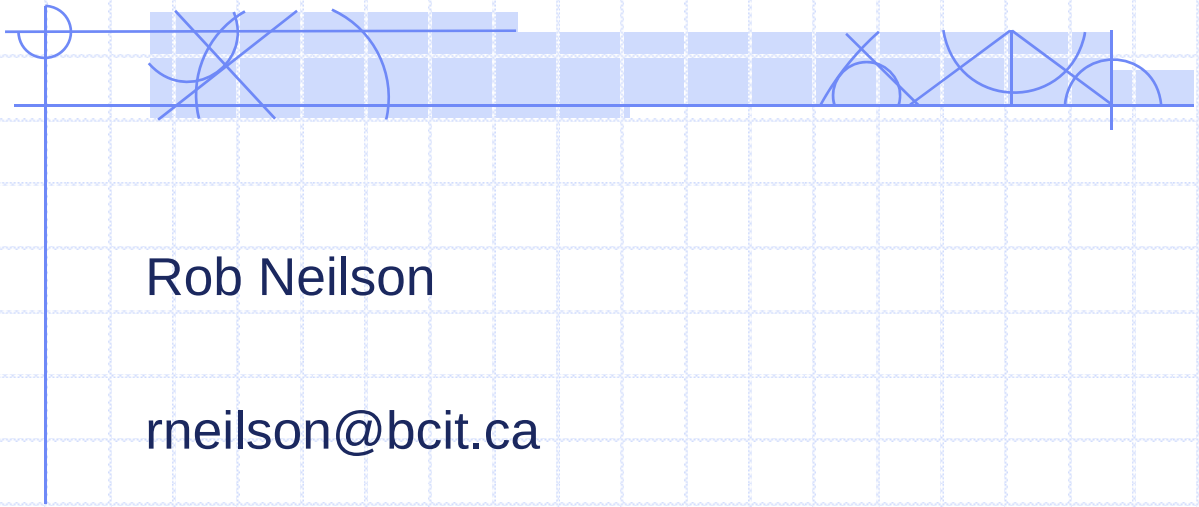
COMP 4735: Operating Systems

Lecture 4: OS Concepts



Rob Neilson

rneilson@bcit.ca

Reading

- The following sections should be read before next Monday
  - it would also help to read this before your lab this week
  - Textbook Sections: 1.5, 1.6, 1.7 (Theory Part)
    10.2 (Case Study Part)
- Yes, there will be a quiz next Monday in lecture on the above sections
  - A sample quiz will be posted on webct on Friday
  - This quiz (Quiz 2) covers material from all of the sections listed above

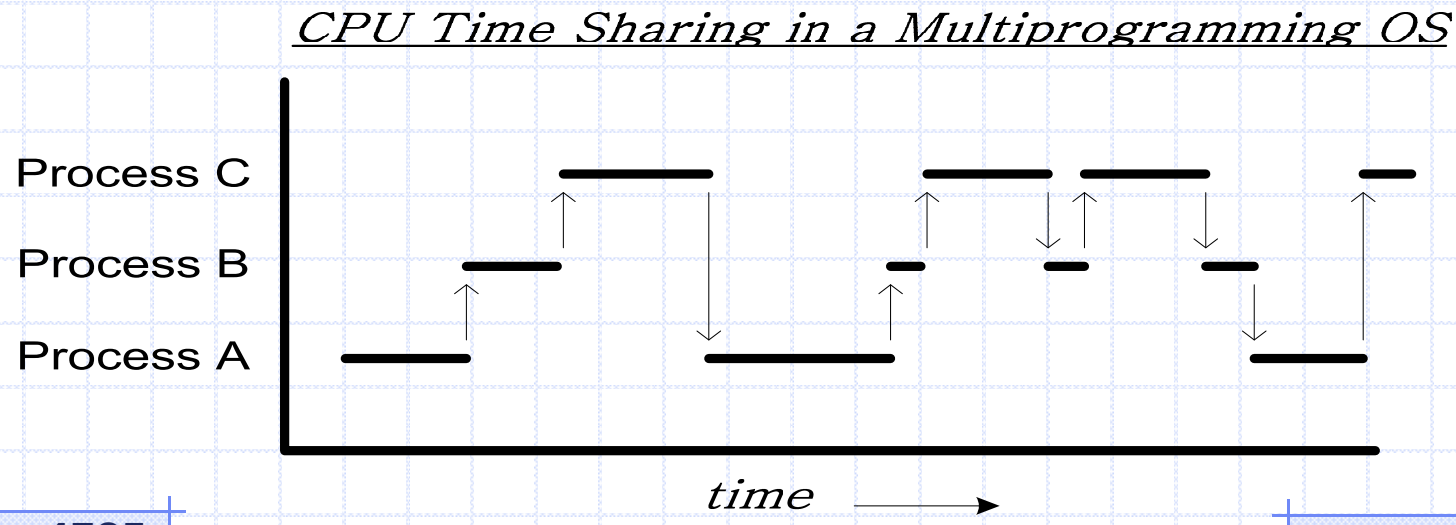
Agenda

Key concepts for this lesson:

- Multi-processing
- Processes
- Address Space
- The File System
- System Resources

Key Concept: Multiprocessing

- Idea is that many programs can run ***concurrently*** on the same computer
- To make programs run concurrently, the OS has to implement the idea of an ***abstract machine*** (sometimes called a *virtual machine*)
- Most OS's implement an abstract machine with an abstraction called a ***process***
- *The OS is responsible for coordinating the execution of processes*



Key Concept: Process

What is a Process?

- a ***process*** is a ***program in execution***
- there are many processes executing on a computer at any given time
- *each process is self-sufficient*
  - ie: each process contains everything that is needed for its program to execute
  - we call this the ***process state*** (or status) information

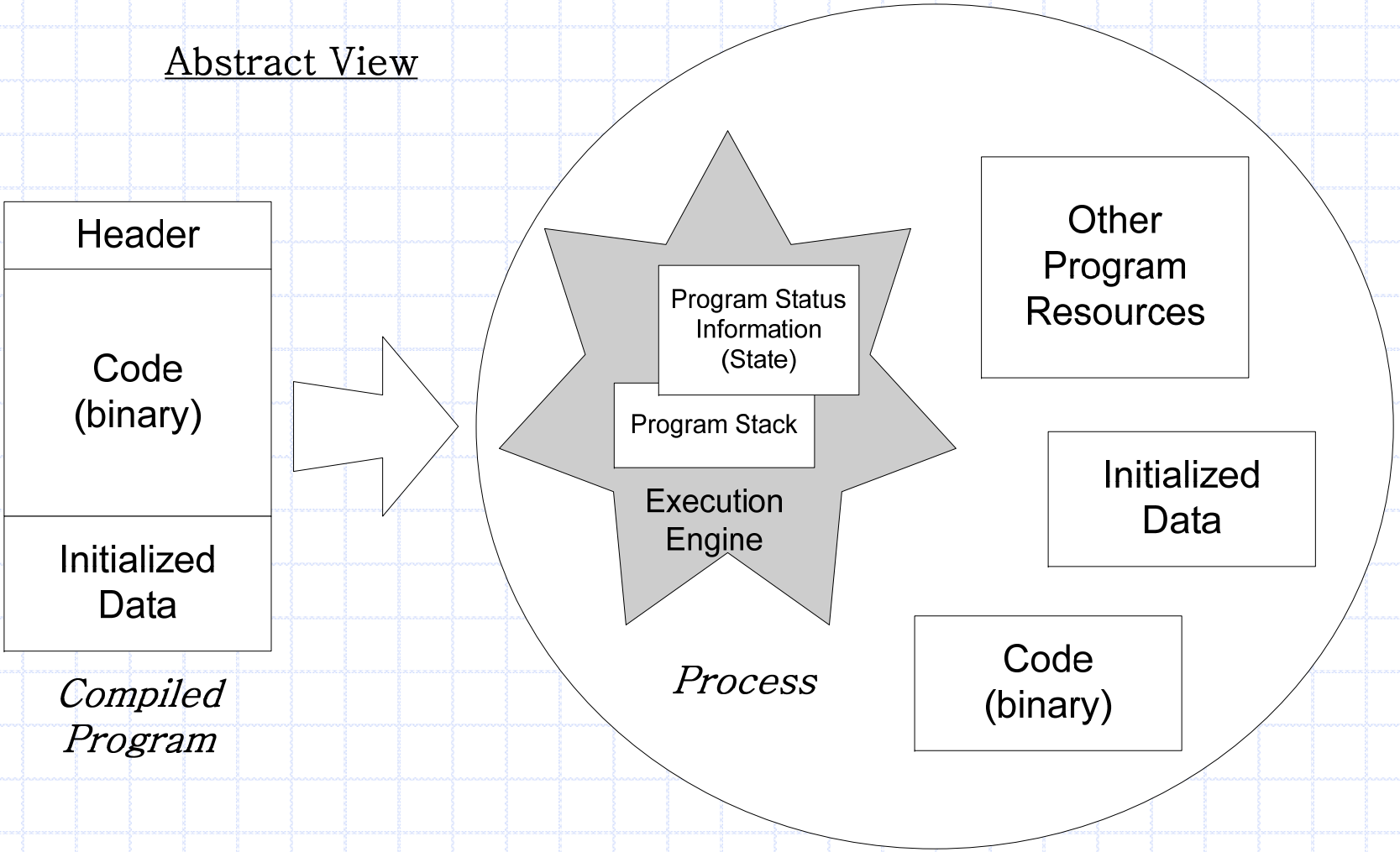
Process Table

A program requires the following things to be able to execute

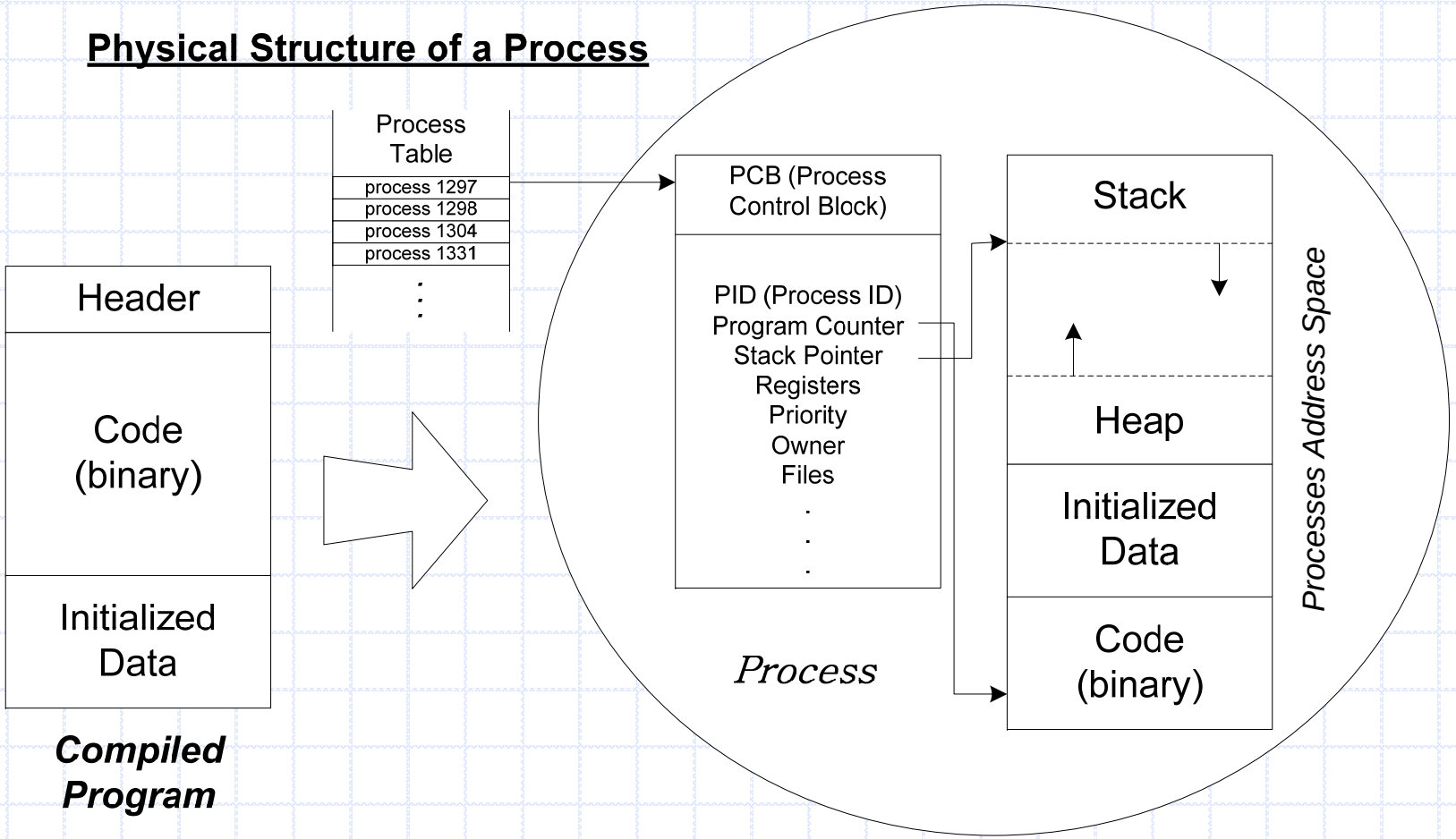
- an address space
- program code (binary)
- program data (static vars, constants etc)
- process stack (state info for function calls)
  - parameters, return address, local variables, etc
- registers
  - program counter (indicates next instruction)
  - stack pointer
  - other general purpose registers
- system resources ... handles/pointers for things like:
  - open files
  - network connections

Note: sometimes we call a process a "job" or a "task"

Anatomy of a Process

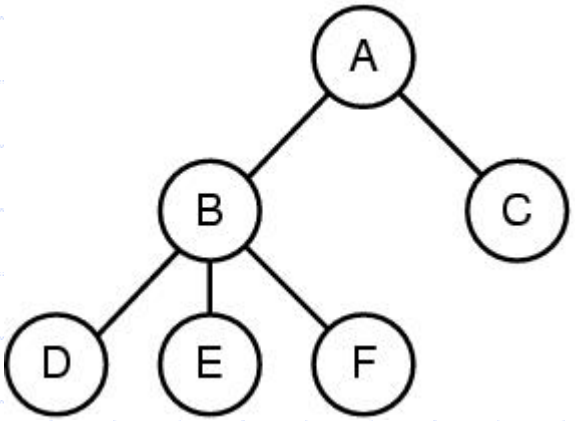


Process Table and PCB



Process Hierarchy

- processes are created using a system call
  - eg: fork() or CreateProcess()
- this suggests a parent-child relationship between processes

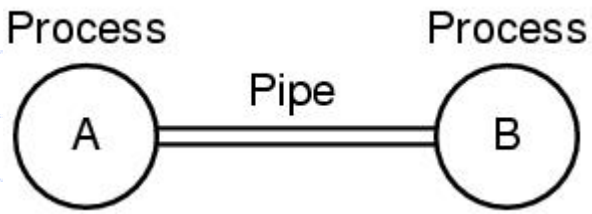


Key Concept: Address Space

- the Address Space is an abstraction for memory (RAM)
- each process has its own Address Space, which is simply the range of addressable memory on the computer
  - for example, processes in a 32-bit computer have an address space from **`0x00000000 - 0xFFFFFFFF`**
- each process maintains a memory map for its address space, and the map is different for each process
  - pages of memory are loaded, using the virtual memory idea, as needed for execution of the current process
- we make the assumption that a processes address space is private and can only be accessed by that process

Cooperating Processes

- often there is a need for processes to share information with each other, ie, to communicate
- this idea is referred to as Inter-process Communication (IPC)
- can be implemented in many different ways
  - shared memory
  - pipes
  - sockets
  - messages

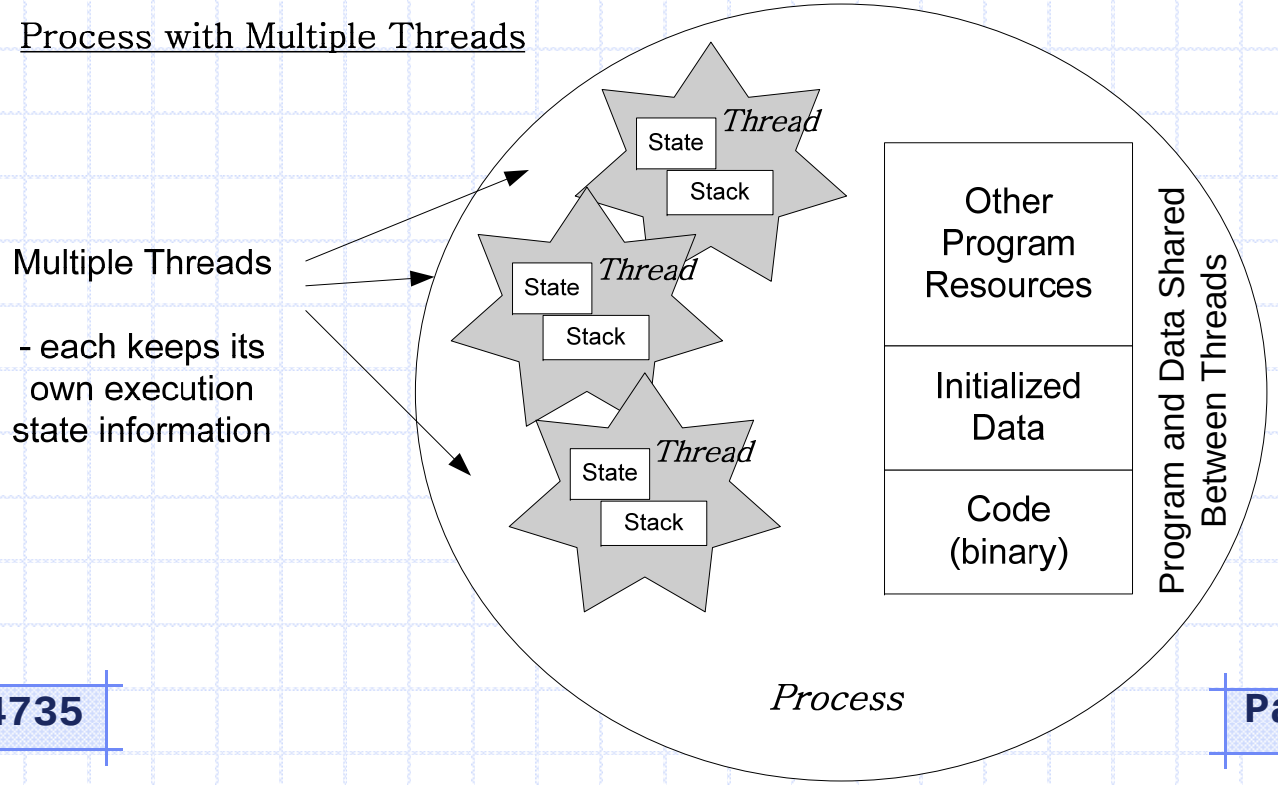


Shared Address Space

- what if we want to have two processes that share a single program?
  - we would need shared address space support
  - this is difficult to implement and introduces various protection issues
- ***thread model*** simplifies the problem
  - define the concept of a thread, where multiple threads of execution exist within a single process
  - all threads in a process share the processes address space, program, data etc
  - unrelated threads do not have access to the address space
  - each thread has its own stack and state information so it can execute independently of other threads in the process
- this makes it trivial for threads to share a program and data
  - If you want sharing, encode your work as threads in a process
  - If you do not want sharing, place threads in separate processes

What is a Thread?

- there may be ***one or more threads executing within a process***
- in a thread model
  - the ***process defines the execution environment*** for all the threads
  - each ***thread maintains its own execution state info***
- all the threads in a process ***execute the same program***, but ...
  - each thread has it's own: stack, program counter, program status info

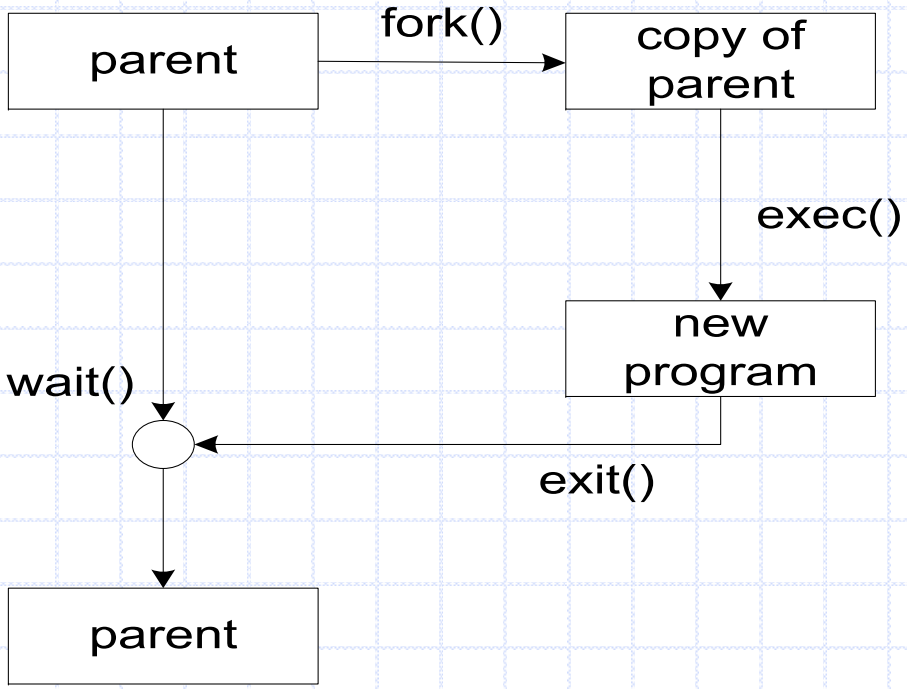


Process Creation (Unix)

- the system boot program creates the *first process*
- all other processes are created from the first process
- processes have a parent / child relationship
  - ie: we have a process tree
- processes are created with a system call: **`fork()`**
  - Creates a new address space
  - Copies text, data, & stack into new address space
  - Provides child process with access to open files
- a program can be loaded into the new process with the system call **`exec()`**
- **`wait()`** allows a parent to wait for a child to terminate

Fork Diagram

Fork / Exec / Wait Example




```
main {
    int childPID;

    childPID = fork();

    if(childPID == 0)
        <code for child>
    else {
        <code for parent>
        wait();
    }
}
```

The concept of `fork()` - 1

- **`fork()`** creates a child process that is an exact copy of the parent
  - inherits all parents variables
  - inherits the contents of the parents memory
  - inherits all register values and state (except PID)
  - inherits a copy of the parents file descriptors

- parent and child each have their own *context*
  - address space, PCB, etc

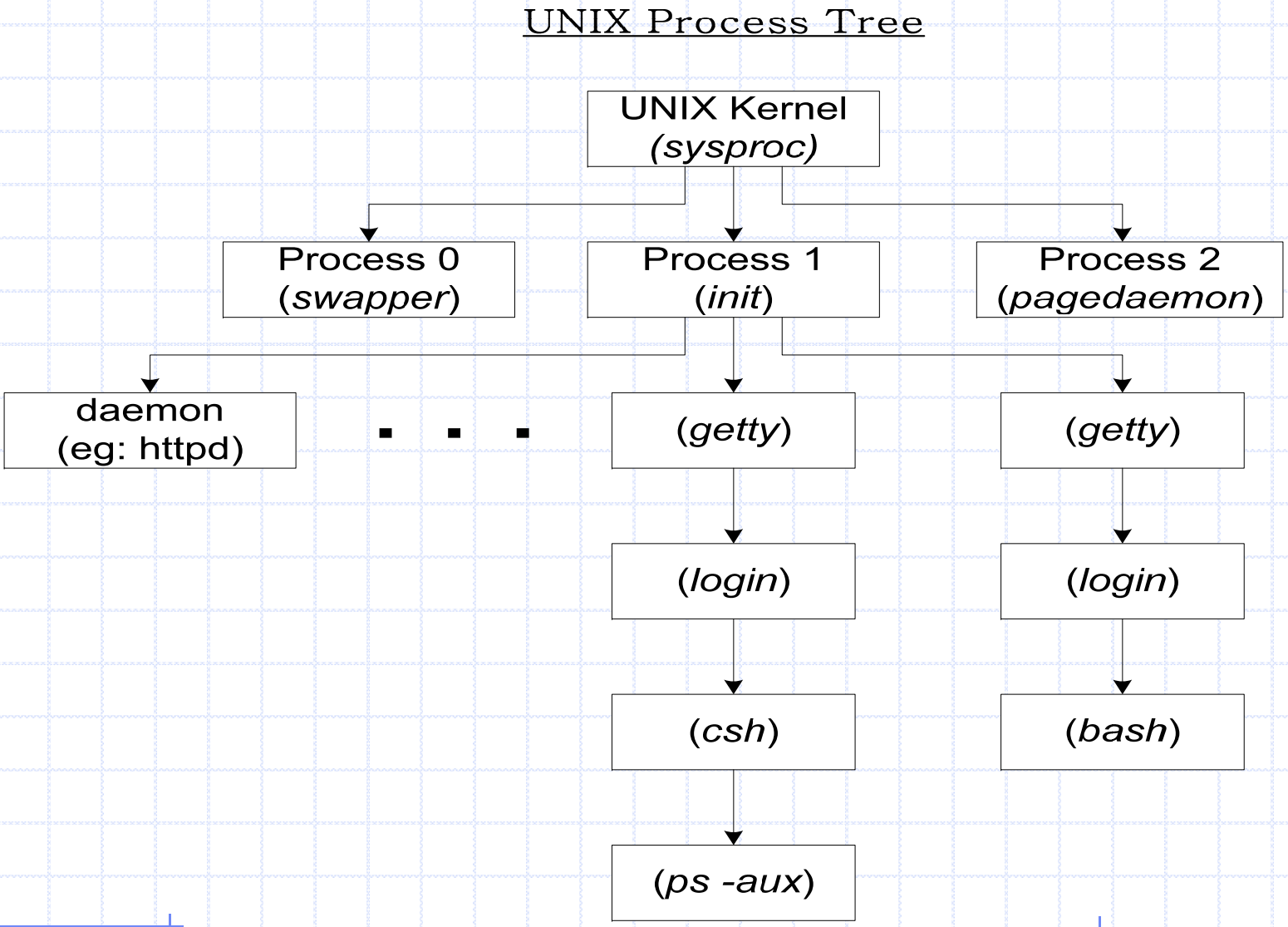
The concept of **`fork()` - 2**

- both parent and child start executing at same place (in their own copy of the code) after the fork

```
childPID = fork();
if(childPID == 0)
        <code for child>
else
        <code for parent>
```

- parent can choose to wait (**`wait()`**) for the child to finish, or it can continue to execute in parallel
- child can choose to load a completely different program (**`exec()`**) and start executing the new program at the default entry point

Example of fork: Unix Process Tree



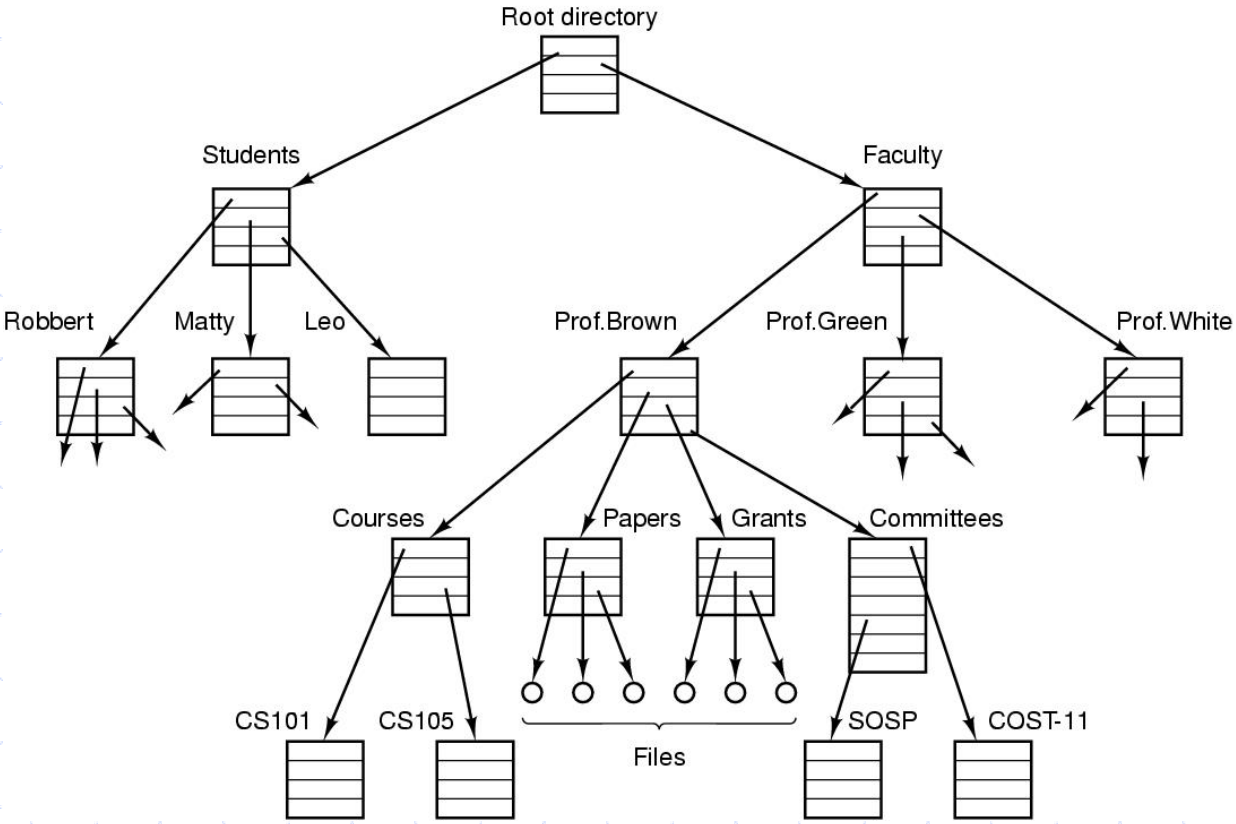
Key Concept: System Resources

- Anything that a process requests from an OS
  - Available ⇒ allocated
  - Not available ⇒ process is blocked
- Examples
  - Files
  - Actual primary memory (“physical memory”)
  - Devices
    - e.g., window, mouse, keyboard, serial port, etc
  - Network port
  - … many others
- The list of resources allocated to a process is stored in a table that is referenced through the PCB

Key Concept: The File

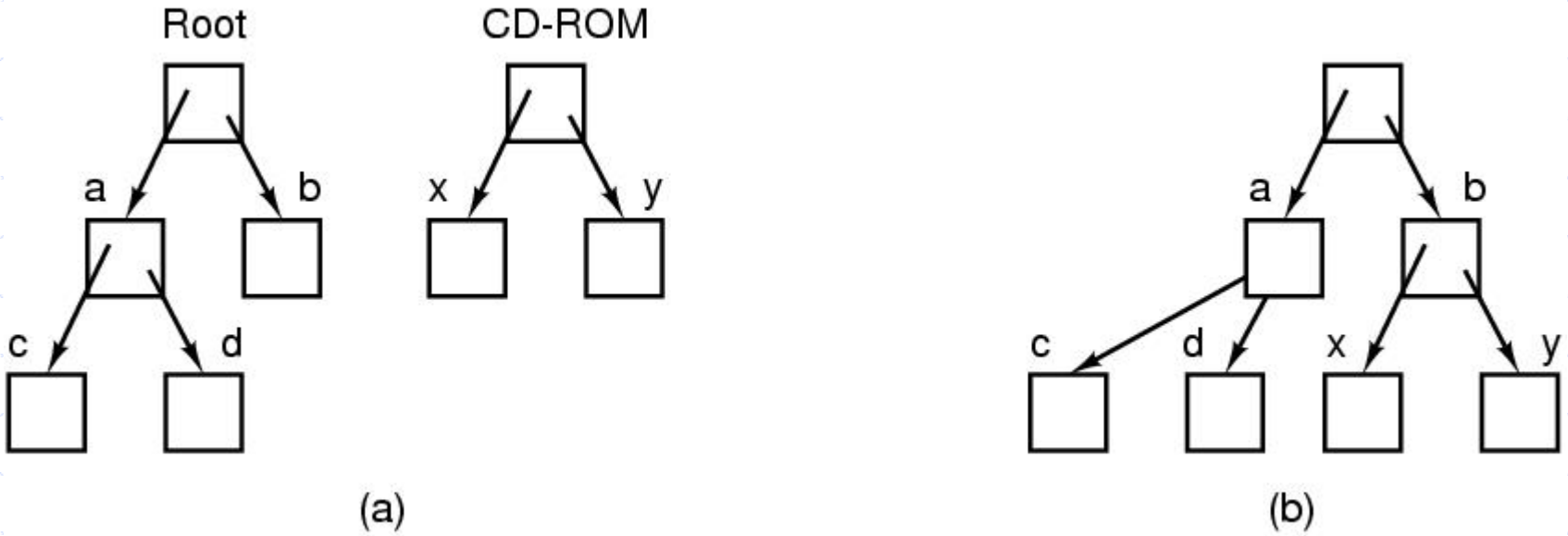
- Data must be read into (and out of) the machine
- Storage devices maintain persistent copy of data
- Need an abstraction to hide device specific details of storage devices
  - the abstraction is the ***file***
- A file is a linearly-addressed sequence of bytes
  - From/to an input device
  - Including a storage device

The File System



- directories are further abstractions
  - used to organize files
- the complete collection of files and directories is called the ***File System***

Mounting File Systems



(a) Before mounting, the files on the CD-ROM are not accessible.

(b) After mounting, they are part of the file hierarchy.

Special Files: Extending the File Abstraction

- Unix designers so much liked the file as an abstraction that they used it for pretty much everything
- Devices (eg: printer, tape drive, terminal etc) can all be thought of as files
  - all receive or supply an ordered series of bytes
- There was a desire to use the same system calls for these devices as are used for regular files (eg: **`read()/write()`** etc)
- Created special files to represent these devices, and mount or place the special files in the file system
  - in reality the special files are references to the device drivers for these devices

Device Special Files

- there are two types of special files for devices

1. character special files: for stream oriented devices such as tty's, printers etc

2. block special files: for block oriented devices such as hard drives

The File Abstraction Again: Pipes

- in Unix a pipe is a mechanism for inter-process communication
- processes communicate by reading and writing from the pipe using the standard read and write system calls
- to establish communication using a pipe:
  - a process creates a new pipe
  - the process forks a child
  - recall that the child process inherits all the files of its parent
  - so now both parent and child have access to the pipe, and can read/write to it

Summary

- Multi-processing requires a process abstraction
- Each process is a program in execution
- The process defines everything that is needed for a program to execute
- Each process has an entry in a process table
- Each process has its own Address Space abstraction
- Each process maintains a list of system resources that it is using
- The File System is an abstraction of physical storage devices
- Unix uses the file abstraction for all sorts of things including most IO devices, IPC mechanisms

The End