Communicating with the Kernel

Azza Abouzied

Process

What is a process

A process is an abstraction that provides the illusion to a program that it has its own abstract machine.

What is a process?

1. Process provides illusion of a private memory system: address space. (Learn how with virutal memory & paging)

2. Process provides illusion of its own CPU to execute instructions: thread of execution

3. Kernel maintains state for each process (Learn how with process management)
    - process descriptor tables
    - kernel stack (different from the user-process stack)
    - run state
    - page tables

Contexts in Operation

At any point in time, a processor is in one of the following three states

1. In user-space, executing user code in a process
2. In kernel-space, in **process context**, executing on behalf of a specific process
3. In kernel-space, in **interrupt context**, not associated with a process, handling an interrupt

Even when nothing is running, the *idle-process* runs.

Application | Kernel: System Call

Application | Kernel

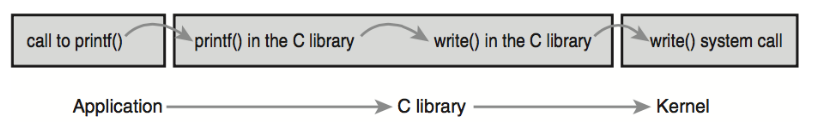
Why do we need a layer (i.e. the kernel) between hardware & user space process

1. **Abstraction**: Read and write files regardless of disk/network/filesystem etc.

2. **Security**: Kernel arbitrates access based on permissions, users, available resources, etc.

3. **Virtualization**: Multitasking, virtual memory, etc.

The System Call



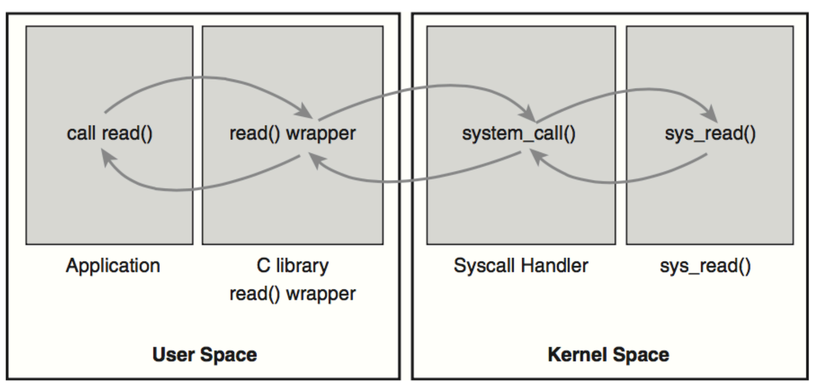


Figure: From Linux Kernel Development by Robert Love

What are examples of system calls?

1. **Process control**: e.g. create, terminate, load, get or set process attributes, wait/sleep (fork(), exit(), getpid(), wait(), sleep())

2. **File management**: create, open, delete, close, read, write, get/set file attributes (open(), read(), write(), close())

3. **Device management** read, write, mount, ...

4. **Information**: time, date, get or set this system data, ... (alarm(), time())

5. **Communication**: pipes, send, receive, ... (pipe(), mmap())

6. **Protection**: get or set permissions, ... (chmod(), chown())

Steps within a system call

1. User space cannot execute kernel code directly via a function call - Why?
2. Each system call has a unique number. User space code sticks that number into the %eax register.
3. Use registers %ebx, %ecx, %edx, %esi, %edi for parameters or a single register to hold a pointer to user-space for more than 5 parameters...[1]
4. Now generate the switch to kernel mode with the help of a software interrupt: `int $0x80`

[1] On some systems, parameters are pushed onto the user-stack.

The int *n* instruction

On the x86, interrupt handlers are defined in an interrupt descriptor table (IDT). The IDT has 256 entries, each giving the %cs and %eip to be used when handling the corresponding interrupt.

The int *n* instruction - what the hardware does?

1. $x80 or 128 in IDT points to the kernel entry point for the system call handler

2. In x86, information about whether we are in user-space or kernel-space is encoded within %eip.

3. If we are switching from user-space to kernel-space (i.e. system call was made from a user process), save user-stack registers (%esp, %ss)

4. Load kernel stack registers from task state segment

5. Push onto the kernel stack the user stack information: e.g. %esp, %ss, %eflags, %cs, %eip,

6. Set the values of %cs, %eip to the kernel entry point for the syscall handler.

After int *n* instruction

1. The system call handler uses a system call table to determine the function to call: `call *sys_call_table(,%eax,4)`

2. Return values in the %eax register

3. Control returned to user with `iret`, which reverses what `int $0x80` does.

Hardware | Kernel: Interrupt

How do you get the attention of the kernel from hardware?

2 options

How do you get the attention of the kernel from hardware?

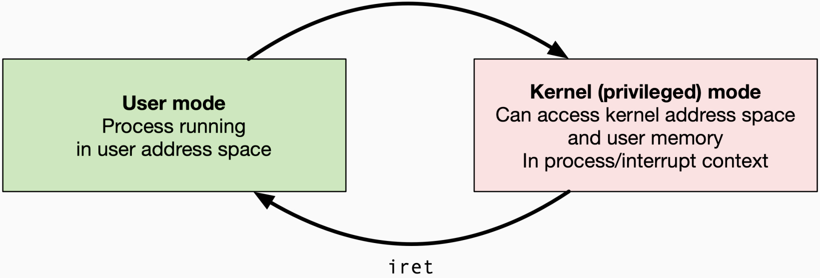
2 options

Polling vs. Interrupts

Recap

Getting the Kernel's Attention

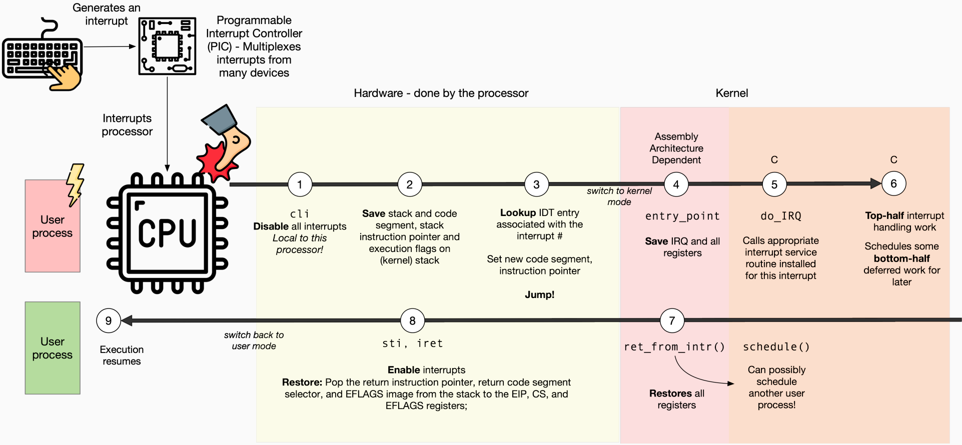
Mechanism	Initiated by	Example
System Call	user process	`int 0x80`
Interrupt	hardware	IRQ 0: Timer; IRQ 1 Keyboard; etc
Trap/Exception	fault	division by 0



The kernel is not a process itself but rather a process manager!

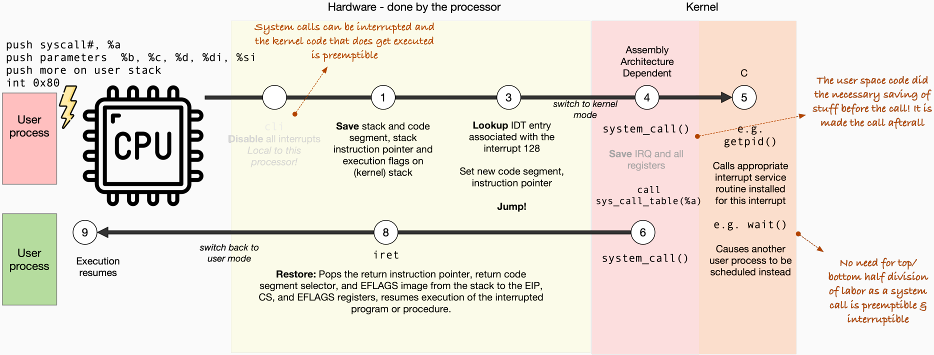
Interrupt Handling

Interrupt Handling



How is exception handling different?

How is this different from system calls?



Differences between System Calls & Interrupts

1. Kernel executes a system call in process-context, not interrupt-context.
    - A system call can block/sleep. Interrupt handler routines cannot!
    - Interrupt handlers in Linux divide labor into a fast top half and a bottom half that can be rescheduled later and has access to blocking calls.
2. Interrupts Handlers still use the kernel stack of the interrupted process. (Some Linux versions however have a special Interrupt stack). Regardless keep memory usage small.

Reentrancy

All modern kernels are **reentrant**: i.e., several processes may be executing in Kernel Mode at the same time.

1. Interrupts on different lines are enabled on other processors and even the currently interrupted one.

2. There is still room for race conditions across processors

3. Kernel data structures can be accessed by other processors so you need some sort of locking

Why do interrupt handlers only form the top half of interrupt processing?

Why?

1. Interrupts are async: they interrupt potentially important code!
2. Other interrupts at least on the same line or possibly all interrupts are disabled preventing hardware from communicating with the OS.
3. Time-critical as they deal with hardware
4. No process context so they cannot block because the kernel cannot put the interrupt handler on the scheduler's queue to be run at a later time when ready.

Rules for Top/Bottom Halves

1. Time sensitive?
2. Requires Hardware?
3. Should not be interrupted especially by this interrupt?

Network Card

1. Network card interrupts when a packet is received

2. The top-half acknowledges and copies data into memory:
    - time-sensitive: network card buffer is small; must free it quickly and move it to memory

3. The bottom-half then takes care of enqueuing the packet for the receiving application.

Class Exercise

Task 1: Pick any system call

- Why is this a system call and not a standard C library function that can be linked into user-address space?
- What does it do?
- List out all the steps involved in executing the system call
- Why do the steps occur the way they do?
- Are there different OS implementations?

Task 2: Pick any interrupt

- Does the interrupt have a dedicated IRQ?
- How is the interrupt handled?
- What actions should the ISR do in the top-half? Why?
- What actions should it do in the bottom-half? Why?

Questions?