Responsibilities of an Operating System

An operating system has two primary responsibilities:

- Managing the computer's resources (processor, memory, I/O devices)
- Providing a standard interface for users and user software

The Kernel

This course focuses on the operating system *kernel* — the small nucleus of the system software that runs in **privileged mode** and mediates access by all user programs to the physical hardware.

User processes (shells, editors, etc.) run independently of the kernel and communicate with it through the use of **system calls**.

At any time after system boot, main memory holds the kernel process and several user processes. When a machine is first powered on, a single user process called `init` is executed when the kernel finishes initializing; `init` is responsible for ultimately spawning all other user processes on the system (shells, GUI's, daemons, etc.)

User Mode vs Kernel Mode

Very early computer architectures were not built with multiprogramming in mind — all CPU instructions could be executed at any time, and applications could access any memory address.

Modern architectures allow different pieces of code to run at different (hardware enforced) privilege levels.

- Certain machine instructions (usually those dealing with memory management, task switching, etc.) are restricted to code running in "kernel mode." Execution while the processor is in "user mode" causes a hardware exception.
- This division between kernel mode and user mode prevents applications from accidentally clobbering each other's memory (or the OS's memory) if something goes wrong.

x86 Protected mode

- Supports four privilege levels (called "rings"), which are numbered from 0 to 3.
- The kernel runs in ring 0 and user mode applications run in ring 3.
- Rings 1 and 2 are generally unused, except by some types of virtualization software
- Kernel code (ring 0) has access to both kernel memory and user space memory. Applications in user space can only access their own memory.

Kernel Entry

Execution switches from user mode to kernel mode on three types of events:

Kernel Entry

Execution switches from user mode to kernel mode on three types of events:

- **Hardware interrupt** — external events from system hardware (I/O devices, network, keyboard, system clock, etc.); hardware interrupts occur **asynchronously**, meaning their execution is not necessarily related to the user process currently running

Kernel Entry

Execution switches from user mode to kernel mode on three types of events:

- **Hardware interrupt** — external events from system hardware (I/O devices, network, keyboard, system clock, etc.); hardware interrupts occur **asynchronously**, meaning their execution is not necessarily related to the user process currently running
- **Hardware trap** — an event (also sometimes referred to as a **hardware exception**) raised by the system hardware that occurs **synchronously** for a process

Kernel Entry

Execution switches from user mode to kernel mode on three types of events:

- **Hardware interrupt** — external events from system hardware (I/O devices, network, keyboard, system clock, etc.); hardware interrupts occur **asynchronously**, meaning their execution is not necessarily related to the user process currently running
- **Hardware trap** — an event (also sometimes referred to as a **hardware exception**) raised by the system hardware that occurs **synchronously** for a process
- **Software trap** — used by the operating system to force an event to happen ASAP (scheduling a new process, processing network data, etc.)

Top Half and Bottom Half

The kernel can be further divided into a **top half** and **bottom half**

Top Half and Bottom Half

The kernel can be further divided into a **top half** and **bottom half**

- **top half** of kernel provides services to processes in response to system calls and traps
  - conceptually a library of routines shared by all processes
  - executes on a per-process kernel stack
  - executes synchronously for a process

Top Half and Bottom Half

The kernel can be further divided into a **top half** and **bottom half**

- **top half** of kernel provides services to processes in response to system calls and traps
  - conceptually a library of routines shared by all processes
  - executes on a per-process kernel stack
  - executes synchronously for a process
- **bottom half** of kernel is a set of routines for handling hardware interrupts
  - runs *asynchronously* in relation to user processes and the top half of the kernel

Top Half and Bottom Half

- FreeBSD 5.x does not allow preemption by another user process while the kernel is running in the top-half, although it may voluntarily give up the processor if it needs to wait for something.

Top Half and Bottom Half

- FreeBSD 5.x does not allow preemption by another user process while the kernel is running in the top-half, although it may voluntarily give up the processor if it needs to wait for something.
- When a hardware interrupt is received, the kernel process associated with that device is scheduled. Since interrupts have a higher priority than user processes or processes executing in the top-half of the kernel, they will usually preempt the currently running process.

Top Half and Bottom Half

- FreeBSD 5.x does not allow preemption by another user process while the kernel is running in the top-half, although it may voluntarily give up the processor if it needs to wait for something.
- When a hardware interrupt is received, the kernel process associated with that device is scheduled. Since interrupts have a higher priority than user processes or processes executing in the top-half of the kernel, they will usually preempt the currently running process.
- Most communication between the top and bottom half of the kernel is performed through **work queues**.

Kernel Entry via System Call

When a user process makes a system call, the following steps are performed:

Kernel Entry via System Call

When a user process makes a system call, the following steps are performed:

- The hardware switches into kernel (supervisor) mode; this allows memory access checks to use kernel privileges, references to the stack will use the per-process kernel stack rather than the user-mode stack, and privileged instructions can be executed

Kernel Entry via System Call

When a user process makes a system call, the following steps are performed:

- The hardware switches into kernel (supervisor) mode; this allows memory access checks to use kernel privileges, references to the stack will use the per-process kernel stack rather than the user-mode stack, and privileged instructions can be executed
- Program counter, processor status flags, information about the type of system call (e.g., the system call ID#), and various registers are pushed onto the *kernel* stack.

Kernel Entry via System Call

When a user process makes a system call, the following steps are performed:

- The hardware switches into kernel (supervisor) mode; this allows memory access checks to use kernel privileges, references to the stack will use the per-process kernel stack rather than the user-mode stack, and privileged instructions can be executed
- Program counter, processor status flags, information about the type of system call (e.g., the system call ID#), and various registers are pushed onto the *kernel* stack.
- An assembly language routine in the FreeBSD kernel saves additional state information that the hardware didn't take care of.

Kernel Entry via System Call

When a user process makes a system call, the following steps are performed:

- The hardware switches into kernel (supervisor) mode; this allows memory access checks to use kernel privileges, references to the stack will use the per-process kernel stack rather than the user-mode stack, and privileged instructions can be executed
- Program counter, processor status flags, information about the type of system call (e.g., the system call ID#), and various registers are pushed onto the *kernel* stack.
- An assembly language routine in the FreeBSD kernel saves additional state information that the hardware didn't take care of.
- The actual system call (a C routine) is executed.

For Next Time

- Try to install FreeBSD on your system if you haven't already done so. If you have problems, it's best to discover that early. The first programming project will probably be given out at Monday's discussion.
- Skim chapter 3 of the textbook. You don't need to understand every tiny detail, but you should understand the general concepts.
- Office location for office hours is still TBA. . . I'll hijack one of the upstairs TA offices in Kemper until I actually get my own office assigned. Please monitor the newsgroup and website for an updated location.