Implementace
    - Compiler barriers
      - barrier();, obecná bariera
    - CPU memory barriers
      - general mb() , wmb(), rmb() - RW/barriers, read_barrier_depends() - data-dependency barrier.
      - Mají i SMP varianty smp_xxx(), které něco dělají jen na SMP
    - MMIO memmory barriers
      - pro memory- mapped operace write
      - mmiowb();
  -

Zamykání v kernelu

- Memory barriers
  - implicitní memmory barrier
    - vpodstě pomocí zámků
    - zámky viz dříve.
      - *A = a;
      - *B = b;
      - LOCK
      - *C = c;
      - *D = d;
      - UNLOCK
      - *E = e;
      - *F = f;

Zamykání v kernelu

- RCU, Read Copy Update,
  - lockless data acces
  - dá se s ní předejít READ zámků
    - tedy WRITERS musí být schopni je měnit „tak, aby to nevadilo“
      - new->next = list->next;
      - wmb();
      - list->next = new;
    - Odpojení ještě jednodušší
      - list->next = old->next;

Zamykání v kernelu

- RCU, Read Copy Update,
  - Kdy můžeme zlikvidovat odpojený element
    - Problematické
    - Ve knihovné list include/linux/list.h je funkce call_rcu()
      - pro registrování funkce, která to zlikviduje, až skončí všichni readři
      - Pozná to, podle toho, že si readři zamykají
        - rcu_read_lock()
        - rcu_read_unlock()
      - nemohou u toho usnout, pokud se přeplánovává, může se zavolat callback procedura
    - http://www.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/x490.html

Zamykání v kernelu

- Konec

Zamykání v kernelu

- Odkazy
  - http://www.kernel.org/pub/linux/kernel/people/rusty/kernel-locking/
  - http://www.linuxjournal.com/article/5833
  - http://www.linuxjournal.com/article/5600
  - http://kerneltrap.org/node/6431 mem bar
  -