CIS-505
Software Systems

Instructor: Matt Blaze

Important Administrative Stuff

- This is CIS-505- Software Systems
  - We meet Tuesday & Thursday, 1330-1500
- There’s a waiting list to enroll
  - contact Mike Felker with:
    - name, status (CIS PhD, CIS MS, other)
  - We’ll let you know before next class
- This is a PhD WPE-I course
  - PhD students should be in the appropriate section

Meet your staff…

- Your Instructor: Matt Blaze
  - Office: Levine 611
  - Office Hours: Tuesday 4:30 – 5:30 (pm!) & by appointment (email me!)
  - blaze@cis.upenn.edu

- That’s me – I’m the person talking right now

Textbooks, course web page

- A.S. Tanenbaum & Maarten van Steen. *Distributed Systems (2nd Edition)*. Prentice-Hall, 2007.
  - it's in the book store
- Optional: W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment (2/e)*. Addison-Wesley Professional. 2005
- Course web page is:
  `www.crypto.com/courses/fall12/cis505/`
  - Check frequently for updates and news
  - Write this URL down. Do it RIGHT NOW.

Prerequisites

- Basic understanding of computer operating systems & architecture (CIS240, CIS380)
- Working knowledge of C language and Unix shell and programming environment
- Shell login on “speclab” Linux cluster
  - Unix shell account
  - Check to make sure you have a working account. (Do this TODAY – really)
- Read the above point and actually do it

Grading

- Midterm (15%), Final (35%)
- Two *Substantial* programming projects (15% and 25%)
  - Done in *two-person* teams
  - putting things of until the last minute doesn't work
- Homeworks (10%)
- fascist, inflexible late homework policy strictly enforced!

A quick word about "PowerPoint"

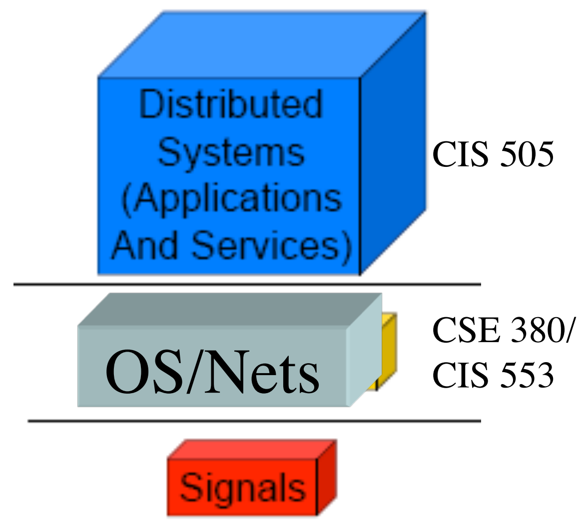
- Some courses are based on *slideware*
  - well defined set of PowerPoint slides in which the content of the course is contained
  - slides more important than textbook, lecture, readings, or homework
- This is not one of those courses
  - Slides useful here as outline of topics, at most
- Slides will be made available *after* each class
  - do not use as substitute

Policy for collaboration

- Individual written homeworks:
  - YES: discuss with friends, acknowledge them, write your own solutions
  - NO: copy past-year solutions word-for-word (or with minor modifications)
  - NO: copy answers scribbled by someone on used textbook
  - NO: use search engines to look for past-year solutions
- Group projects
  - YES: discuss concepts and programming approaches on newsgroup
  - NO: modifying/submitting solutions from other groups current/past year
- All students must contribute fair-share to team projects:
  - Individually fill in a form listing contributions of each group member.
  - Logs in version control to resolve disputes

This is a Distributed Systems Course!

- Fundamentals: how distributed systems are designed
  - Naming and addressing
  - Clock synchronization, logical clocks, events
  - Elections, distributed commits/recovery
  - Failures, Byzantine fault tolerance
- Real-world insights: how are real systems built?
  - How does theory translate into practice?
  - Network File System, Coda FS, Chord FS
  - “Backends” of search technologies
- Skills: distributed system implementation
  - Hands-on programming experience
  - Code repository (Subversion or CVS)
  - Low-level C programming to Java-based distributed programs



First half of course: Fundamentals

- Concurrency: Processes, semaphores, monitors (review of basic OS concepts)
- Inter-process protocols, RPC, RMI, sockets, msg-queues
- Naming, addresses, locations
- Clock synchronization, logical clocks, clock vector
- Coordination and agreements: elections, mutual exclusion, quorums/voting, distributed commit, total ordering
- Consensus in the presence of faults / failures
- Reliable group communication
- Distributed snapshots / checkpoint
- Replication and consistency models

Second half of course: Putting Theory into Practice

- Topics may depend on time and interests.
- Examples:
  - Distributed mail server (not covered in class, learn via project 2)
  - Distributed file systems (Network File System, Andrew File System, Coda File System, Chord File System, Ivy File System)
  - Content distribution networks (Akamai)
  - Large-scale clusters:
    - Google: MapReduce, Google File System, BigTable, Chubby distributed lock system
    - Amazon: Dynamo storage system
    - Microsoft: Dryad, DryadLINQ
    - (If time permits) Cloud computing: Amazon EC2 cloud, Hadoop middleware, consistency models
- Guest lectures on advanced topics

So… let's get started

Operating Systems:

Isolating Programs

Mostly, this course will be about how systems *communicate*

- But first, we need to answer some questions:
  - how does software run on computers?
  - how come you usually don't have to worry about other running programs?
- Operating systems *isolate* running programs
  - allow transparent timesharing
  - provide narrow communication channel
  - how does this work?

Processes / Time Sharing

- OS allows program to act as if it "owns" the machine
  - not worry about other running programs
  - not use more than its fair share of resources
  - get services from the OS
  - communicate with others
- Exploits two hardware / CPU mechanisms:
  - relocatable memory
  - "kernel mode" and "user mode" CPU state
    - "traps"

Relocatable Memory

- CPU feature that allows programs to not know in advance where they will be in memory
  - and also avoid stepping on each other’s memory by mistake
- Many ways to this is implemented:
  - offsets, prefixes, virtual memory
  - details don’t matter yet
- Every running program can have its own private *address space*

“Supervisor mode” / “User mode” and “traps”

- When the CPU is in “supervisor” mode, it can do anything, and can address any part of memory.
  - the OS kernel runs in supervisor mode
  - supervisor mode can switch to user mode at will
- When the CPU is in “user” mode, it has access only to its own address space, and can’t talk directly to devices
  - User process run in user mode
  - But after a trap we can switch to supervisor mode…

“Traps”

- A user mode program can switch to supervisor mode by issuing a “trap”
  - but there’s a catch… when it issues a trap it starts running OS code in the supervisor address space
  - user program can’t overwrite this code
- Traps are used to implement system calls to provide services to user processes that require:
  - communication with a hardware device
  - communication with another process or with the OS
- Traps also happen when devices issue *interrupts*
  - real time clock ticks, message arrives on network, etc.

How an OS runs processes… (an oversimplification)

- OS must keep track of which process are assigned which sections of memory plus other stuff
- To run a new process, assign it some memory and put its code there
  - switch to user mode and start running at the first address of the program
- The OS keeps a record of every process
  - assigned memory is, current program counter, etc.
  - this is called the process' *context*
  - enough information to restart process where it left off

Eventually, a trap happens

- Either because the running program issued it or because a device did (e.g., the clock ticked)
- First the OS records the state of the running process' context in its record
- Next it will do whatever servicing the trap requires
  - e.g., send something to the printer
- finally, it will pick a user process to restart
  - maybe the one that was running, maybe not
  - restarts based on the new process' context record
  - back in user mode now

Processes and System Calls

- First we’ll look at processes in (Unix-like Oss) from the *user’s* perspective
  - what is a process?
  - what’s the interface to the OS for managing them?
- Then we’ll look at how all this is implemented by the OS

What’s a process?

- A “*program in execution*”
  - with associated (data and execution) context
- *Not* the same as “program” or “application”
  - a given program may be running 0, 1, or >1 times
- Each *instance* of a program is a separate process
  - with its own address space and other context
- Some OSs and GUIs obscure this distinction from the user. Don’t be fooled.
  - e.g., what happens when you double click the browser icon the first time? the second time?

Unix Process Hierarchy

- A Unix process comes in to being when another process creates it
  - a newly created process is the “child” of the “parent” that created it
  - every process has exactly one immediate parent
  - a process might create any number of children
- “Root” process, *init*, created when OS is booted
  - every process can be traced back to init
- Processes have a unique *process ID* (PID) number
  - index in the OS to the processes’ context

Creating a process: fork()

- The fork() system call creates a new process
- The child process is an (almost) exact clone of the parent (with it's own copy of parent's address space)
  - starts running as soon as fork() returns
  - both child and parent now running simultaneously
- What good is this?
  - write code to *behave differently* if you're the child
- How can you tell if you're the child or the parent?
  - for the child, fork() return value is 0
  - for the parent, fork() return value is the PID of child

Replacing a process: exec() and friends

- The exec() system call (and variants) *replaces* a process with a new program
  - it doesn't create any new processes
  - the new program is specified by the name of the file containing its executable code plus arguments
- The old code stops running as soon as it calls exec () (if the executable file is successfully run)
- What good is this?
  - usually run after fork()

Creating a child process

- Typically uses both fork() and exec()

```
pid=fork();
if (pid != 0)
  /* do parent stuff */
else
   exec("/bin/child");
```

Annoying details

- Exec is really a complicated family of system calls, with slightly different versions of the way each processes arguments
  - you specify the name of the executable file and its arguments (in an array of string pointers)
  - see the manual pages for execl() and execve()
- Fork might fail
- Exec might fail

Wait()

- Often a process will have nothing to do until its child terminates
  – e.g., what the shell usually does when you type a new command
- wait() *blocks* until the child terminates
  – so it doesn’t return immediately
- Annoying details: status argument, what if there’s no child, etc.

Creating a child process (slightly refined version)

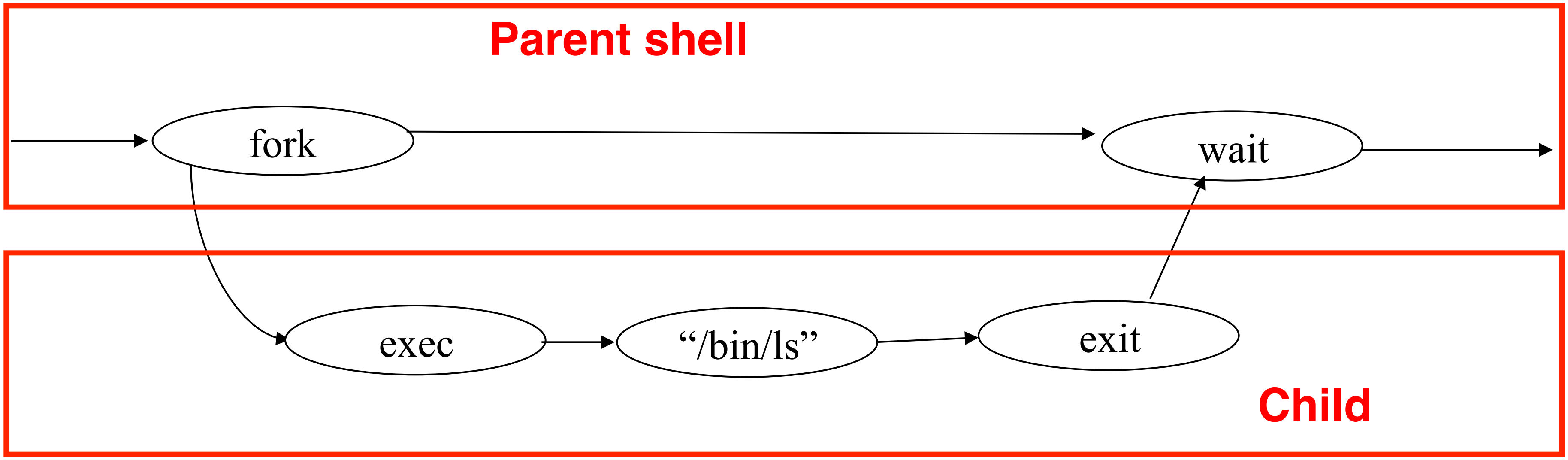
- Using fork(), execve() and wait()…

```
pid=fork();
if (pid != 0)
  wait(&status);
else
   if (execve("/bin/child"...)!=0)
       exit(1);
```

Wait() just a second here: What was this about “blocking?”

- Some system calls can “return” immediately
  - fork() and exec(), for example
- Others are asking for something that takes a while
  - wait(), for example
  - no sense in having the OS restart the process until the request is ready
- A process might be in one of three basic states:
  - *running*, *blocked* and *ready*
  - more on this later

How the Unix shell runs commands



- when you type a Unix “command” like “ls”, your shell process first *forks* an (almost) identical clone of itself
- the new “child” process makes an *exec* call (to the ls program), which causes it to stop executing the shell and start executing your command
- the original “parent” process, still running the original shell, *waits* for the child process to terminate

Communicating with processes: exit(), kill()

- exit() forces the current process to terminate
  - takes a return value that can be communicated back to parent via wait()
  - also called automatically when process ends
- kill() sends a *signal* to a process
  - if the process has set up a *signal handler*, it's called, much like an interrupt
    - otherwise the process terminates
  - restrictions on sending signals
    - only can send signals to the same user's processes

More system calls: I/O with read() and write()

- Much I/O is based on a streaming model
  - sequence of bytes
- write() sends a stream of bytes somewhere
- read() blocks until a stream of input is ready
- Annoying details:
  - might fail, can block for a while
  - file descriptors…
  - arguments are pointers to character buffers
  - see the read() and write() man pages

File descriptors open(), close()

- A process might have several different I/O streams in use at any given time
- These are specified by a kernel data structure called a *file descriptor*
  - each process has its own table of file descriptors
- Open() associates a file descriptor with a file
- Close() destroys a file descriptor
- Standard input and standard output are usually associated with a *terminal*
  - more on that later

More I/O stuff

- It's possible to hook the output of one program into the input of another
  - pipe()
- It's possible to block until one of several file descriptor streams is ready
  - select()
- Special calls for dealing with network
  - sockets, etc.

What’s the point here?

- System calls are the main interface between processes and the OS
  - system calls are like an extended “instruction set” for user programs that hide many details
  - first Unix system had a couple dozen system calls
  - current systems have a couple *hundred*
- Understanding the system call interface of a given OS lets you write useful programs under it
- Natural questions to ask:
  - is this the right interface? how to evaluate?
  - how can these system calls be implemented?

Implementing processes in Unix

- The OS kernel manages processes
- Has to solve various problems, including:
  - keeping track of the processes' states
    - including enough information to stop and restart them
  - keeping track of the available resources
    - those allocated to specific processes
    - reclaiming resources when a process releases them (or exits)
  - deciding who to run next (scheduling/dispatching)
  - managing communication
    - pipes, file I/O, etc.

Keeping track of process state

- A kernel data structure, usually called the *process table*, keeps track of each active process
  - entries in the table are *process control blocks* (PCBs), which contain data on state of each process
  - some of the data is directly in the PCB (e.g., registers)
  - other data in form of pointers (e.g., memory)
- In other words, a process' *context* is described by its process control block
  - this allows the OS kernel to stop and restart processes from where they left off

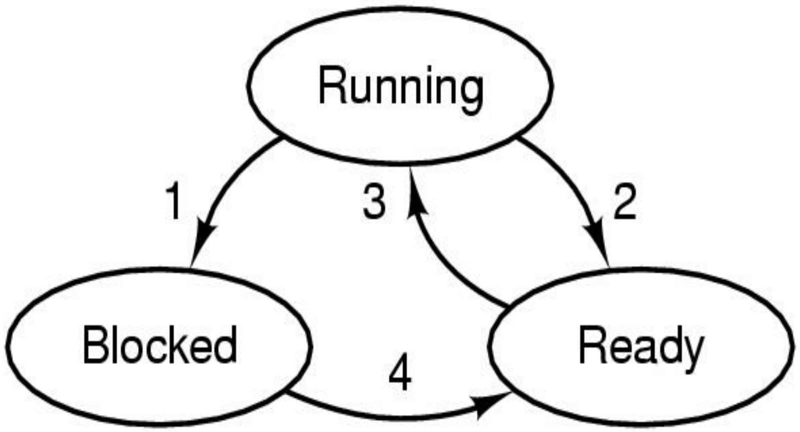
Process Context

- The full context of a process includes:
  - contents of main memory
  - contents of CPU registers, including the program counter, etc.
  - other info (open files, I/O in progress, etc.)
- Main memory -- three logically distinct regions of memory:
  - text region: contains executable code (typically read-only)
  - data region: storage area for dynamically allocated data structure, e.g., lists, trees (typically heap data structure)
  - stack region: run-time stack of activation records
- Registers: general registers, PC, SP, PSW, segment registers
- Other information:
  - open files table, status of ongoing I/O
  - process status (running, ready, blocked), user id, ...

Who’s running when?

- Most of the time, a regular (user mode) process is running
- When a system call or an interrupt occurs, the kernel runs (in kernel mode)
  - as soon as it’s finished, the *scheduler / dispatcher* picks a new process to run
  - it uses the PCB of that process to restart it

Basic Process States



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possible process states
  - Running: executing (only one per CPU)
  - Blocked: waiting (e.g., for I/O)
  - Ready: waiting to be scheduled

Implementing system calls

- When a system call or interrupt occurs:
  - first do whatever is requested
    - schedule I/O, update real time clock, etc.
  - maybe move some processes from *blocked* to *ready*
  - go through the list of *ready* processes, pick one and restart it based on its PCB (in the process table)
    - how to pick? it's a policy question.
      - fairness, efficiency, etc.
      - we'll discuss in detail later

Implementing Unix System Calls (oversimplified)

- fork()
  - duplicate process table entry, allocate and copy memory
  - both parent and child become ready immediately
- exec()
  - open file, load (or map) contents to memory, reset context
  - process goes to ready state as soon as this happens
  - might block if opening or loading the file takes a while

Implementing Unix System Calls (continued)

- wait()
  - if child not running, put process in ready state
  - if child is running, put process in blocked state
- exit()
  - terminate process and release resources
  - if parent is blocked in wait(), change parent's state to ready

Implementing Unix System Calls (continued)

- read()
  - if error, put process in ready state
  - if input is available, put process in ready state
  - if no input available, put process blocked state
- write()
  - if error, go to ready state
  - if I/O channel not available, put process in blocked state, otherwise put process in ready state
  - if another process was blocked waiting for I/O, put that process in ready state
- On many OSs, I/O goes through a *scheduler*

Other system calls

- signal(), sleep(), select(), etc all ask OS kernel to perform specific functions for process
  - block until something happens
  - send a message to another process
  - interact with I/O devices
  - etc
- System calls documented in section 2 of the Unix / Linux manual
  - type “man 2 read” for example

A question…

What’s the difference between a system call (like write()) and a standard library function (like printf())?

What’s the point here?

- System calls are the main interface between processes and the OS
  - system calls are like an extended “instruction set” for user programs that hide many details
  - first Unix system had a couple dozen system calls
  - current systems have a couple *hundred*
- Understanding the system call interface of a given OS lets you write useful programs under it
- Natural questions to ask:
  - is this the right interface? how to evaluate?
  - how can these system calls be implemented?

So far…

- System calls are the interface that processes (running programs) use to communicate with the OS
  - called like functions, but implemented in the OS
- We’ve looked at the basic Unix process management system calls
  - fork(), exec(), wait(), kill(), exit()
- And some I/O system calls
  - read(), write()

Why bother?

- Understanding the system call interface of a given OS lets you write useful programs under it
- Natural questions to ask:
  - is this the right interface? how to evaluate?
  - how can these system calls be implemented?

Implementing processes in Unix

- The OS kernel manages processes
- Has to solve various problems, including:
  - keeping track of the processes' states
    - including enough information to stop and restart them
  - keeping track of the available resources
    - those allocated to specific processes
    - reclaiming resources when a process releases them (or exits)
  - deciding who to run next (scheduling/dispatching)
  - managing communication
    - pipes, file I/O, etc.

Keeping track of process state

- A kernel data structure, usually called the *process table*, keeps track of each active process
  - entries in the table are called *process control blocks* (PCBs), which describe the running state of each process
  - some of this is stored in the PCB (e.g., registers)
  - other data in form of pointers (e.g., memory)
- In other words, a process' *context* is described by its process control block
  - this is enough information for the OS kernel to stop and restart processes from where they left off

Process Context

- Process' full context (process table entry) includes:
  - the memory it's using for code and data
  - contents of CPU registers, including the program counter, etc.
  - other things (open files, I/O in progress, etc.)
  - details depend on the particular hardware architecture
- Memory -- three logically distinct regions of memory:
  - text region: contains executable code (typically read-only)
  - data region: storage area for dynamically allocated data structure, e.g., lists, trees (typically heap data structure)
  - stack region: run-time stack of activation records
- Registers: general registers, PC, SP, PSW, etc.
- Other things:
  - open files table, status of ongoing I/O
  - process status (running, ready, blocked), user id, ...

Running Programs: Who's running when?

- Most of the time, a (*user mode*) process is running
- When a system call or an interrupt occurs, the kernel runs (in *kernel mode*)
  - kernel does what it has to, then the *scheduler / dispatcher* part of the kernel picks the next process to run
    - maybe the same one, maybe a different one
  - scheduler uses the PCB (process table entry) of the selected process to find out how to restart it

Non-Blocking vs. Blocking

- Some system calls are “non-blocking”
  - OS can return to the calling process immediately
  - ask the OS to do something, but start running again right away
  - fork()
- But others explicitly “block” the calling process until something happens
  - process goes to sleep until conditions have changed to where it makes sense to run
  - wait(), read()

Blocking let’s us avoid “polling” system calls

```
pid=fork();
if (pid != 0) /* parent */
  wait(&status); /* parent blocks
  */
else /* child */
   if (execve(“/bin/child“…)!=0)
      exit(1);
```

Why is this a good way to do things?

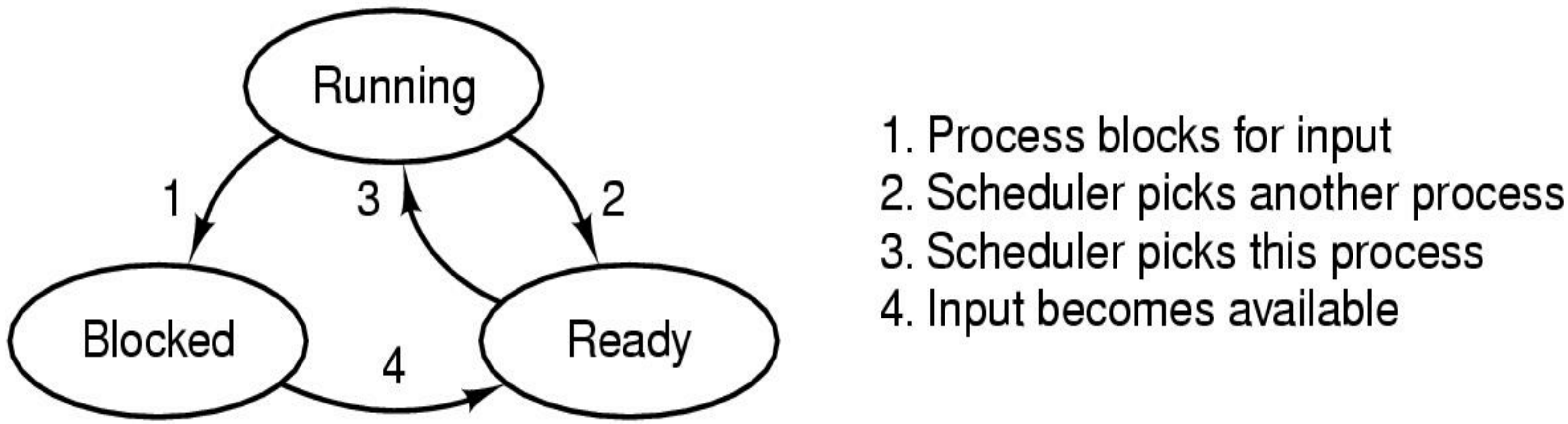
(hint: think about *The Simpsons*)

What if wait() didn't block?

- We'd need an "ischilddeadyet()" syscall
  - returns TRUE if child is dead, else FALSE

```
pid=fork();
if (pid != 0) /* parent */
  while (ischilddeadyet(pid)==FALSE)
     ;  /* "are we there yet?" */
else /* child */
  if (execve("/bin/child"…)!=0)
     exit(1);
```

Basic Process States



- Processes can be in any of three states:
  - ***Running***: currently executing; at most one per CPU
    - may become *blocked* or *ready* after a system call
  - ***Blocked***: waiting for something (e.g., I/O)
    - currently *not* eligible to be scheduled to run
  - ***Ready***: not waiting for anything
    - not running, but eligible to be scheduled to run

What does the OS do when a system call occurs?

- First, do whatever actions are requested
  - schedule I/O, update real time clock, etc.
  - based on these actions, we might move some processes from *blocked* to *ready*
- Next, go through the list of *ready* processes and select one to run next
  - how to pick? different OSs have different policies.
    - fairness, efficiency, etc.
    - we'll discuss in detail later
- Finally, restart selected process based on its PCB entry (in the process table)

Implementing Unix System Calls: Process Management

- fork()
  - duplicate process table entry, allocate and copy memory
  - both parent and child become *ready* immediately
- exec()
  - open file, load (or map) contents to memory, reset context (e.g., set PC to first instruction)
  - process usually goes to *ready* state as soon as this happens
    - might *block* if opening or loading the file takes a while

Implementing Unix System Calls: Process Management cont'd

- wait()
  - if child not running, put process in *ready* state
  - if child is running, put process in *blocked* state
- exit()
  - terminate process and release resources
    - remove PCB entry
  - if parent is blocked in wait(), change parent's state to *ready*

Implementing Unix System Calls: I/O

- read()
  - if error, put process in *ready* state
    - return error code
  - if input is available, put process in *ready* state
  - if no input available, put process *blocked* state
- write()
  - if error, go to *ready* state
  - if I/O channel not available, put process in blocked state, otherwise put process in *ready* state
  - if another process was blocked waiting for input from this write, put that process in *ready* state
- On many OSs, I/O doesn't happen immediately
  - goes through a *scheduler*

Other system calls

- signal(), sleep(), select(), etc ask OS kernel to perform various other services for process
  - *block* until something happens
  - send a message to another process
  - interact with I/O devices
  - etc
- System calls are documented in section 2 of the online Unix / Linux manual
  - try "**`man 2 read`**" for example

Details: Interrupts, System Calls and Processes

- A more detailed look a how basic OS services are implemented on real CPUs
  - Traps and system calls
  - User/supervisor mode
  - Context switching
  - Serving interrupts
  - Managing processes
- Efficiency issues

What’s a system call, really?

- Looks a lot like a function as far as the programmer is concerned
  - parameters, return value, etc
  - function call wrapper
- Main difference is that the system call executes code in the OS (in supervisor mode)
  - In this sense, can think of system call as like an extension of the instruction set
- How to communicate parameters and return value between the program and the OS?

Calling a regular function (not a system call)

- Call (typical CPU architecture):
  - Push calling data onto stack
    - parameters
    - state (general registers, control registers, etc)
  - Jump to (load PC with address of) first instruction of function
- Return:
  - pop state off stack and reload registers
  - push return value onto stack
  - jump to next instruction of caller (as read from stack)

Invoking a system call is similar, except:

- The code for the “function” is in the OS
  - operates in **supervisor** mode
  - different address space
- Limited number of entry points (“functions”)
  - a user process can’t just jump to anywhere it likes in the OS
  - enforced by semantics of TRAP instruction
- A system call might not return right away
  - process’ state might change to *blocked* as a result of the system call’s actions
  - OS might decide to schedule a different program

Calling a system call

- Push data onto stack
  - parameters to system call
  - system call number
  - registers, current state, etc
- Execute a TRAP instruction
  - this causes several things to happen
    - change CPU mode bit from **user** to **supervisor**
    - jump to specific address in OS address space
      - as indexed by system call number

Calling a System Call (continuted)

- OS pops and saves calling process's state from user stack
  - saved in process table entry
- OS pops syscall number and parameters (from stack)
- OS calls appropriate system call function
  - pushes return value onto process's stack
- OS calls scheduler/dispactcher to select and load next process to run
  - maybe the one that issued the syscall, maybe not

Returning from a system call

- OS sets mode bit back to USER
  - and reloads state registers (PC, etc)
- Now we're back in the calling process
  - next instruction is the one after the TRAP that issued the syscall
- Process pops syscall return value from stack and gets on with its business

Next class

- Concurrency:
  - Processes, threads, semaphores, monitors
  - First homework assigned
- Readings (which you should have done already)
  - Chapters 1-3
- Your TODO list:
  - Log onto speclab cluster
  - Get your textbook!
  - Visit course website: http://www.crypto.com/courses/spring12/cis505/
  - Note Homework 1:
    - http://www.crypto.com/courses/spring12/cis505/hw1.html
  - Start looking for a project partner.