Today

- Review: C
- Process abstraction
- Dual mode execution

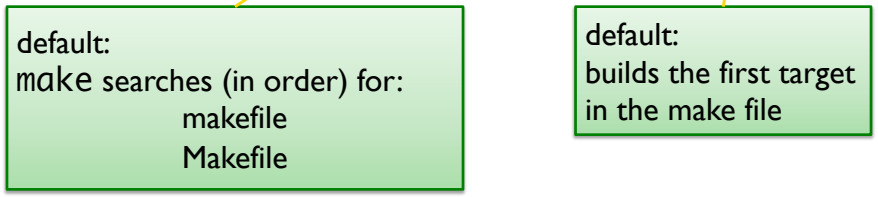
Review

- What are the standard streams?
  - How do you refer to them in C?
- What is a `struct`?
  - How do you refer to the fields in a struct?
- What is `make`?
  - How do you execute `make`?

Review: Using make

Invocation:

```
make  [ -f  makeFileName ]  [ target ]
```



default:
make searches (in order) for:
makefile
Makefile

default:
builds the first target in the make file

Understand the example given in assign2

Using files in C

```
FILE *fp;
…
fp = fopen(filename, "r");

if (fp == NULL) {
    … give error message and exit …
}
… read and process file …
int status = fclose(fp);
if (status == EOF) {
    … give error message…
}
```

Using files in C

```
FILE *fp;
…
fp = fopen(filename, "r");

if (fp == NULL) {
   … give error message and exit …
}
… read and process file …
int status = fclose(fp);
if (status == EOF) {
   … give error message…
}
```

We need error checking in C.

Review: C Pointers

- Consider the following function definition:

```
void my_function( char* my_ptr )
```

What can `my_ptr` point to?

a) One character
b) An array of characters
c) Either one character or an array of characters

Review: C Pointers

- Consider the following function definition:

```
void my_function( char* my_ptr )
```

What can `my_ptr` point to?

a) One character
b) An array of characters
c) Either one character or an array of characters

`struct` Example

What does this code do?
How would the function be used?

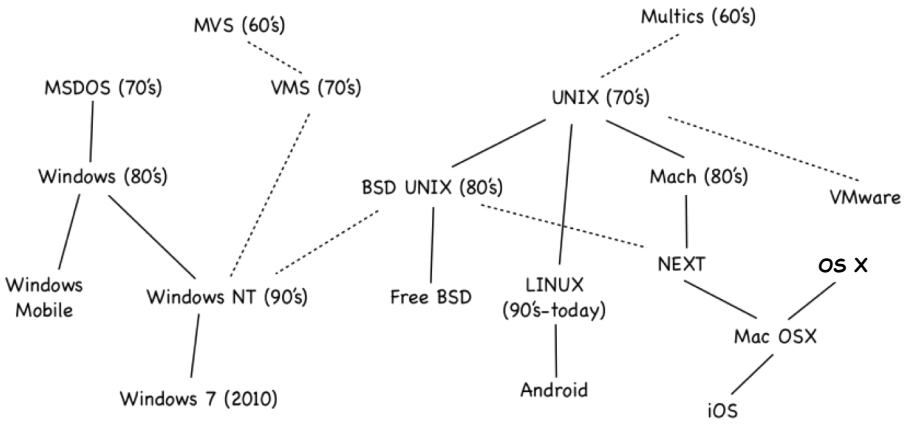
```
struct TIME {
  int seconds; int mins; int hours;
};

void differenceBetweenTimePeriod(struct TIME start,
                    struct TIME stop, struct TIME *diff) {
  if(stop.seconds > start.seconds){
    start.mins--;
    start.seconds += 60;
  }
  diff->seconds = start.seconds - stop.seconds;
  if(stop.mins > start.mins){
    start.hours--;
    start.mins += 60;
  }
  diff->mins = start.mins - stop.mins;
  diff->hours = start.hours - stop.hours;
}
```

Example Use

```
int main() {
  struct TIME startTime, stopTime, diff;

  printf("Enter start time: \n");
  printf("Enter hours, minutes and seconds respectively: ");
  scanf("%d %d %d", &startTime.hours, &startTime.mins,
            &startTime.seconds);

  printf("Enter stop time: \n");
  printf("Enter hours, minutes and seconds respectively: ");
  scanf("%d %d %d", &stopTime.hours, &stopTime.mins,
            &stopTime.seconds);

  differenceBetweenTimePeriod(startTime, stopTime, &diff);
  printf("\nTIME DIFFERENCE: %d:%d:%d - ", startTime.hours,
            startTime.mins, startTime.seconds);
  printf("%d:%d:%d ", stopTime.hours, stopTime.mins,
            stopTime.seconds);
  printf("= %d:%d:%d\n", diff.hours, diff.mins, diff.seconds);
  return 0;
}
```

OPERATING SYSTEMS

Genealogy of Modern Operating Systems



The Kernel

- Today, all "real" operating systems have protected kernels
  - The kernel resides in a well-known file
    - the "machine" automatically loads it into memory (*boots*) on power-on/reset
  - Our "kernel" is called the *executive* in some systems
- The kernel is (mostly) a library of service procedures shared by all user programs, but the *kernel is* ***protected***:
  - User code cannot access internal kernel data structures directly, and it can invoke the kernel only at well-defined entry points (*system calls*).
- Kernel code is like user code, but the *kernel is* ***privileged***:
  - The kernel has direct access to all hardware functions, and defines the entry points of handlers for *interrupts* and *exceptions* (*traps* and *faults*).

More on all of this later…

Terminology: Kernel vs. OS

- "OS" & "Kernel" - interchangeable in this course
- Compiled Linux kernel: ~5-10 MB
- Fully installed system - a few GB
  - Mostly user-level programs that get executed as processes
  - System utilities, graphical window system, shell, text editor, etc.

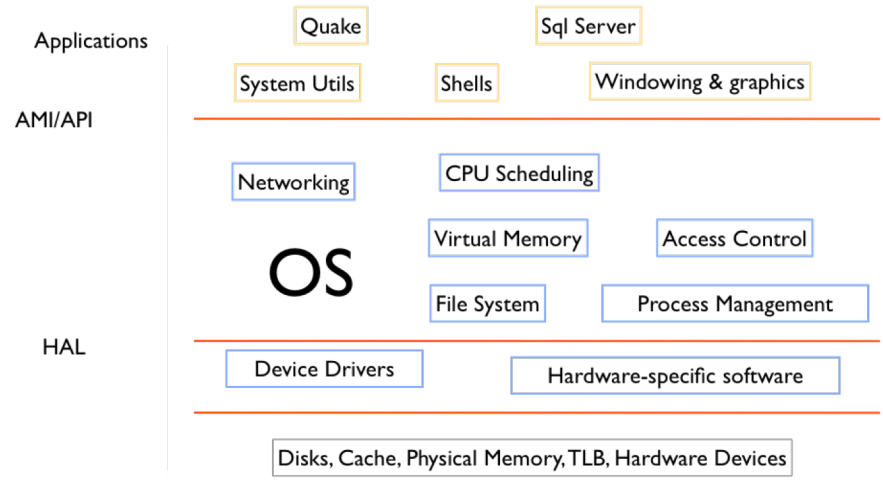
OPERATING SYSTEMS: DUAL MODE

Review: OS Interfaces

- Abstract Machine Interface (AMI)
  - OS's representation of the machine
  - between OS and apps: API + memory access model + legally executable instructions
- Application Programming Interface (API)
  - function calls provided to apps
- Hardware Abstraction Layer (HAL)
  - abstracts hardware *internally to the OS*

Why do we need these interfaces?

Logical OS Structure



If Applications Had Free Rein…

Buggy or malicious applications could:

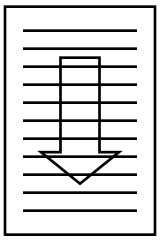
- crash other applications
- violate privacy of other applications
- hog all the resources
- change the OS
- crash the OS

We would be trusting every software developer!

PROCESS ABSTRACTION

Primary Abstraction: The Process

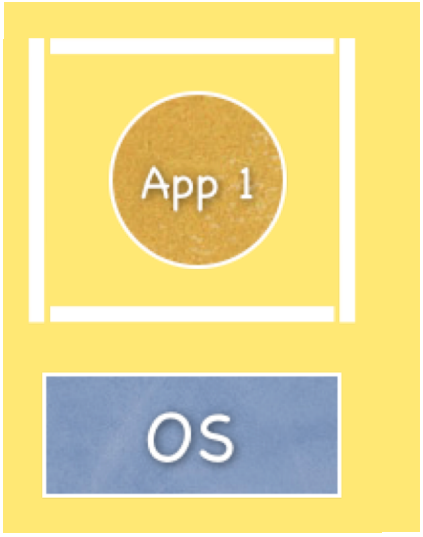
- Abstraction of a running program
  - a dynamic "program in execution"
  - Program = static file (image)
  - Process = executing program = program + execution state



- Program: blueprint
- Process: constructed building

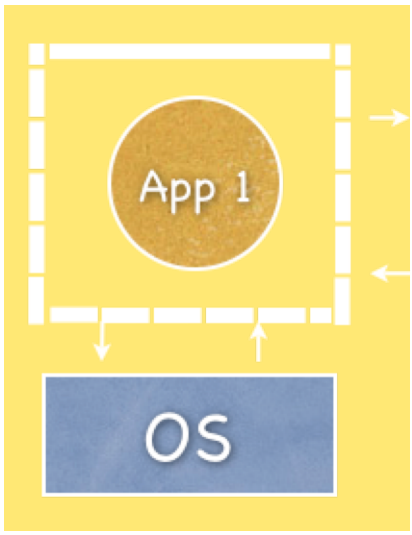
- Program: class
- Process: instance

The Process: Boxes in the Application



- An abstraction for protection
  - Represents an application program executing with ***restricted*** rights
- Restricting rights must not hinder functionality
  - Must still allow efficient use of hardware

The Process: Boxes in the Application



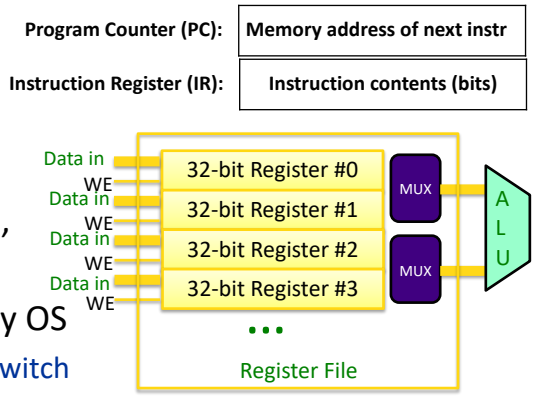
- An abstraction for protection
  - Represents an application program executing with restricted rights
- Restricting rights must not hinder functionality
  - Must still allow efficient use of hardware
  - Must still enable safe communication

What is a Process?

- A process is
  - A program during execution
  - The basic unit of execution in an OS
- Different processes may run different instances of the same program
  - e.g., my gcc and your gcc process both run the GNU C compiler
- At a minimum, process execution requires
  - Memory to contain the program code and data
  - A set of CPU registers to support execution

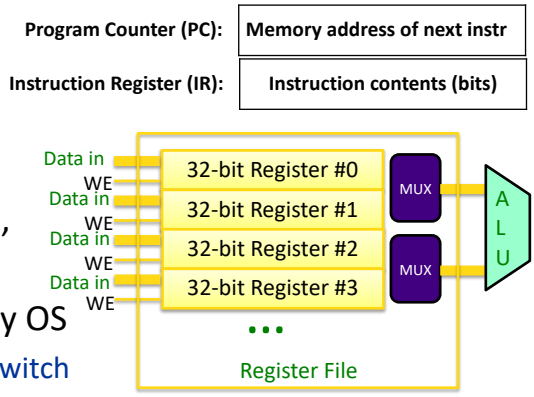
Process Resource: CPU Time

- CPU: Central Processing Unit
- PC points to next instruction
- CPU loads instruction, decodes it, executes it, stores result
- Process "given" CPU by OS
  - **Mechanism**: context switch
  - **Policy**: CPU scheduling



Process Resource: CPU Time

- CPU: Central Processing Unit
- PC points to next instruction
- CPU loads instruction, decodes it, executes it, stores result
- Process "given" CPU by OS
  - **Mechanism**: context switch
  - **Policy**: CPU scheduling



Required for process to execute and make progress!

Looking Ahead

- Assignment 2 – due Friday before class
- Dual Execution