


Comp 310
Computer Systems and Organization

Lecture #5

The Process and Communication

Prof. Joseph Vybihal



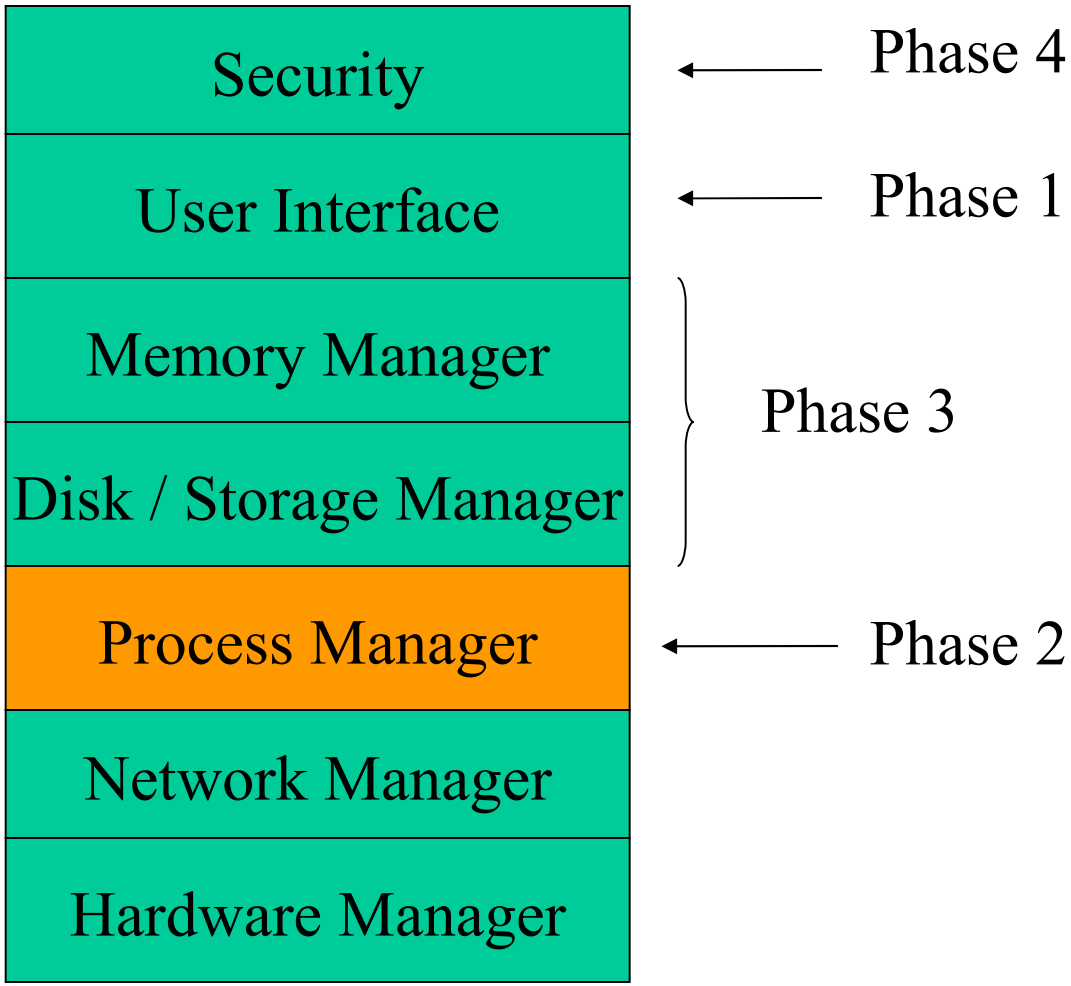

Announcements

- Assignment #1 out, did you see it?
- Unix tutorial Monday & Tuesday




Basic OS Architecture

(Course Table of Contents)



Process inter-comm.
Management
Scheduling
CPU Sharing



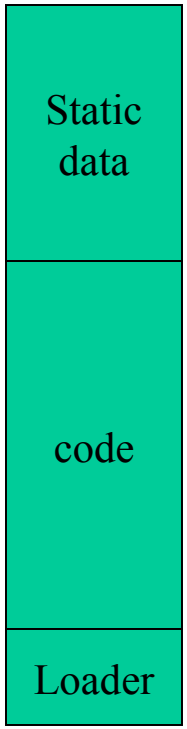

Part 1

The Process Concept

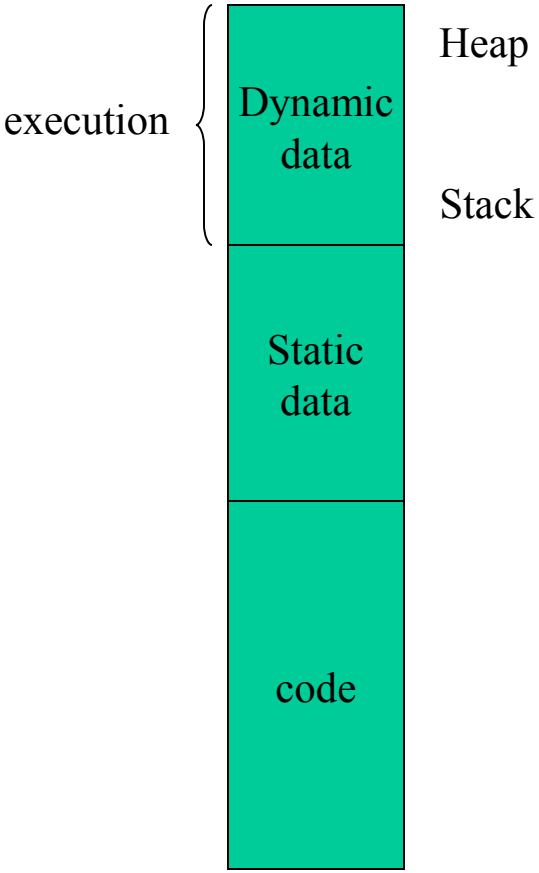



Process Vs Program

What is a loader?
Made from text or code?



Program
(HDD)

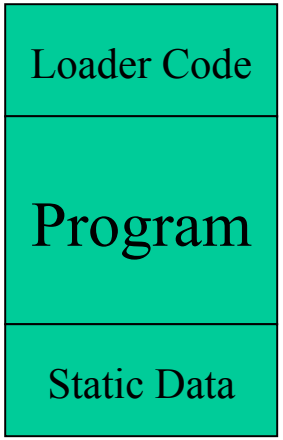


Process
(RAM)

Process Types

A Program

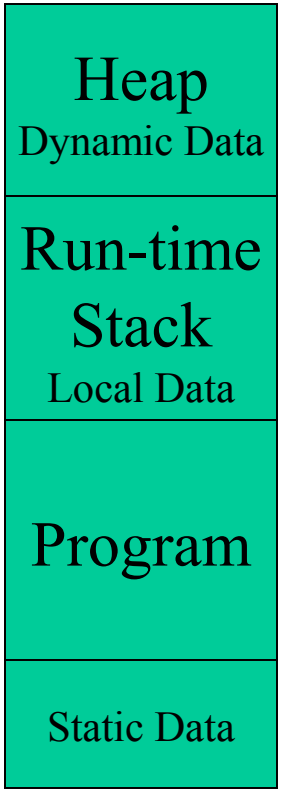
ON DISK



Passive

A Process

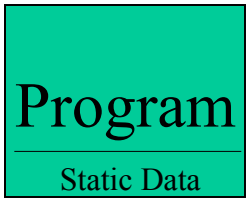
IN RAM



Dynamic

A Dynamic Link Library

DLL



- Loaded when called
- Managed by OS
- Shared with all P_i

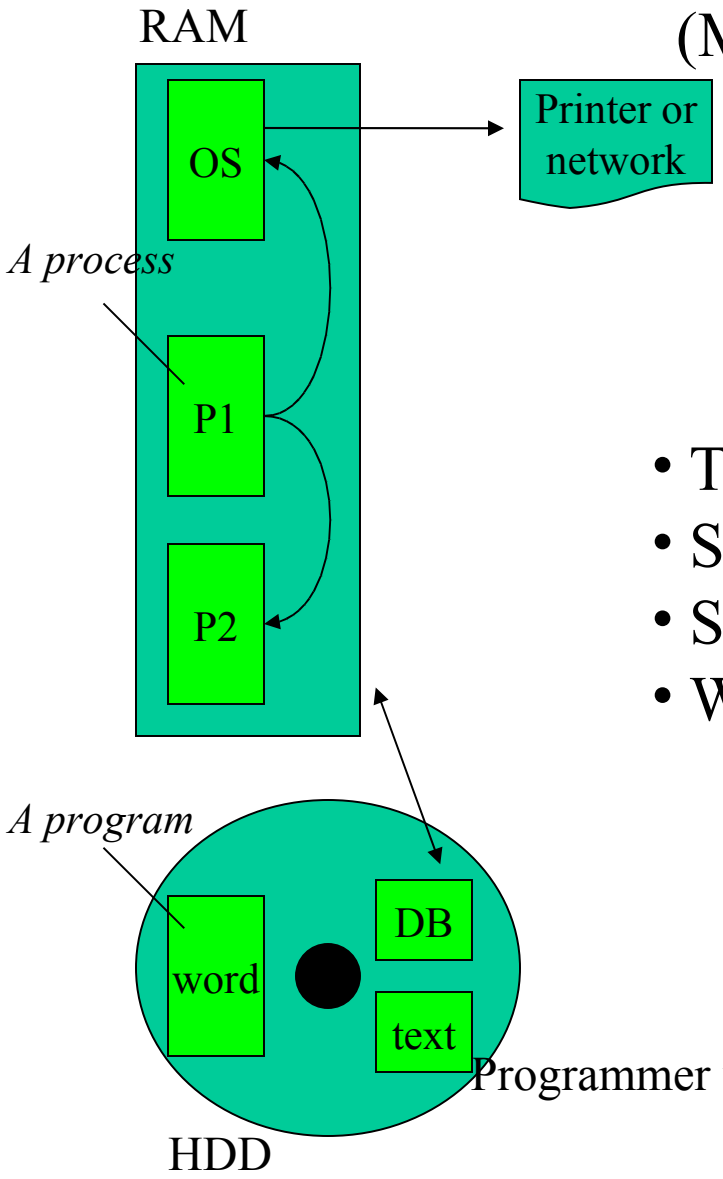
Shared Code

(same on disk and RAM)
- needs parent program for dynamic memory to be able to execute

Static Libraries?
OS Libraries?
OS Drivers?

Processes need to communicate

(Motivation)



- Take advantage of libraries
- Share devices
- Share data
- Work together on a common problem

Programmer version (how could we do that in C?)




Part 2

Process to OS Communication




Types of Process Communication

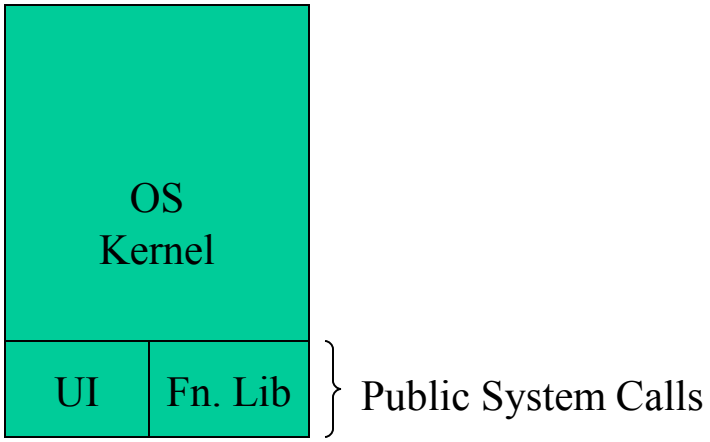
- No process communication
  - direct assembler, or
  - programming libraries
- Process using OS Services (e.g. malloc)
- Process using OS to connect to peripheral drivers
  - Printers
  - Network
- Process using OS to talk to other process
- Process using OS to share data




System Calls

Process to OS Communication

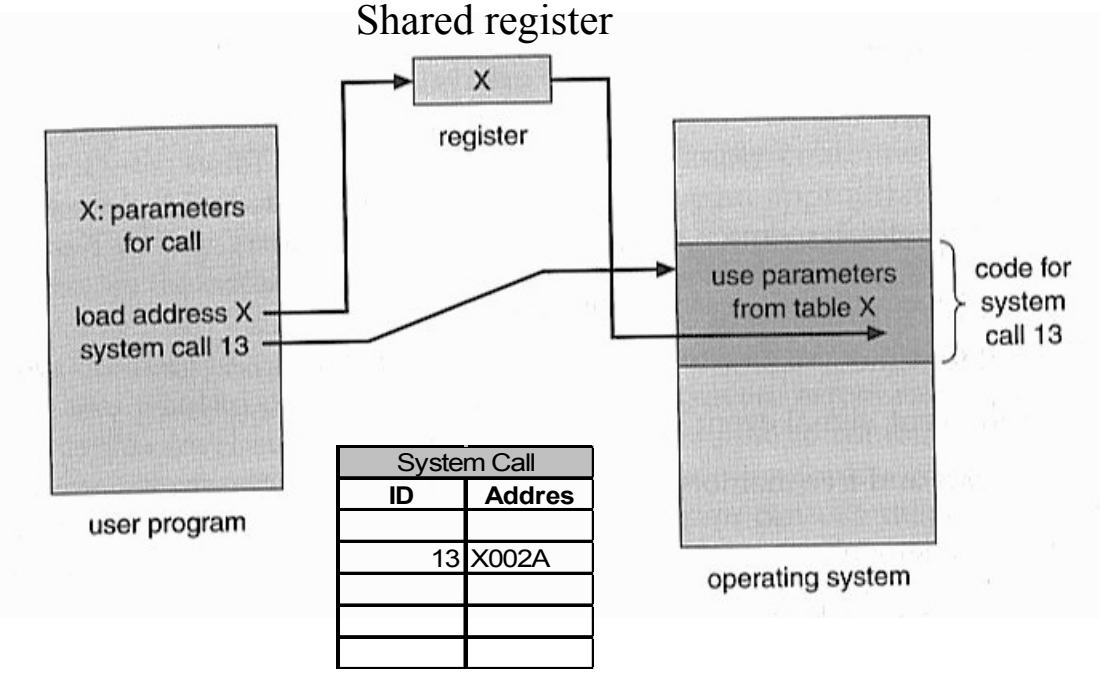
- Provide the interface between a process and the OS.
- C, C++ and Perl are examples of languages that have functions that trigger the assembler system calls.
- Window's Win32 API interface contain these systems calls available to all programming languages.



Accessible by:
- Command-line commands
- Function calls

System Calls

System Call Run-time Environment

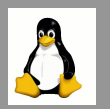


System Call Table

When the assembler command issues system call 13 the jump to address x where the code exists for system call 13 is contained in the System Call Table, created when the OS initially loads and is updated as the computer operates.

How do you build this in code?






Hidden System Calls

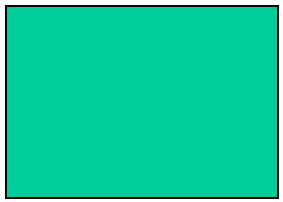
Language Library System Call Example

Executing Program

Process looses control of CPU

- assembler: syscall x
- get address for index x from sys table
- jump to OS routine
- put process to sleep? (optional)

printf("Hi");



System call x to draw on screen

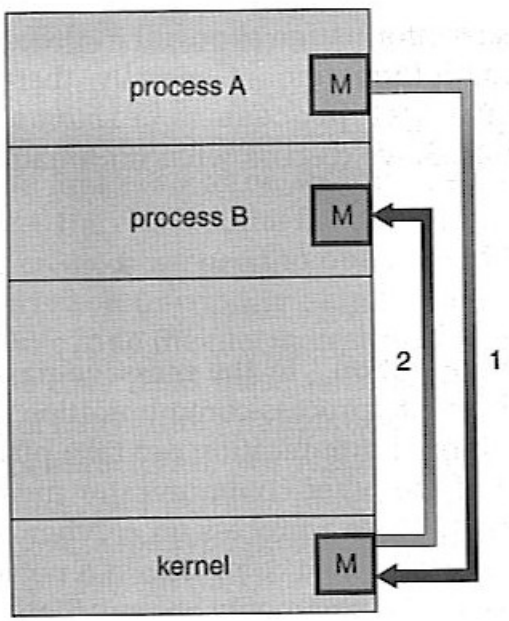
Control is returned to process

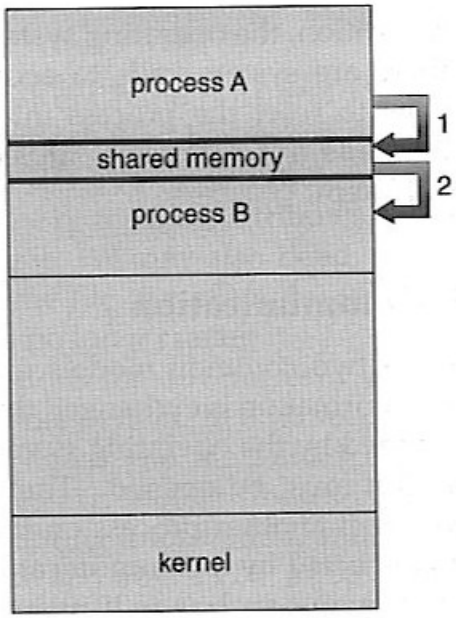
Executing OS

How do you build this in code?

Memory Sharing

Process to Process Communication





Message Passing

- Issue a system call
- Data and Process ID
- OS manages
- Function call and return
- Controlled

Shared Memory

- At process load, OS assigned share space
- Independent of OS
- Variable names to the same space
- Conflicts

How do you build this in code?




Unix Command-line Pipes

- Command-line: $ ls | more
- Implementation:
  - ASCII Streams
    - OS controlled streams
    - Stdin
    - Stdout
  - Buffers
    - Arrays
    - Temp files




Unix Programming Pipes

```
#include <stdio.h>    /* standard I/O routines.                */
#include <unistd.h>   /* defines pipe(), amongst other things. */

void do_child(int data_pipe[]) {/* this routine handles the work of the child process. */
   int c;      /* data received from the parent. */
   int rc;     /* return status of read().       */

   /* first, close the un-needed write-part of the pipe. */
   close(data_pipe[1]);

   /* now enter a loop of reading data from the pipe, and printing it */
   while ((rc = read(data_pipe[0], &c, 1)) > 0) {
            putchar(c);
   }

   /* probably pipe was broken, or got EOF via the pipe. */
   exit(0);
}
```




```
void do_parent(int data_pipe[]) {/* this routine handles the work of the parent process. */
   int c;       /* data received from the user. */
   int rc;      /* return status of getchar().  */

   /* first, close the un-needed read-part of the pipe. */
   close(data_pipe[0]);

   /* now enter a loop of read user input, and writing it to the pipe. */
   while ((c = getchar()) > 0) {

                /* write the character to the pipe. */
                 rc = write(data_pipe[1], &c, 1);
                if (rc == -1) { /* write failed - notify the user and exit */
                   perror("Parent: write");
                   close(data_pipe[1]);
                   exit(1);
                }
   }

   /* probably got EOF from the user. */
   close(data_pipe[1]); /* close the pipe, to let the child know we're done. */
   exit(0);
}
```



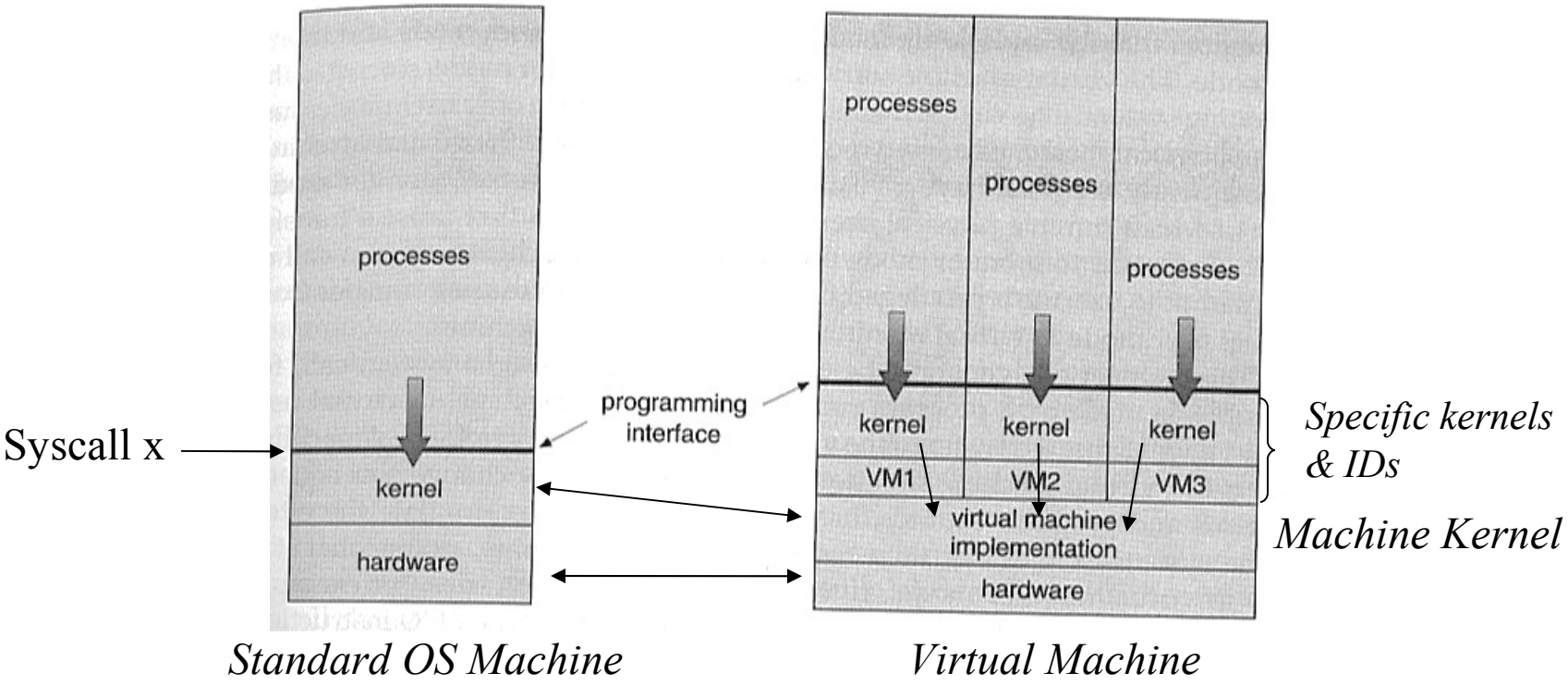

```
int main(int argc, char* argv[]) {/* and the main function. */
    int data_pipe[2]; /* an array to store the file descriptors of the pipe. */
    int pid;       /* pid of child process, or 0, as returned via fork.    */
    int rc;        /* stores return values of various routines.           */

    /* first, create a pipe. */
    rc = pipe(data_pipe);
    if (rc == -1) {
                perror("pipe");
                exit(1);
    }

    /* now fork off a child process, and set their handling routines. */
    pid = fork();

    switch (pid) {
                case -1:        /* fork failed. */
                   perror("fork");
                   exit(1);
                case 0:                         /* inside child process.  */
                   do_child(data_pipe);
                   /* NOT REACHED */
                default:        /* inside parent process. */
                   do_parent(data_pipe);
                   /* NOT REACHED */
    }

    return 0;   /* NOT REACHED */
}
```

How is data_pipe implemented?
What is fork?




Virtual Machines

(Communication between machines) Pseudo or real...



- Bottle-necks: File Systems, CPU (any shared resource)
- Benefits: Can run favorite applications
- The OS kernels are also now processes scheduled by the Machine Kernel

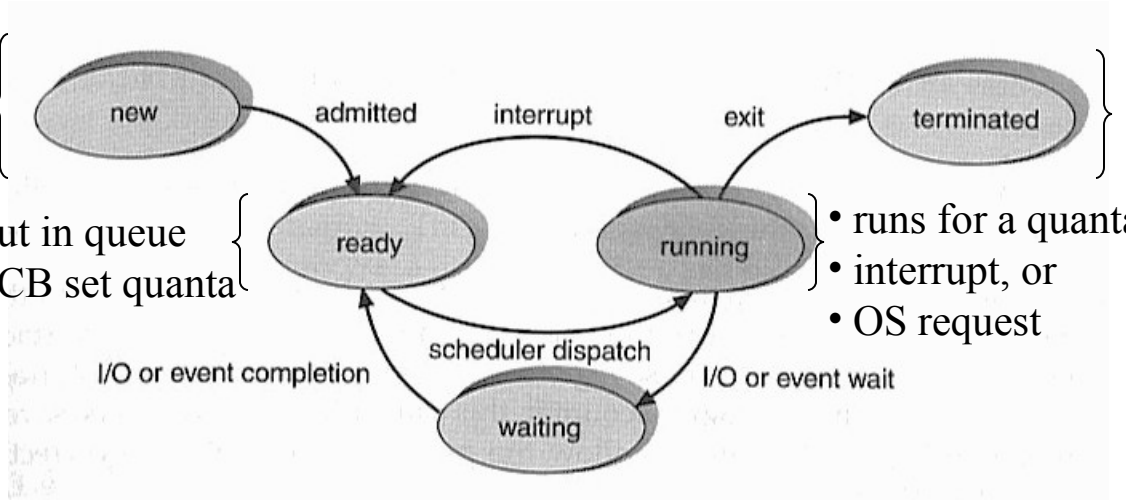



Part 3

Process Management

Process State

- Load from disk
- Create process
- Create PCB



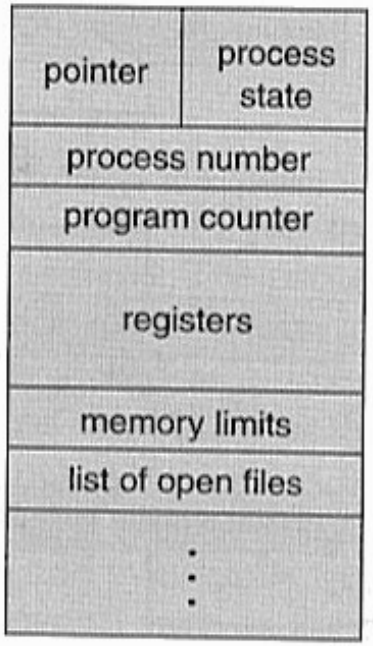
- Put in queue
- PCB set quanta

- runs for a quanta, or
- interrupt, or
- OS request

- Free RAM
- Delete PCB
- Free buffers

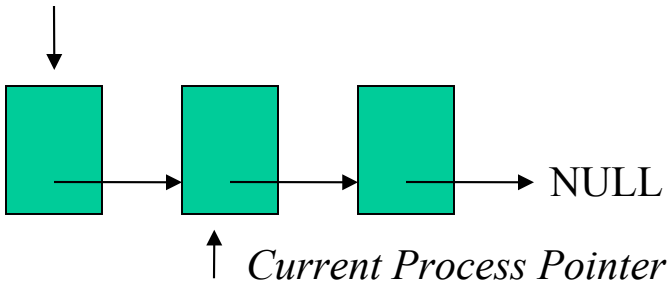
- Ready: is the run-time process queue. Each link is a PCB
- Waiting: is the sleep list. Not really a queue. PCBs on this list wait for their OS request to end.

The Process Control Block
(PCB)

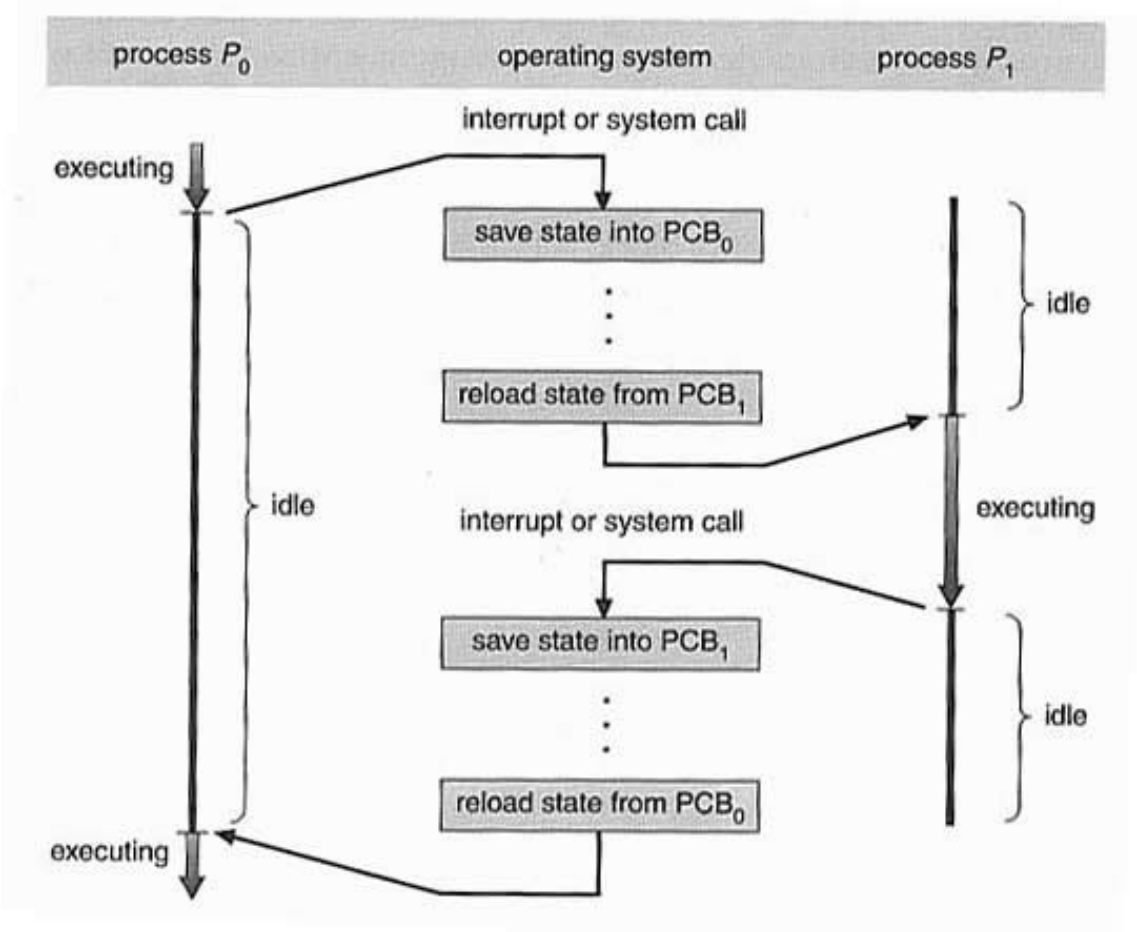


- The process is known by its ID number
- A pointer points to the beginning of the process
- An integer counter indicates which byte is currently being executed
- A space is reserved to store the registers for the process switch

Process Run-Time Queue (linked list or circular array)



Process Execution Simulation



"The context switch"




Threads

- By definition: Execution path through code
- What if…
  - Two users share the same MSWORD application at the same time but with different documents - how many threads? How many processes?
    - 2 threads!
  - What properties do threads then have?
    - Represent a chain of execution, regardless of code source
    - Each thread has its own PCB on the process run-time queue (even if PCB pointer points to the same code source)
- Other threads:
  - Spawn your own
  - Event Processing

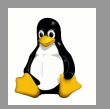
Discuss...




Part 4

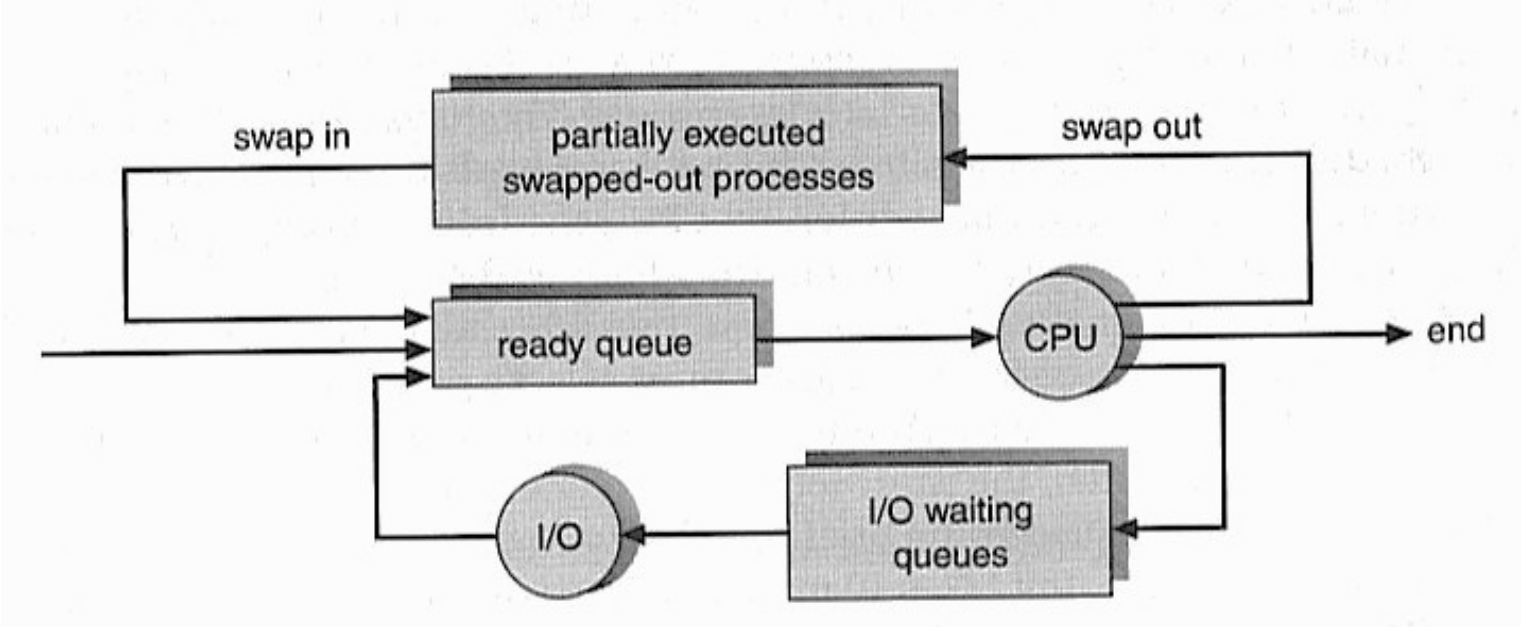
Process Scheduling





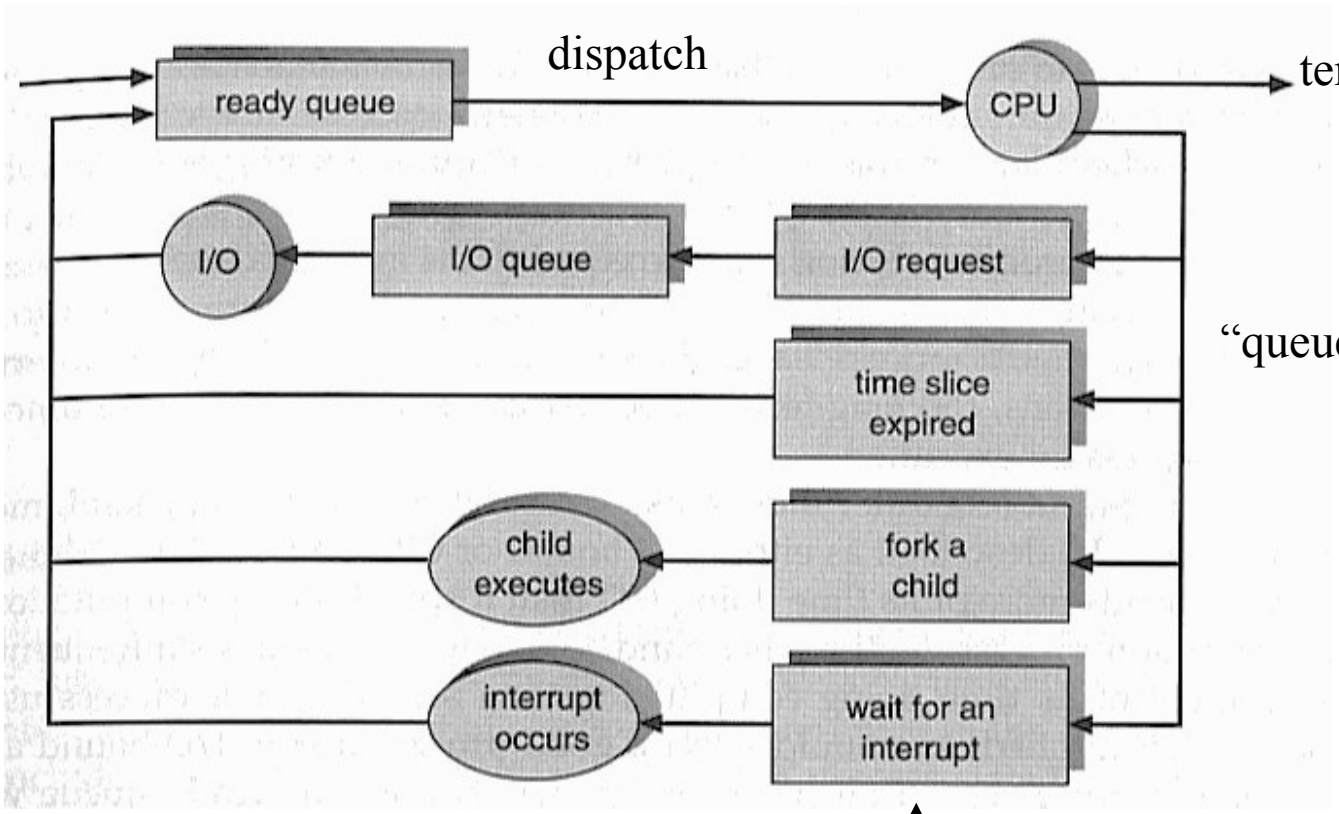

Deeper Schedules

(Mid-level Scheduler)



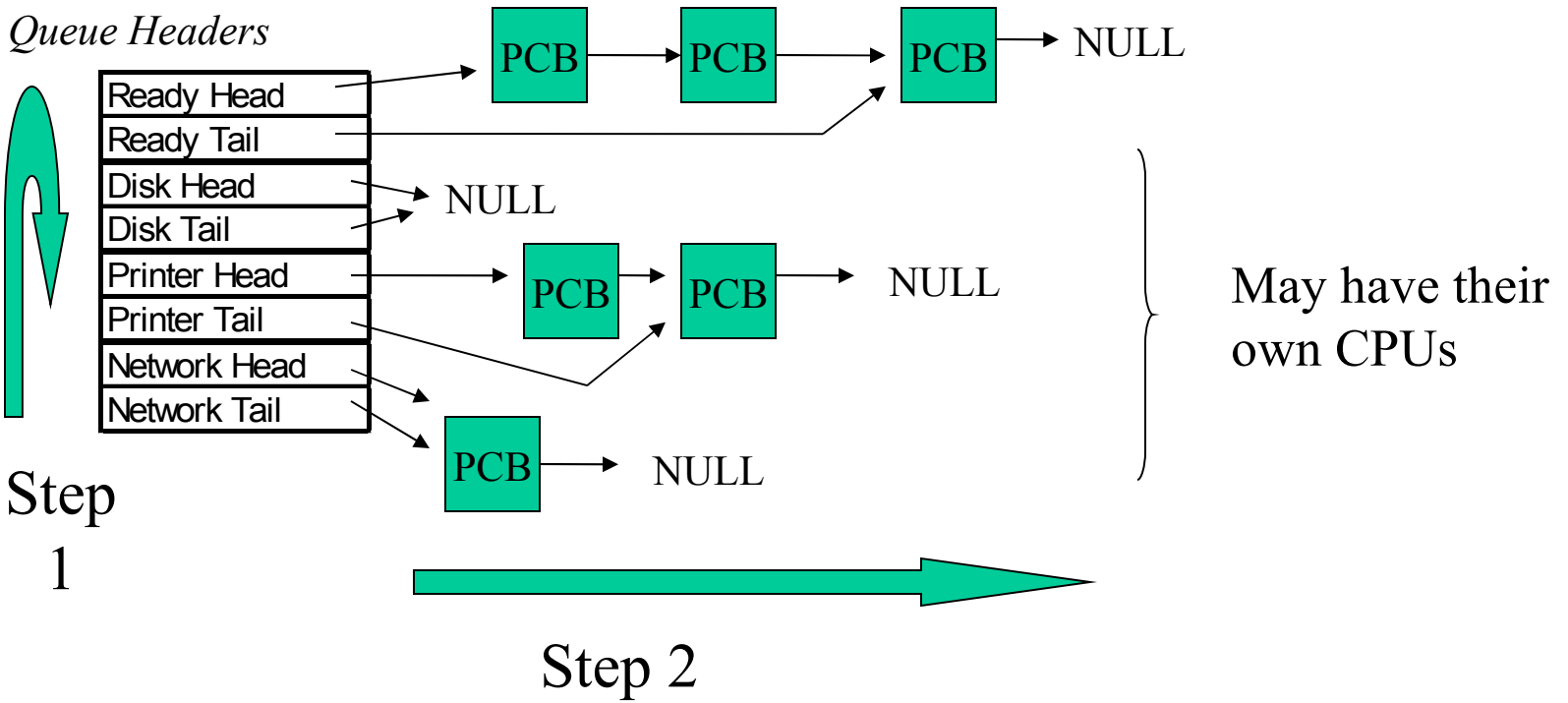
Did not have time to execute… Or system load too heavy…
Therefore, swap out of ready queue, put on Overload Queue.

Process Scheduling



Pseudo-code and assembler discussion...

Multiple OS Run-Time Queues



This is a double nested loop:

- For each queue
- Execute next PCB




Part 5

At Home




Things to try out

1. If we did not have system calls, what would be an alternative way of providing the services that system calls provide?
2. Find out how the printer queue on your home computer or laptop functions.