CMPT 300 — Operating Systems I

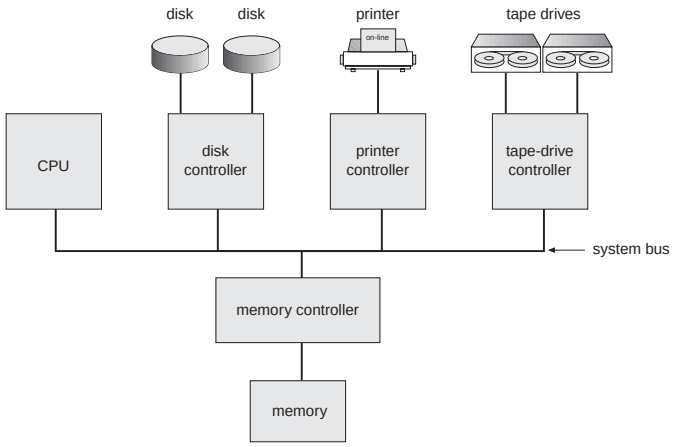
Summer 1999

Segment 3:

Computer System Architecture

Melissa O'Neill

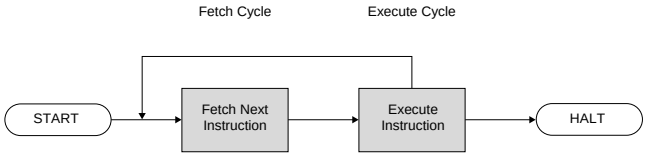
Computer System Architecture



Computer System Operation

- I/O devices and the CPU can execute concurrently.
- Each device controller is in charge of a particular device type.
- Each device controller has a local buffer.
- CPU moves data from main memory to the local buffers and vice versa.
- I/O is from the device to local buffer of controller.
- *Device controller informs CPU that it has finished its operation by causing an interrupt.*

Processor Operation (basic)



- Memory contains program instructions and program data
- Processor registers maintain processor state. Registers include:
  - General Purpose (Address & Data) Registers
  - Instruction Pointer (aka Program Counter)
  - Stack Pointer(s)
  - Control and Status Registers

I/O Structure

Two alternatives — either:

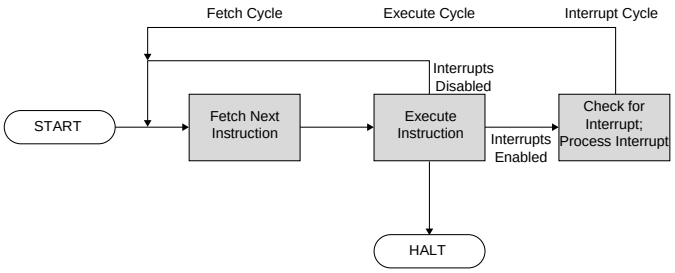
1. *Programmed I/O* — After I/O starts, control returns to user program only on I/O completion.
   - CPU waits until I/O completes.
   - at most one I/O request is outstanding at a time; no simultaneous I/O processing.

I/O Structure (continued)

or:

2. *Interrupt driven I/O* — After I/O starts, control returns to user program without waiting for I/O completion.
   - System call – request to the operating system to allow user to wait for I/O completion.
   - Device-status table contains entry for each I/O device indicating its type, address, and state.
   - Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt.

Processor Operation Revisited (with interrupts)



- Interrupt transfers control to the interrupt service routine, generally, through the interrupt vector, which contains the addresses of all the service routines.
- Interrupt architecture must save the address of the interrupted instruction (and anything else need to resume the code that was running)
- *What if an interrupt occurs during interrupt processing?*

Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter.
- Determines which type of interrupt has occurred, using either
  - polling
  - vectored interrupts
- Separate segments of code determine what action should be taken for each type of interrupt.

Types of Interrupts

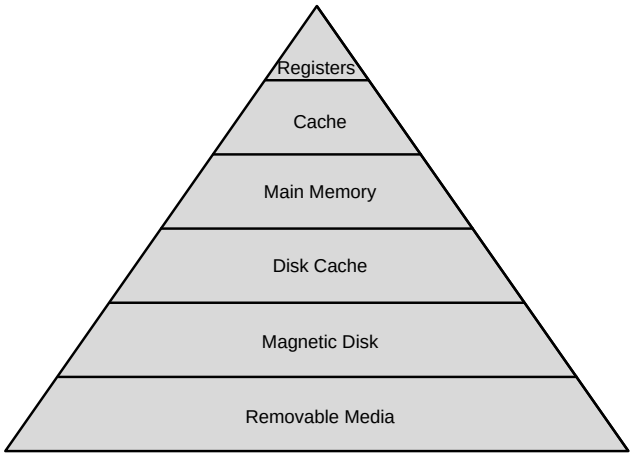
- Software exception (also called a trap)
- Timer
- I/O
- Hardware failure

An operating system is *interrupt driven.*

Storage Hierarchy

- Storage systems organized in hierarchy:
  - speed
  - size
  - cost
  - volatility
- Caching – copying information into faster storage system; main memory can be viewed as a fast cache for secondary storage.

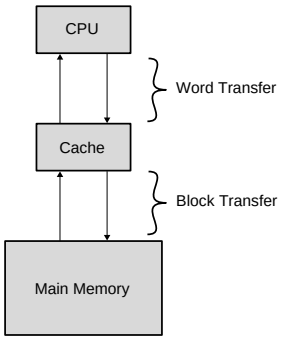
Storage-Device Hierarchy



Cache Issues

- Cache size
- Block size
- Mapping function
- Replacement algorithm
- Write policy

Class Exercise:
What problems might the addition of cache cause?



Dual-Mode Operation

Sharing system resources requires the operating system to ensure that a buggy program cannot cause other programs to execute incorrectly.

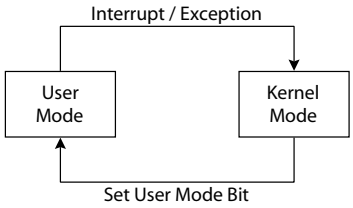
Solution: *Dual-Mode Operation*

- Provide hardware support to differentiate between at least two modes of operations.
  1. User mode – execution done on behalf of a user.
  2. Kernel mode (aka privileged mode, supervisor mode, system mode or monitor mode)—execution done on behalf of operating system.

Class Exercise: Would more modes be helpful?

Dual-Mode Operation (continued)

- *Mode bit* added to computer hardware to indicate the current mode: privileged (0) or user (1).
- When an interrupt or fault occurs, hardware switches to privileged mode:



- Privileged instructions can be issued only in kernel mode.

I/O Protection

We need to protect, the I/O devices from the actions of errant programs.

Solution: *I/O Protection*

- Only the kernel is allowed to interact with the I/O hardware.
- All I/O instructions are privileged instructions.
- Must ensure that a user program can never run in kernel mode (e.g., a user program that, as part of its execution, stores a new address in the interrupt vector).

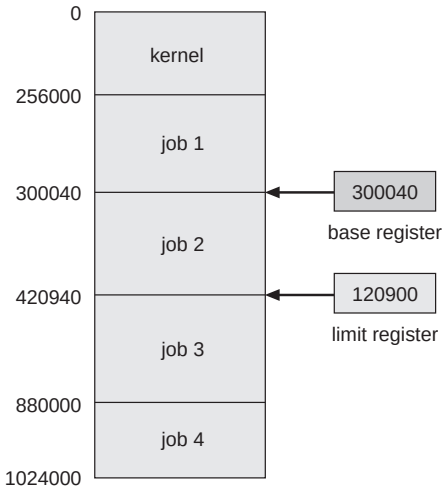
Memory Protection

We need to protect, the interrupt vector, the interrupt service routines and operating system data structures from the actions of errant programs.

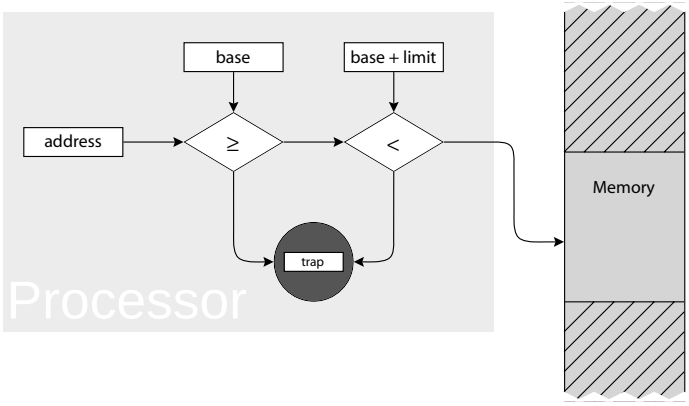
Solution: *Memory Protection*

- Processor can use two special registers that determine the range of legal addresses that a program may access:
  - *Base register* – holds the smallest legal physical memory address.
  - *Limit register* – contains the size of the range.
- Memory outside the defined range may not be accessed by user-mode code.

Memory Protection — Example



Protection Hardware (basic)



- When executing in kernel mode, the process has unrestricted access to all memory.
- The load instructions for the base and limit registers are privileged instructions.

Protection Hardware (other answers)

When I teach CMPT-363, I initially tell my students:

Modes are almost always a mistake ! 

In that case, I'm talking about user interfaces, but perhaps the same lesson applies here?

Class Exercise: Consider the case where a process asks to read one block from the disk. Discover how our memory protection scheme adds additional overhead.

Rethink the idea of modes. What problem are we trying to solve, and how else could we solve it?

CPU Protection

If a program hangs, it shouldn't hang the machine.

- Timer – interrupts computer after specified period to ensure operating system maintains control.
  - Timer is decremented every clock tick.
  - When timer reaches zero, an interrupt occurs.
  - Timer commonly used to implement time sharing.
  - Timer also used to compute the current time.
  - Load-timer is a privileged instruction.

General-System Architecture

Given that I/O instructions are privileged, how does the user program perform I/O?

Solution: *System call*

- System call — the method used by a process to request action by the operating system.
  - Provides only user access to kernel-mode–only functionality.
  - Usually takes the form of a *software interrupt*
  - Is just like an interrupt — control passes through the interrupt vector to a service routine in the OS, and the mode bit is set to kernel mode.
  - Requires that the kernel verify that the parameters are correct and legal, execute the request, and return control to the user instruction following the system call.

System Calls (continued)

- Generally available as assembly-language instructions.
  - Wrappers for these instructions are available for higher-level languages.
- Three general methods are used to pass parameters between a running program and the operating system:
  - Pass parameters in processor registers
  - Store the parameters in a table in memory, passing the table address to the operating system in a register
  - Push the parameters onto the stack

System Calls — Example (hello.S)

Here's our classic "Hello World" program, written using direct system calls.

```
#include <syscall.h>
.data
greeting:   .ascii "Hello World\n"      // As usual...
.text
start:      // Print "Hello World" using write
            movel   #SYS_write, d0
            movel   #1, d1
            movel   #greeting, d2
            movel   #12, d3
            trap    #4                  // Syscall!
            bcs     failed              // Okay/Failed?
okay:       movel   #0, d1
            bra     exit
failed:     movel   d0, d1
exit:       movel   #SYS_exit, d0
            trap    #4                  // Syscall!
            illegal                     // Crazy! Panic!
```


System Call Wrappers

Programming languages like *C* generally want to see system calls as a function in their language, so most operating systems define library wrappers for their system calls.

- The *C* Library on NEXTSTEP (m68k) has this wrapper for the rename system call:

```
rename:
            movel   sp,a0               // args on stack
            movel   d2,sp@-             // save registers
            movel   #SYS_rename,d0
            movel   a0@(0x4:w),d1       // old filename
            movel   a0@(0x8:w),d2       // new filename
            trap    #4                  // Syscall!
            bcc     rename_okay         // Okay/Failed?
rename_failed:
            jsr     cerror:l            // set errno
rename_okay:
            movel   sp@+,d2             ; restore d2
            rts
```

Beyond System Calls — Library Interfaces

System calls tend to be minimal and low level. Generally, we prefer to use higher-level routines found in libraries provided with the operating system.

Class Exercise: What is the key difference between system calls and library calls?

Beyond System Calls — Operating System Shells and System Programs

- "User interface" for the operating system
- Allow control of:
  - Process creation and management
  - I/O handling
  - Secondary-storage management
  - Main-memory management
  - File-system access
  - Protection
  - Networking

Beyond System Calls — Operating System Shells and System Programs (continued)

- Shells include:
  - Control-card interpreters
  - Command line interpreters
  - GUI-based managers
- To many users, these programs *are* the operating system (even if the operating system core sees them as user-mode programs)

System Design Goals

- User goals
  - Convenient to use
  - Easy to learn
  - Reliable
  - Safe
  - Fast
- System goals
  - Easy to design, implement, and maintain
  - Flexible
  - Reliable
  - Error-free
  - Efficient

Mechanism and Policy

- Mechanisms determine how to do something
- Policies decide what will be done.
- Policy may vary while mechanism changes (and different mechanisms can implement the same policy).

Class Exercise: Some examples of mechanism and policy…

System Implementation

- Traditionally written in assembly language, now often written in higher-level languages.
- Code written in a high-level language:
  - Can be written faster.
  - Is more compact.
  - Is easier to understand and debug.
- An operating system is far easier to port (move to some other hardware) if it is written in a high-level language.

System Structure — Non-modular Approach

- One big program, not very modular; examples
  - MS-DOS
  - Early Unix kernels
- Maintenance nightmare

System Structure — Layered Approach

- OS functionality built up through layers, with each layer depending on the previous layers

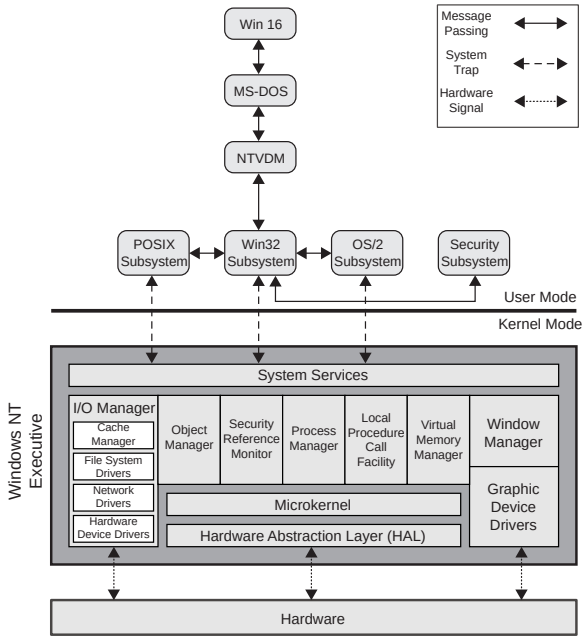
Class Exercise: You're asked to design a small OS with about five layers?

What would they be?

(No peeking at your textbook!)

System Structure — Layered Approach (contd)

- Stallings lists a thirteen-layer conceptual framework
- Real operating systems are a little different



Segment Review

You should be able to:

- Determine which aspects of a computer system operate concurrently
- Compare and contrast programmed and interrupt-driven I/O
- Explain *hardware protection* and design hardware protection schemes
- Explain the trade-offs associated with caching