CS632/SEP564: Embedded Operating Systems (Fall 2008)

Kernel Synchronization

KAIST

Atomic Operations (1)

- **What is "atomic operations"?**
  - Execute atomically – without interruption.
  - Guarantee consistency but not order.
  - Impossible to race.

- **Atomic operations**
  - Kernel provides two atomic operations
    - Integers
    - Individual bits
  - Implementation is very architecture-specific.
    - On most architecture (including x86), a word-size read/write is atomic.

Atomic Operations (2)

- **atomic_t**
  - <asm/atomic.h>
  - typedef struct { volatile int counter; } atomic_t;

- **Why another type?**
  - Enable type checking during atomic operations and function calls
  - Prevent compiler from optimizing access to the variable.
  - Hide any architecture-specific differences in its implementation

Atomic Operations (3)

- **Using atomic integer operations**
  - #define ATOMIC_INIT(i) { (i) }
  - #define atomic_set(v,i) (((v)->counter) = (i))
  - #define atomic_read(v) ((v)->counter)

  - void atomic_add(int i, atomic_t *v);
  - void atomic_sub(int i, atomic_t *v);
  - void atomic_inc(atomic_t *v);
  - void atomic_dec(atomic_t *v);

Atomic Operations (4)

- **Using atomic integer operations (cont'd)**
  - int atomic_sub_and_test(int i, atomic_t *v);
    - Atomically subtract i from v and return true if the result is zero; otherwise false.
  - int atomic_add_and_negative(int i, atomic_t *v);
    - Atomically add i to v and return true if the result is negative; otherwise false.
  - int atomic_dec_and_test(int i, atomic_t *v);
    - Atomically decrement v by one and return true if zero.
  - int atomic_inc_and_test(int i, atomic_t *v);
    - Atomically increment v by one and return true if zero.

Atomic Operations (5)

- **Atomic integer operations implementations**
  - IA32 implementations

```
void atomic_add(int i, atomic_t *v)
{
   __asm__ __volatile__ (
      LOCK “addl %1,%0”
      :“+m” (v->counter)
      :“ir” (i));
}
void atomic_inc(atomic_t *v)
{
   __asm__ __volatile__ (
      LOCK “incl %0”
      :“+m” (v->counter));
}
```

Atomic Operations (6)

- **Using atomic bitwise operations**
  - <asm/bitops.h>
  - void set_bit(int nr, void *addr);
  - void clear_bit(int nr, void *addr);
  - void change_bit(int nr, void *addr);
  - int test_and_set_bit(int nr, void *addr);
  - int test_and_clear_bit(int nr, void *addr);
  - int test_and_change_bit(int nr, void *addr);
  - int test_bit(int nr, void *addr);
  - The nonatomic form: __test_bit(), etc.
    - Faster, when you already have a lock.

Atomic Operations (7)

- **Atomic bitwise operations implementations**
  - IA32 implementations

```
void set_bit(int nr, volatile unsigned long *addr)
{
    __asm__ __volatile__ (
        LOCK “btsl %1,%0”
        :“+m” (ADDR)
        :“Ir” (nr));
}

int test_and_set_bit(int nr, volatile unsigned long *addr)
{
    int oldbit;
    __asm__ __volatile__ (
        LOCK “btsl %2,%1\n\tsbbl %0,%0”
        :“=r” (oldbit), “+m” (ADDR)
        :“Ir” (nr) : “memory”);
    return oldbit;
}
```

Interrupt Disabling

- **Local interrupt disabling**
  - Effective to protect critical section for UP system.
  - Do not protect against concurrent accesses to data structures by interrupt handlers running on the other CPUs.

  - void local_irq_disable();
  - void local_irq_enable();
  - void local_irq_save();
  - void local_irq_restore();
  - int local_irq_disabled();

Spin Locks (1)

- **Spin lock**
  - A lock that can be held by at most one thread of execution.
    - Spins while waiting for the lock to become available.
    - Wasteful if a spin lock is held for a long time.
  - Spin locks can be used in interrupt handlers.
    - Threads do not block.
  - Spin locks are not recursive.
    - If you attempt to acquire a lock you already hold, you will spin.
    - If a lock is used in an interrupt handler, disable local interrupts before obtaining the lock.
    - You need to disable interrupts only on the current processor.
  - On UP, locks are compiled away and do not exist.

Spin Locks (2)

- **Using spin locks**
  - <linux/spinlock.h>
  - void spin_lock_init(spinlock_t *lock);
  - void spin_lock(spinlock_t *lock);
  - void spin_unlock(spinlock_t *lock);
  - void spin_lock_irq(spinlock_t *lock);
  - void spin_unlock_irq(spinlock_t *lock);
  - void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
  - void spin_unlock_irqrestore(spinlock_t *lock,
    unsigned long flags);
  - int spin_trylock(spinlock_t *lock);

Reader-Writer Spin Locks (1)

- **Reader-writer spin locks**
  - Used when accesses to a data structure can be clearly divided into readers and writers.
  - Provide separate reader and writer variants of the lock.
  - One or more readers can concurrently hold the reader lock.
  - The writer lock can be held by at most one writer with no concurrent readers.
  - A read lock can not be upgraded into a write lock.
  - Linux favors readers over writers.
    - If the read lock is held and a writer is waiting, readers that attempt to acquire the lock will continue to succeed.

Reader-Writer Spin Locks (2)

- **Using reader-writer spin locks**
  - <linux/spinlock.h>
  - void rw_lock_init(rwlock_t *lock);
  - void read_lock(rwlock_t *lock);
  - void read_unlock(rwlock_t *lock);
  - void write_lock(rwlock_t *lock);
  - void write_unlock(rwlock_t *lock);
  - void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
  - void read_unlock_irqsave(rwlock_t *lock,
    unsigned long flags);
  - void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
  - void write_unlock_irqsave(rwlock_t *lock,
    unsigned long flags);

Semaphores (1)

- **Semaphores**
  - Sleeping locks
  - Semaphores are well suited to long critical sections.
  - Semaphores allow for an arbitrary number of simultaneous lock holders.
    - Binary semaphore or mutex: count = 1
    - Counting semaphore: count = n
  - Semaphores can be obtained only in process context because interrupt context is not schedulable.
  - Unlike spin locks, semaphores do not disable kernel preemption and, consequently, code holding a semaphore can be preempted.

Semaphores (2)

- **Creating and initializing semaphores**
  - <asm/semaphore.h>
  - DECLARE_SEMAPHORE_GENERIC(name, count);
  - DECLARE_MUTEX(name);
  - DECLARE_MUTEX_LOCKED(name);

  - void sema_init(struct semaphore *sem, int val);
  - void init_MUTEX(struct semaphore *sem);
  - void init_MUTEX_LOCKED(struct semaphore *sem);

Semaphores (3)

- **Using semaphores**
  - void down(struct semaphore *sem);
    - If unavailable, sleeps in the TASK_UNINTERRUPTIBLE state.
  - int down_interruptible(struct semaphore *sem);
    - If unavailable, sleeps in the TASK_INTERRUPTIBLE state.
    - If the task receives a signal while waiting for the semaphore, it is awakened and down_interruptible() returns –EINTR.
  - int down_trylock(struct semaphore *sem);
    - Try to acquire the given semaphore and immediately return nonzero if it is contended.
  - void up(struct semaphore *sem);
    - Release the given semaphore and wakes up a waiting task, if any.

Reader-Writer Semaphores

- **Using reader-writer semaphores**
  - <asm/rwsem.h>
  - DECLARE_RWSEM(name);
  - void init_rwsem(struct rw_semaphore *sem);
  - void down_read(struct rw_semaphore *sem);
  - int down_read_trylock(struct rw_semaphore *sem);
  - void down_write(struct rw_semaphore *sem);
  - int down_write_trylock(struct rw_semaphore *sem);
  - void up_read(struct rw_semaphore *sem);
  - void up_write(struct rw_semaphore *sem);
  - void downgrade_write (struct rw_semaphore *sem);

Spin Locks vs. Semaphores

- **What to use?**

Requirement	Recommended lock
Low overhead locking	Spin lock is preferred.
Short lock hold time	Spin lock is preferred.
Long lock hold time	Semaphore is preferred.
Need to lock from interrupt context	Spin lock is required.
Need to sleep while holding lock	Semaphore is required.

Completion Variables

- **Using completion variables**
  - <linux/completion.h>
  - DECLARE_COMPLETION(name);
  - void init_completion(struct completion *x);
    - Initialize the dynamically created completion variable.
  - void wait_for_completion(struct completion *x);
    - Wait for the given condition variable to be signaled.
  - int wait_for_completion_interruptible(struct completion *x);
  - void complete(struct completion *x);
    - Signal any waiting tasks to wake up
  - void complete_all(struct completion *x);

Seq Locks (1)

- **Seq lock**
  - Useful to provide a very lightweight and scalable lock for use with many readers and a few writers.
  - Maintains a sequence counter.
  - Writer
    - Increase the counter before and after writing the data
    - Grabbing the write lock makes the value odd
  - Reader
    - Read the counter prior to and after reading the data.
    - If the values do not match or they are odd numbers, retry.
  - Seq locks favor writers over readers.

Seq Locks (2)

- **Using seq locks**
  - <linux/seqlock.h>
  - void seqlock_init(seqlock_t *sl);
  - void write_seqlock(seqlock_t *sl);
  - void write_sequnlock(seqlock_t *sl);
  - int write_tryseqlock(seqlock_t *sl);
  - unsigned read_seqbegin(const seqlock_t *sl);
  - int read_seqretry(const seqlock_t *sl, unsigned iv);

Seq Locks (3)

- **Examples**

```
seqlock_t sl =
   SEQLOCK_UNLOCKED;

int write ()
{
   …
   write_seqlock(&sl);

   // write

   write_sequnlock(&sl);
      …
}
```

```
int read ()
{
   unsigned long seq;
   …
   do {
      seq = read_seqbegin(&sl);

      // read

   } while (read_seqretry(&sl, seq));
   …
}
```

Per-CPU Variables (1)

- **Per-CPU variables**
  - A per-CPU variable is an array of data structures, one element per each CPU in the system.
  - A CPU should not access the elements of the array corresponding to the other CPUs.
  - The elements of the per-CPU array are aligned in main memory so that each data structure falls on a different line of the hardware cache.
  - A kernel control path should access a per-CPU variable with kernel preemption disabled.

Per-CPU Variables (2)

- **Using per-CPU variables**
  - DEFINE_PER_CPU(type, name)
    - Statically allocate a per-CPU array
  - alloc_percpu(type)
    - Dynamically allocates a per-CPU array and returns its address
  - free_percpu(pointer)
    - Release a dynamically allocated per-CPU array

Per-CPU Variables (3)

- **Using per-CPU variables (cont'd)**
  - per_cpu(name, cpu)
    - Selects the element for a given CPU
  - __get_cpu_var(name)
    - Selects the local CPU's element
  - get_cpu_var(name)
    - Disable kernel preemption, then select the local CPU's element of the per-CPU array
  - put_cpu_var(name)
    - Enable kernel preemption
  - Per_cpu_ptr(pointer, cpu)
    - Returns the address of the element for *cpu* of the per-CPU array

RCU (1)

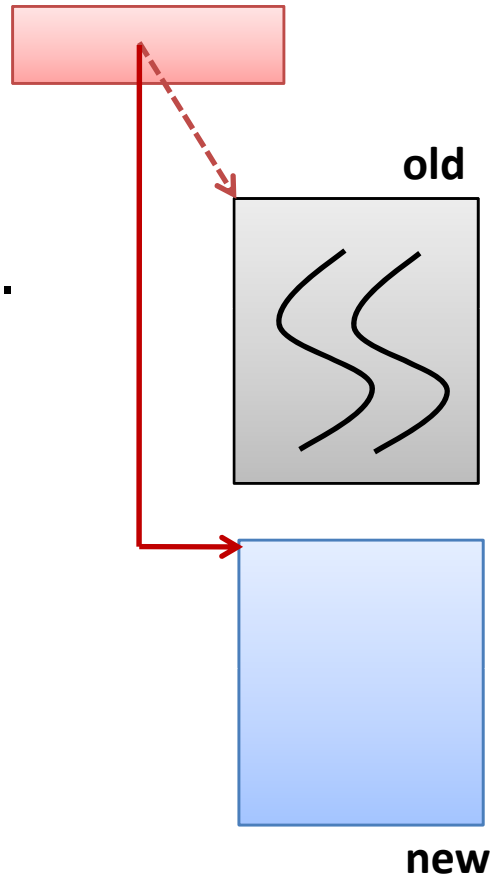
- **Read-copy update**
  - Protect data structures that are mostly accessed for reading by several CPUs.
  - Lock-free: no lock or counter shared by all CPUs
    - Advantageous over read/write spin locks and seqlocks, which have a high overhead due to cache line-snooping and invalidation.
  - Restrictions
    - Only data structures that are dynamically allocated and referenced by means of pointers can be protected.
    - No kernel control path can sleep inside a critical region protected by RCU.

RCU (2)

- **Reading**
  - Call rcu_read_lock();
    - Same as preempt_disable()
  - Dereference the pointer to the data structure
  - Start reading it using the pointer
    - The reader cannot sleep until it finishes reading the data structure
  - Call rcu_read_unlock();
    - Same as preempt_enable();
    - Mark the end of the critical region

RCU (3)

- **Updating**
  - Dereference the pointer and makes a copy of the whole data structure.
  - Modify the copy.
  - Change the pointer to the updated copy.
    - Changing the value of the pointer is an atomic operation.
    - A memory barrier is required to ensure that the updated pointer is seen by the other CPUs only after the data structure has been modified.
  - Old copy cannot be reclaimed right away. Why?



RCU (4)

- **Reclaiming old copies**
  - void call_rcu(struct rcu_head *head,
    void (*func)(struct rcu_head *head)));
  - Invoked by the writer to get rid of the old copy.
  - Once every tick, the kernel checks whether the local CPU has gone through a quiescent state:
    - The CPU performs a process switch.
    - The CPU starts executing in User Mode.
    - The CPU executes the idle loop.
  - When all CPUs have gone through a quiescent state, a local tasklet executes all callbacks registered by call_rcu(), which frees old copies.

Avoiding Synchronization (1)

- **Observations**
  - All interrupt handlers acknowledge the interrupt on the PIC and also disable the IRQ line.
    - Further occurrences of the same interrupt cannot occur until the handler terminates.
  - Interrupt handlers, softirqs, and tasklets are both nonpreemptible and nonblocking.
    - Their execution can be slightly delayed due to interrupts.
  - A kernel control path performing interrupt handling cannot be interrupted by a kernel control path executing a deferrable function or a system call service routine.

Avoiding Synchronization (2)

- **Observations (cont'd)**
  - Softirqs and tasklets cannot be interleaved on a CPU.
  - The same tasklet cannot be executed simultaneously on several CPUs.

- **Implications**
  - Interrupt handlers and tasklets need not to be coded as reentrant functions.
  - Per-CPU variables accessed by softirqs and tasklets only do not require synchronization.
  - A data structure accessed by only one kind of tasklet does not require synchronization.

Transactional Memory (1)

- **Problems with locks**
  - Pessimistic approach
  - Granularity
    - Coarse-grained: lock contention, poor scalability
    - Fine-grained: hard to write, extra overhead
  - Deadlock
  - Convoying effect
  - Priority inversion
  - Not composable
  - Error-prone
    - Forget to unlock
    - Thread termination with holding a lock, …

Transactional Memory (2)

- **Optimistic concurrency**
  - Execute <body> without taking any locks
  - Each read and write in <body> is logged
  - Writes go to the log only, not to memory
  - At the end, the transaction tries to commit to memory
  - Commit may fail; then transaction is re-run

```
atomic {
    <body>
}
```

- **Implementation**
  - STM (Software Transactional Memory)
  - HTM (Hardware Transactional Memory)

Summary

- **Protecting kernel data structures**

Kernel control paths	UP protection	MP further protection
Exceptions	Semaphore	None
Interrupts	Local interrupt disabling	Spin lock
Deferrable functions	None	None (one tasklet) Spin lock (softirqs, many tasklets)
Exceptions + Interrupts	Local interrupt disabling	Spin lock
Exceptions + Deferrable functions	Local softirq disabling	Spin lock
Interrupts + Deferrable functions	Local interrupt disabling	Spin lock
Exceptions + Interrupts + Deferrable functions	Local interrupt disabling	Spin lock