CS 5460/6460
Operating Systems

Fall 2009

Instructor: **Matthew Flatt**

Lecturer: **Kevin Tew**

TAs: Bigyan Mukherjee, Amrish Kapoor

Reminders

- Join the Mailing List!
- Make sure you can log into the CADE machines

Modern Operating System Functionality

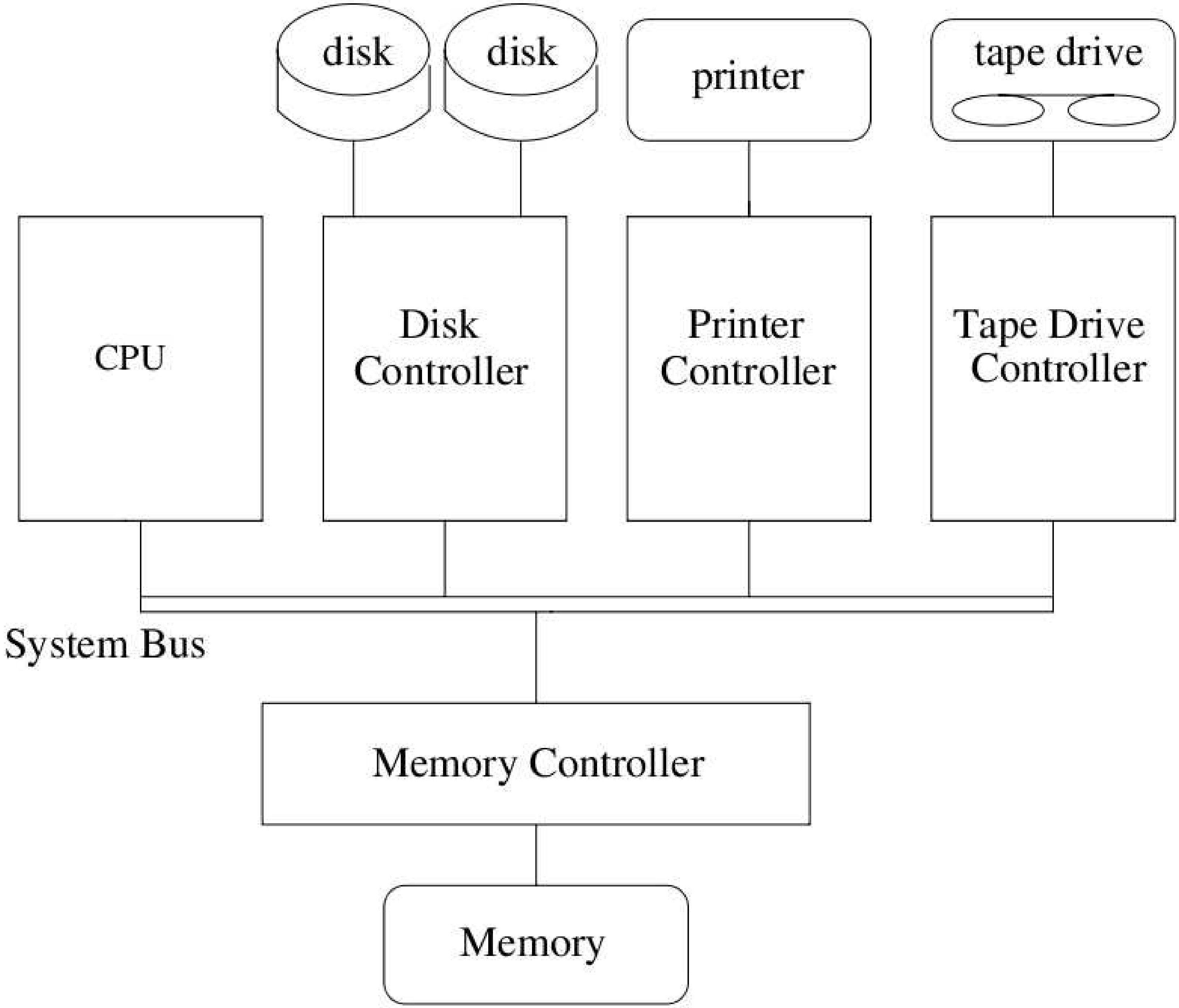
- Concurrency - doing many things at the same time (I/0, processing, multiple programs, etc.)
  - Several users work at the same time as if each has a private machine
  - Threads (unit of OS control) - one thread on the CPU at a time, but many threads active concurrently
  - Virtualization different OS or multiple copies of the same OS running on a single system
- Namespacing - separation or isolation

Modern Operating System Functionality cont.

- I/O devices - let the CPU work while an I/O device is working (especially a device like a slow terminal.)
- Memory management - OS coordinates allocation of memory and moving data between disk and main memory.
- Files - OS coordinates how space is used for files, in order to find files and to store multiple files
- Distributed systems & networks - allow a group of workstations to work together on distributed hardware

Some Operating System Principles

- OS as juggler: providing the illusion of a dedicated machine with infinite memory and CPU.
- OS as government: protecting users from each other, allocating resources efficiently and fairly, and providing secure and safe communication.
- OS as complex system: keeping OS design and implementation as simple as possible is the key to getting the OS to work.
- OS as history teacher: learning from past to predict the future, i.e., OS design tradeoffs change with technology.


disk
disk
printer
tape drive
CPU
Disk Controller
Printer Controller
Tape Drive Controller
System Bus
Memory Controller
Memory

Generic Computer Architecture

- CPU - the processor that performs the actual computation
- I/O devices - terminal, disks, video board, printer, etc.
- Memory - RAM containing data and programs used by the CPU
- System bus - the communication medium between the CPU, memory, and peripherals

OS Provides a High-Level Version of Hardware:

- CPU → Processes, Threads
- Main Memory → Address spaces
- Disk → Hierarchical Filesystem
- Devices → Virtual devices

Example Concepts

Services	Processes
Threads	CPU Scheduling
I/O Redirection	Pipes
Concurrency	Synchronization
Deadlock	Memory Management
Paging	Segmentation
Virtual Memor	Page Replacement
File Systems	I/O Systems
Distributed Systems	Networks
RPC	Distributed Filesystems
Security	Embedded Systems

Architectural Features Motivated by OS Services

OS Service	**Hardware Support**
Protection	Kernel/User mode Protected Instructions Base and Limit Registers
Virtual memory	Translation look-aside buffers
System calls	Trap instructions and trap vectors
Asynchronous	I/O Interrupts
CPU scheduling and accounting	Timer interrupts
Interprocessor communication	Interprocessor interrupts
Synchronization	Atomic instructions

Protection

From who?

ANY CODE

that violates the OS security policy

faulty or malicious

local or remote

Restricted Instructions

Some instructions are restricted to use only by the OS

From **user-mode** you cannot:

- access I/O devices directly
- access out-of-bounds memory
- use instructions that manipulate the state of memory protection
  (page table pointers, TLB load, etc.)
- set the mode bits that determine user or kernel mode
- disable and enable interrupts
- halt the machine

but in **kernel mode**, the OS can do all these things.

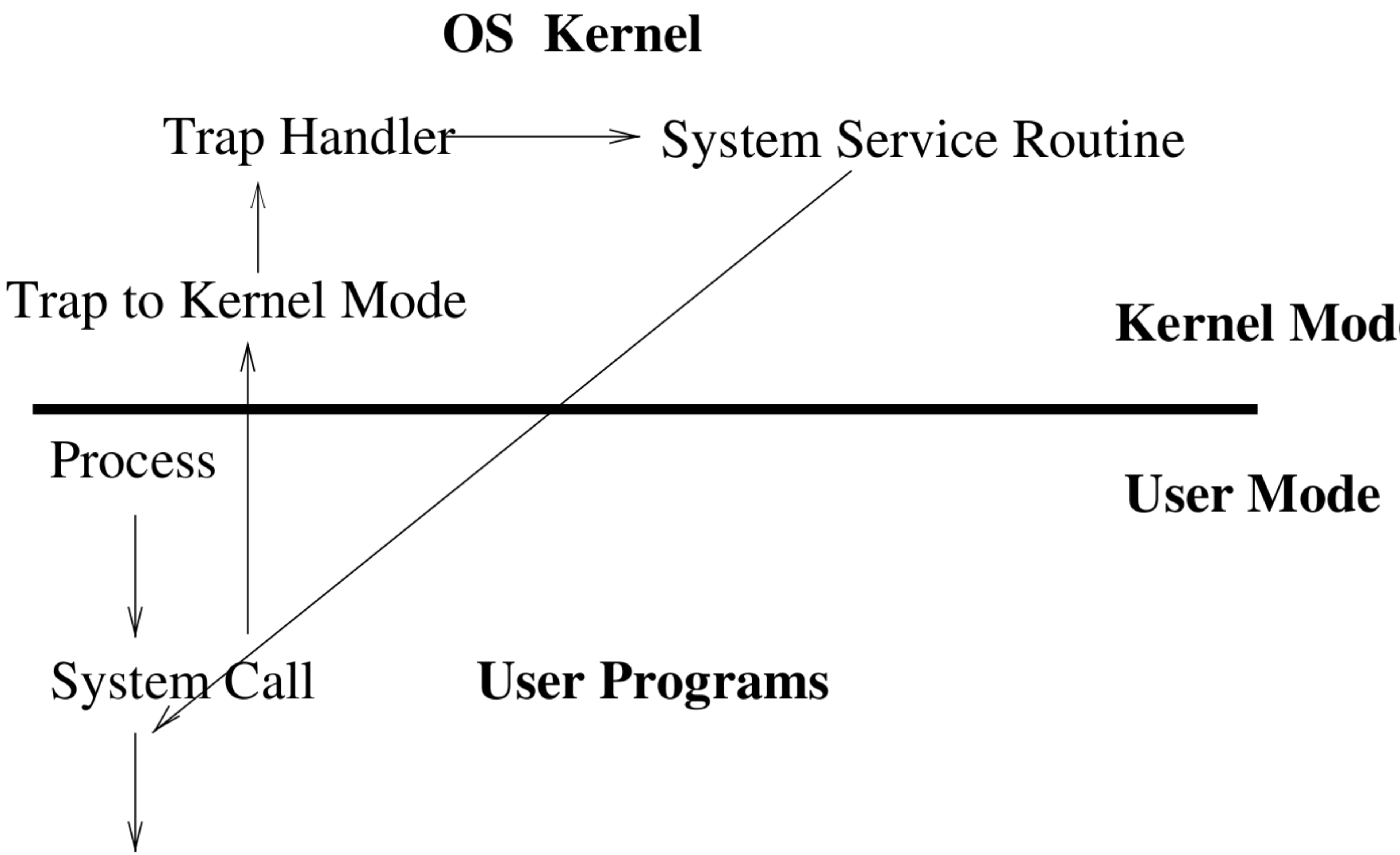
Kernel mode vs. User mode:

The architecture must support at least kernel and user mode.

- A status bit in a protected processor register indicates the mode.
- Protected instructions can only be executed in kernel mode.
- User programs get access to protected functionality through system calls

Important: OS must never permit arbitrary code to be executed in kernel mode

Syscall Kernel Trap



Crossing Protection Boundaries

Users make a system call to an OS procedure to execute privileged instructions (e.g., I/O)

A System Call:

- Program puts system call parameters in registers.
- Program executes a trap:
  - Minimal processor state (PC, PSW) pushed on the stack.
  - CPU vectors (jumps) to the trap handler in the OS kernel.
- The trap handler uses the parameter to the system call to jump to the appropriate handler (fork, exec, open, etc.).
- The architecture must permit the OS to verify the caller's parameters.
- The architecture must also provide a way to return to user mode when finished.

Unix System Calls

Familiar examples: open, close, read, write, select, fork, exec, sleep
Esoteric: brk, setsid, unlink, waitpid

Arguments tend to be:

- pointers to blocks of memory (strings, file buffers, socket buffers)
- integer constants (buffer size, file offset, permission mask, delay time)
- names
  - file descriptor: small int naming a file (process-relative)
  - pid: small int naming a process
  - path: string naming a location in the filesystem

Often, return value of -1 indicates an error.

- See man pages for exceptions
- As a programmer, always always check return codes

Unix System Calls

Principle: Dialogue between user-mode and kernel should be semantically simple.

Why?

Unix System Calls

Principle: Dialogue between user-mode and kernel should be semantically simple.

Why?

- Simple interfaces are easier to work with (from both sides)
- Simple interfaces can sometimes be implemented correctly (complex ones almost never can)
- Simple interfaces tend to be broadly useful
- Simple interfaces tend to have efficient implementations)

Example: A System Call in Linux

You write:

```
static char buf[] = "hello\n";
int main (void)
{
  write (1, buf, 6);
}
```

Example: A System Call in Linux

You write:

```
static char buf[] = "hello\n";
int main (void)
{
  write (1, buf, 6);
}
```

main() contains:

```
movl        $0x1,(%esp,1)
movl        $0x80a034c,0x4(%esp,1)
movl        $0x6,0x8(%esp,1)
call        804cd40 <__libc_write>
```

Example: A System Call in Linux

You write:

```
static char buf[] = "hello\n";
int main (void)
{
  write (1, buf, 6);
}
```

main() contains:

```
movl        $0x1,(%esp,1)
movl        $0x80a034c,0x4(%esp,1)
movl        $0x6,0x8(%esp,1)
call        804cd40 <__libc_write>
```

libc write() is:

```
mov 0x10(%esp,1),%edx
mov 0xc(%esp,1),%ecx
mov 0x8(%esp,1),%ebx
mov $0x4,%eax
int $0x80
cmp $0xfffff001,%eax
jae 804d550 <__syscall_error>
ret
```

Example: A System Call in Linux

You write:

```
static char buf[] = "hello\n";
int main (void)
{
  write (1, buf, 6);
}
```

main() contains:

```
movl        $0x1,(%esp,1)
movl        $0x80a034c,0x4(%esp,1)
movl        $0x6,0x8(%esp,1)
call        804cd40 <__libc_write>
```

libc write() is:

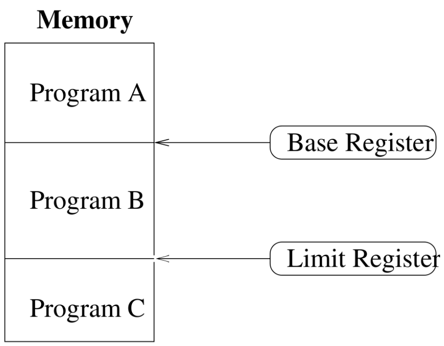
```
mov 0x10(%esp,1),%edx
mov 0xc(%esp,1),%ecx
mov 0x8(%esp,1),%ebx
mov $0x4,%eax
int $0x80
cmp $0xfffff001,%eax
jae 804d550 <__syscall_error>
ret
```

Question: Why load "4" into eax?

Note: For this class you will need to read assembly code at this level

Memory Protection

- Architecture must provide support so the OS can
  - protect user programs from each other, and
  - protect the OS from user programs.
- Architecture may or may not protect User programs from the OS.
- The simplest technique is to use base and limit registers.



- Base and limit registers are loaded by the OS before starting a program.
- The CPU checks each user reference (instruction and data addresses), ensuring it falls between the base and limit register values.
- Virtual memory and segmented memory have additional requirements and more complex solutions.

Virtual Memory

- Virtual memory allows users to run programs without loading the entire program in memory at once.
- Instead, pieces of the program are loaded as they are needed.
- The OS must keep track of which pieces are in which parts of physical memory and which pieces are on disk.
- In order for pieces of the program to be located and loaded without causing a major disruption to the program, the hardware provides a translation lookaside buffer to speed the lookup.

Traps

Architecture must detect special conditions:

- page fault
- write to a read-only page
- bad address trap
- floating point exception
- privileged instruction trap
- system call
- etc...

Traps

We call these traps.

When the processor detects these conditions, it must

- save state on trap (PC, stack, etc.), so that the process may be restarted after the trap is serviced, and then
- The CPU transfers control to the appropriate trap handler (OS routine) via a memory-mapped trap vector.
  - The CPU indexes the trap vector with the trap number,
  - then jumps to the address given in the vector, and
  - starts to execute at that address.
  - When the handler completes, the OS resumes the execution of the process that caused the trap.

Traps

Modern OS use Virtual Memory traps for many functions: debugging, distributed VM, garbage collection, copy-on-write, etc.

Trap Vector:

0:	0x00080000	Illegal Address
1:	0x00100000	Memory Violation
2:	0x00100480	Illegal Instruction
3:	0x00123010	System Call
...	...	

Question: Why are traps so useful?

Traps

Modern OS use Virtual Memory traps for many functions: debugging, distributed VM, garbage collection, copy-on-write, etc.

Trap Vector:

0:	0x00080000	Illegal Address
1:	0x00100000	Memory Violation
2:	0x00100480	Illegal Instruction
3:	0x00123010	System Call
...	...	

Question: Why are traps so useful?

Because they are efficient. Instead of using software to test for a condition, special-purpose logic in the processor does the work without slowing down your application.

I/O Control

Concurrency in I/O

- I/O devices run concurrently with main processor (and may contain their own little processors)
- CPU issues commands to I/O devices, and continues
- Completion of the command is detected in one of two ways:
  - When the I/0 device completes the command, it issues an interrupt: CPU stops whatever it was doing and services the interrupt
  - CPU periodically asks the device if it is done (polling)

The question of polling vs. interrupts is an extremely common theme in programming (and not just for OS code).

Questions: When is polling better? When are interrupts better?

Too Many Interrupts

Some devices (like gigabit network cards) can generate interrupts faster than the OS can handle them. **Receiver livelock** happens when all time is spent handling interrupts and no time is spent doing useful work.

$\Longrightarrow$ Polling cures receiver livelock but overall is a bad alternative because it wastes a lot of CPU time when the device is quiet.

Clever solution: Adaptively switch between interrupts and polling depending on offered load.

Question: How would you decide when to switch? Would it depend on the device, the CPU, or both?