CSci 4061

Introduction to Operating Systems

OS Concepts and Structure

Today

- OS Kernel
- Systems programmer view
- OS concepts/abstractions provided to systems programmer

Reading

Read Chapter 1 (R&R)

Opt:

Chapter 1 (LSP)

Chapter 1 (MOS) or

Chapters 1 and 2 (S&G)

The Kernel: core layer of the OS

- The kernel is a library of procedures shared by all user programs, **but** the kernel is protected:
  - User code cannot access internal kernel data structures (and associated code) directly
  - User code can invoke the kernel only at well-defined entry points, and these are?

system calls

The Kernel: cont’d

- Kernel code is like user code, but the kernel is privileged:
  - Kernel has direct access to all hardware, and handles interrupts and hardware exceptions
  - CPU is either executing OS code (kernel-mode) or your code (user-mode)

- An aside: some OS code may be in user-mode too

Systems Programming

- Low level, interfacing directly with the kernel and/or core system libraries (*libc.a*, *glibc.a*)

- What is a .a file?
  - Archive: bundle of .o files
  - What is a .o file? object file

Systems Programmer Viewpoint

- Systems programmer can use system calls **directly** (in assembly)
  - called in user-mode
  - executed by the OS (i.e. kernel mode)
  - when efficiency demands it
  - assembly code: x86 “`int`” instruction, e.g. `int` 48

identifies syscall #

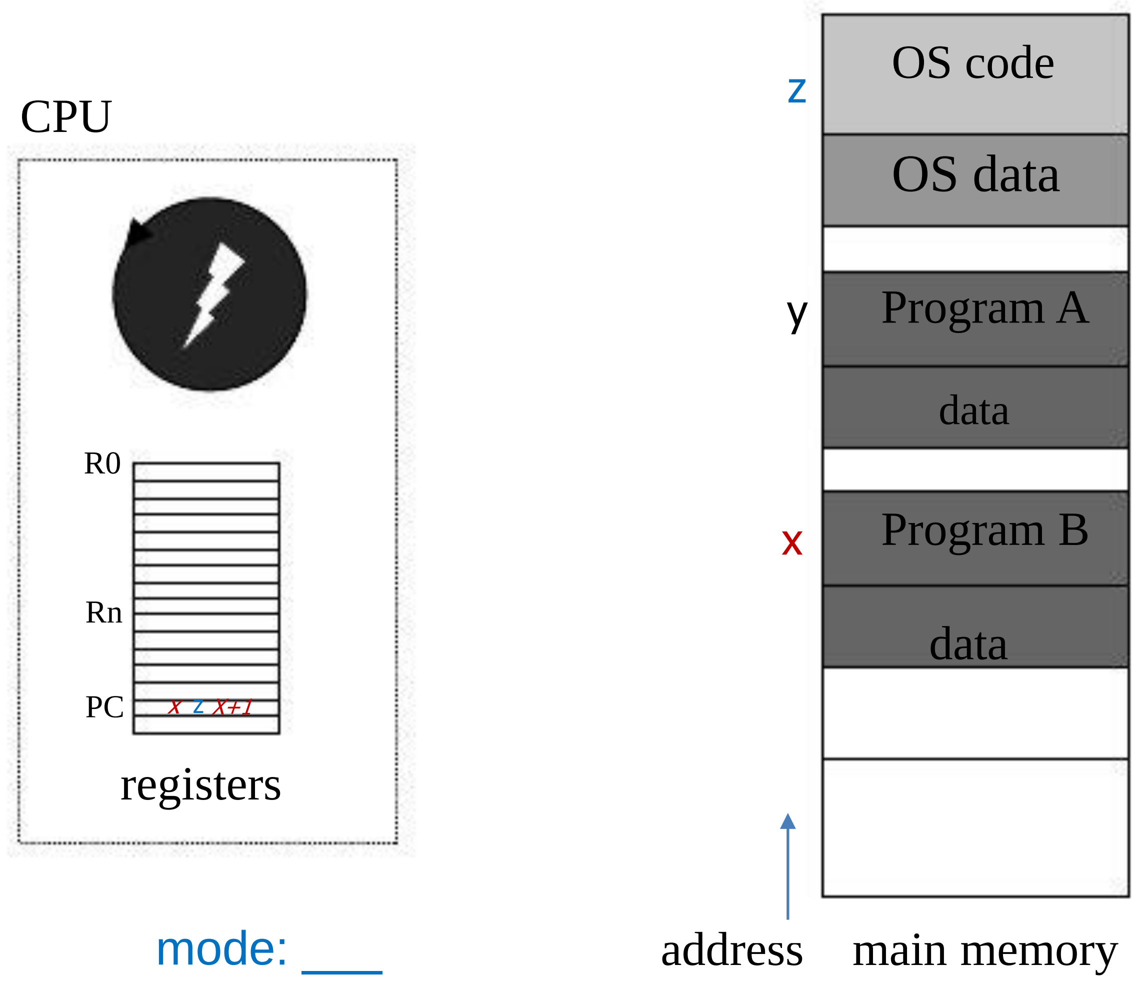
- Alternatively, language-specific libraries can be used to access system calls
  - C programming language libraries (libc.a, glibc.a)
  - E.g. `read (…)`

Terminology Alert!

- I will often refer to low-level library calls as system calls
  - e.g. `read (…);`
  - becomes `int #`

- Library (or system calls) are not part of the C language

Running programs: memory and the CPU



mode: ___

Program B makes a system call

System call completes, mode?

Let’s Look At

OS Concepts and **Abstractions**
Above the Hardware

Abstraction

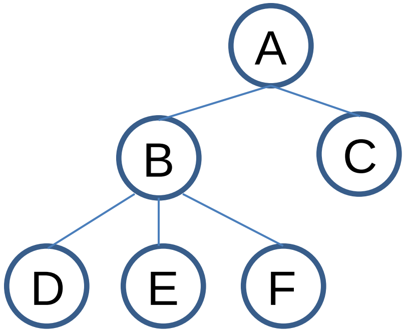
High-level construct

Useful, easy-to-use, understand

Hides lower-level details

PL: class or structure data-type

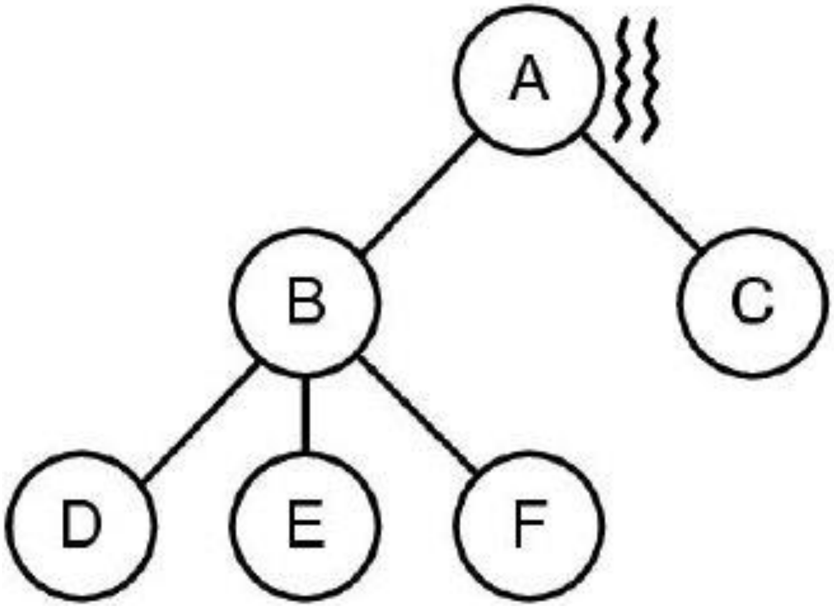
Operating System Concepts: Process



- Process is an executing program: container for computing resources (abstraction)
  - Process tree
  - A created two child processes, B and C
  - B created three child processes, D, E, and F

What resources?

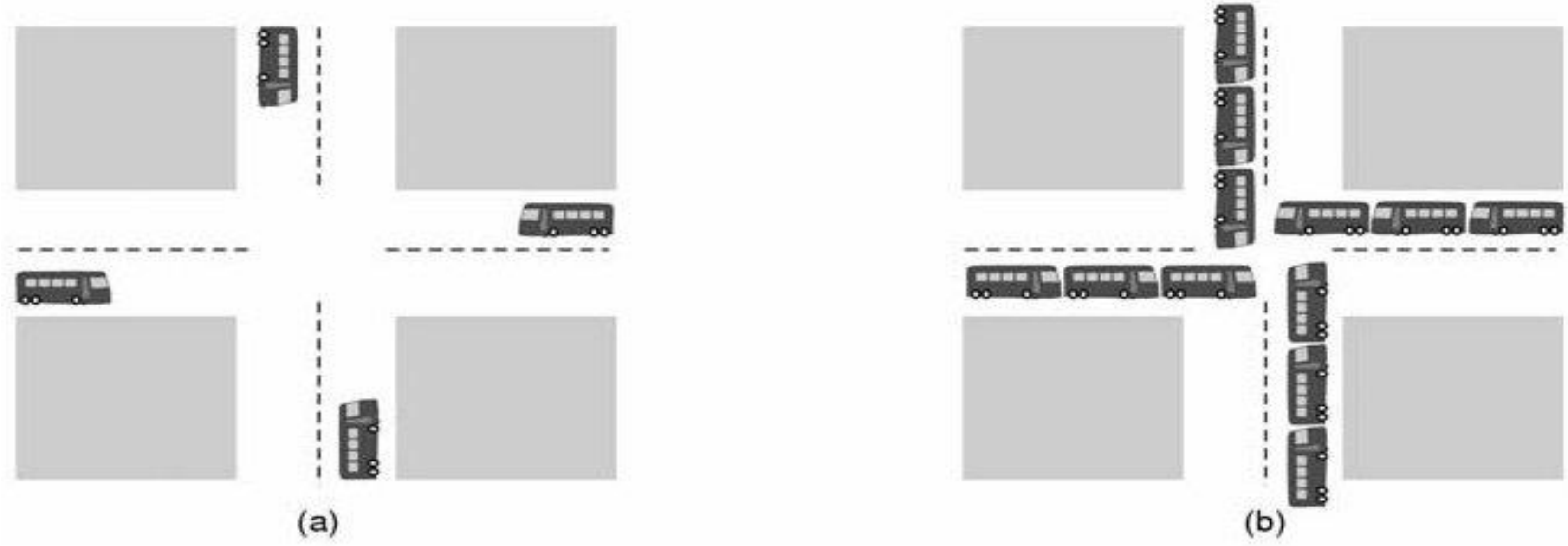
Operating System Concepts: Threads



- A thread is an executing stream of instructions normally within a process
  - A has two threads; share A's resources
  - Every process has at least one thread
  - Threads can also exist in the OS

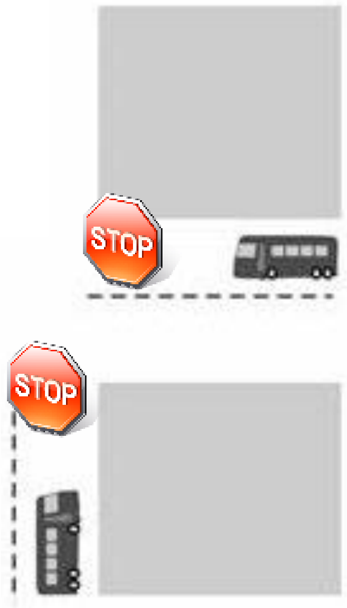
```
main () {
    int i;
    i=2;
}
```

Operating System Concepts: Synchronization



- Concurrency (processes/threads run together) and shared resources can lead to problems:
  - (a) Race condition
  - (b) Deadlock
- Solution: Synchronization, e.g. case (a)?

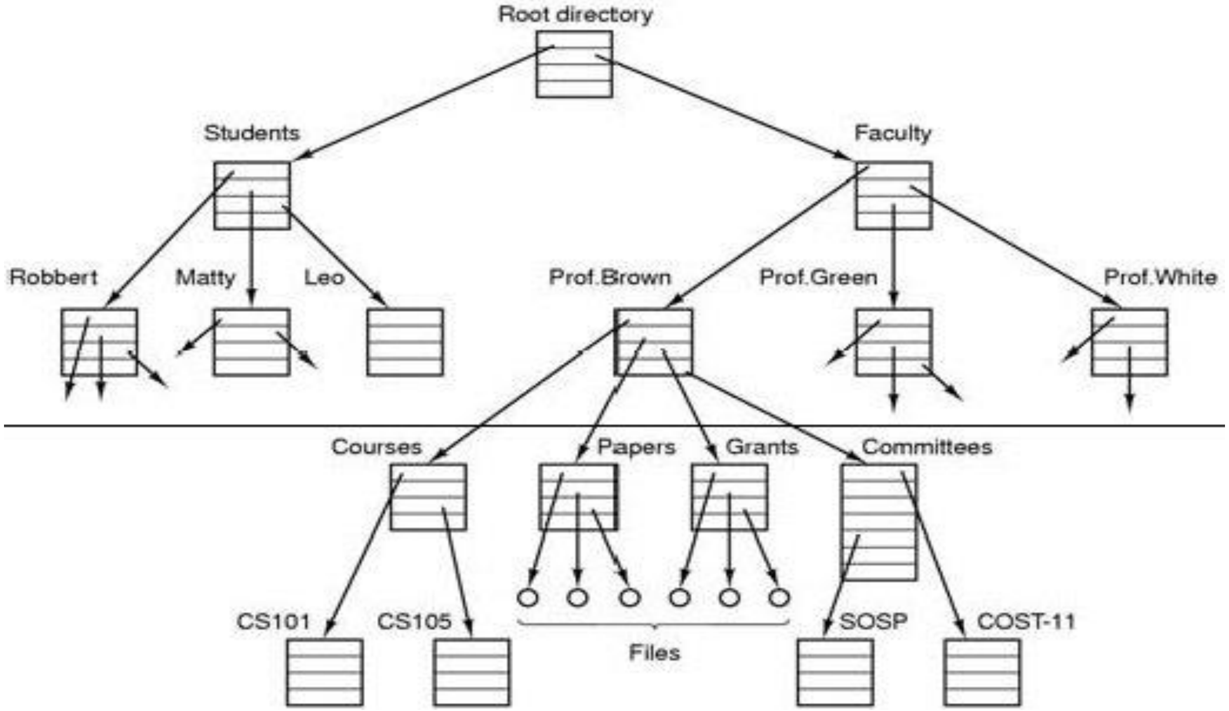
Operating System Concepts: Synchronization Issues



Livelock! (aka “Minnesota Nice”)
No one makes progress

Deadlock/Livelock is often caused by poor use of synchronization

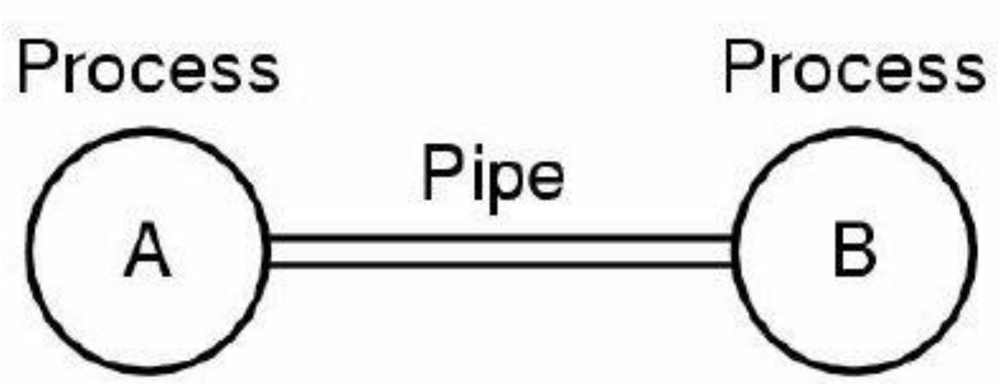
Operating System Concepts: File system



Files/directories are an OS **abstraction** to make data storing and data sharing easier

What are they abstracting?

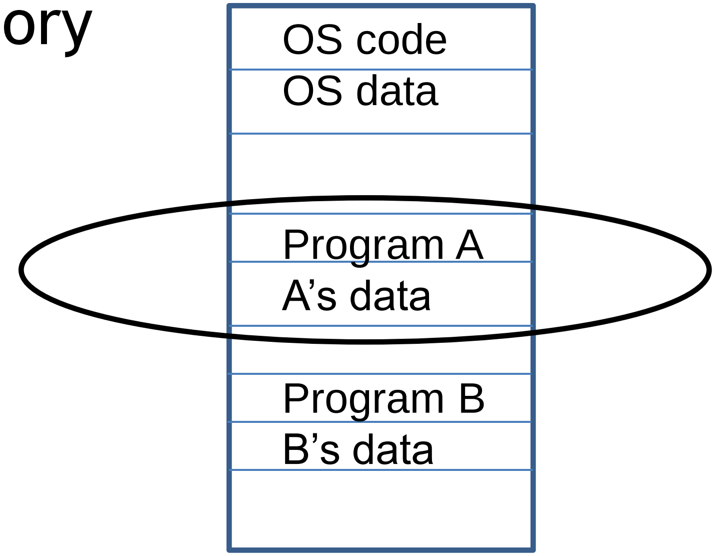
Operating System Concepts: Communication



- Two processes connected by a “pipe”, channel
- Processes need to communicate – why?

Operating System Concepts: Memory Management

- How is memory allocated to programs?
  - Largely an "inside" issue but ….
  - We will see how a program can make good use/bad use of memory

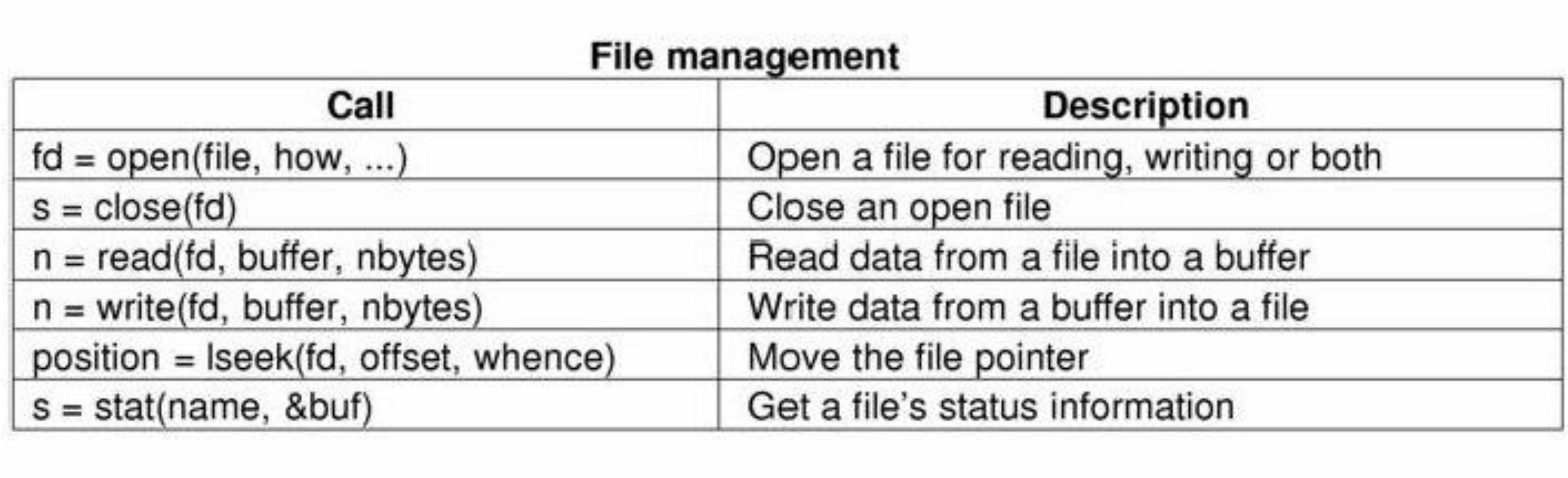


- **Abstraction** = virtual memory

Operating System Concepts: System Calls

- System calls are how user programs interact with the OS
  - Generally available as assembly-language instructions
  - C-Unix provides a library interface to system calls to avoid this messiness
  - e.g. **`read (…)`** gets compiled into the appropriate syscall linkage/assembly code
  - **`a = read(b, c)`** vs. **`a = myfunc (d, e)`**

Example: Some “System Calls” For File Management



File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file’s status information

In this course, we will use the term system call to refer to the C-Unix/Linux interface, e.g. **`open`**

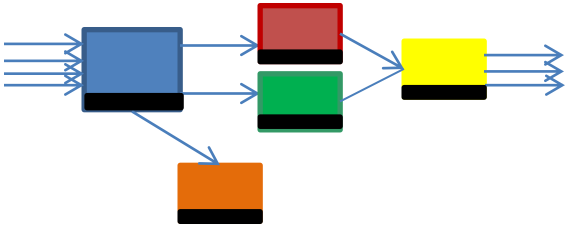
UNIX Standards

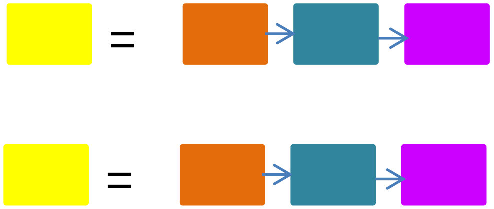
- POSIX: Portable Operating System Interface

- Linux is POSIX compliant

Systems Concepts: Taming Complexity

“systems”: OS, Web, Internet, …





Granularity

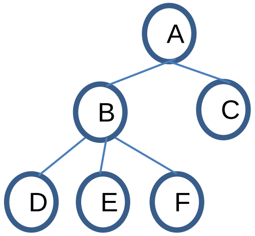
Modularity

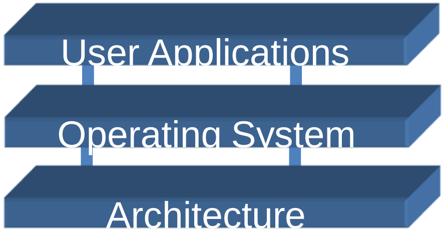
Abstraction

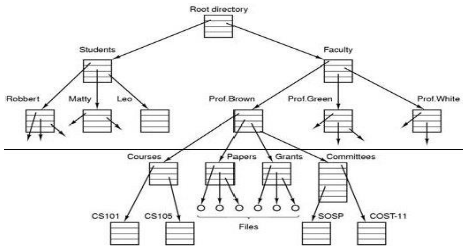
Layering

Hierarchy

Complexity







Complexity is Hard

- Different stakeholders => different metrics and requirements
  - Programmer => ease-of-problem-solving
  - End-user(**s**) => performance, ease-of-use
  - Owner (~ system) => fairness/priority, efficiency or utilization
  - Admin => security
  - OS Vendor => extensible, secure, reliable, …

- Tradeoff and conflict lead to complexity

This Weekend

- C/Linux Refresh

1. Edit and write a simple C program
2. Compile and run it
3. Look at a debugger such as DDD, GDB

Next Time

Programs and Processes in C and UNIX/Linux

Read Chapter 2,3 (R&R)

Lab on Monday: must attend

Quiz based on this week's material

this weekend

Have a great weekend!