System calls

<Chapter 5. System calls>

Seo Bon Keun
System call

- An *extended* function call
  - From applications in user space
  - To kernel functions
  - Access system resources safely

```
User mode
write(…)

sys_enter

Kernel mode
sys_write(…)

sys_exit
```
Overview of system call

- System call is usually called by standard C library
  - libc, glibc, uclibc, ...
  - User application uses the library
  - Library implements POSIX API

- System call is called by a number
  - System call table

<table>
<thead>
<tr>
<th>Number</th>
<th>Kernel function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sys_restart_syscall()</td>
</tr>
<tr>
<td>1</td>
<td>sys_exit()</td>
</tr>
<tr>
<td>2</td>
<td>sys_fork()</td>
</tr>
<tr>
<td>3</td>
<td>sys_read()</td>
</tr>
<tr>
<td>4</td>
<td>sys_write()</td>
</tr>
</tbody>
</table>
System call invocation

- Old x86 used **interrupt**
  - `int 80h` (MS-DOS: `int 21h`)
  - EAX register holds the system call number

- **Special instruction**
  - Recent x86: `sys_enter`
    - EAX: system call number
  - ARM: `SWI <n>`
  - MIPS: `syscall`
    - Use v0 register to pass system call number
Passing arguments & return value

• Argument passing
  – Registers : integer value
    • If # of arguments exceeds # of registers, it is stored in a region of memory, and a register holds the address of the memory
  – Memory : large object
    • copy_from_user(void *to, void __user *from, unsigned long n);
    • copy_to_user(void __user *to, void *from, unsigned long n);

• Return value
  – Register: EAX for x86
Handling system calls in kernel

sysenter, eax=4

ia32_sysenter_target() <entry.$>

sys_call_table
<arch/x86/syscalls/sySCALL_32.tbl>

<table>
<thead>
<tr>
<th>Kernel function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sys_restart_syscall()</td>
</tr>
<tr>
<td>1</td>
<td>sys_exit()</td>
</tr>
<tr>
<td>2</td>
<td>sys_fork()</td>
</tr>
<tr>
<td>3</td>
<td>sys_read()</td>
</tr>
<tr>
<td>4</td>
<td>sys_write()</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

sys_write()

Interrupts

<Chapter 7. Interrupts and Interrupt Handlers>
<Chapter 8. Bottom Halves and Deferring Work>
Interrupt hardware

- Programmable interrupt controller (PIC)
  - 16 Interrupt request (IRQ) lines
  - Each IRQ line can be selectively disabled (cli/sti)

Which interrupt handling routine should be executed?
• Each device has an IRQ line assigned to it
• Many devices can share an IRQ line

Interrupt vector

- ISA 0: System timer
- ISA 1: Standard 101/102-Key Keyboard
- ISA 4: Communications Port (COM1)
- ISA 6: Standard floppy disk controller
- ISA 8: System CMOS/real time clock
- ISA 9: Microsoft ACPI-Compliant System
- ISA 12: Logitech-compatible Mouse PS/2
- ISA 13: Numeric data processor
- ISA 14: Primary IDE Channel
- ISA 15: Secondary IDE Channel
Interrupt handling

Generic C function
<kerneld/irq/handle.c>
handle_irq_event()

Device drivers

Architecture-dependent C function

Assembly routine
<entry.S>
Nested interrupts

• An interrupt handler may be interrupted
  – Must never block: careful synchronization!
  – *Interrupt handlers never generate page faults*
Linux interrupt handler

- **struct irq_desc**

```
struct irq_data    irq_data;
irq_flow_handler_t handle_irq;
struct irqaction  *action;    /* IRQ action list */
unsigned int      depth;       /* nested irq disables */
unsigned int      wake_depth;  /* nested wake enables */
unsigned int      irq_count;   /* For detecting broken IRQs */
struct cpumask    *percpu_enabled;
wait_queue_head_t wait_for_threads;
```

- A list of **irqactions**
  - For each device mapped to the IRQ line

```c
struct irqaction {
    irq_handler_t handler;
    void *dev_id;
    struct irqaction *next;
    irq_handler_t thread_fn;
    struct task_struct *thread;
    ...
}
```
Implementing a device driver

- request_irq() function to add a handler

```c
int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev)
```

- Implement an interrupt handler
  - irqreturn_t handler(int irq, void *data);

- May use bottom halves
Interrupt handlers need to run as quickly as possible.

- It may interrupt other potentially important code, including other interrupt handlers.

**Critical actions**
- Acknowledge an interrupt to the PIC.
- Reprogram the PIC or the device controller.
- Update data structures shared by multiple devices

**Reenable interrupts**
- Exchange command/data/status with the device (e.g., reading the scan code from the keyboard)

**Return from interrupts**
- Actions may be delayed.
- Copy buffer contents into the address space of some process (e.g., sending the keyboard line buffer to the terminal handler process).
- **Bottom half (Softirqs, Tasklets, and Work Queues)**
• Statically registered *important* bottom halves
  – Used to implement tasklets
  – Raising softIRQ: `raise_softirq(int nr)`
  – Executed
    • In the return from hardware interrupt code
      – In `irq_exit()` function in `do_IRQ()` function
    • In the `ksoftirqd` kernel thread
    • When raised by kernel code
      – `do_softirq()`

```c
#include/linux/interrupt.h>

struct softirq_action
{
    void (*action)(struct softirq_action *);
};
```

```c
softirq_vec[]
<kernel/softirq.c>
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>HI</td>
<td>0</td>
</tr>
<tr>
<td>TIMER</td>
<td>1</td>
</tr>
<tr>
<td>NET_TX</td>
<td>0</td>
</tr>
<tr>
<td>NET_RX</td>
<td>0</td>
</tr>
<tr>
<td>BLOCK</td>
<td>0</td>
</tr>
<tr>
<td>BLOCK_IOPOLL</td>
<td>0</td>
</tr>
<tr>
<td>TASKLET</td>
<td>0</td>
</tr>
<tr>
<td>SCHED</td>
<td>0</td>
</tr>
<tr>
<td>HRTIMER</td>
<td>0</td>
</tr>
<tr>
<td>RCU</td>
<td>0</td>
</tr>
</tbody>
</table>
• Softirqs can be enabled frequently (nested interrupt!)
  – irq_exit() function runs softIRQ handlers once
  – Repeated softIRQ harms user process performance

• A kernel thread to process deferred softIRQs (per CPU)

```c
static void run_ksoftirqd(unsigned int cpu)
{
    local_irq_disable();
    if (local_softirq_pending()) {
        __do_softirq();
        rcu_note_context_switch(cpu);
        local_irq_enable();
        cond_resched();
        return;
    }
    local_irq_enable();
}
```
Tasklet

• A bottom half built on top of softIRQ
  – User friendly API!
  – Only one tasklet of a kind runs at the same time
  – Supports priority

```
tasklet_hi_action()

tasklet_hi_vec[cpu]

func, data, cnt

func, data, cnt
```

```
tasklet_action()

func, data, cnt

func, data, cnt
```

```
tasklet_struct

func, data, cnt

func, data, cnt

func, data, cnt
```
Tasklet API

- **Creating a tasklet**
  - DECLARE_TASKLET(name, func, data)

- **Scheduling tasklets**
  - *Tasklet function is runs every time softIRQ is executed*
  - void tasklet_schedule( struct tasklet_struct *t );
  - void tasklet_hi_schedule( struct tasklet_struct *t );

- **Enabling/disabling tasklets**
  - tasklet_disable() / tasklet_kill()
  - tasklet_enable()
• Defers work into a kernel thread
  – Runs in process context!
  – Useful for situations where
    • allocate a lot of memory
    • obtain a semaphore
    • perform block I/O
    • ...
  – Worker thread: kworker/n
Work queue API

• Creating work queue
  – struct workqueue_struct *create_workqueue(char *name);

• Creating work
  – DECLARE_WORK(name, void (*func)(void *), void *data);
  – INIT_WORK(struct work_struct *work, 
    void (*func)(void *), void *data);

• Scheduling work
  – int schedule_work(struct work_struct *work);
  – int flush_scheduled_work(void);
<table>
<thead>
<tr>
<th>Bottom half</th>
<th>Context</th>
<th>Serialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoftIRQ</td>
<td>Interrupt (may be process)</td>
<td>None</td>
</tr>
<tr>
<td>Tasklet</td>
<td>Interrupt (may be process)</td>
<td>Against the same tasklet</td>
</tr>
<tr>
<td>Work queues</td>
<td>Process</td>
<td>None</td>
</tr>
</tbody>
</table>
Kernel synchronization

<Chapter 9. Introduction to Kernel Synchronization>
<Chapter 10. Kernel Synchronization Methods>
Race conditions

• Causes
  – Interrupts
    • SoftIRQs and tasklets
  – Kernel preemption
  – Sleeping and synchronization with user-space
  – Symmetrical multiprocessing

• Tools
  – Atomic operations
  – Spinlocks
  – Semaphores, mutexes, completion variables
  – Big kernel lock
  – Disabling interrupt, memory barrier
Atomic integer operations

• Atomic access to integer variables
  – Do not require any kind of locks → efficient!

• **Data type**: atomic_t

• **Functions**
  – atomic_set(&v, n)
  – atomic_add(n, &v)
  – atomic_inc(&v)
  – atomic_read(&v)
  – atomic_dec_and_test(&v)
  – ...

Spinlocks

- A lock ‘spins’ over a variable (do not sleep!)
  - Used in a multi-processor environment
  - Protecting code blocks with small size

- Data type: spinlock_t

- Functions
  - spin_lock()
  - spin_unlock()
  - spin_lock_irqsave()
  - spin_unlock_irqrestore()
Reader-writer spin locks

- A lock that is read frequently, written rarely
  - A number of readers can read
  - One writer exclusively updates

- Data type: rwlock_t

- Functions
  - read_lock()
  - read_unlock()
  - write_lock()
  - write_unlock()
Semaphores

- **Sleeping lock**
  - count > 0, enter
  - count <= 0, sleep

- **Data type:** struct semaphore <asm/semaaphore.h>

- **Functions**
  - sema_init(&name, count)
  - up(), down()
  - down_interruptible(), down_trylock()
• **Binary** sleeping lock
  – Recently added to the kernel
  – Caution: *Whoever locked a mutex must unlock it*

• **Data type:** `struct mutex <include/linux/mutex.h>`

• **Functions**
  – `mutex_init()`
  – `mutex_lock()`
  – `mutex_unlock()`
  – `mutex_trylock()`
  – `mutex_is_unlocked()`

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Recommended Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low overhead locking</td>
<td>Spin lock is preferred.</td>
</tr>
<tr>
<td>Short lock hold time</td>
<td>Spin lock is preferred.</td>
</tr>
<tr>
<td>Long lock hold time</td>
<td>Mutex is preferred.</td>
</tr>
<tr>
<td>Need to lock from interrupt context</td>
<td>Spin lock is required.</td>
</tr>
<tr>
<td>Need to sleep while holding lock</td>
<td>Mutex is required.</td>
</tr>
</tbody>
</table>
Completion variables

• An easy way to synchronize between two tasks
  – Condition variables of monitors in OS concepts

• **Data type:** `struct completion <linux/completion.h>`

• **Functions**
  – `init_completion(struct completion *)`
  – `wait_for_completion(struct completion *)`
  – `complete(struct completion *)`
Big kernel lock

• A global spin lock
  – Used for SMP before fine-grained locking implementation
  – Still exist

• Please, don’t use
  – Only one thread can run in a multi-processor system
Preemption disabling

- A protection mechanism for per-processor variables
  - Avoid using spinlocks
  - Increases `thread_info.preempt_count`

- Functions
  - `preempt_disable()`
  - `preempt_enable()`
  - `preempt_enable_no_resched()`
Memory barrier

- Ordering memory accesses without locking
  - CPU caches competes
  - Write barrier: *writes before the barrier guaranteed to happen*
  - Read barrier: *reads ...*
  - Data dependency barrier: *pointer assignment ...*

![Diagram of memory barrier](image)

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>&amp;a</td>
<td>&amp;b</td>
</tr>
</tbody>
</table>
```

- `b = 4`
- `p = &b`
- `q = p`
- `d = &q`