CS571: Operating Systems

CLASS 1 : 27 AUG 2003
16:30 – 19:10

Class 1 Overview

- course details
- review of computer architecture
- OS as service provider
- The Process
CS571: Where and When

- ‘live’ classroom presentation in STI:126, Wednesdays 16:30 to 19:10
- live over the Internet using GMU’s NEW software for distance learning:
  [http://netlab.gmu.edu/disted](http://netlab.gmu.edu/disted)
CS571: Where and When

- ‘live’ classroom presentation in STI:126, Wednesdays 16:30 to 19:10
- live over the Internet using GMU’s NEW software for distance learning: http://netlab.gmu.edu/disted
- recorded as given in class, may be played back anytime

NEW: Network Education Ware

- provides, live, in real-time:
  - overheads of class presentation with
  - instructor annotations and
  - instructor voice, synchronized to overheads
  - live video of classroom
- lets students ask questions:
  - by typing them in
  - by speaking them (if client does audio)
- entire lecture recorded for later Internet playback (excluding live video)
Using NEW

- need client software; runs only on Windows platforms
- minimal machine configuration:
  - P200 or faster
  - IE 4.0 or later, Netscape (6.0 or later)
  - standard sound card
  - ≥ 56 kbps connection to Internet
- see http://netlab.gmu.edu/disted to try it and/or sign up (need password to get courses)

Course Resources

- required text:
  Applied Operating System Concepts
  Silberschatz, Galvin and Gagné
  John Wiley ISBN: 0–471–26314–1
- course web site:
  http://cs.gmu.edu/~csnow/cs571/2003A
  - contains printed copies of slides (pdf)
  - other course related info
  - background material
Resources: Instructor + TA

- Instructor: Charles Snow  
  - office hours: Wed 14:00 to 15:30  
  - ST2, room 435 (across from CS dept office)  
  - email: csnow@cs.gmu.edu

- TA: TBA (see course website for update)

Resources: Systems & Software

- students must have IT&E UNIX system accounts  
- students will need ssh clients for remote access  
- programming assignments are to be written in C unless otherwise noted
Students must:

- course pre-requisites:
  - CS310 Computer Science III,
  - CS365 Computer Systems Architecture

- be able to program in C

CS571 Grading:

- Assigned work: 35%
- Midterm: 25%
- Final exam: 40%

- assignments due by end-of-day on due date
  - assignments handed via
    http://cs.gmu.edu/course-upload

- grades are proficiency based
CS571 Grading, cont’d:

- important that you demonstrate:
  - understanding of material
  - ability to reason with it
- final exam
  - is comprehensive
  - scheduled for ?? Dec 2003, 16:30 to 19:15
- basic rule: if we cover it in class, it can be on an exam
  - includes reading material suggested but not necessarily explicitly covered in class

Class 1 Overview

- course details
- review of computer architecture
- OS as service provider
- The Process
What is a computer?

• what’s a computer?
  • has one (or more) processing units: why?

• what’s a computer?
  • has one (or more) processing units
  • has memory where instructions and data reside
What is a computer? What’s an OS?

- what’s a computer?
  - has one (or more) processing units
  - has memory where instructions and data reside
  - has one (or more) peripheral units to allow for:
    - interaction with other components
    - persistence of data
What is a computer?

- what's a computer?
  - has one (or more) processing units
  - has memory where instructions and data reside
  - has one (or more) peripheral units to allow for:
    - interaction with other components
    - persistence of data
  - has necessary electrical and mechanical infrastructure needed to operate its components
A basic computer:

- **CPU**:
  - executes instructions retrieved from memory,
  - has registers and status word(s),
  - is bus master most of the time,
  - may surrender bus to other devices for brief intervals,
  - can generate traps for failure conditions

- **BUS**:
  - provides data and control signal connection between units,
  - transfers data between a source and a destination as requested by bus master,
  - can generate traps for certain conditions,
  - ‘creates’ the address space: most addresses are in memory, some are I/O devices (memory mapped I/O)
A basic computer:

MEMORY:
- stores bit patterns (instr, data),
- organized as flat, linear sequence of bytes,
- is byte addressable (usually),
- may generate traps for error conditions

memories...

- place to hold a value (data or instruction) while it is worked on or until it is needed to be worked on
Memories...

- place to hold a value (data or instruction) while it is worked on or until it is needed to be worked on
- may require electrical power to hold a value: volatile memory
- may not require electrical power: non-volatile memory
  - earliest computer memories, magnetic core, were non-volatile

Memories...

- not all memories are the same:
  - smaller access time ⇔ higher co$t per bit
  - bigger capacity ⇔ smaller co$t per bit
  - higher capacity ⇔ higher access time
Memories...

REGISTERS:
- contained within CPU
- GPRs: general purpose registers
- PC: program counter
- PSW: program status word
- access at CPU speed
- volatile

CACHE:
- small amount of high-speed mem between CPU and bus
- may be multi-staged
- volatile

CPU

memory
input device
output device

cache
Memories…

**MAIN MEMORY:**
- byte addressable
- flat, linear address space
- typical sizes: 256 Mb to 2 Gb
- volatile

**DISKS (magnetic, optical):**
- are peripheral devices
- interact through interfaces/controllers
- magnetics: now in 100s of Gb
- opticals: now in 10s of Gb
- mechanical access method (e.g., read/write head must be moved to correct position)
- non-volatile
  - magnetics subject to magnetic fields
Memories…

**MAGNETIC TAPE:**
- are peripheral devices
- interact through interfaces/ controllers
- now in 100s of Gb
- mechanical access method: tape must be moved to correct position)
- sequential access only
- non-volatile
  - magnetics subject to magnetic fields

- lower cost per bit
- higher capacity
- higher access time
- lower frequency of access by CPU

small, expensive, fast memory supplemented by larger, cheaper, slower memory
About caches

- cache contains a copy of small piece of main memory
  - usually a block ("line") of n consecutive bytes
- CPU accesses an address: if not present in cache, ⇒ cache miss
  - get contents of memory location from memory
  - transfer block of n consecutive bytes of memory into cache
  - next access to same address or ‘nearby’ address will be in cache, saving access time
- cache hit: accessed location is in cache

\[
\text{cache hit ratio} = \frac{\text{number of accesses content in cache}}{\text{total number of accesses}}
\]

- how high are typically cache hit ratios?
About caches

cache hit ratio = \( \frac{\text{number of accesses content in cache}}{\text{total number of accesses}} \)

- how high are typically cache hit ratios?
  - 95% or better

- how is this possible?
  - locality: how close together addresses are that are accessed
    - e.g., if PC points to a word at address k, there is high probability that next PC access is word at address k+4, i.e., next word in memory
  - operands often show similar local clustering
  - but operands and instructions may not be local w.r.t. each other ⇒ separate caches for I and D
  - example: loop to zero an array...
About caches

```c
#define SIZE 500
main()
{
    int i,j;
    int fobble[SIZE];
    for (i = 0; i < SIZE; i++) {
        fobble[i] = 0;
    }
}
```

L30:
```
    mov %g0,%i5
cmp %g0,500
bge .L97
nop
```

L_y0:
```
    add %fp,-2012,%o2
```

L98:
```
    sll %i5,2,%o0
    st %g0,[%o2+%o0]
    add %i5,1,%i5
    cmp %i5,500
    bl .L95
    nop
```

Our improved computer:

- why do input/output devices have bi-directional connection to bus?
  - data
  - control/status
Input/Output Devices

- may be polled:
  - check device to see if it needs attention
    - yes: do what needs to be done and go back to checking
    - no: go back to checking

Input/Output Devices: Polling

- may be polled:
  - e.g., keyboard:
    - while (no new data to read);
    - loop takes ≈ e.g., 15 ns (> 65e6 iterations/sec)
    - how fast do you type?
    - what else is CPU doing when it's doing this?
Input/Output Devices: Using Interrupts

- may be polled
- may be interrupt-driven:
  - when a device needs attention it uses an interrupt to assert the need for service
  - CPU:
    - suspends current processing
    - executes a special routine that performs the service (an interrupt handler)
    - can then resume suspended work
  - are all interrupts equally important?

- interrupts have different priorities:
  - are processed by priority
  - an interrupt can interrupt an interrupt handler
Input/Output Devices: Using Interrupts

- Interrupts have different priorities:
  - Are processed by priority
  - An interrupt can interrupt an interrupt handler

- Keyboard example:
  - When data is ready, device asserts interrupt
  - CPU switches from one context to another
    - That of the interrupt handler
    - Switching designed to be fast (e.g., 40 ns)
    - Keyboard handler is short (moves 1 byte: e.g., 10 ns)
    - Return to previous context (e.g., 20 ns)

Fast Input/Output Devices

- E.g., a hard disk (read)
  - Delivers data at, e.g., 25 Mb/sec
    - So new byte ready every 38 ns
  - If use interrupts, can get a byte every ≈ 50 ns
  - Bottleneck is context switch & CPU intervention every single byte
Fast Input/Output Devices: DMA

- cut out the middleman: let device move data on bus directly

- called direct memory access (DMA)
- what can CPU do while DMA is happening?

Fast Input/Output Devices: DMA

- requires:
  - "smart" devices that are capable of interacting directly with bus (becoming bus master)
  - set-up for transfer: device is given instructions
    - e.g., head/sector/cyl, count, read (or write)
  - DMA xfer begins some time later, when device is ready
- such transfers are usually small in size
  - i.e., short duration
- why?
What the CPU does:

- CPU repeatedly performs this loop:
  1. fetch word from memory pointed at by PC
  2. decode word as instruction
  3. fetch any operand(s) instruction needs
  4. execute the instruction
  5. update status flags as consequence (e.g., NZVC)
  6. store any result(s)

What could possibly go wrong?

- CPU repeatedly performs this loop:
  1. fetch word from memory pointed at by PC
  2. decode word as instruction
  3. fetch any operand(s) instruction needs
  4. execute the instruction
  5. update status flags as consequence (e.g., NZVC)
  6. store any result(s)

  - Address in PC may not correspond to a word in memory,
  - Address may not be readable

  ⇒ bus/operand error
What could possible go wrong?

- CPU repeatedly performs this loop:
  1. fetch word from memory pointed at by PC
  2. decode word as instruction
  3. fetch any operand(s) instruction needs
  4. execute the instruction
  5. update status flags as consequence (e.g., NZVC)
  6. store

  ⇒ illegal instruction error

- Address may not correspond to a location where there is anything in bus-space,
  ⇒ bus/operand error
What could possibly go wrong?

- CPU repeatedly performs this loop:
  1. fetch word from memory pointed at by PC
  2. decode word as instruction
  3. fetch any operand(s) instruction needs
  4. execute the instruction
  5. update status flags as consequence (e.g., NZVC)
  6. store any result(s)

  Instruction may result in an error condition, e.g., integer or floating-point divide by 0, integer overflow

  ⇒ execution error

What could possibly go wrong?

- CPU repeatedly performs this loop:
  1. fetch word from memory pointed at by PC
  2. decode
  3. fetch any operand(s)
  4. execute ⇒ bus/operand error
  5. update status flags as consequence (e.g., NZVC)
  6. store any result(s)

  Address may not correspond to a location where there is anything in bus-space,
  Address may not be writeable
Dealing with errors:

- any of these error conditions cause a **trap**.
- upon a trap happening, CPU:
  - suspends current processing
  - executes a special routine that performs the service (a **trap handler**)
  - could resume suspended work, but usually doesn’t because it usually doesn’t make sense to

- sounds just like interrupts, **except**:
  - traps don’t have priorities
  - traps are handled instantly as they occur
What the operating system operates:

- basic hardware configuration

```
+-------+      +------+
| memory|      | cache |
|       |      | CPU   |
| input device          BUS              output device
```

Class 1 Overview

- course details
- review of computer architecture
- OS as service provider
- The Process
What the OS gives us:

- a set of resources created, maintained and controlled by the OS
- presented as a set of services for use

Different OS for different settings:

- OS can make resource collection apt for one or another kind of use, e.g.,
  - batch: run programs, often from beginning to end, in their entirety (e.g., VISA month-end billing)
    - typically what “mainframe” computers did
  - interactive: run programs of many different concurrent users, usually logged in on terminals
  - real–time: run program(s) subject to the constraint that some events must occur within certain time, e.g., autopilot
- generally can do any of these with ‘plain’ hardware: the OS makes the difference
OS as supervisor:

- computers provide (in hardware) for different levels of user:
  - not all instructions are available at all levels

- OS uses higher privilege level instructions to manage machine's resources

- ‘user’ programs usually have least privilege level

OS as supervisor:

- e.g., when running a user program (lowest privilege level) an interrupt occurs:
  - suspend work on user’s stuff (so that it can be resumed)
  - get address of interrupt handler
  - raise priority level to ‘supervisor’ level to perform handler
  - often handled simply through PSW (holds priority and privilege level)
Ask the OS…

- user program (low privilege) cannot, e.g.,
  - directly control an I/O device like a disk
- so how does a user get disk input to program?

Ask the OS…

- user program (low privilege) cannot, e.g.,
  - directly control an I/O device like a disk
- user program asks the OS to do the I/O work on the user’s behalf: it makes a supervisor call to invoke OS services
- what kinds of services does the OS provide?
OS: Service Provider

1. **Program Execution:**
   - load program into memory,
   - run it
   - deal with its ending (normal or otherwise)

2. **I/O Operation:** operates devices on behalf of users

3. **File System:** structure, protection, access, operation

4. **Communications:** between programs
   - on same system
   - on different systems

5. **Error Detection & Recovery**
   - traps described earlier: program errors
   - device errors/failure: disk errors, network down, …)

6. **Resource allocation:**
   - brokers access to resources to programs
   - some allocated at start-up: cpu, memory
   - some allocated dynamically: memory, disk space

7. **Accounting:** who did what, when, & for how long

8. **Protection** of resources under OS management against accidental or intentional attack
OS: Service Provider

- these are the services OS provides, so these are what supervisor calls (a.k.a.: system calls) invoke
- user programs (e.g., in C) rarely invoke system calls directly
  - use library functions that invoke the system call
  - e.g., `read(fd, mybuf, COUNT);`

OS: Where it Lives

- the OS is a collection of instructions and data items
OS: Where it Lives

- the OS is a collection of instructions and data items ⇒ must live in memory
- how much of it has to be present at all times?
  - some basic OS code
  - interrupt and trap handlers (and their vectors)
- the part of the OS always resident in memory is the kernel

Class 1 Overview

- course details
- review of computer architecture
- OS as service provider
- The Process
The Consumer

- who uses the resources managed by the OS?
- services are intended for the running of programs
- we introduce a model for such a ‘consumer’ of these services: the process

The Process

- standard model of user of OS services/resources
- is a program, hence uses:
  - cpu
  - memory
  - possibly, other resources
    - e.g., disk, graphics display, network
- has a status
  - e.g., running, waiting
- because it’s standard, OS has standard way to handle
The Process

- each new process gets:
  - a unique identifier (PID: process ID)
  - an amount of memory to work with (apart from its instruction space)
  - a privilege level (can do some things, can't do others)
  - possibly, some or all of resources requested up-front