CS162
Operating Systems and Systems Programming
Lecture 2

Concurrency: Processes, Threads, and Address Spaces

January 24th, 2011

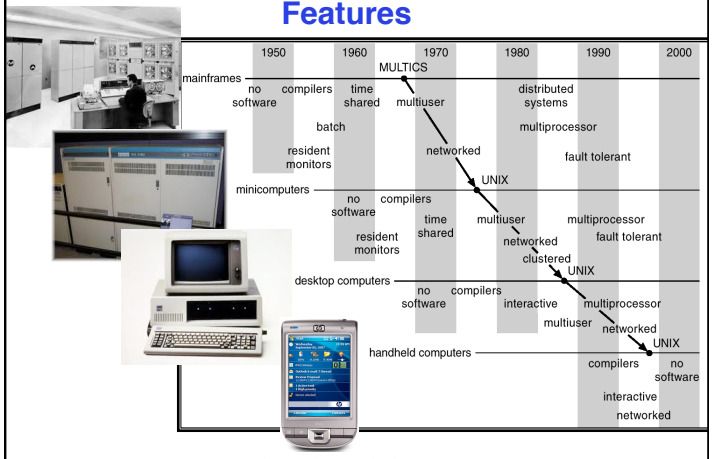
Ion Stoica

http://inst.eecs.berkeley.edu/~cs162

Very Brief History of OS

- Several Distinct Phases:
  - Hardware Expensive, Humans Cheap
    - » Eniac, … Multics
  - Hardware Cheaper, Humans Expensive
    - » PCs, Workstations, Rise of GUIs
  - Hardware Really Cheap, Humans Really Expensive
    - » Ubiquitous devices, Widespread networking
- Rapid Change in Hardware Leads to changing OS
  - Batch ⇒ Multiprogramming ⇒ Timeshare ⇒ Graphical UI ⇒ Ubiquitous Devices
  - Gradual Migration of Features into Smaller Machines
- Situation today is much like the late 60s
  - Small OS: 100K lines/Large: 10M lines (5M browser!)
  - 100-1000 people-years

Review: Migration of OS Concepts and Features



Implementation Issues (How is the OS implemented?)

- Policy vs. Mechanism
  - Policy: What do you want to do?
  - Mechanism: How are you going to do it?
  - Should be separated, since policies change
- Algorithms used
  - Linear, Tree-based, Log Structured, etc…
- Event models used
  - Threads vs. event loops
- Backward compatibility issues
  - Very important for Windows 2000/XP/Vista/…
  - POSIX tries to help here
- System generation/configuration
  - How to make generic OS fit on specific hardware

Goals for Today

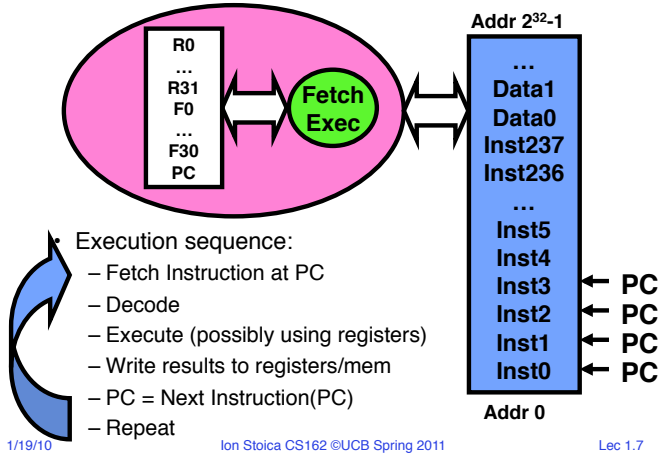
- How do we provide multiprogramming?
- What are **processes**?
- How are they related to **threads** and **address spaces**?

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from lecture notes by Kubiatowicz.

Threads

- Unit ("thread") of execution:
  - Independent Fetch/Decode/Execute loop
  - Unit of scheduling
  - Operating in some **address space**
- Uniprogramming: *one thread at a time*
  - MS/DOS, early Macintosh, Batch processing
- Multiprogramming: *more than one thread at a time*
  - Multics, UNIX/Linux, OS/2, Windows NT/2000/XP, Mac OS X
- ManyCore ⇒ Multiprogramming, right?

Recall (61C): What happens during execution?



- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

Uniprograming vs. Multiprograming

- Uniprogramming: *one thread at a time*
  - MS/DOS, early Macintosh, Batch processing
  - Easier for operating system builder
  - Get rid of concurrency (only one thread accessing resources!)
  - Does this make sense for personal computers?
- Multiprogramming: *more than one thread at a time*
  - Multics, UNIX/Linux, OS/2, Windows NT/2000/XP, Mac OS X
  - Often called "multitasking", but multitasking has other meanings (talk about this later)

Challenges of Multiprograming

- Each applications wants to own the machine → **virtual machine abstraction**
- Applications compete with each other for resources
  - Need to arbitrate access to shared resources → **concurrency**
  - Need to protect applications from each other → **protection**
- Applications need to communicate/cooperate with each other → **concurrency**

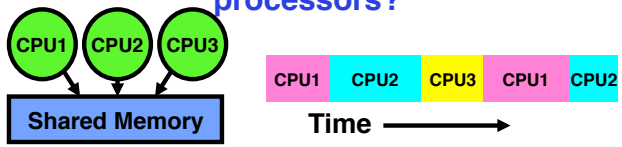
Processes

- **Process:** unit of resource allocation **and** execution
  - Owns memory (address space)
  - Owns file descriptors, file system context, …
  - Encapsulate one or more threads sharing process resources
- Why **processes**?
  - Navigate fundamental tradeoff between protection and efficiency
  - Processes provides memory protection while threads don't (share a process memory)
  - Threads more efficient than processes (later)
- Application instance consists of one or more processes

The Basic Problem of Concurrency

- The basic problem of concurrency involves resources:
  - Hardware: single CPU, single DRAM, single I/O devices
  - Multiprogramming API: processes think they have exclusive access to shared resources
- OS has to coordinate all activity
  - Multiple processes, I/O interrupts, …
  - How can it keep all these things straight?
- Basic Idea: Use Virtual Machine abstraction
  - Simple machine abstraction for processes
  - Multiplex these abstract machines
- Dijkstra did this for the "THE system"
  - Few thousand lines vs 1 million lines in OS 360 (1K bugs)

How can we give the illusion of multiple processors?



- Assume a single processor. How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual "CPU" needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch from one CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, other things

Properties of this simple multiprogramming technique

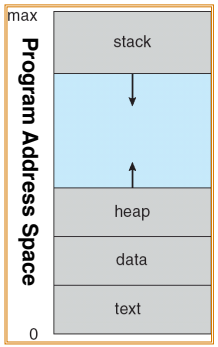
- All virtual CPUs share same non-CPU resources
  - I/O devices the same
  - Memory the same
- Consequence of sharing:
  - Each thread can access the data of every other thread (good for sharing, bad for protection)
  - Threads can share instructions (good for sharing, bad for protection)
  - Can threads overwrite OS functions?
- This (unprotected) model common in:
  - Embedded applications
  - Windows 3.1/Machintosh (switch only with yield)
  - Windows 95—ME? (switch with both yield and timer)

How to protect threads from one another?

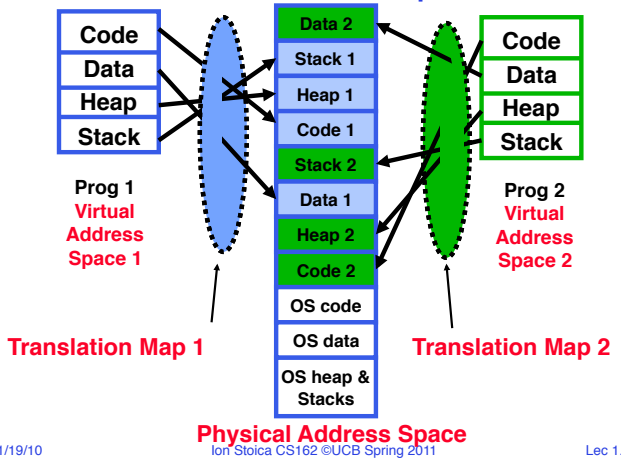
1. Protection of memory
   - Every task does not have access to all memory
2. Protection of I/O devices
   - Every task does not have access to every device
3. Protection of Access to Processor: preemptive switching from task to task
   - Use of timer
   - Must not be possible to disable timer from usercode

Recall: Program's Address Space

- Address space ⇒ the set of accessible addresses + associated states:
  - For a 32-bit processor there are 2^{32} = 4 billion addresses
- What happens when you read or write to an address?
  - Perhaps nothing
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation
    - » (Memory-mapped I/O)
  - Perhaps causes exception (fault)



Providing Illusion of Separate Address Space: Load new Translation Map on Switch



Project Signup

- Project Signup: Watch "Group/Section Signup" Link
  - 4-5 members to a group
    - » Everyone in group must be able to *actually* attend same section
    - » The sections assigned to you by Telebears are temporary!
  - Only submit once per group!
    - » Everyone in group must have logged into their cs162-xx accounts once before you register the group
    - » Make sure that you select at least 2 potential sections
    - » **Hard deadline: due Friday (1/28) by 11:59pm**
- Sections:
  - Watch for section assignments next Monday/Tuesday
  - Attend new sections next week

Section	Time	Location	TA
101	Th 10:00A-11:00A	3105 Etcheverry	Jorge Ortiz
102	Th 11:00A-12:00P	4 Evans	
104	Th 1:00P-2:00P	85 Evans	Stephen Dawson-Haggerty
105	Th 2:00P-3:00P	B56 Hildebrand	
103	Th 3:00P-4:00P	4 Evans	David Zhu
106	Th 4:00P-5:00P	320 Soda	

Others…

- We are going to use Piazzza instead of the newsgroup
  - Got to http://www.piazzza.com/class#cs162/
  - Make an account
  - Join the Berkeley version of cs162 (it's open.)
- Final exam conflict with cs184
  - Unfortunately, you'd have to pick one class or another
  - We can only accommodate very few exceptions

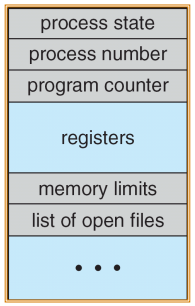
5min Break

Traditional UNIX Process

- Process: *Operating system abstraction to represent what is needed to run a single program*
  - Often called a "HeavyWeight Process"
  - Formally: a single, sequential stream of execution in its *own* address space
- Two parts:
  - Sequential Program Execution Stream
    - » Code executed as a *single, sequential* stream of execution (i.e., thread)
    - » Includes State of CPU registers
  - Protected Resources:
    - » Main Memory State (contents of Address Space)
    - » I/O state (i.e. file descriptors)
- Important: There is no concurrency in a heavyweight process

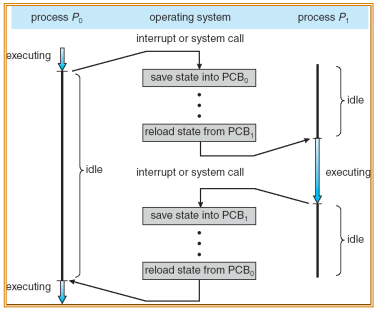
How do we multiplex processes?

- The current state of process held in a process control block (PCB):
  - This is a "snapshot" of the execution and protection environment
  - Only one PCB active at a time
- Give out CPU time to different processes (Scheduling):
  - Only one process "running" at a time
  - Give more time to important processes
- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
  - Sample mechanisms:
    - » Memory Mapping: Give each process their own address space
    - » Kernel/User duality: Arbitrary multiplexing of I/O through system calls



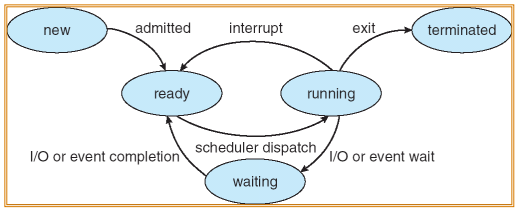
Process Control Block

CPU Switch From Process to Process



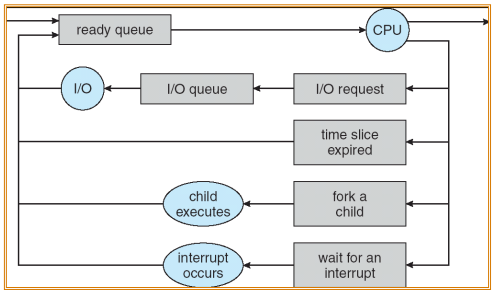
- This is also called a "context switch"
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but… contention for resources instead

Diagram of Process State



- As a process executes, it changes *state*
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution

Process Scheduling

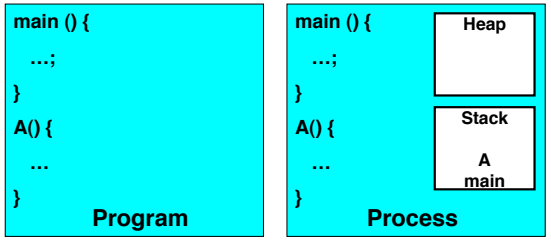


- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible (few weeks from now)

What does it take to create a process?

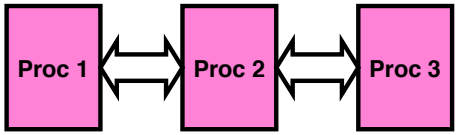
- Must construct new PCB
  - Inexpensive
- Must set up new page tables for address space
  - More expensive
- Copy data from parent process? (Unix `fork()`)
  - Semantics of Unix `fork()` are that the child process gets a complete copy of the parent memory and I/O state
  - Originally *very* expensive
  - Much less expensive with "copy on write"
- Copy I/O state (file handles, etc)
  - Medium expense

Process =? Program



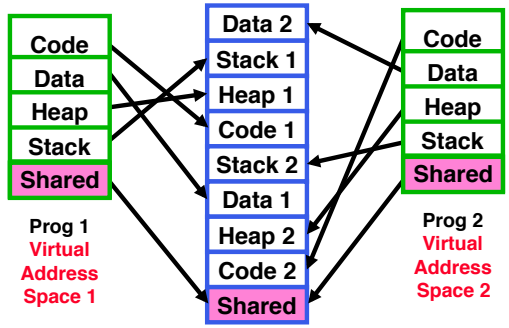
- More to a process than just a program:
  - Program is just part of the process state
  - I run emacs on lectures.txt, you run it on homework.java – Same program, different processes
- Less to a process than a program:
  - A program can invoke more than one process
  - cc starts up cpp, cc1, cc2, as, and ld

Multiple Processes Collaborate on a Task



- High Creation/memory Overhead
- (Relatively) High Context-Switch Overhead
- Need Communication mechanism:
  - Separate Address Spaces Isolates Processes
  - Shared-Memory Mapping
    - » Accomplished by mapping addresses to common DRAM
    - » Read and Write through memory
  - Message Passing
    - » `send()` and `receive()` messages
    - » Works across network

Shared Memory Communication



- Communication occurs by "simply" reading/writing to shared address page
  - Really low overhead communication
  - Introduces complex synchronization problems

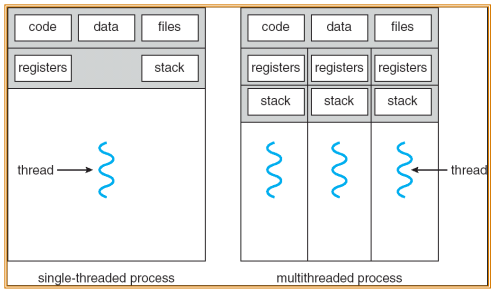
Inter-process Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - `send(message)` – message size fixed or variable
  - `receive(message)`
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus, systcall/ trap)
  - logical (e.g., logical properties)

Modern "Lightweight" Process with Threads

- Thread: *a sequential execution stream within process* (Sometimes called a "Lightweight process")
  - Process still contains a single Address Space
  - No protection between threads
- Multithreading: *a single program made up of a number of different concurrent activities*
  - Sometimes called multitasking, as in Ada…
- Why separate the concept of a thread from that of a process?
  - Discuss the "thread" part of a process (concurrency)
  - Separate from the "address space" (protection)
  - Heavyweight Process ≡ Process with one thread

Single and Multithreaded Processes



- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

Examples of multithreaded programs

- Embedded systems
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations
- Most modern OS kernels
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- Database Servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done

Examples of multithreaded programs (con't)

- Network Servers
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems
- Parallel Programming (More than one physical CPU)
  - Split program into multiple threads for parallelism
  - This is called Multiprocessing
- Some multiprocessors are actually uniprogrammed:
  - Multiple threads in one address space but one program at a time

Thread State

- State shared by all threads in process/addr space
  - Contents of memory (global variables, heap)
  - I/O state (file system, network connections, etc)
- State "private" to each thread
  - Kept in TCB ≡ Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack – what is this?
- Execution Stack
  - Parameters, Temporary variables
  - return PCs are kept while called procedures are executing

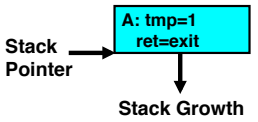
Execution Stack Example

```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```

- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

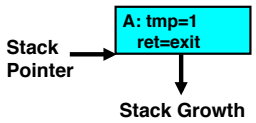
```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

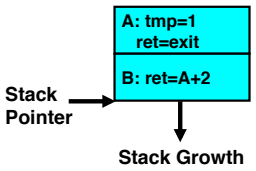
```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

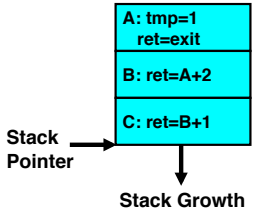
```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

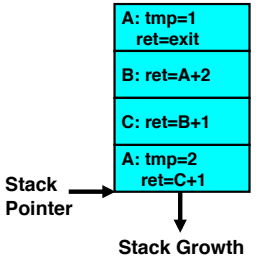
```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Execution Stack Example

```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Classification

# threads Per AS: / # of addr spaces:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, JavaOS,etc) JavaOS, Pilot(PC)	Mach, OS/2, Linux Windows 9x??? Win NT to XP, Solaris, HP-UX, OS X

- Real operating systems have either
  - One or many address spaces
  - One or many threads per address space
- Did Windows 95/98/ME have real memory protection?
  - No: Users could overwrite process tables/System DLLs

Summary

- Processes have two parts
  - Threads (Concurrency)
  - Address Spaces (Protection)
- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (`yield()`, I/O operations) or involuntary (timer, other interrupts)
- Protection accomplished restricting access:
  - Memory mapping isolates processes from each other
  - Dual-mode for isolating I/O, other resources
- Book talks about processes
  - When this concerns concurrency, really talking about thread portion of a process
  - When this concerns protection, talking about address space portion of a process