

CSE 120—Principles of Operating Systems

Summer, 2005

Day 1

Introduction

Overview

Threads and Processes

Instructor: Neil Rhodes

Introduction

What will you be learning?

- What an operating system is
 - Various capabilities provided by an OS
- Distinction between an OS/Kernel and the rest of the system
- How an operating system is written
- The spectrum of services provided by an OS
 - Lightweight/small/fast
 - More capable/larger/slower

Why?

- All programmers interact with an OS
- Some programmers must write an OS (or modify)

What is the structure?

- Combination of theory and practice

People

Instructor: Neil Rhodes

Lecture TA: Samory Kpotufe

Projects TA: Bhavjit Walha

Times

Class: 11:30-2:30 Monday Wednesday

- break

Lecture Discussion section

- 3-4 Wednesday (Samory) Peterson 102

Projects Discussion section

- 4-5 Monday (Bhavjit) Peterson 102

Office Hours

- Instructor: M/W 2-3
- Samory: TBD

Attended Lab Hours (EBU1-3327)

- Monday
 - 8PM-9PM (Instructor)
- Tuesday
 - 7-10PM (Bhavjit)
- Wednesday
 - 8PM-9PM (Instructor)
- Thursday
 - 5-6:50 PM (Bhavjit)

Administrative Details

Homework

- Assigned every week
- Need not be turned in

Reading

- Textbook
 - Modern Operating Systems by Tanenbaum
- Read assigned reading before class
- Print slides before class (available sometime the previous day)
- Take notes on slides during class
- Read slides (including notes) after class
- Read assigned reading again, after class

Quizzes

- From first day to last day, exclusive:)
- Taken directly from homework

Midterm

- Day 6

Final

- Saturday, fifth week (7/30/05) 8-11 AM Center 216

Projects!

...

Five weeks is not much time! This class will take a *significant* amount of outside time

Projects

Three programming projects

- Work in groups of 1-4
 - We need group assignments ASAP (no later than Wednesday at class).
 - Email group assignments to Bhavjit
 - One email: "Our group consists of Sally Smith, Joe Jones, Aaron Anderson, and Mike Martin"
- Not much time: Projects due 3rd, 4th, and 5th week
- Modifying an existing operating system: Nachos
 - Nachos is written in Java. You'll modify/extend the OS by writing Java code
 - Nachos runs applications for a MIPS architecture. It includes a simulator for a MIPS machine.
- Important overview discussion section
 - This Thursday, 5-6:50 PM EBU1-3327

Resources

Webpage (including announcements, schedules, etc.)

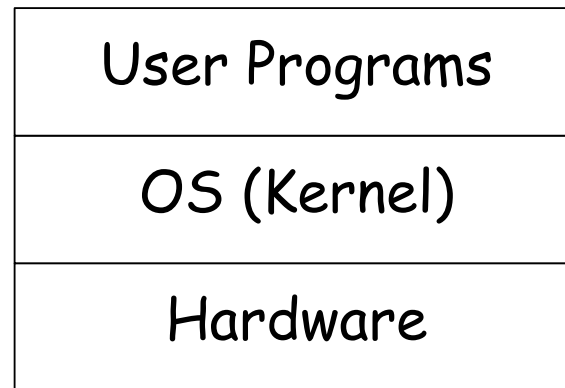
- <http://www.cse.ucsd.edu/classes/su05/cse120>

Discussion board (monitored by Instructor and Bhavjit)

- <http://www.discus.ucsd.edu>

Operating System

A level of software between programs and the raw hardware



History

First generation—Direct control

- Machine is used directly by a single person at a time
- All programmed in assembly

Second generation—Single program

- High-level languages
- One program runs at a time; has complete control of the machine
- Batch processing
- Library routines
 - Output to printer
 - Write to tape
- Procedure
 - Card decks handed to operators.
 - Operators load batches of jobs
 - Programmer waits for output

Third generation—Multiple programs (System/360)

- Multiple jobs loaded into memory at once
 - Hardware support usually provided to protect one program from another
- Can run one program while another is waiting
 - Usually done preemptively
- Timesharing introduced

Fourth generation—Personal Computers

- GUI
- Direct control again

Operating System Distinctions

PCs not the only game out there

- cellphones
- PDAs
- MP3 players

Minimal operating system

- Low-level interface to I/O and memory

More advanced features

- Multiple applications
 - Cooperation vs.
 - Preemption
- Virtual memory
- File system

A major distinction

- Supervisor mode/kernel—applications are prohibited from certain operations; only the kernel can do them
- Does the operating system enforce restrictions, or is it a convention?

For most of this class, we'll look at “modern” operating systems, like those on Unix/Linux/Windows XP/Mac OS X

What an Operating System Does (one view)

Manages resources needed by various programs

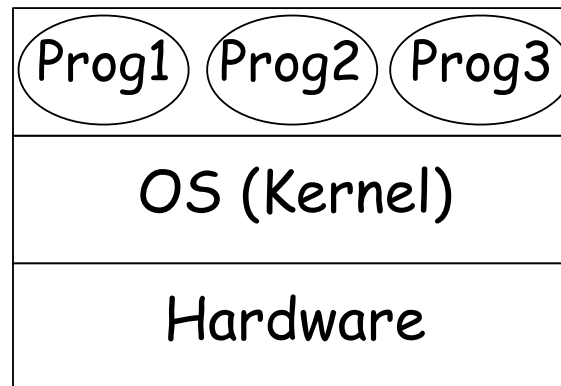
- Memory
- CPU
- Disk
- Peripherals
 - Handles conflicts between various programs

Time multiplexing

- CPU
- Mouse
- Printer

Space multiplexing

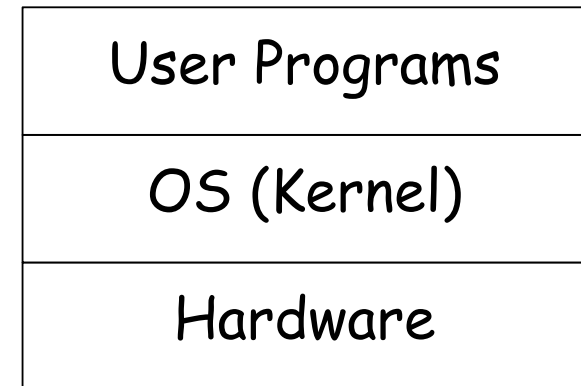
- Memory
- Disk



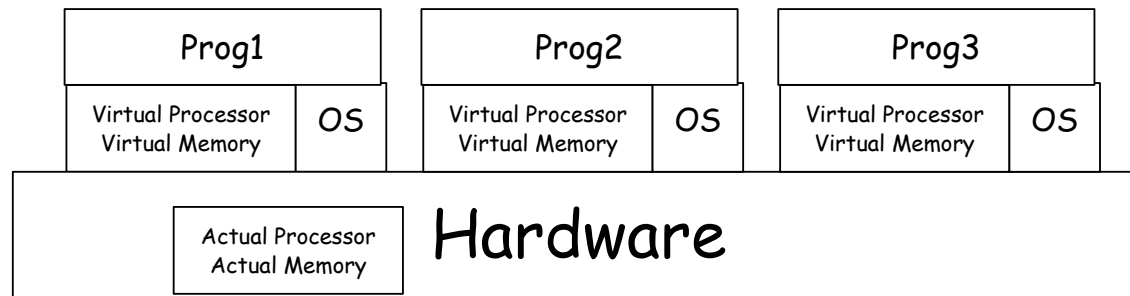
What an Operating System Does (another view)

Operating system as extended machine

- The operating system provides an interface higher than that of the raw machine
 - Level 1: write to the floppy disk controller directly
 - Start motor spinning
 - Move the disk arm
 - ...
 - Stop motor spinning
 - Level 2: write to floppy disk sector-by-sector
 - Impose your own structure on the floppy.
 - Level 3: Open/close files.
 - use file system
- Provide abstractions
 - Same low-level functions to write to IDE drive as SCSI drive as USB drive
 - Same high-level functions to write to a file on disk as to a network connection



What an Operating System Does



The Kernel

Uses supervisor mode

- Certain operations only available if processor is running in supervisor mode
 - Enable/Disable *interrupts*
 - Change memory map
 - Switch to supervisor mode

Normal applications run in user mode

- Hardware disallows some operations
- How to call to kernel?
- Make System call
 - Put system call number and parameters in special location (registers?)
 - Issue special TRAP instruction
 - Causes execution to switch to kernel's trap handler (now in supervisor mode)
 - Makes call to appropriate system call handler
 - Returns to user mode with special instruction

Handling input

- I/O device generates an *interrupt*
- Causes execution to switch to kernel's interrupt handler (now in supervisor mode)
- Deals with that particular interrupt
- Returns to where it was interrupted from with special instruction

Overview of the Quarter

Processes/Threads

Synchronization/Inter-process Communication (IPC)

Scheduling/Deadlock

Memory Management/Virtual Memory

Page Replacement/Segmentation

File Systems

Input/Output

Multiple Processor Systems

Computer System Design

Processes/Threads

Multiprocessing:

- Multiprogramming: One CPU switching quickly between various programs
- Multiprocessor: Multiple CPUs each running a program simultaneously

All Software Organized as Processes

- Program + state information
 - Code + register values, stack, memory, ...
- Analogy
 - Program $\approx$ Class
 - Process $\approx$ Object

Kernel switches between processes

- Cooperative: Each process *yields* when willing to give up control
- Preemptive: Timer goes off every so often. (How often?)
- *Context Switch*: switching from one process to another:
 - Swap registers
 - including Program Counter (PC) and Stack Pointer (SP)
 - Change memory map

Each process has the illusion that it runs continuously, independent of other processes

- At times, the programmer must be aware that it's just an illusion

Advantages of Multiprogramming

Easier to organize/abstract

- Imagine if entirety of GNU/Linux was one giant program!

Better Throughput

- If process A is waiting for a read from disk to complete, process B can profitably use the CPU

Potential Problems

- Process that have real-time requirements (deadlines)
 - Video, Audio good examples

Memory Model of a Process

Text: The code

Data:

- The global variables
- The dynamically allocated memory (heap)

Stack

- activation records (stack frames). One per in-progress subroutine
 - Parameters
 - Saved program counter
 - Pointer to previous activation record
 - Any saved registers
 - Local variables

All are memory addresses in the process's *address space*

How Processes are Created

The first process is created by hand (at boot time)

Remaining processes are spawned from existing processes

- Unix: `fork()`
- Windows: `CreateProcess(...)`

Examples

- `% ls`

Sharing the CPU

Three processes: their illusion

Three processes: the reality

Each process gets a slice of time (the *quantum* or *timeslice*)

How Processes are Destroyed

Normal exit—voluntary

- return from main()

Error exit—voluntary

- exit(error_code)

Fatal error—involuntary

- Divide-by-zero
- Segmentation violation
- Bus error
- ...

Killed by another process—involuntary

- % kill -SIGKILL 1234 # sends SIGKILL signal (9) to process ID 1234
- or, kill(SIGKILL, 1234)

How Time is Shared (Cooperative)

Cooperative multiprogramming

Process A calls Yield()

- next process in list of waiting process now gets to run.

What does Yield do?

- Must do Context Switch
- Switch registers
- Switch memory map

Yield must cause execution to begin in Process B

- Where?

When Process B calls yield, Process A runs

- From A's point of view, it called yield, which just returned (even though B executed in the meantime).

Details of Context Switch

How does Yield work?

One implementation

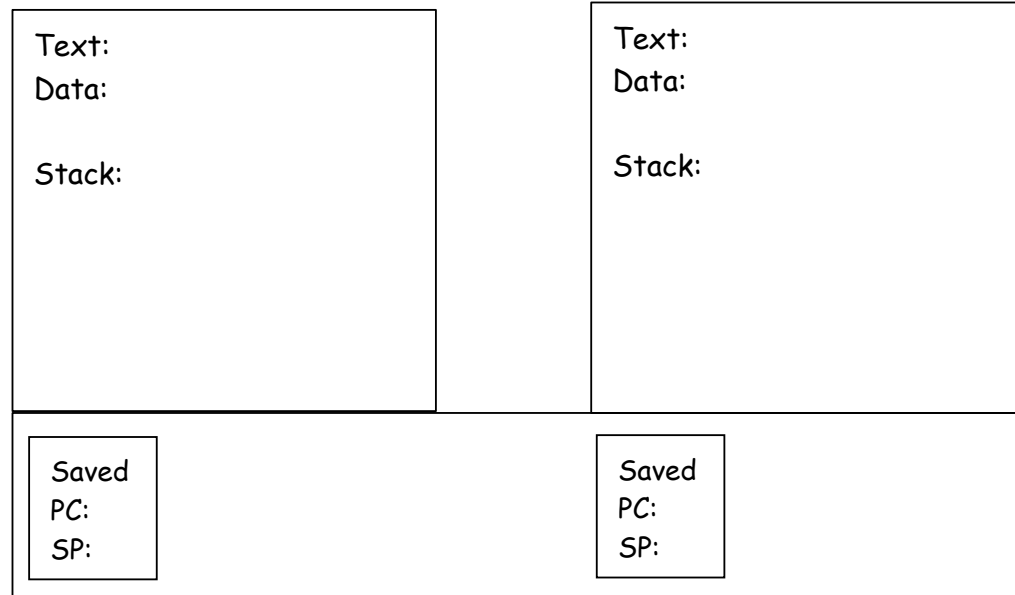
```
▪ Yield()  
{  
    volatile int magic = 0;  
  
    Save context of current process (memory map, general registers, SP, PC)  
    if (magic == 1)  
        return;  
    magic = 1;  
    restore context of next process (memory map, general registers, SP, PC)  
}
```

Context-switch Example

A:
int y = 1;
void main() {
 ...
 Yield();
 y = 2;
}

B:
int x = 10;
void main() {
 Yield();
 x = 11;
 Yield();
 x = 12;
}

```
Yield()  
{  
    volatile int magic = 0;  
  
    Save context of current process  
    if (magic == 1)  
        return;  
    magic = 1;  
    restore context of next process  
}
```



How is Time Shared? (Preemption)

Preemptive

- Advantages:
 - Doesn't require cooperation from all applications
 - Doesn't require special code in an application
- Disadvantages:
 - OS might not have as much information as application
 - Context switch occurs anytime, rather than at well-defined locations in code

How implemented?

- Kernel starts hardware timer
- If process finishes before timer expires, no problem
 - Makes blocking I/O call
 - Gives up CPU explicitly (yield)
 - Process destroyed
- Otherwise, timer interrupt occurs
 - Hardware dispatches to interrupt handler
 - Interrupt handler saves state of process A
 - Figures out which is the next eligible process
 - Restores state of process B
- When process A runs again, it'll continue from where it left off

Possible States of a Process

Running

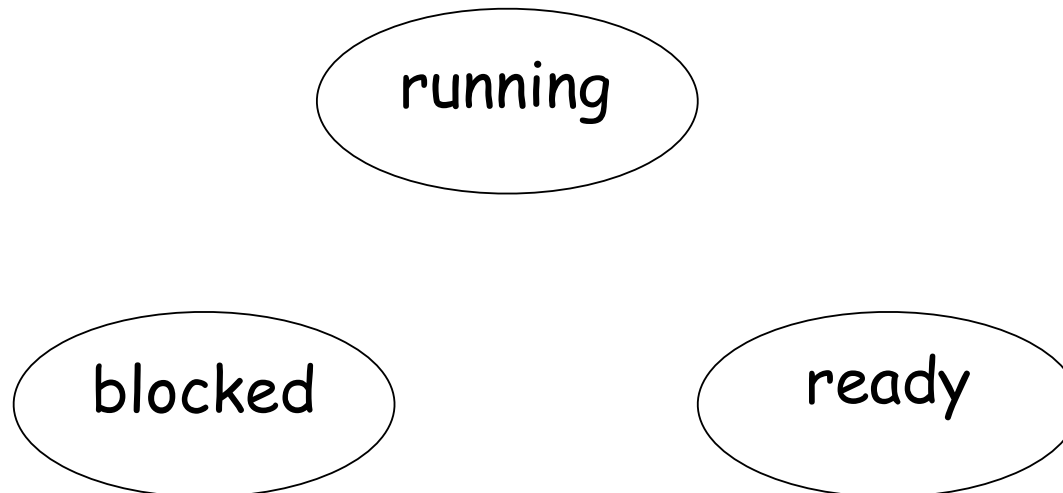
- Actually executing on the CPU
- Only one process can be in this state

Ready

- Ready to use the CPU (in a queue).
- What is the ordering of the queue?

Blocked

- Not able to use the CPU (waiting for something)



Context-switching/Scheduling

Context-switching

- Switching from one process to another
- Has overhead because
 - Requires system call
 - Change memory map
 - May flush cache

Scheduling

- Deciding which process should next run
- We'll see more about this next class

Process table

Kernel stores information about each process in a table

Entries in a table: Process Control Block (PCB)

- Program Counter
- Stack Pointer
- Other registers
- Open files/sockets
- Scheduling information
- Memory map information
- Priority
- History
- Accounting information
 - CPU time spent by this process
 - CPU time spent by the kernel on behalf of this process

Kernel vs. Process

Kernel supports processes (is not a process itself)

Well-defined entry to the kernel

- Trap: system call by application
- Interrupt: generated by hardware (timer, I/O device, etc.)

How does kernel get control?

- Hardware (on trap or interrupt)
 - Switches from user mode to supervisor mode
 - Hardware determines new PC using a vector (Stored at predefined location)
 - 68K: 16 traps. x86: 256 software traps
- Software (trap or interrupt handler)
 - Stores state
 - Calls appropriate Kernel routine (based on specific trap or interrupt)
 - Restores state (if context switch, does *not* restore state).

user mode/supervisor mode

- Usually stored in Processor Status Word (PSW)
- In user mode, some instructions not allowed
 - Write PSW (Read PSW?)
 - Change memory map
 - Others that could breach security

Threads

Threads are like lightweight processes

Processes have their own:

- Address space
- Open Files

Threads are within a given process

- Share the same address space (memory)
- Share open files
- Share other resources

Can be cooperative or preemptive

User-level/Kernel-level threads

User-level (sometimes called *fibers*)

- Kernel knows nothing about threads
- Advantages:
 - Can be added to *any* OS
 - Lightweight context-switch between threads (no kernel call)
 - Ability to have process-specific scheduling algorithm
 - Context-switch happens at well-defined times; can reduce race conditions
- Disadvantages
 - If a thread blocks on an I/O call, all threads in the process are blocked
 - Must wait on a thread to yield before another thread in the process runs
 - Unless roll own preemption with alarms

Kernel-level

- Kernel knows about threads, will schedule/context-switch, etc.
- Advantages:
 - Can schedule preemptively within a process
 - Can schedule another thread within a process if one thread blocks
- Disadvantages:
 - Heavier-weight context switch
 - If many threads, kernel must store lots of state information

Thread table

- Per-thread saved information:
 - Registers (including PC)
 - Stack
 - State

Multithreaded Process

OS with kernel-level threads:

- Process starts with one thread

If process spawns another thread, it is now *multithreaded*.

Issues of synchronization between threads/process and Inter-Process Communication (IPC)

