System calls

<Chapter 5. System calls>

Seo Bon Keun
• **An extended function call**
  – From applications in user space
  – To kernel functions
  – Access system resources safely
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>

Application

```
printf()
```

```
write()
```

<table>
<thead>
<tr>
<th>System call</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>syscall()</td>
<td>sys_write()</td>
</tr>
</tbody>
</table>

VFS | Network | IPC
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/entry/syscalls/sysexec/sysexec_32.tbl>

<table>
<thead>
<tr>
<th>Kernel function</th>
<th></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>

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?
Interrupt vector

- Each device has an IRQ line assigned to it
- Many devices can share an IRQ line

![Interrupt request (IRQ)](image)
Interrupt handling

- do_IRQ()
- handle_irq_event()
- ret_from_intr()

Architecture-dependent C function

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

Assembly routine
<entry.S>

Device drivers

1. Hardware
2. generates an interrupt
3. processor interrupts the kernel
4. is there an interrupt handler on this line?
   - yes: run all interrupt handlers on this line
   - no: return to the kernel code that was interrupted
5. Interrupt controller
6. Processor
Nested interrupts

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

![Diagram showing nested interrupts and mode transitions](image-url)
- **struct irq_desc**

  ```c
  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**
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

**Noncritical actions**
- 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 *);
};
```

```
<kernel/softirq.c>

softirq_vec[]

<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>
```

```c
<kernel/softirq.c>

irq_stat[cpu]
```
• 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();
}
```

Launched using smp_hotplug_thread API
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
• Creating a tasklet
  – DECLARE_TASKLET(name, func, data)

• Scheduling tasklets
  – Tasklet function 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
• 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);
## Summary

<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>
Interrupt summary

Example

Application → *ksoftirqd/0*

- **SoftIRQ** (Tasklet)
- **Pending SoftIRQs** (Tasklets)

Context switch

Application → *ksoftirqd/0* → *kworker/0*

- **Work queues**

<table>
<thead>
<tr>
<th>HI</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMER</td>
</tr>
<tr>
<td>NET_TX</td>
</tr>
<tr>
<td>NET_RX</td>
</tr>
<tr>
<td>BLOCK</td>
</tr>
<tr>
<td>BLOCK_IOPOLL</td>
</tr>
<tr>
<td>TASKLET</td>
</tr>
<tr>
<td>SCHED</td>
</tr>
<tr>
<td>HRTIMER</td>
</tr>
<tr>
<td>RCU</td>
</tr>
</tbody>
</table>
Manipulating proc entry

- Became a bit complex in Linux v4.x

- `proc_create(name, mode, parent, proc_fops)`
  - `struct file_operations -> struct seq_operations`

```c
static const struct file_operations led_proc_fops {
    .read   = seq_read,
    .write  = led_proc_write,
};
```

- `proc_cmdline_init()` in `fs/proc/cmdline.c`
- `proc_modules_init()` in `kernel/module.c`
- `led_proc_write()` in `drivers/parisc/led.c`

- `proc_remove(entry)`
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)`
  - `...`
• 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()
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()`
• **Sleeping** lock
  – count > 0, enter
  – count <= 0, sleep

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

• **Functions**
  ```c
  - mutex_init()
  - mutex_lock()
  - mutex_unlock()
  - mutex_trylock()
  - mutex_is_unlocked()
  ```

---

**Table 10.8  What to Use: Spin Locks Versus Semaphores**

<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()`
• 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 …*

```
CPU1
Cache
b p
4 &b

CPU2
Cache
d q
1 &a

Memory

Device
```

```
b = 4
p = &b
```

(write barrier)

```
q = p
d = &q
```

(data dependency barrier)