CS 378 (Spring 2003)

Linux Kernel Programming

Yongguang Zhang

(ygz@cs.utexas.edu)

This Lecture

- Time Management
- Synchronization

- Questions?

Clocks in Linux

- Real-Time Clock (RTC)
  - Independent of CPU
  - May be expensive to access
- CPU Cycle Counter (TSC Register)
  - For accurate time measurements
  - Granularity: CPU cycle (e.g. 1GHz)
- Kernel Clock (Programmable Interval Timer)
  - Each tick: cause timer interrupt at a fixed interval
  - For the kernel to keep track of time
  - Granularity: much larger: in terms of 10 Hz-1000 Hz

CPU Cycle Counter

- Architecture dependent
  - i386: TSC (Time Stamp Counter) register, 64 bits
  - Updated by hardware, read by "rdtsc" instruction
- C functions
  - In include/asm-i386/{msr.h,timex.h}

    rdtsc(), rdtscl(), rdtscll(), cycles_t get_cycles (void)
- Example uses:
  - Gettimeofday()
  - Networking (timestamping, scheduling)
  - Some device drivers

Kernel Timer

- Frequency: HZ (per second)
  - Defined in include/asm/param.h
  - Linux chooses 100Hz: #define HZ 100
- jiffies: number of ticks since system boot time
  - Global variable declared in include/linux/sched.h

    extern unsigned long volatile jiffies;
  - Incremented in each timer interrupt
  - 32-bit, so jiffies can wrap around
    - Never: if (jiffies > last_jiffies)
    - Always: if ((long)(jiffies – last_jiffies) > 0)

Timer Interrupt Handler

- timer_interrupt()
  - Architecture-dependent: arch/i386/kernel/time.c
  - Save TSC value (lower 32bit) into last_tsc_low
  - Then calls do_timer_interrupt()
- do_timer_interrupt()
  - Acks the interrupt (if necessary)
  - Calls the architecture independent do_timer()
  - Updates the RTC every 660 ticks if configured

do_timer()

- Architecture independent timer interrupt routine
  - Increments jiffies
  - Update process time (add 1 tick to user or system time)
  - Schedule timer bottom half
  - Essentially, (in kernel/timer.c)

```
(*(unsigned long *)&jiffies)++;
update_process_times(user_mode(regs));
mark_bh(TIMER_BH);
```

  - Q: why is jiffies updated in “top half”?

Timer Button Half

- A high priority tasklet to run at a later time
  - Button Half: backward compatible only, will go away
  - TIMER_BH → timer_bh()
- timer_bh()
  - Architecture independent, in kernel/timer.c
  - Calls the following two functions
  - update_times(): update the coarse wall clock (xtime)
  - run_timer_list(): process the list of kernel event timers

Wall Clock Management

- Wall clock
  - Time-of-day clock, real-time clock
  - To read precise wall clock: do_gettimeofday()
  - Two parts (each a long integer)
    - Second since 1/1/1970 (valid until 1/19/2038 03:14:07 UTC)
    - Microsecond within the second
- Implementation
  - Read hardware RTC only when system boot
  - Kernel maintains a coarse clock (xtime)
  - Accurate value calculated with help from TSC

xtime

- Coarse timer, updated in timer BH only
  - Defined in include/linux/sched.h
  - Two fields: xtime.tv_sec, xtime.tv_usec
  - timer_bh() calls update_times(), which does

```
ticks = jiffies – wall_jiffies;
if (ticks) {
        wall_jiffies += ticks;
        update_wall_time(ticks);
}
```

  - Q1: what is the actual granuty of xtime.tv_usec?
  - Q2: can xtime drift?

do_gettimeofday()

- Return precise time of day clock value (usec)
  - Need to compensate xtime.tv_usec
  - Add the lost jiffies: jiffies – wall_jiffies;
  - Add the elasped cycles: read TSC value and substrate last_tsc_low
- Use of do_gettimeofday()
  - Serving system call gettimeofday()
  - Other places need accurate time of day (e.g. logging)
- Kernel programming
  - Use do_gettimeofday() or get_cycles()
  - Don't use xtime directly

Kernel Event Timers

- Function calls scheduled at a specified time
- Data structure
  - In kernel/linux/timer.h

```
struct timer_list {
        struct list_head list;
        unsigned long expires;
        unsigned long data;
        void (*function)(unsigned long);
};
```

  - expires: function call at this jiffies value
  - function, data: function and argument to call

Using Kernel Timers

- Create a struct timer_list object
  - Add initialize it with init_timer(struct timer_list *)
- Setup the fields
- Insert it into the kernel timer list: add_timer()
- Before expiration, you can
  - Reschedule it: mod_timer(t, new_expires)
  - Delete timer: del_timer_sync()

Event Timer Handling

- Checking and executing event timers
  - By run_timer_list()
  - During timer BH only, so timing can be imprecise
- Kernel data structure for managing event timers
  - Group events into 512 lists by their expiration time
    - First 256 lists: events expiring in next 1, 2, 3, ..., 256 ticks
    - Next 64 lists: in next 1×2^8, 2×2^8, 3×2^8, ..., 64×2^8 ticks
    - Next 64 lists: in next 1×2^{14}, 2×2^{14}, 3×2^{14}, ..., 64×2^{14} ticks
    - Next 64 lists: in next 1×2^{20}, 2×2^{20}, 3×2^{20}, ..., 64×2^{20} ticks
    - Last 64 lists: in next 1×2^{26}, 2×2^{26}, 3×2^{26}, ..., 64×2^{26} ($=2^{32}$) ticks
  - Code in kernel/timer.c

Deferred Execution

- Scheduling execution in a safe time, ASAP
  - Use tasklet
  - In interrupt context (task cannot sleep)
  - Mostly used by interrupt handler/device driver
- Scheduling execution in a specific time
  - Use kernel event timer
  - Executed by timer BH, which is a tasklet afterall – still in interrupt context, cannot sleep
- Create your own list and scheduling
  - Use task queue (may go away in 2.6 kernel)

Task Queues

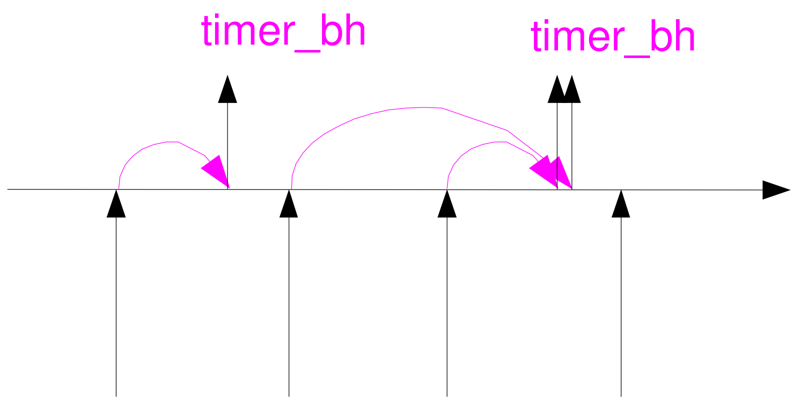
- Use task queue (see include/linux/tqueue.h)
  - Declare a queue
    DECLARE_TASK_QUEUE(name)
  - Add task to a queue
    int queue_task(struct tq_struct *, task_queue *)
  - Run all tasks currently in the queue
    void run_task_queue(task_queue *)
- Task queue may run in a different context
  - May run in interrupt context (by tasklet)
  - May run in a different thread (e.g., ksoftirqd, keventd)
  - May run in a different CPU

Time Illustration

User mode

Kernel mode

Interrupt mode



Timer Interrupts

Synchronization

- You all know why we need synchronization
  - Whenever a resource can be accessed by others
  - This task can sleep and schedule to others
  - This task can be interrupted and tasklet can run
  - There are other tasks in other CPUs
- Synchronization mechanims in Linux kernel
  - Atomic operations
  - Disabling interrupt
  - Locking: spin locks, semaphores

Hardware Support

- Fundamentally, mutual exclusion requires hardware support
  - Bootstrap from hardware-supported atomic action
- Single processor
  - cli and sti instruction: disable (and enable) all interrupts
- SMP architecture
  - The "lock" instruction prefix: lock the memory bus for this instruction (so no other CPU can access the memory until this instruction is done)

Atomic Operations

- Execute a single instruction in an "atomic" way, even under multiprocessor system
  - Supported by SMP hardware (with the "lock" instruction prefix)
  - Two types: bit ops and atomic integer variables
- Bit ops: change a bit in any memory address
  - void set_bit(int nr, volatile void * addr)
  - void clean_bit(int nr, volatile void * addr)
  - int test_and_set_bit(int nr, volatile void * addr)
  - And more (see include/asm-i386/bitops.h)

Atomic Integer Variable

- Atomic operations on integers
  - Defined in include/asm-i386/atomic.h
  - Data Structure: type atomic_t
- Functions:
  - atomic_read(v)
  - atomic_set(v,i)
  - atomic_add(i,v)
  - atomic_inc(v)
  - …

Interrupt Disabling

- Disable/enable all interrupts in this CPU
  - Use local_irq_disable() and local_irq_enable()
  - Implemented by cli or sti instruction
  - In SMP system: has no effect on other CPUs
- Global interrupt disabling/enabling
  - Use cli() and sti()
  - In uniprocessor system: same as cli and sti instruction
  - In SMP, cli() and sti() are implemented with spin lock to delay interrupt handlers in other CPUs

Saving eflags Register Content

- Must save register content before interrupt disabling and restore it after re-enabling
  - Register includes the interrupt flag (IF)
  - See include/asm-i386/system.h
- For local interrupt disabling
  - __save_flags(long) and __restore_flags(long)
- For global interrupt disabling
  - save_flags(long) and restore_flags(long)

Protecting Critical Session

- Using interrupt disabling (the simple way)

```
…
unsigned long flags;
 save_flags(flags);
 cli();
 … critical session …
 restore_flags(flags);
 …
```

  – Disadvantage: stops all other CPUs,
- Protecting from deferred tasklet/BH only?
  – Use local_bh_disable(), local_bh_enable()

Spin Lock

- A locking mechanism for SMP system
  - Through a shared variable
  - Acquire the lock by setting the variable
  - " Spin" in a busy-wait loop until the variable is unset
  - Should be used with care: holding a spin lock too long may cause other CPUs to waste time in busy waiting
  - Data type: spinlock_t (in include/asm/spinlock.h)
- In a uniprocessor system
  - Implemented as no-op (because it is the only process running)

Using Spin Lock

- Include <linux/spinlock.h>
- Define spin lock variable:
  - spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
- To lock, call spin_lock(my_lock)
  - With local interrupt disabled: spin_lock_irq()
  - Also with eflags also: spin_lock_irqsave()
  - With tasklet/BH disabled: spin_lock_bh()
- To unlock, call spin_unlock(my_lock)
- To check, spin_is_locked(my_lock) returns 1/0
- Variants: spin_trylock(), spin_unlock_wait()

Read/Write Spin Lock

- Allow multiple readers but only one writer
- Data type: rwlock_t
  - In include/spinlock.h
  - Ex: rwlock_t my_lock = RW_LOCK_UNLOCKED;
- Operations:
  - void read_lock(rwlock_t *rw)
  - void read_unlock(rwlock_t *rw)
  - void write_lock(rwlock_t *rw)
  - void write_unlock(rwlock_t *rw)
  - And more (with _irq, _irqsave, _bh, …)

Kernel Semaphores

- Concept of Semaphore
  - A number of resource available for claimed by task
  - Task put on the wait queue if resource unavailable
  - Task waits up when resource available (released)
- Data Structure
  - In include/asm/semaphore.h

```
struct semaphore {
        atomic_t count;        // > 0: available, <=0: busy
        int sleepers;
        wait_queue_head_t wait;
};
```

Using Kernel Semaphores

- To Initialize: sema_init(struct semaphore *, int)
  - If number of resource is 1, it is called MUTEX
  - init_MUTEX(sem), init_MUTEX_LOCKED(sem)
- To use a resource: down(struct semaphore * sem)
  - Atomically decrease count
  - Put current on the wait queue if count<0
  - Variants: down_trylock(), down_interruptible()
- To release: up(struct semaphore * sem)
  - Atomically increase count
  - Wake up one task on wait queue if count<=0

Synchronization Mechanisms

- Atomic operations
- Disabling interrupts
  - Simple way, but reduce parallelism
- Spin locks
  - Better way, good for protecting against other CPUs
- Semaphores
  - Good for synchronization among different tasks
  - Invoke scheduling, cannot be used in interrupt context

Summary

- Time Management:
  - LKP §3.2.5
  - ULK §6
  - LDD2 §7
- Synchronization:
  - LKP §5.1, §10
  - ULK §5
  - LDD2 §9
- Next Lecture: Device Driver