G22.2250-001
Operating Systems

Computer and Operating System Structures
Processes, Threads, and Process Cooperation

September 11, 2007

Outline

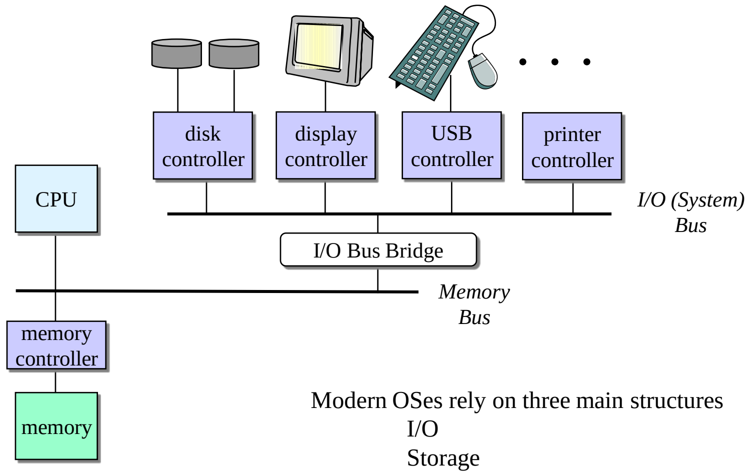
Announcements

- TA: Jay Chen (jchen@cs), 7th Floor 715 B'way
  - Office hours: Mondays 5:00 – 6:00pm, Thursdays 4:00 – 5:00pm
  - Primary point of contact for questions about Nachos labs

- Computer system structures
  - (Review) I/O, Storage, Protection
- Operating system structures
- Processes
  - process scheduling
- Threads
  - Multithreading models
- Process cooperation
  - Shared memory vs. message passing

[Silberschatz/Galvin/Gagne: Chapters 1, 2, 3.1-3.3, 4.1-4.5, 6.1]

(Review)
The Hardware of a Modern Computer System



Modern OSes rely on three main structures
- I/O
- Storage
- Protection

(Review)
Computer-System Structures (1): Input/Output

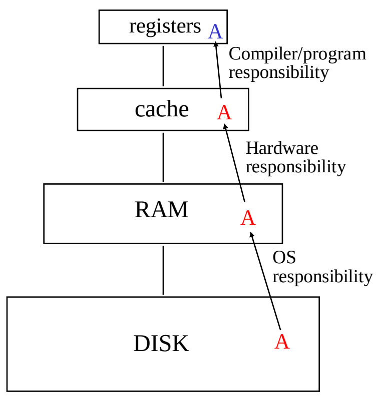
- CPU interaction with I/O devices via device controllers
  - Special-purpose processors with local storage and registers
  - CPU requests service by writing to control registers
  - Device controllers interrupt the CPU upon completion of action
    - CPU support for interrupts
      - Saving state of currently running process (registers)
      - Vectoring to interrupt service routine (ISR)
      - Return from ISR restores process state, resuming as if the interrupt never happened
    - Traps are like interrupts, except triggered by special CPU instructions
- I/O operations can be synchronous or asynchronous
  - Direct memory access (DMA) mechanisms help reduce CPU involvement
- Memory-mapped I/O permits controller storage to be accessed using CPU ld/st instructions
  - Essential for high-speed I/O devices (e.g., video)

Computer-System Structures (2): Storage

- Primary storage: *Main memory* (volatile)
  - accessed directly using load/store instructions
    - 1 cycle (registers), 2-5 cycles (cache), 20-50 cycles (RAM)
    - *before*: only one outstanding memory operation, CPU waits for completion
    - *now*: several outstanding operations
- Secondary storage: *Disks* (non-volatile)
  - accessed using a disk controller
  - supports random access but with non-uniform cost
- Tertiary storage: *Tapes, Optical disks* (non-volatile)
  - typically used only for backup
  - very inefficient support for random access
- Organized as a hierarchy
  - small amount of faster, more expensive storage closer to the CPU
  - larger amounts of slower, less expensive storage further away

Storage Hierarchy

- Rationale
  - keep CPU busy: lots of fast memory
  - keep system cost down
- How does it work
  - *caching:* upon access, move datum or instruction and its neighbors into higher levels of the hierarchy
  - *replacement* when a level fills up
  - copies need to be kept coherent
- Why does it work
  - Real programs demonstrate *locality*
    - e.g.: rows and columns of a matrix
    - e.g.: sequential instructions
  - once a *datum* or *instruction* is used, things "near" them are likely to be used "soon"



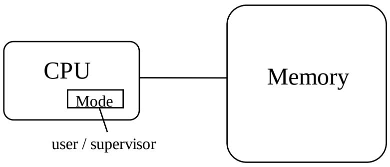
Computer-System Structures (3): Protection

- Goal: Prevent user processes from accidentally/maliciously damaging
  - the OS structures
  - parts of other process's memory space
  - other user's I/O devices

- Mechanisms address different ways in which protection breaks down
  1. dual-mode operation
     - Prevent user process taking over part of the OS and using this to overwrite other processes or even modify the OS itself (as in MS-DOS)
  2. privileged instructions
     - Prevent user process intervening in I/O of another process via control of the I/O handlers and indirectly causing damage
  3. memory protection
     - Prevent user process directly accessing another user process' storage
  4. CPU protection via timers
     - Prevent hanging the OS -- e.g., via an infinite loop

Protection Mechanisms (1): Dual-mode Operation and Privileged Instructions

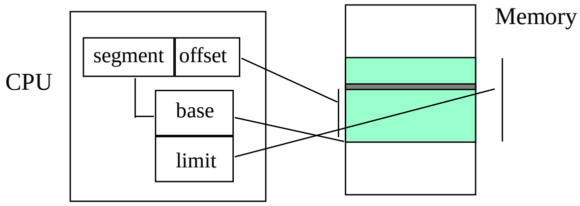
- Dual-mode operation
  - *supervisor* and *user* modes
  - system starts off in supervisor mode and reenters it for interrupt processing
  - operating system gains control in supervisor mode



- Privileged instructions
  - restrict use of certain instructions to supervisor mode
    - I/O, including interrupt control
      - exception is instructions which generate interrupts
      - may be done by memory mapping
    - affect memory mapping
    - affect CPU mode (user/supervisor)
  - hardware support crucial for performance and for atomicity

Protection Mechanisms (2): Memory Protection

- Basic method: Memory is divided into segments



- Furthermore
  - logical addresses are mapped to physical addresses
    - provides sharing, etc.
  - hardware support for address mapping
  - a memory protection violation is detected
    - user process *traps* to (interrupts) the OS

Protection Mechanisms (3): Timers

- OS code can enforce policies only if it gets a chance to run

- Timers maintain a count of elapsed (system) clock ticks
  - when timer expires, the CPU is interrupted → run the OS code

- Used for
  - interrupting hung processes
  - context switching in time-shared systems

- Access to timers is (usually) privileged
  - WHY?

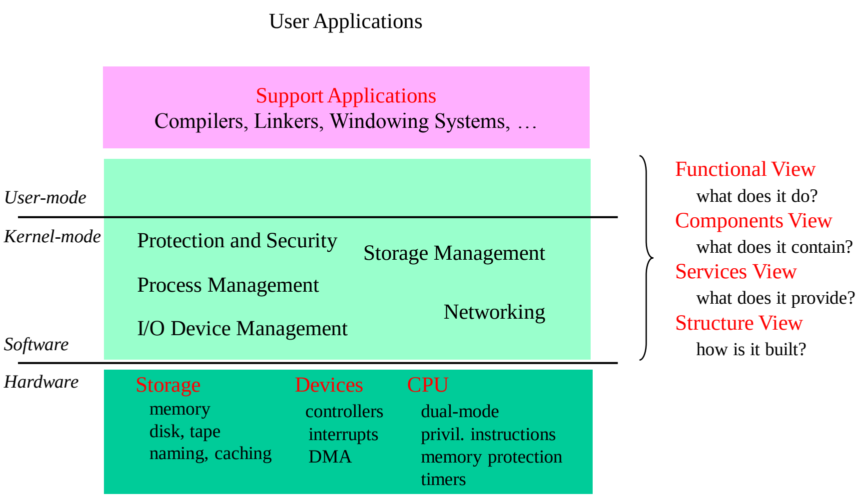
Outline

Announcements

- TA: Jay Chen (jchen@cs), 7th Floor 715 B'way
  - Office hours: Mondays 5:00 – 6:00pm, Thursdays 4:00 – 5:00pm
  - Primary point of contact for questions about Nachos labs
- Computer system structures
  - (Review) I/O, Storage, Protection
- Operating system structures
- Processes
  - process scheduling
- Threads
  - Multithreading models
- Process cooperation
  - Shared memory vs. message passing

[Silberschatz/Galvin/Gagne: Chapters 1, 2, 3.1-3.3, 4.1-4.5, 6.1]

Hardware and OS Structures



OS Views (1): Functional View

- What are the functions performed by an OS?

- Explicit operations
  - program execution and handling
  - I/O operations
  - file-system management
  - inter-process communication
  - exception detection and handling
    - e.g., notifying user that printer is out of paper

- Implicit operations
  - resource allocation
  - accounting
  - protection
    - e.g., maintaining data integrity, logging invalid login attempts

OS Views (2): Components View

- Processes: run-time representations of user programs
  - create, terminate, suspend, resume
  - access to shared resources (e.g., printers)

Lectures 2-9

- Storage
  - allocation of memory among resident processes
  - disk management (e.g., scheduling of disk accesses)
- I/O
  - device drivers, handling of device interrupts
  - files and directories

Lectures 10-13

- Protection
  - user access to system resources

Lectures 14-15

► Course organization follows this view

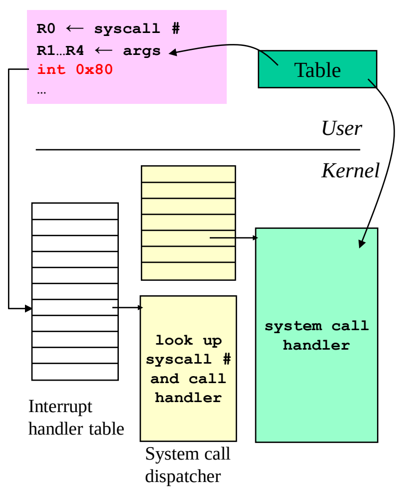
OS Views (3): Services View

- Two issues
  - What services does an OS provide? (same as functional view)
  - How do users and user programs access these services?
- Interface between the user and the OS: Command Interpreter
  - typical commands
    - process creation and (implicitly) destruction
    - I/O handling and file system manipulation
    - communication: interact with remote devices
    - protection management: changing file/directory access control, etc.
  - different varieties
    - the interpreter contains the code for the requested command (e.g., **delete**)
    - the interpreter calls a system routine to handle the request
    - the interpreter spawns new process(es) to handle the request
      - process lookup through some general procedure

  ▸ you will implement a simple shell in Nachos Lab 5

OS Views (3): Services View (contd.)

- Interface between a user program and the OS: System Calls
  - arguments passed in registers, a memory block, or on the stack
  - entry into the kernel using the *trap* mechanism
- Standard system calls
  - process control
  - file manipulation
  - device manipulation
  - information maintenance
    - *get/set* system data (time, memory/cpu usage), process and device attributes
  - communications



OS Views (4): Structure View

- How to structure OS functionality
  - Layering
  - Microkernels
  - Virtual machines
- Designing and implementing an OS

- Read Sections 2.6-2.9, Silberschatz, Galvin, and Gagne
- Look at Nachos source code
  - Thomas Narten's roadmap

Outline

Announcements

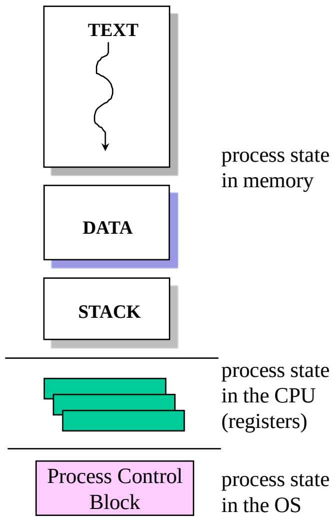
- TA: Jay Chen (jchen@cs), 7th Floor 715 B'way
  - Office hours: Mondays 5:00 – 6:00pm, Thursdays 4:00 – 5:00pm
  - Primary point of contact for questions about Nachos labs

- Computer system structures
  - (Review) I/O, Storage, Protection
- Operating system structures
- Processes
  - process scheduling
- Threads
  - Multithreading models
- Process cooperation
  - Shared memory vs. message passing

[Silberschatz/Galvin/Gagne: Chapters 1, 2, 3.1-3.3, 4.1-4.5, 6.1]

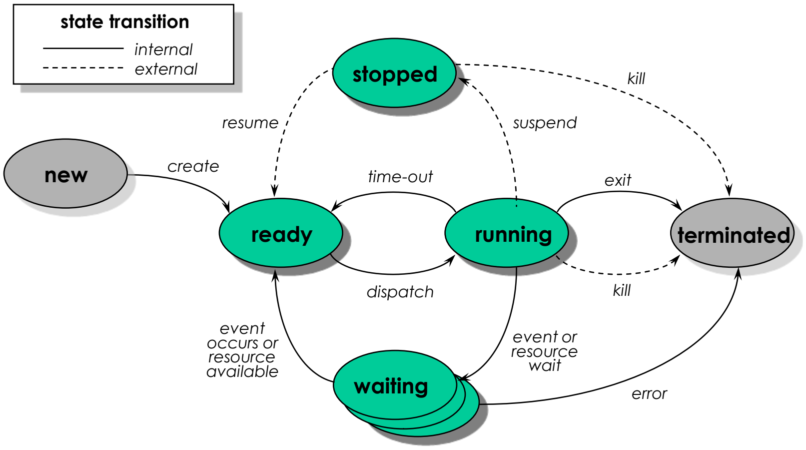
What is a Process?

- A process is a program in execution.
- The components of a process are:
  - the program to be executed
  - the data on which the program will execute
  - the resources required by the program—such as memory and file(s)
  - the status of the execution
- A process is the unit of
  - resource ownership
  - protection
  - dispatching



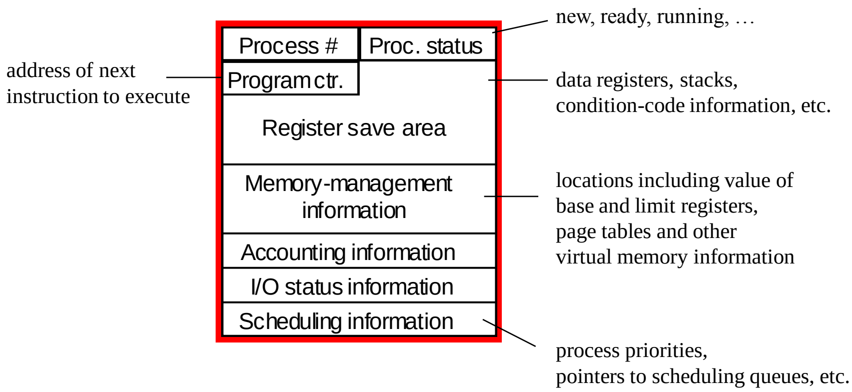
The State of a Process

- Can be one of: New, Ready, Running, Waiting, Stopped, Terminated



Process Control Information

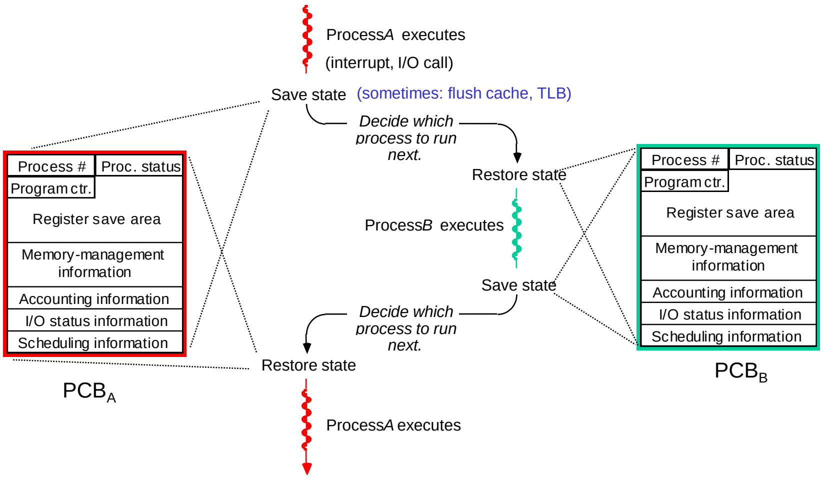
Process Control Block (PCB)



Scheduling Processes

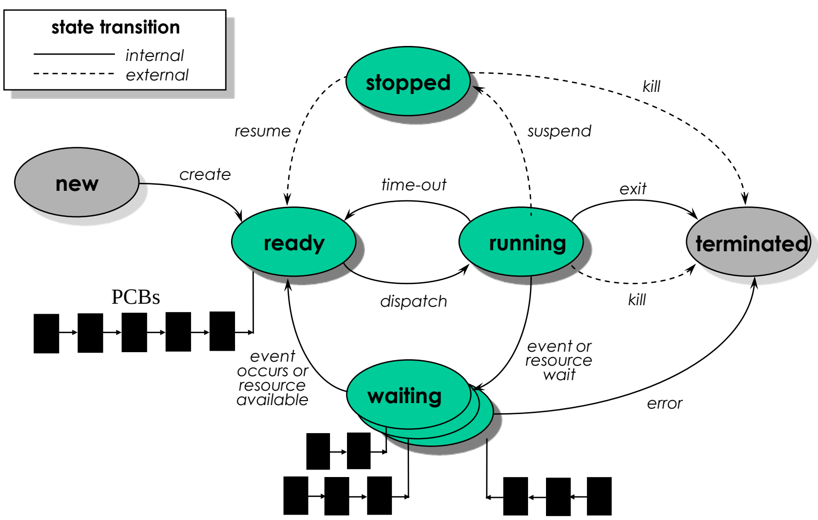
- "Decide which process to run next"
- Some reasons for doing this
  - move a *running* process to a *waiting* state in a multiprogrammed OS
    - multiplex CPU among ready processes
  - swapping in a time-shared system when a process' *time-slice* is over
    - typically controlled by a *timer* (process)
  - start and stop processes for accessing secondary memory and I/O
    - this may cause *spawning* of appropriate new processes
- Three main concerns
  - what happens to the process currently using the CPU?
  - how do you keep track of what each process should be doing?
  - how do you decide which process does what?

Concern 1: Process Context Switch



Look at the Nachos code: Thread::Yield, SWITCH
Nachos Lab 1: Consequences of asynchronous context switches

Concern 2: Process Queues



Concern 3: Schedulers

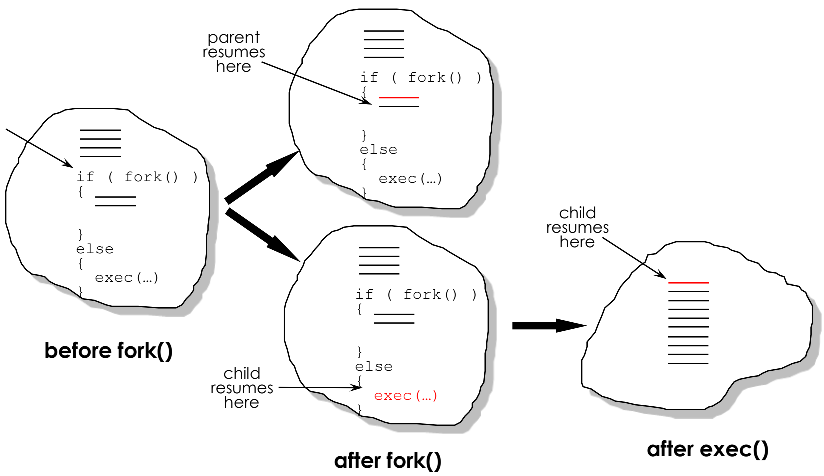
- The long-term scheduler
  – *operation*: creates processes and adds them to the ready queue
  – *frequency*: infrequent, ~minutes
  – *objective*: maintain good throughput by ensuring mix of I/O and CPU jobs
- The short-term scheduler
  – *operation*: allocates CPU and other resources to ready jobs
  – *frequency*: frequent, ~100 ms (a context switch takes ~10s of μsecs)
  – *objective*: ensure good response times in time-sharing systems
- The medium-term scheduler
  – *operation*: swaps some processes out of the short-term scheduler's loop
  – *frequency*: somewhere between the short- and long-term schedulers
  – *objective*: to prevent over-multiprogramming (thrashing)
    - required when the long-term scheduler underestimates process requirements

System Calls for Process Management

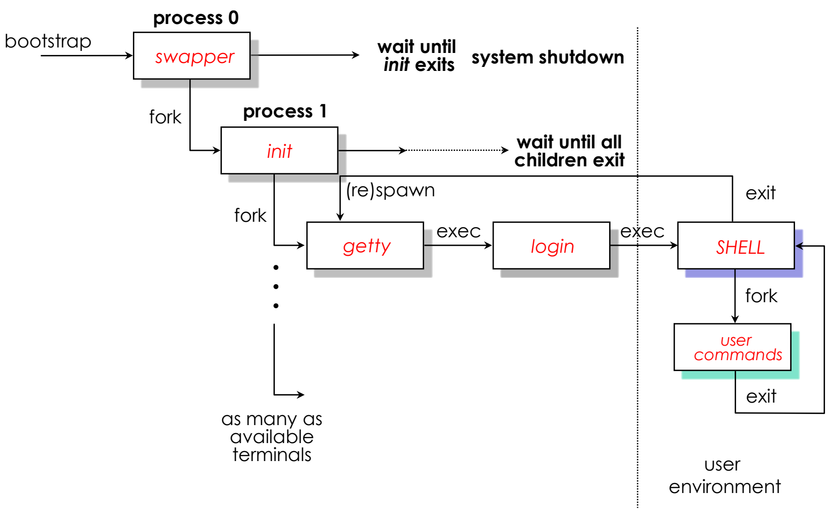
- Creation
  – a "parent" process spawns a "child" process; a *fork* in UNIX
    - child may or may not inherit parent's memory
    - child is added to the ready queue
  – the parent-child association is maintained via process IDs (PIDs)
- Termination
  – normal: a process asks the OS to delete it; an *exit* in UNIX
    - all resources of a terminated process are deallocated and reclaimed
    - on termination, the child's PID and output may be passed back to the parent
  – abnormal: another process (typically the parent) can cause termination
    - if the child exceeds its usage, becomes obsolete, or the parent is exiting the system due to some other problem
    - a process (almost always) terminates when its parent does
- Communication: Lecture 5
- Coordination: Lectures 6, 9-11

Example: Process Creation in UNIX

Two system calls: fork, exec



UNIX System Initialization



Outline

Announcements

- TA: Jay Chen (jchen@cs), 7th Floor 715 B'way
  - Office hours: Mondays 5:00 – 6:00pm, Thursdays 4:00 – 5:00pm
  - Primary point of contact for questions about Nachos labs

- Computer system structures
  - (Review) I/O, Storage, Protection
- Operating system structures
- Processes
  - process scheduling
- Threads
  - Multithreading models
- Process cooperation
  - Shared memory vs. message passing

[Silberschatz/Galvin/Gagne: Chapters 1, 2, 3.1-3.3, 4.1-4.5, 6.1]

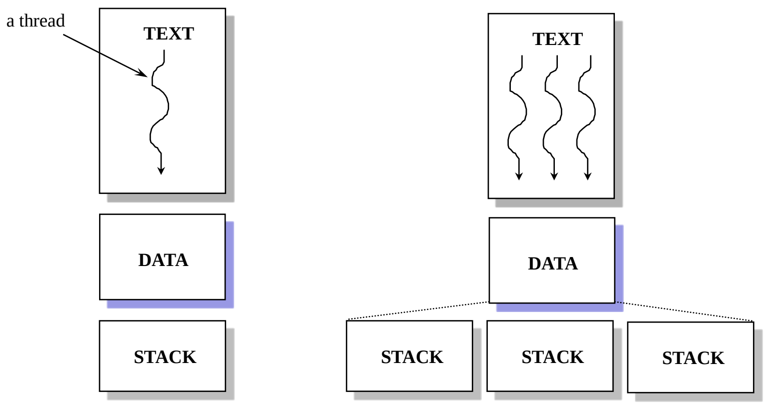
Threads

- A thread is similar to a process
  - sometimes called a *lightweight process*
  - several threads (of control) can execute within the same address space

- Like a process, a thread
  - is a basic unit of CPU utilization
  - represents the state of a program
  - can be in one of several states: *ready*, *blocked*, *running*, or *terminated*
  - has its own program counter, registers, and stack
  - executes sequentially, can create other threads, block for a system call

- Unlike a process, a thread
  - shares with peer threads, its code section, data section, and operating-system resources such as open files and signals
  - is *simpler and faster*

Threads versus Processes (contd.)



a traditional process

a multi-threaded process

Threads: Why Simpler?

Threads share the process address space

- Benefits for the user:
  - communication is easier
  - communication is more efficient
  - security may not be necessary
    assumed to operate within the same protection domain
  - one blocking thread need not block other threads in the process
- Benefits for the OS
  - context switching is more efficient
    - memory mappings can remain unchanged
    - cache need not be flushed
  - can run a process across multiple nodes of a multiprocessor
    - performance advantages if threads can execute in parallel (e.g., web servers)

Types of Threads

- User-level threads (e.g., pthreads: Section 4.3.1, Java threads: Section 4.3.3)
  - OS does not know about them
  - implemented/scheduled by library routines
  - ⬆ operations are faster (context switch, communication, control)
  - ⬇ blocking operations block the entire process (even with ready threads)
  - ⬇ operations based on local criteria may be less effective (e.g., scheduling)

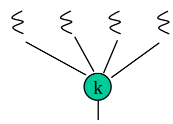
- Kernel-level threads (e.g., Solaris 2, WinXP: Section 4.5.1, Linux: 4.5.2)
  - known to the OS
  - scheduled by the OS
  - ⬆ process need not block if one of its threads blocks on a system call
  - ⬇ thread operations are expensive
    - switching threads involves kernel interaction (via an interrupt)
  - ⬆ the kernel can do a better job of allocating resources

Multithreading Models

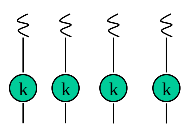
- Most systems provide support for both user and kernel threads

Three dominant models for mapping threads to kernel resources

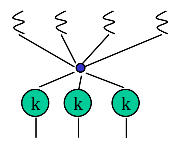
- Many-to-one
  - Thread management done in user space
  - Entire process blocks if a thread does a blocking operation
  - E.g., systems without kernel threads



- One-to-one
  - Each user-thread mapped to a kernel thread
  - Allows more concurrency
  - E.g., Windows 2000, XP (fibers: many-to-one)



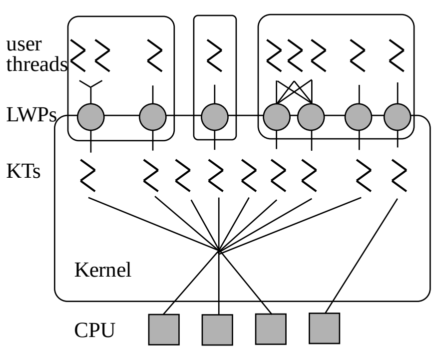
- Many-to-many
  - Combination of the above two
  - E.g., Solaris 2



POSIX Threads (pthreads)

- A portable API for multithreaded programs
  - Some pthreads implementations do map threads to kernel threads
  - Most rely on user-level threading support
    - Assembly instructions to save/restore registers
- Calls for creating, exiting, joining pthreads
  - pthread_create: start execution of this thread
    - Takes function pointer as an argument
  - pthread_exit:terminate execution of this thread
  - pthread_join: wait for a particular thread to exit
- Other calls
  - Help set thread attributes (stack size, scheduling behavior, etc.)
  - Specify signal handling
    - Signals are a way of allowing processes to respond to events
      - Interrupts (Ctrl-C), others
    - Multithreaded systems need to define a way for signals to be communicated to individual threads (see Section 4.4.3)
      - All threads , a specific thread, only those threads that do not block the signal, …

Processes and Threads in Solaris 2



- OS schedules execution of kernel threads (KTs)
  - runs them on the CPUs
  - a KT can be pinned to a CPU
- A task consists of one or more lightweight processes (LWPs)
  - LWPs in a task may
    - contain several *user-level threads*
    - issue a system call
    - block
- A LWP is associated with a KT
- There are KTs with no LWP

- Linux has a simpler model (see Section 4.5.2): only kernel threads (tasks)
  - System calls for process creation (fork) and thread creation (clone)

Outline

Announcements

- TA: Jay Chen (jchen@cs), 7th Floor 715 B'way
  - Office hours: Mondays 5:00 – 6:00pm, Thursdays 4:00 – 5:00pm
  - Primary point of contact for questions about Nachos labs
- Computer system structures
  - (Review) I/O, Storage, Protection
- Operating system structures
- Processes
  - process scheduling
- Threads
  - Multithreading models
- Process cooperation
  - Shared memory vs. message passing

[Silberschatz/Galvin/Gagne: Chapters 1, 2, 3.1-3.3, 4.1-4.5, 6.1]

Process Cooperation

- Why do processes cooperate?
  - *modularity*: breaking up a system into several sub-systems
    - e.g.: an interrupt handler and device driver that need to communicate
  - *convenience*: users might want to have several processes share data
  - *speedup*: a single program is run as several sub-programs
- How do processes cooperate?
  - communication abstraction: *producers* and *consumers*
    - *producers* produce a piece of information
    - *consumers* use this information
  - abstraction helps deal with general "phenomena" and simplifies correctness arguments
- Two general classes of process cooperation techniques
  - shared memory
  - message passing

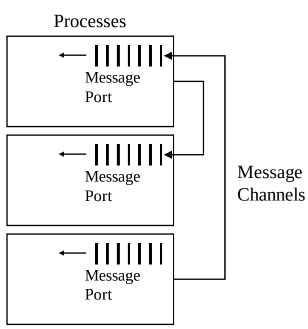
Shared Memory (Procedure-oriented System)

- Processes can directly access data written by other processes
  - examples: POSIX threads, Java, Mesa, small multiprocessors
- A finite-capacity shared buffer

```
N: integer                          -- buffer size
nextin = nextout = 1 initially;     -- start of buffer
buffer: array of size N
Producer:
   Repeat
     -- produce an item in tempin
     while (nextin+1) mod n = nextout do wait-a-bit;
     buffer[nextin] := tempin;
     nextin := (nextin+1) mod n;
Consumer:
   Repeat
     while nextin = nextout do wait-a-bit;
     tempout := buffer[nextout];
     nextout := (nextout+1) mod n;
     -- consume the item in tempout
```


Message Passing (Message-oriented System)

- Execution is in separate address spaces
  - communication using message channels
  - examples: UNIX processes, large multiprocessors, etc.
- Components
  - messages and message identifiers
  - message channels and ports
    - channels (pipes) must be bound to ports
    - queues associated with ports
  - message transmission operations
    - SendMessage[channel, body] returns id
    - AwaitReply[id]
    - RecvMessage[port] returns id
    - SendReply[id, body]



- Many variants: See Section 4.5
- ➡ Focus on shared memory for next few lectures

Bounded Buffers Using Counters

```
N: integer                              -- buffer size
counter: integer = 0 initially;
nextin = nextout = 1 initially;   -- start of buffer
buffer: array of size N
Producer:
  Repeat
    -- produce an item in tempin
    while counter = N do wait-a-bit;
    buffer[nextin] := tempin;
    nextin := (nextin+1) mod n;
    counter := counter+1;
Consumer:
  Repeat
    while counter = 0 do wait-a-bit;
    tempout := buffer[nextout];
    nextout := (nextout+1) mod n;
    counter := counter-1;
    -- consume the item in tempout
```

Producer and Consumer processes are asynchronous! execution of these two statements can be interleaved (e.g., because of interrupts)

Interleaving of Increment/Decrement

- Each of increment and decrement are actually implemented as a series of machine instructions on the underlying processor

Producer	Consumer
register1 := counter	register2 := counter
register1 := register1 + 1	register2 := register2 - 1
counter := register1	counter := register2

- An interleaving
  - counter = 5; a producer followed by a consumer

Producer	Consumer	
register1 := counter		{register1 = 5}
register1 := register1 + 1		{register1 = 6}
	register2 := counter	{register2 = 5}
	register2 := register2 - 1	{register2 = 4}
counter := register1		{counter = 6}
	counter := register2	{counter = 4}

The Problem

- Increment and decrement are not *atomic* or *uninterruptable*
  - two or more operations are executed atomically if the result of their execution is equivalent to that of some serial order of execution
  - operations which are always executed atomically are called atomic
    - byte read; byte write;
    - word read; word write
- The code containing these operations creates a race condition
  - produces inconsistencies in shared data
- Reasons for non-atomic execution
  - interrupts
  - context-switches

The Solution

- The producer and consumer processes need to synchronize
  - so that they do *not* access shared variables at the same time
  - this is called mutual exclusion
    - the *shared* and *critical* variables can be accessed by only one process at a time
  - access must be serialized even if the processes attempt concurrent access
    - in the previous example: counter increment and decrement operations
- General framework for achieving this: Critical Sections
  - work independent of the particular context or need for synchronization

Critical Sections

- Critical sections: General framework for process synchronization

```
ENTRY-SECTION
CRITICAL-SECTION-CODE
EXIT-SECTION
```

  - the **`ENTRY-SECTION`** controls access to make sure that no more than one process P_i gets to access the critical section at any given time
    - acts as a *guard*
  - the **`EXIT-SECTION`** does bookkeeping to make sure that other processes that are waiting know that P_i has exited

- How can we implement critical sections?
  - turn off interrupts around critical operations
  - ✓ build on top of atomic memory load/store operations
  - ✓ provide higher-level primitives