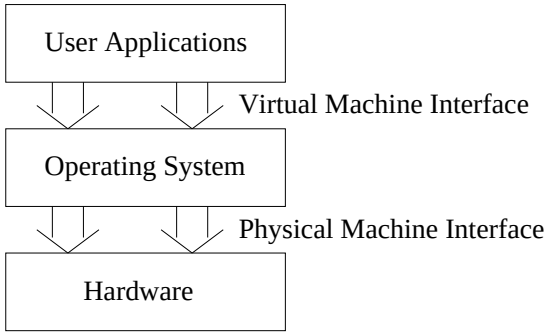
Introduction to Operating System — Last Time

Key concepts:

- Basic OS introduction
- History lesson in change

An operating system is the interface between the user and the architecture.



OS as juggler: providing the illusion of a dedicated machine with infinite memory and CPU.

OS as government: protecting users from each other, allocating resources efficiently and fairly, and providing secure and safe communication.

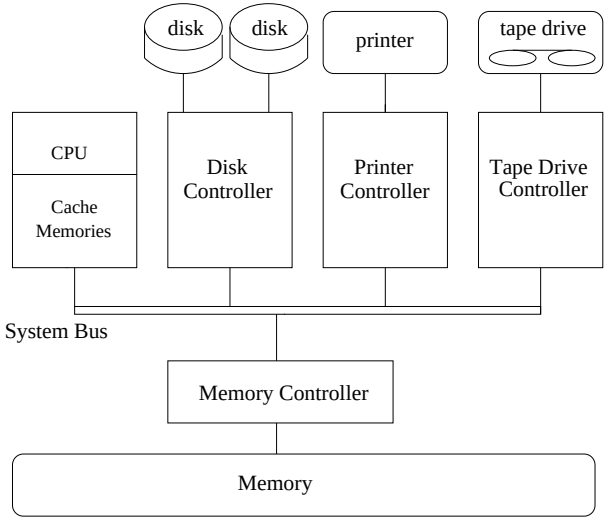
OS as complex system: keeping OS design and implementation as simple as possible is the key to getting the OS to work.

OS as history teacher: learning from past to predict the future, i.e., OS design tradeoffs change with technology.

Computer Architecture & Processes — Today

- Basic architecture reminder
- What the OS can do is dictated in part by the architecture.
- Course theme: architectural features can greatly simplify or complicate the OS.
- Process: unit of execution
  - How are processes represented in the OS?
  - What are possible execution states and how does the system move from one state to another?

Generic Computer Architecture



- CPU - the processor that performs the actual computation
- I/O devices - terminal, disks, video board, printer, etc.
- Memory - RAM containing data and programs used by the CPU
- System bus - the communication medium between the CPU, memory, and peripherals

From Architectural to OS to Application, and Back

Hardware	Example OS Services	User Abstraction
Processor	Process management, Scheduling, Traps, Protection, Billing Synchronization	Process
Memory	Management, Protection, Virtual memory	Address space
I/O devices	Concurrency with CPU, Interrupt handling	Terminal, Mouse, Printer, (System Calls)
File system	Management, Persistence	Files
Distributed systems	Network security Distributed file system	RPC system calls, Transparent file sharing

OS Service	Hardware Support
Protection	Kernel/User mode
	Protected Instructions
	Base and Limit Registers
Interrupts	Interrupt Vectors
System calls	Trap instructions and trap vectors
I/O	Interrupts or Memory-Mapping
Scheduling, error recovery, billing	Timer
Synchronization	Atomic instructions
Virtual memory	Translation look-aside buffers

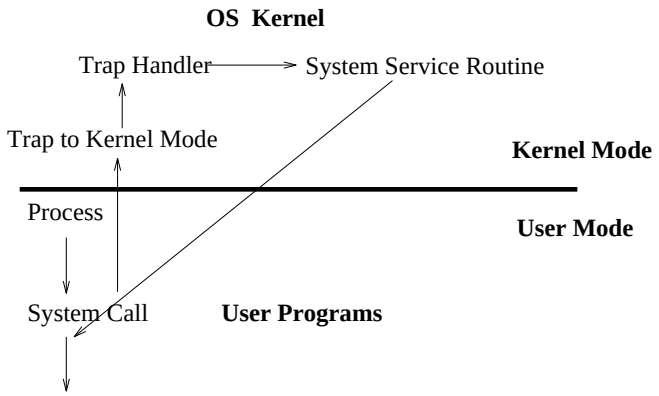
Interrupts - Moving from Kernel to User Mode

User processes may not:

- address I/O directly
- use instructions that manipulate OS memory (e.g., page tables)
- set the mode bits that determine user or kernel mode
- disable and enable interrupts
- halt the machine

but in kernel mode, the OS does all these things.

- A status bit in a protected processor register indicates the mode.
- Protected instructions can only be executed in kernel mode.
- On interrupts (e.g., time slice) or system calls



What's in a Process?

A **process** is the dynamic execution context of an executing program.

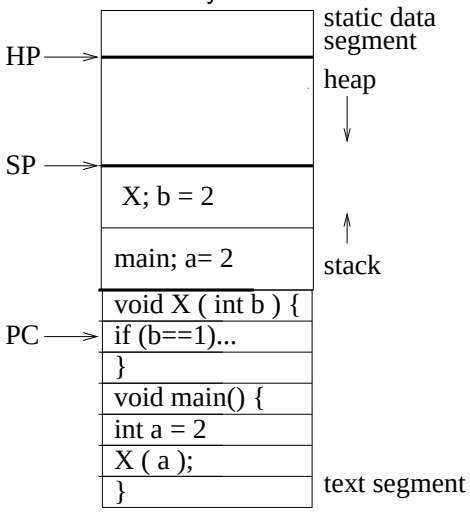
- Several processes may run the same program, but each is a distinct process with its own state (e.g., emacs).
- A sequential process is a program in execution that yields a sequential, one instruction at a time, ordering of the instructions.

Example Process State in Memory

What you wrote:

```
    void X ( int b ) {
PC→ if (b == 1) ...
    }
    main() {
      int a = 2;
      X ( a );
    }
```

What's in memory:



Process state consists of at least:

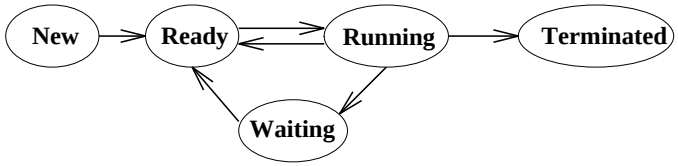
- the code for the running program,
- the Program Counter (PC), indicating the next instruction,
- an execution stack with the program's call chain (the stack), the stack pointer (SP)
- the static data for the running program,
- space for dynamic data (the heap), the heap pointer (HP),
- values of CPU registers
- a set of OS resources in use (e.g., open files)
- process execution state (ready, running, etc.).

Process Execution State

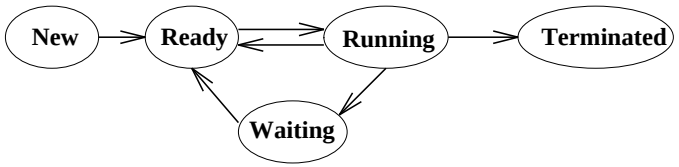
Each process has an *execution state* which indicates what it is currently doing.

new:	the OS is setting up the process state
running:	executing instructions on the CPU
ready:	ready to run, but waiting for the CPU
waiting:	waiting for an event to complete (e.g., I/O)
terminated:	the OS is destroying this process

- As the program executes, it moves from state to state, as a result of the program actions (e.g., system calls), OS actions (scheduling), and external actions (interrupts).



Process Execution State



Example:

```
void main() {
  printf("Hello World");
}
```

state sequence
new
ready
running
waiting for I/O
ready
running
terminated

Since multiple processes are active at any one time, the OS manages them by placing all the processes in the same state in a *state queue* (e.g., the *ready queue*). [More on this in a minute ....]

OS Process Data Structures

The OS uses a *Process Control Block* (PCB), its own dynamic data structure, to keep track of all the processes (this data structure is in addition to the process state in memory). The PCB represents the execution state and location of each process when it is not executing (e.g., when it is *waiting*).

The PCB contains:

- Process state (running, waiting, etc.)
- Process number
- Program Counter (PC)
- Stack Pointer (SP)
- General Purpose Registers
- Memory Management Information (HP, etc.)
- Username of owner
- List of open files
- Queue pointers for state queues (e.g., the *waiting queue*)
- Scheduling information (e.g., priority)
- I/O status
- ...

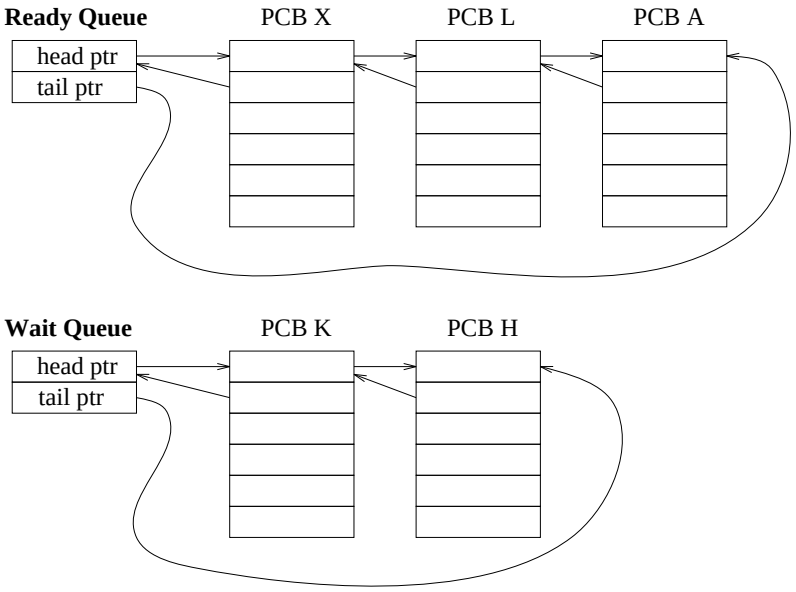
When a process is created, the OS allocates and initializes a new PCB, and then places the PCB on a state queue.

When a process terminates, the OS deallocates the PCB.

OS Process State Queues

The OS maintains the PCBs of all the processes in the system by using *state queues*.

- The OS places the PCBs of all the processes in the same *execution state* in the same queue.
- When the OS changes the state of a process, the PCB is unlinked from its current queue and moved to its new state queue.
- The OS can use different policies to manage each queue.
- Each I/O device has its own wait queue.



PCBs and Hardware State

Starting and stopping processes is called a **context switch**, and is a relatively expensive operation.

- When the OS is ready to start executing a waiting process, it loads the hardware registers (PC, SP, and other registers) from the values stored in that process' PCB.
- While a process is running, the CPU modifies the Program Counter (PC), Stack Pointer (SP), registers, etc.
- When the OS stops a process, it saves the current values of the registers, (PC, SP, etc.) into the PCB for that process.
- This process of switching the CPU from one process to another (stopping one and starting the next) is the context switch.
- Time sharing systems may do 100 to 1000 context switches a second. (The cost to do the context switch and the time between process switches are closely related.)

Creating a Process

- One process can create other processes to do work. These are *child* processes, and the creator is the *parent* process.
- In some systems, the parent defines (or donates) resources and privileges to its children.
- When a parent creates a child, it can either wait for the child to complete, or continue in parallel.
- In Unix, subprocesses are created by a call to the system routine *fork*
  - the variable values and program counter are copied from the parent to the new child process
  - the return value of *fork* is different in the parent and child process. In the parent process, the return value is the child process id. In the child process, the return value is 0. *This is the only difference between the parent and child processes*.
  - The parent can wait for the child to terminate by executing the *wait* system call to wait for the process identified by fork to terminate, or
  - The parent can continue to execute.
  - The child often starts a new and different program within itself, via a call to *exec*, another Unix System call.

Creating a Process

Example:

- When you log in to a machine running Unix, you create a shell process.
- Every command you type into the shell is a child of your shell process and is an implicit *fork* and *exec* pair.
- For example, you type emacs, the OS *"forks"* a new process and then *"exec"* (executes) emacs.
- If you type an & after the command, Unix will run the process in parallel with your shell, otherwise, your next shell command must wait until the first one completes.

Example Unix Program

fork() forks a new child process that is a copy of the parent.

execlp() replaces the program of the current process with the named program.

sleep() suspends execution for at least the specified time.

waitpid() waits for the named process to finish execution.

gets() reads a line from a file.

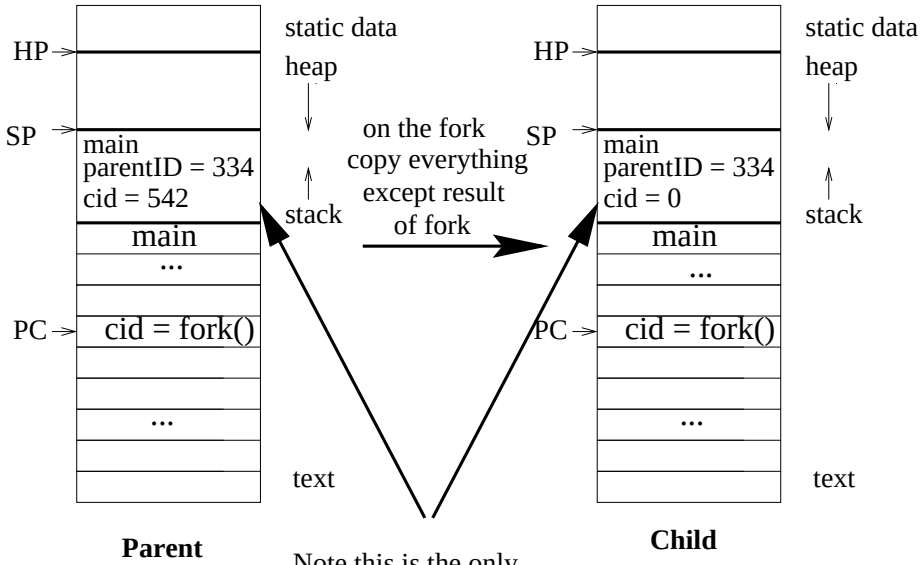
```
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
main() {
  int parentID = getpid();    /* ID of this process */
  char  prgname[1024];
  /* read the name of program we want to start */
  gets(prgname);

  int cid = fork();
  if(cid == 0) {  /* I'm the child process */
    execlp( prgname, prgname, 0);  /* Load the program */
    /* If the OS starts  prgname, we never get to this line,
       because the OS replaces the program running in the
       child process by prgname.   */
    printf("I didn't find program %s\n", prgname);

  } else { /* I'm the parent process */
    sleep (1);  /*  Give my child time to start. */

    /*  Wait for my child to terminate. */
    waitpid(cid, 0, 0);
    printf("Program %s finished\n", prgname);
} }
```


What is happening on the Fork



Note this is the only difference between the parent and the child at the time of the fork.

Process Termination

- On process termination, the OS reclaims all resources assigned to the process.
- On Unix
  - a process can terminate itself using the *exit* system call.
  - a process can terminate a child using the *kill* system call.

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
main() {
  int parentID = getpid();    /* ID of this process */
  int cid = fork();
  if(cid == 0) {  /*  I'm the child process  */
    /* I'll exit myself after 5 seconds. */
    sleep (5);
    printf ( "Quitting child\n" );
    exit (0);
    /* We should never get here. */
    printf ( "Ack!  After exit call.!");
  } else {  /*  I'm the parent process  */
    /* The user can kill the child by typing
       any character. */
    printf ( "Type any character to kill the child.\n" );
    char answer[10];
    gets (answer);
    if ( !kill(cid, SIGKILL) ) {
      printf("Killed the child.\n");
} } }
```


Summary

Processes:

- A process is the unit of execution.
- Processes are represented as Process Control Blocks in the OS that encapsulate program state, memory management information, scheduling information, I/O status, etc.
- At any time, a process is either New, Ready, Waiting, Running, or Terminated.
- On a uniprocessor, there is at most one Running process at a time.
- The program currently executing on the CPU is changed by performing a context switch

Next Time:

- Scheduling Processes on the CPU