LINUX INTERRUPTS AND SYSTEM CALLS

Shiguang Wang

Feb 12th, 2014

Obtaining Kernel Source

- http://www.kernel.org the latest stable kernel *3.13.2*
- Untar bzip2 tarball: *tar -jxvf linux-x.y.z.tar.bz2*
- Untar GNU zip tarball: *tar -zxvf linux-x.y.z.tar.gz*
- *Now the *tar* command can **automatically** identify the compressing method, so simply *tar -xvf [tarball]*

Kernel Source Tree

Directory	Description
arch	Architecture-specific source
crypto	Crypto API
Documentation	Kernel source documentation
drivers	Device drivers
fs	The VFS and the individual file systems
include	Kernel headers
init	Kernel boot and initialization
ipc	Interprocess communication code
kernel	Core subsystems, such as the scheduler
lib	Helper routines
mm	Memory management subsystem and the VM
net	Networking subsystem
scripts	Scripts used to build the kernel
security	Linux Security Module
sound	Sound subsystem
usr	Early user-space code (called initramfs)

Compiling Kernel

- *cd* into the top source directory
    - *# make oldconfig*
    - *# make*
    - *# make modules*
- Must specify what kind of kernel you want
    - Configuration targets, like *xconfig*, *menuconfig*, *oldconfig*
        - *oldconfig* means “the same as last time”
    - Asking for “everything” may not be able to boot at all!

Booting A New Kernel

- *# make modules_install*
  - At this point, you should find */lib/modules/[version]* in your system
- *# make install*
  - This command will create the corresponding files in */boot*
    - *vmlinuz-[version]* -- the actual kernel
    - *System.map-[version]* -- the symbols exported by the kernel
    - *initrd.img-[version]* -- initrd image is temporary root file system used during boot process
    - *config-[version]* -- the kernel configuration file
  - *grub.cfg* will be updated automatically!!!
- *# reboot* and use *$ uname -r* to see the updated kernel version

Dual Mode of Operation

- Why? -- Need for protection
  - Kernel privileged, cannot trust user processes
    - User processes may be malicious or buggy
  - Must protect
    - User processes from one another
    - Kernel from user processes

Hardware Mechanisms for Protection

- Memory protection
  - Segmentation and paging
    - E.g. kernel sets segment/page table
- Timer interrupt
  - Kernel periodically gets back control
- Result -- Dual mode of operation
  - Privileged operations in kernel mode
  - Non-privileged operations in user mode

x86 Protection Modes

- Four modes (0-3), but often only 0 & 3 used
  - Kernel mode: 0
  - User mode: 3
- Segment has Descriptor Privilege Level (DPL)
- Current Privilege Level (CPL) = current code segment's DPL
  - Can only access data segments when CPL <= DPL

System Calls

- A set of interfaces that user-space processes interact with the system
  - Give applications access to hardware and other OS resources
  - Applications issues requests, the kernel fulfilling them
  - A mechanism for OS to regulate the behavior of applications
    - Providing a stable system, avoiding a big mess

A Middle Layer
Btw Hardware And User-Space Processes

- Purposes
  - Abstracted hardware interface for user-space
    - E.g., reading or writing from a file
  - Ensure system security and stability
    - E.g., preventing app. from incorrectly using hardware
  - Virtualized system provided to processes
    - E.g., multitasking and virtual memory
- System calls are only legal entry point into the kernel other than exceptions and traps
  - Other interfaces, such as */proc*, ultimately accessed via system calls

APIs, POSIX, and C Library

- Applications uses APIs not system calls directly
  - Benefits of no direct correlation
    - API can be implemented as a system call, through multiple system calls, or without system call
    - Same API exist on multiple systems with same interface, but different implementations of itself

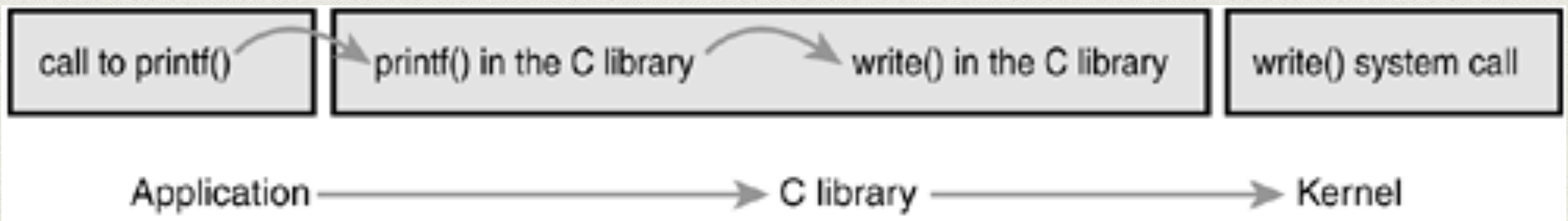
APIs, POSIX, and C Library

- POSIX standard
  - Comprises a series of standards from the IEEE*
  - Provides a portable OS standard roughly based on Unix
- On most Unix systems, the POSIX-defined API calls have a strong correlation to the system calls
  - On the other hand, some systems that are far from Unix, such as Windows NT, offer POSIX-compatible libraries

*IEEE (eye-triple-E) is the Institute of Electrical and Electronics Engineers. It is a nonprofit professional association involved in numerous technical areas and responsible for many important standards, such as POSIX. For more information, visit http://www.ieee.org

APIs, POSIX, and C Library

- Interface provided in part by the C library
  - Wrapped by other programming languages



- "provide mechanism, not policy"
  - System calls exist to provide a specific function in a very abstract sense

Syscalls

- System calls in Linux often called **syscalls**
  - Typically accessed via function calls
- Syscalls return value of the `long` type
  - Negative return usually denotes an error
  - Zero usually denotes a sign of success
- Write error code into the global `errno` variable
- The error code can be translated to human-readable via `perror()`

The `getpid()` Syscall

- Defined to return an integer of current process's PID
- The implementation of this syscall is simple:

```
asmlinkage long sys_getpid(void)
{
        return current->tgid;
}
```

- `asmlinkage` is a required modifier for all system calls
- Syscalls are prefixed with `sys_`

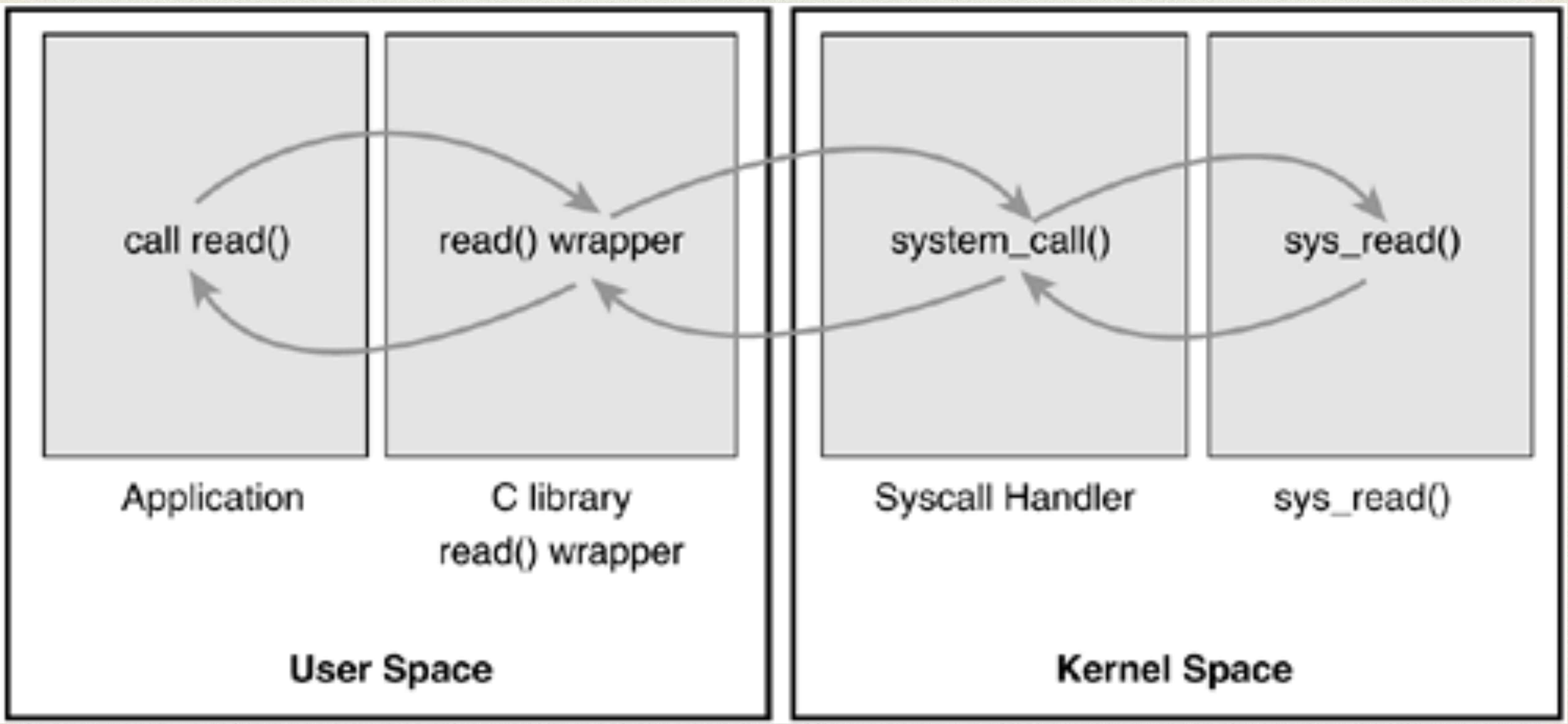
Syscall Numbers

- In Linux, each syscall is assigned a *syscall number*
    - Unique number used to reference a specific syscall
    - User-space process refer to the syscall by number
- Once assigned, the syscall number cannot change
    - Likewise, if a syscall is removed, its number cannot be recycled
- `sys_call_table`
    - A list of all registered syscalls
    - Architecture dependent and typically defined in `entry.S`, for x86 in `/arch/i386/kernel`

Denote the Correct Syscall

- Enter the kernel in the same manner for all system calls
- The syscall number must be passed to the kernel
    - via the `eax` register
- The `system_call()` function checks the validity of the syscall number
    - Comparing to `NR_syscalls`
        - If larger than or equal to `NR_syscalls`, returns `-ENOSYS`
        - Otherwise, invoke: `call *sys_call_table(, %eax, 4)`

Invoking Syscall Handler and Executing a Syscall



Syscall Handler

- Software interrupt to signal the kernel
    - Incur an exception, then system will switch to kernel mode, and execute the exception handler
    - Exception handler is actually syscall handler
- Defined software interrupt on x86 is `int $0x80`
    - switch to kernel mode
    - execute the syscall handler `system_call()`
    - Architecture dependent and typically implemented in assembly in `entry.S`
    - http://www.cs.dartmouth.edu/~sergey/cs108/rootkits/entry.S

Syscall Implementation

- Adding a new syscall to Linux is easy
- Designing and implementing a syscall is hard!

Syscall Implementation

- Steps in implementing a syscall
  - Define its purpose -- What will it do?
    - The syscall should have exactly one purpose
    - Multiplexing syscalls is discouraged in Linux
  - Clean and simple interface
    - With smallest number of arguments
    - The semantics and behavior of a syscall musts not change
    - Designing with an eye toward the future
  - Write a system call
    - Realize the need for portability and robustness, not just today but in the future
    - The basic Unix system calls have survived the test of time

“Provide mechanism, not policy.”

-Unix motto

Syscall Implementation

- Verifying the Parameters
  - Syscalls must carefully verify all their parameters
    - File I/O syscalls must check whether the file descriptor is valid
    - Process-related functions must check whether the provided PID is valid.
    - Invalid input is dangerous, e.g. access a protected memory space in kernel by passing an unchecked pointer

Syscall Implementation

- Two methods for performing the desired copy to and from user-space
  - `copy_to_user()`, with parameters:
    - The destination memory address in the process's address space
    - The source pointer in the kernel-space
    - The size in bytes of the data to copy
  - `copy_from_user()`, analogous to the above:
    - Read from the second parameter into the first parameter the number of bytes specified in the third parameter.

A Bad Example

```
/*
 * silly_copy - utterly worthless syscall that copies the len bytes from
 * 'src' to 'dst' using the kernel as an intermediary in the copy for no
 * good reason.  But it makes for a good example!
 */
asmlinkage long sys_silly_copy(unsigned long *src,
                               unsigned long *dst,
                               unsigned long len)
{
        unsigned long buf;

        /* fail if the kernel wordsize and user wordsize do not match */
        if (len != sizeof(buf))
                return -EINVAL;

        /* copy src, which is in the user's address space, into buf */
        if (copy_from_user(&buf, src, len))
                return -EFAULT;

        /* copy buf into dst, which is in the user's address space */
        if (copy_to_user(dst, &buf, len))
                return -EFAULT;

        /* return amount of data copied */
        return len;
}
```

Check for Valid Permission

- Older version of Linux, user `suser()`
  - Merely checked whether a user was root or not
- New system allows *specific access checks on specific resources*

```
asmlinkage long sys_am_i_popular (void)
{
        /* check whether the user possesses the CAP_SYS_NICE capability */
        if (!capable(CAP_SYS_NICE))
                return EPERM;

        /* return zero for success */
        return 0;
}
```

- http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/include/linux/capability.h

Final Steps in Binding a Syscall

- After a system call is written, it is trivial to register it as an official system call:
  - First, add an entry to the end of the system call table
  - Second, for each architecture supported, the syscall number needs to be defined in `<asm/unistd.h>`
  - Third, the syscall needs to be compiled into the kernel image (as opposed to compiled as a module)
    - This can be as simple as putting the system call in a relevant file in `kernel/`, such as `sys.c`

Example of Implementing A Syscall `foo()`

- First, add `sys_foo()` to the system call table in `entry.S`

```
ENTRY(sys_call_table)
        .long sys_restart_syscall    /* 0 */
        .long sys_exit
        .long sys_fork
        .long sys_read
        .long sys_write
        .long sys_open               /* 5 */

    ...

        .long sys_mq_unlink
        .long sys_mq_timedsend
        .long sys_mq_timedreceive       /* 280 */
        .long sys_mq_notify
        .long sys_mq_getsetattr
        .long sys_foo                /* 283 */
```

- http://www.cs.dartmouth.edu/~sergey/cs108/rootkits/entry.S

Example of Implementing A Syscall foo()

- Next, the system call number is added to <asm/unistd.h>

```
/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall  0
#define __NR_exit             1
#define __NR_fork             2
#define __NR_read             3
#define __NR_write            4
#define __NR_open             5

...
#define __NR_mq_unlink        278
#define __NR_mq_timedsend     279
#define __NR_mq_timedreceive  280
#define __NR_mq_notify        281
#define __NR_mq_getsetattr    282
#define __NR_foo              283
```

- http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/arch/arm/include/asm/unistd.h

Example of Implementing A Syscall foo()

- Finally, the actual `foo()` system call is implemented in `kernel/sys.c`

```
#include <asm/thread_info.h>

/*
 * sys_foo  everyone's favorite system call.
 *
 * Returns the size of the per-process kernel stack.
 */
asmlinkage long sys_foo(void)
{
        return THREAD_SIZE;
}
```

- You should put it wherever the function is most relevant
  - `kernel/sys.c` is home to miscellaneous system calls
  - E.g., if the function is related to scheduling, you could put it in `kernel/sched.c`

Example of Implementing A Syscall foo()

- Accessing the System Call from User-Space

```
#define __NR_foo 283
__syscall0(long, foo)

int main ()
{
        long stack_size;

        stack_size = foo ();
        printf ("The kernel stack size is %ld\n", stack_size);

        return 0;
}
```

- http://www.cs.albany.edu/~sdc/CSI500/Fal11/Labs/L06/OwnSyscall.html

- It is not hard to implement a new system call!!!

Why Not to Implement A System Call?

- Pros:
  - Syscalls are simple to implement and easy to use
  - Syscalls performance on Linux is blindingly fast
- Cons:
  - You need a syscall number, which needs to be officially assigned to you during a developmental kernel series.
  - After the system call is in a stable series kernel, it is written in stone.
  - Each architecture needs to separately register the system call and support it
  - System calls are not easily used from scripts and cannot be accessed directly from the filesystem.
  - For simple exchange of information, a system call is overkill.
- The slow rate of addition of new system calls is a sign that Linux is a relatively stable and feature-complete OS.

Why Interrupts?

- A primary responsibility of the kernel is managing the hardware
- Processors are typically magnitudes faster than the hardware they talk to
    - Not ideal for the kernel to issue a request and wait for a response from slow hardware
    - Polling -- kernel periodically check the status of the hardware and respond accordingly (incurs overhead!)
    - Interrupts -- the hardware signal the kernel when attention is needed (better solution!)

What are Interrupts?

- Allow hardware to communicate with the processor
  - When you type, keyboard controller signals the processor a newly available key press
  - The processor receives the interrupt and signals the OS to respond to the new data
- Interrupts are generated asynchronously
- Consequently, the kernel can be interrupted at any time

How Interrupts?

- Physically produced by electronic signals originating from hardware devices and directed into input pins on an interrupt controller
- The interrupt controller sends a signal to the processor
- The processor detects this signal and interrupts its current execution to handle the interrupt
- The processor can then notify the OS to handle the interrupt appropriately

Interrupt Values

- Different devices associated with unique values
  - Enable OS differentiate between interrupts from different hardware devices
  - OS services each interrupt with a unique handler
- Interrupt values called *interrupt request (IRQ)* lines
  - IRQ zero is the timer interrupt, IRQ one is the keyboard interrupt
  - Not all interupt numbers are so regidly defined!
- Specific interrupt is associated with a specific device, and the kernel knows this

Exceptions

- Occur synchronously *w.r.t.* the processor clock
  - *synchronous interrupts*
- Produced by the processor
  - E.g., *divide by zero, a page fault,* or *a system call*
- The kernel infrastructure for handling the two is similar

Interrupt Handlers

- Function in response to a specific interupt
- A.k.a. *interrupt service routine* (ISR)
- Each device that generates interrupts has an associated interrupt handler
  - Part of the device's *driver* -- the kernel code that manages the device
- Imperative that the handler runs quickly
  - Needs to immediately respond to the hardware
  - Also needs to resume the execution of the interrupted code *asap*.
  - The two halves approach!

Top Halves vs. Bottom Halves

- At the very least, the handler just acknowledge the receipt to the hardware
- Often, however, the handlers have a large amount of work to perform
    - E.g., the gigabit Ethernet cards.
- Contrast goals -- execute quickly and perform a large amount of work!
- Because of the conflicting goals, the processing of interrupts is split into two parts:
    - The handler is the *top half -- immediate response and perform only time critical work*
    - What that can be performed later is delayed to the *bottom half*

Example --Network Card

- When network cards receive incoming packets off the network, they need to alert the kernel
  - Need to do this immediately
    - To optimize network throughput and avoid timeouts
- The kernel responds by executing the network card's registered interrupt
- Inside interrupt (top half), acknowledge the hardware, copy the packets into main memory, ready the network card for more packets
  - These jobs are important, time-critical, and hardware-specific work
- The rest of processing and handling in the bottom half
  - Push the packets down to the appropriate protocol stack or application

Registering an Interrupt Handler

- Responsibility of the device driver
- Register an interrupt handler and enable a given interrupt line for handling

```
/* request_irq: allocate a given interrupt line */
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long irqflags,
                const char *devname,
                void *dev_id)
```

- `irq`, specifies the interrupt number to allocate
- `handler`, is a function point to the actual interrupt handler service
- `irqflags`, might be either zero or a bit mask of one or more of the following:
  - `SA_INTERRUPT`, `SA_SAMPLE_RANDOM`, `SA_SHIRQ`
- `devname`, device ASCII text representation
- `dev_id`, is used primarily for shared interrupt lines

Registering an Interrupt Handler

- On registration, an entry corresponding to the interrupt is created in `/proc/irq`
  - The function `proc_mkdir()` to is used to create new procfs entries, which in turn call `kmalloc()` to allocate memory
    - Since `kmalloc()` can sleep, the function `request_irq()` can sleep.
    - Never use it in interrupt context!
- In a driver, requesting an interrupt line and installing a handler is done via `request_irq():`

```
if (request_irq(irqn, my_interrupt, SA_SHIRQ, "my_device", dev)) {
        printk(KERN_ERR "my_device: cannot register IRQ %d\n", irqn);
        return -EIO;
}
```

- Free an interrupt handler by:
- ```
  void free_irq(unsigned int irq, void *dev_id)
  ```
  ```

Example -- RTC Driver

- http://lxr.free-electrons.com/source/drivers/char/rtc.c

- `rtc_init()` invoked to initialize the driver

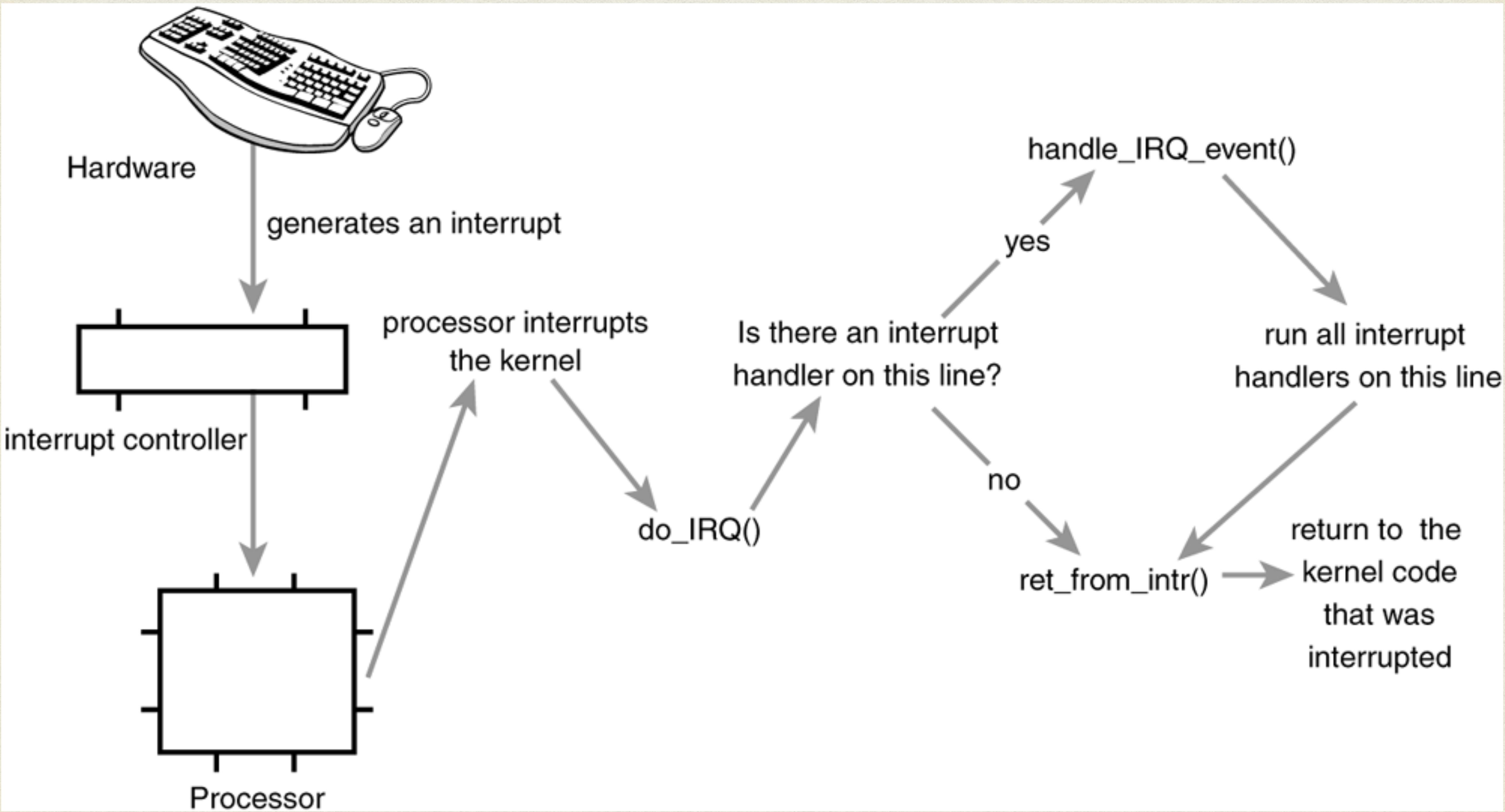
```
/* register rtc_interrupt on RTC_IRQ */
if (request_irq(RTC_IRQ, rtc_interrupt, SA_INTERRUPT, "rtc", NULL) {
        printk(KERN_ERR "rtc: cannot register IRQ %d\n", RTC_IRQ);
        return -EIO;
}
```

```c
/*
 * A very tiny interrupt handler. It runs with SA_INTERRUPT set,
 * but there is a possibility of conflicting with the set_rtc_mmss()
 * call (the rtc irq and the timer irq can easily run at the same
 * time in two different CPUs). So we need to serialize
 * accesses to the chip with the rtc_lock spinlock that each
 * architecture should implement in the timer code.
 * (See ./arch/XXXX/kernel/time.c for the set_rtc_mmss() function.)
 */
static irqreturn_t rtc_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        /*
         * Can be an alarm interrupt, update complete interrupt,
         * or a periodic interrupt. We store the status in the
         * low byte and the number of interrupts received since
         * the last read in the remainder of rtc_irq_data.
         */

        spin_lock (&rtc_lock);

        rtc_irq_data += 0x100;
        rtc_irq_data &= ~0xff;
        rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) & 0xF0);

        if (rtc_status & RTC_TIMER_ON)
            mod_timer(&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);

        spin_unlock (&rtc_lock);

        /*
         * Now do the rest of the actions
         */
        spin_lock(&rtc_task_lock);
        if (rtc_callback)
                rtc_callback->func(rtc_callback->private_data);
        spin_unlock(&rtc_task_lock);
        wake_up_interruptible(&rtc_wait);

        kill_fasync (&rtc_async_queue, SIGIO, POLL_IN);

        return IRQ_HANDLED;
}
```

Interrupt Context

- Process context is the mode of operation the kernel is in while it is executing on behalf of a process
  - `current` macro points to the associated task
  - Because a process is coupled to the kernel in process context, process context can sleep
- Interrupt context is NOT associated with a process
  - `current` macro is not relevant (although it points to the interrupted process)
  - Without a backing process, interrupt context cannot sleep
    - *How would it ever reschedule? No way!*

The Path an Interrupt Takes



Demo: Statistics of Interrupts

- *# cat /proc/interrupts*

Bottom Halves

- Tasklets and Work Queues
- *Covered in MP1 Q&A already :-)*

References

- http://www.makelinux.net/books/lkd2/?u=ch06lev1sec6
- http://www.win.tue.nl/~aeb/linux/lk/lk.html#toc4
- http://www.thegeekstuff.com/2013/06/compile-linux-kernel/
- http://lxr.free-electrons.com/source/