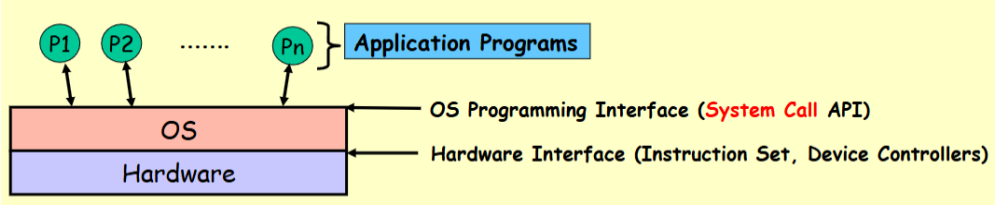
Topic 1 - Intro to Operating Systems

What is an Operating System?

- A **software layer** to abstract away and manage details of hardware resources
- A **virtual machine** that is easier to program than the raw hardware
- A **resource allocator** (manages and allocates resources such as CPU, memory, disk…)
    - Kernel: the one program running **at all times**



- Moderates the access programs have to hardware resources (acts as a middle man) such as:
    - Computation (CPU)
    - Volatile Storage (memory or RAM)
    - Persistent Storage (Disks)
    - I/O Devices
    - Network Communications (TCP/IP stack, Ethernet cards, etc.)
- Application benefits?
    - Programming Simplicity
        - Programs see high-level abstractions (files/directories) instead of low-level hardware details
    - Portability
        - I/O device independence
- User benefits?
    - Efficiency (cost and speed)
        - share one computer across many users and process
    - Safety
        - protects process from each other
    - Security
        - secure against outsiders
- Major Components:
    - Processes Manager
    - Memory Manager
    - I/O Manager
    - Protection & Security
    - Accounting
    - Shells (command line interpreter - CLI)

Single-Tasking Systems (MS-DOS)

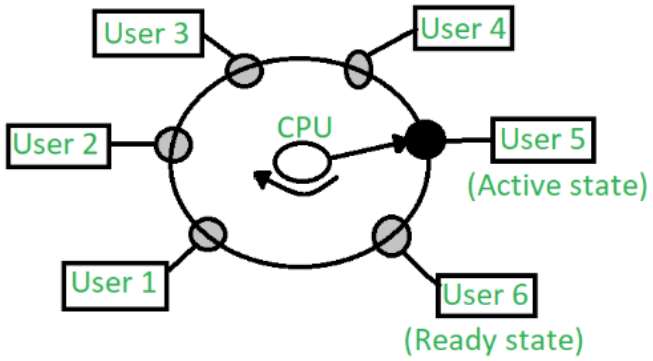
- Prior implementation
- OS was stored in a portion of the primary memory
- The OS has a Command Line Interface (CLI) that users use to load the next program into memory from the disk
- Problem: **CPU is idle** when a program interacts with a peripheral (I/O) during execution

Multi-Tasking Systems

- Issue with Single-tasking Systems was that the CPU was idle when a program did I/O
    - Then came multi-tasking systems
- Keeps multiple runnable jobs (processes) loaded in memory at once
- While one job waits for I/O completion, the OS runs instructions from another job
    - Need some way to know when I/O is complete via **interrupts**
- Goal: optimize system throughput

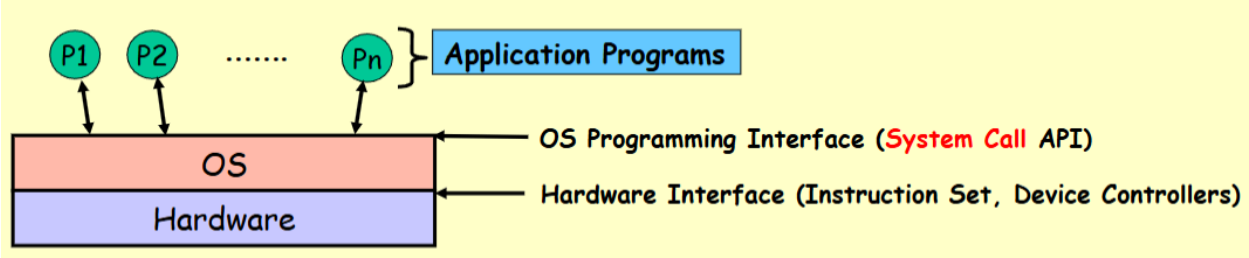
Timesharing

- A timesharing OS is for interactive use
- Multiple terminals into one machine
- Each user has the illusion of the entire machine to him/herself



- Timeslicing
    - Divide CPU equally among the users
    - Permits users to interactively view, edit, and debug running programs

OS API



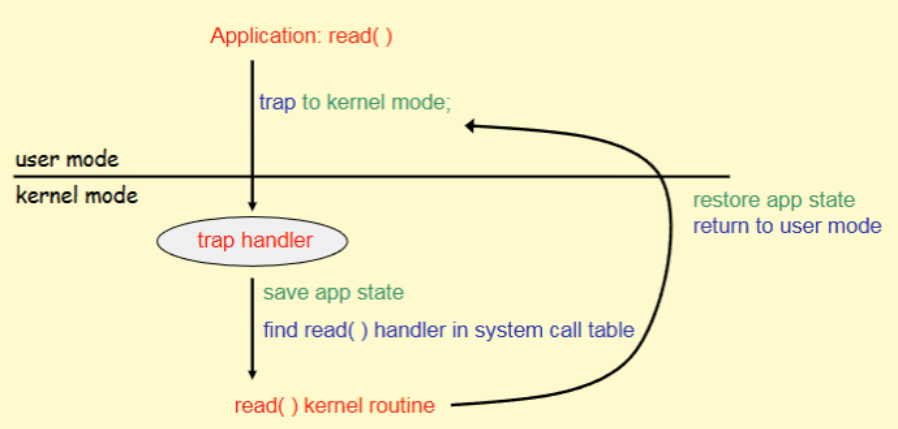
- OS exports a system call API to user programs to provide services from the OS
- The system call API consists of several functions that the user can call to ask for certain services from the OS. Below are some examples:
    - Create a process, terminate a process
    - open/read/write/close
    - send/receive a network packet
- Each OS defines its own system call API

Invoking System Calls

- In OSs designed for general purpose MPUs, we want to protect processes from each other
    - We do **not** want a user application to be able to execute **all** instructions in the instruction set
- To safely invoke these system calls, we use **dual mode operation**

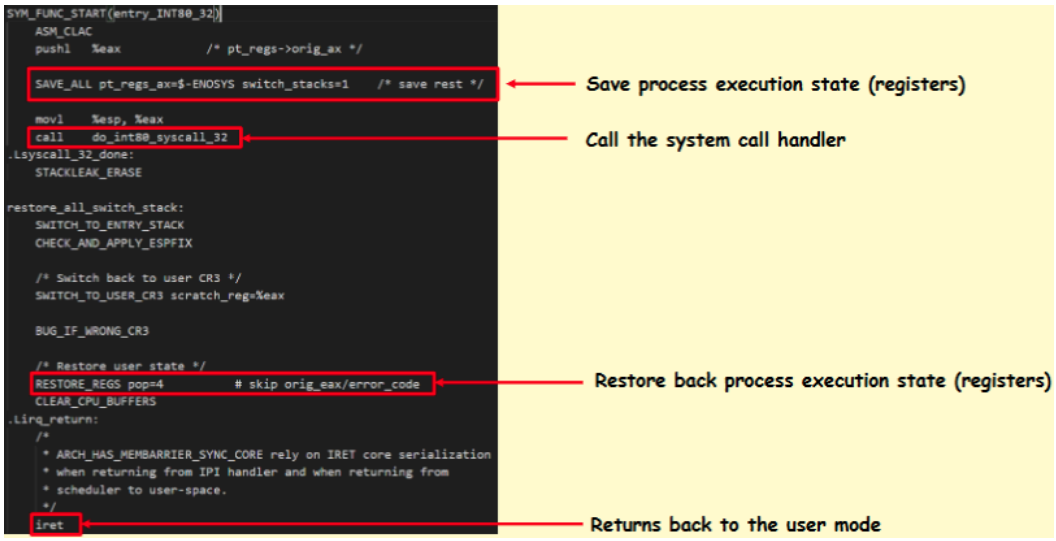
Dual Mode Operation

- **Kernel**/System mode: CPU executing **OS code**
    - CPU can execute **every** instruction in the instruction set
- **User** mode: CPU executing **user program code**
    - CPU can execute only a subset of the allowed instructions
    - Instructions involving **I/O** and **memory protection** are disallowed
- You “trap” (switch) to kernel mode using a **trap instruction**, and the “trap handler”



Trapping to the Kernel

- The **trap instruction** must automatically save instruction & flag registers, and change the CPU execution mode from “user mode” to “kernel mode”
    - Older versions of 32-bit Linux systems use int 0x80/iret instruction pair to enter/exit kernel mode
    - Modern 32-bit Linux uses sysenter/sysexit, which is more efficient
    - Modern 64-bit uses syscall/sysret
- The **trap handler** is the main entry point for ALL system calls. Upon entry, the handler must:
    - Save the current process execution state (registers) to the stack
    - Call the function within the kernel that handles the system call, which must verify the system call arguments
- To return, the trap handler must:
    - Restore the current process execution state (registers) from the stack
    - Return back to the user mode (iret/sysexit/sysret)



Executing the System Call

- The kernel assigns a unique number to each system call
- This system call number is passed to the kernel from the application
    - Usually through accumulator register (eax/rax)
- The trap handler uses unique number to invoke the function that handles the system call
- The address of the actual system call handler functions are stored in an array of function pointers called “sys_call_table”
    - **sys_call_table**[3] = sys_read() ; **sys_call_table**[4] = sys_write() …
- Summary using int 0x80:
    - User runs a library function, and this **library function** assigns the system call number using eax and traps to the kernel using an interrupt (int 0x80)

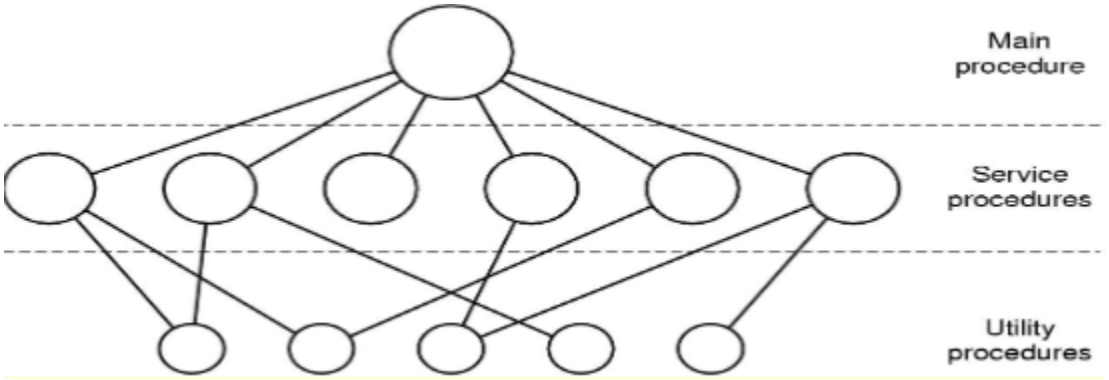
    - Calling int 0x80 matches the interrupt to a system_call function using the Interrupt Vector Table (**IVT**)
    - The function stored in the **IVT** saves the application states, calls the function in the sys_call_table, restores the app state, and returns to user mode (iret)
- Summary using syscall (64-bit Linux)
    - User runs a library function, system call number and parameters are passed to registers, and traps to kernel using an interrupt (**syscall**)
    - Immediately saves application state, calls function using sys_call_table, restores application state, and returns to user mode (sysret)
    - Using **syscall**/**sysret** is **more efficient** than int 0x80 because
        - Saving one level of indirection by not using **IVT**, and we only save/restore the necessary app state to the stack

Kernel Stack

- When we switch to the kernel mode and start executing kernel code, we must switch to a **kernel stack** allocated for the process by the **kernel**
    - Kernel must run on a trusted stack stored in the kernel address space
- During a trap to the kernel, the trap instruction automatically switches the stack pointer to point to the kernel stack of the process
- When we return from the system call, we switch back to the user stack

Monolithic (Macro) Kernels

- Traditionally, OS's were built as a monolithic (macro) kernel
- All system calls TRAP to a main procedure
- The main procedure looks at the system call number and invokes the appropriate service procedure
    - Service procedure may use one or more utility procedures to do its job
    - Utility procedures may be shared by different system call service procedures

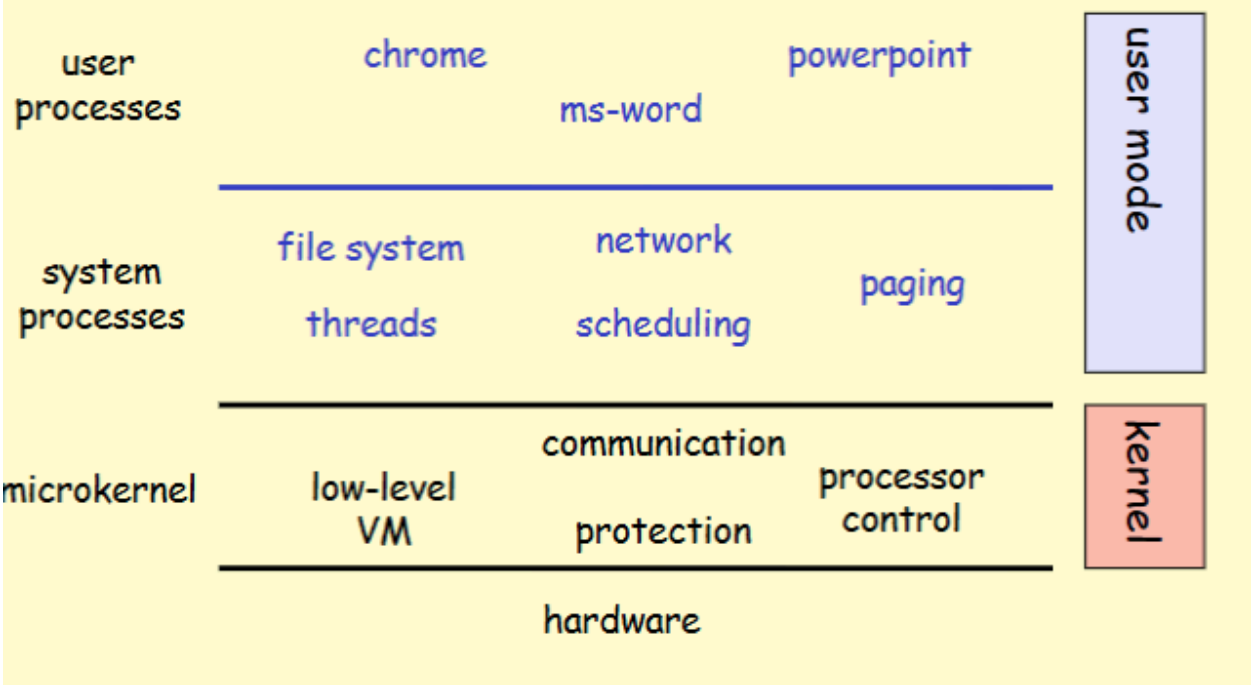


- **Advantage**:
    - The cost of module interactions is low (procedure call)
- **Disadvantages**:
    - Hard to understand

- Hard to modify
- Unreliable
- Hard to maintain

Microkernels

- Monolithic kernels are fast but hard to maintain, extend, and debug. Enter Microkernels
- The goal is to minimize what goes in the kernel and organize the rest of the OS as user-level processes
- This results in better reliability (isolation between components) and ease of extension and customization!
- Kernel's job is to handle communication between client & server processes
- Very small kernel BUT leads to poor performance



Practice Problems:

1. Short Answer Questions

a. List 2 benefits of having an OS.
You do not have to interact directly with the hardware, the OS does that for you. It's also very secure against outsiders

b. What are the major components of an operating system? Briefly explain their responsibilities.
The major components of an operating system are the process manager, memory manager, and I/O manager.

The process manager manages the creation, scheduling, and termination of processes.
The memory manager manages the allocation and deallocation of memory.
The I/O manager manages input devices such as the keyboard, mouse, disks, etc.

c. Briefly explain what multitasking is and what the main motivation for it is. Is Linux a multitasking OS? Why or why not?
Multitasking is allowing the CPU to continue executing a process while another process waits for I/O. This makes it so that the CPU is not idle while a process waits for I/O, optimizing throughput! Linux **is** a multitasking OS

d. Briefly explain when you will use a timesharing OS? Is Windows a time-sharing OS?

You want to use a time-sharing OS when you have multiple users using one centralized system but want to give the illusion that they all have their own independent machine. An example of this would be employees at a company each using remote desktop connections that are connected to a centralized device that all employees are using at the same time. Windows is an example of a time-sharing OS, as it supports multitasking and allows multiple users to share system resources.

e. Define the essential properties of the following 2 types of operating systems: batch, timesharing.
Batch: Execute jobs in batches without human interaction during execution
Timesharing: Timesharing Operating Systems allow users to access system resources at the same time through rapid context switching

f. Briefly define the following terms:

(1) Multi-tasking: allow the CPU to execute processes while another process is waiting on I/O to optimize throughput
(2) Timesharing: allow multiple users to use one centralized device as if they are the only ones using it

Does every time-sharing OS have to be multitasking?
Yes! To be a timesharing OS, you need to execute multiple processes for multiple users at once, and to do that, you need to be a multitasking system so the CPU won't idle any time a process waits on I/O and the CPU can switch between processes rapidly.

g. If your workload consists only of batch jobs, i.e., jobs that work in the background requiring no user interaction, would you use a timesharing OS? Why or why not? Briefly explain.

h. Consider a system where I/O devices can only be programmed (controlled) with polling. Would a multi-tasking OS be able to increase CPU utilization in this system? Justify your answer.

A multi-tasking OS would not be able to increase CPU utilization in this system, mainly because the CPU would be busy polling the I/O devices and waiting for a response. In this case, it would not be able to continue another process while another waits for I/O.

i. Explain why it is necessary to re-implement your Linux program under Windows (for most nontrivial code).

j. Why is it necessary to have two modes of operation, e.g., user and kernel mode, in modern OSs? Briefly explain the difference between the user mode and kernel mode.

This is important because we need to be able to protect the system from malicious users! We do not want to give users the ability to run any system call anytime they want! That is why we have user mode and kernel mode, where, while in user mode, only certain permitted system calls can be made. In kernel mode, all system calls are accessible and can be called, which makes it so that the user must first securely trap to the kernel to access these system calls!

k. How does the OS change from the user mode to the kernel mode and vice versa?

The OS changes from user mode to kernel mode when it gets a certain system interrupt (int 0x80/syscall). Once made, a bit is flipped that signifies that the system is now in kernel mode, allowing the execution of system calls. Then, the trap handler saves the application state and calls the system call function on behalf of the user. Once completed, the trap handler restores the application state and flips the bit, returning to user mode using iret/sysret!

l. Explain why it is not possible to use "call/ret" pair of instructions to call and return from an OS system call function in modern general-purpose OSs?

m. Which of the following instructions should be privileged, i.e., not allowed in user mode?
(1) Read the system clock,
(2) turn off interrupts,
(3) switch from user to kernel mode.
Briefly justify your answer.

Reading the system clock should be allowed in user mode! There aren't any harmful side effects from reading the system clock. But allowing users to switch from user to kernel mode whenever they want would be very harmful, as they would be able to execute any system call they like, and that's why they have to use specific libraries to access system calls. Also, turning off interrupts SHOULD be privileged. Turning off interrupts would be very harmful for the system, as interrupts are vital to many of the operations/properties of the system.

n. What's the difference between a trap and an interrupt?
A trap involves receiving the specific **interrupt** (0x80, syscall) to switch to kernel mode, while an interrupt in general can be anything from the Interrupt Vector Table (IVT) that all don't necessarily involve trapping to the kernel. Many involve exceptions, I/O, and more!

o. Describe 2 methods by which arguments are passed from a user process to the kernel in system calls. How does Linux pass arguments to the kernel during a system call?

One way arguments are passed from a user process to the kernel in system calls is using registers such as ebx, ecx, edx, etc. so that the values can be accessed while in the kernel. Another way is by passing them to the kernel stack. The method Linux uses is registers, but it only has 6 registers it can use for this. If there are over 6 parameters, Linux puts the first 6 in registers and the rest on the stack.

p. List the names of 5 system calls in Linux.
sys_read
sys_write
sys_open
sys_close
sys_exit

q. Briefly explain why it is necessary to switch to a kernel stack allocated for the process during the execution of a system call?.
The kernel must use a "trusted stack" stored in the kernel address space in order to complete the execution of privileged system calls.

r. List 1 advantage and 1 disadvantage of monolithic kernels.
One advantage is that the cost of module interactions is low. A disadvantage is that they are large and hard to maintain/change.

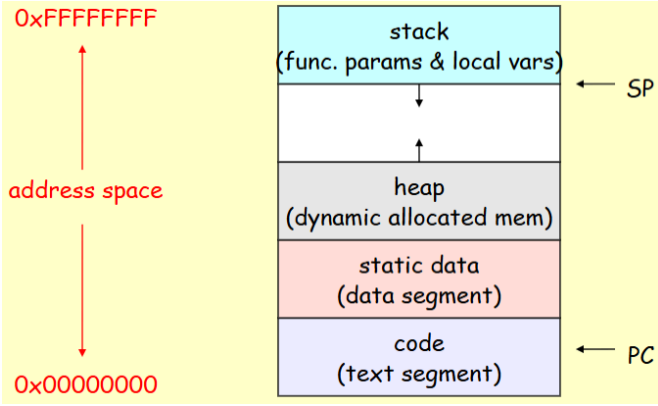
s. List 1 advantage and 1 disadvantage of microkernels.
One advantage is that it is easier to maintain/change, but one disadvantage is that it has a small kernel, which leads to poor performance

Topic 2 - Process Management

Program vs. Process

- Program: code stored in memory on hardware (passive instructions & static data stored on disk)
- Process: a program in execution

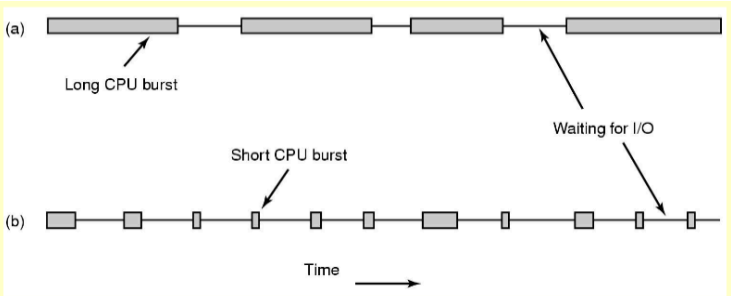
Process



- The process address space has four main parts
  - Stack (function params & local variables)
  - Heap (Dynamically allocated memory)
  - Static Data (data segment)
  - Code (text segment)
- A process consists of:
  - Code for the running program
  - Data for the running program (static/dynamic)
  - An execution stack
    - Traces state of function calls made (function parameters & local variables)
  - Program Counter (PC): holds the address of the next instruction to run
  - General-purpose processor registers and their values
  - A set of OS resources
- The process is a container for all of this state - named by a unique Process ID (PID)
- Process is represented in the kernel by an OS Data Structure called the **Process Control Block** (PCB), which contains many fields:
  - PID
  - Hardware state (PC, Stack Pointer, general purpose register values)
  - Process state (Ready, Running, Waiting)
  - Memory management info
  - Pointer to kernel stack used by the process
  - Scheduling information
  - Accounting information
  - Descriptors to open files, I/O devices, network sockets
  - Pointers into state queues

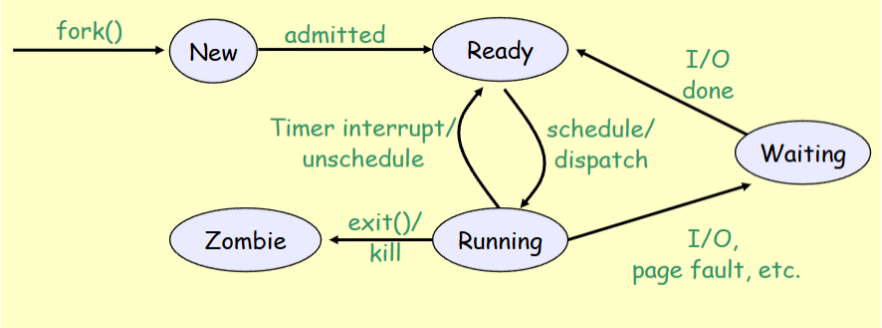
Process Behavior & States

- All processes alternate between bursts of computing with I/O requests

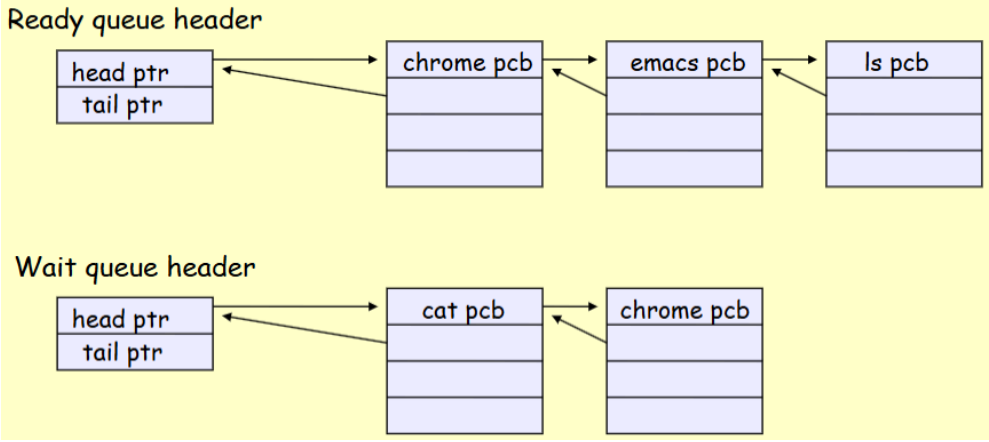


(a) a CPU bound process
(b) an I/O bound process

- Each process has an execution state, which indicates:
  - What it is currently doing
  - Ready: waiting to be assigned to CPU
    - It could run, but another process is using the CPU
  - Running: executing on the CPU
    - Currently controls the CPU
  - Waiting: waiting for an event, e.g. I/O
    - Cannot progress until the event occurs



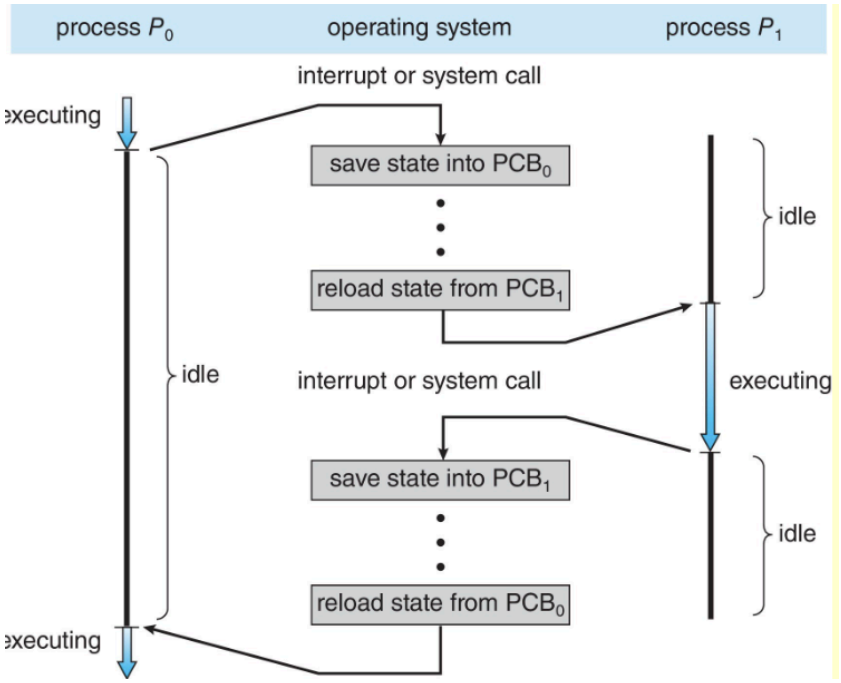
- The OS maintains a collection of **queues** that represent the state of all processes in the system
  - Typically one queue for each state
    - Different types of I/O have their own queue (KeyboardQ, SocketQ…)
  - Each PCB is queued onto a state queue according to its current state
    - As a process changes state, its PCB is unlinked from a queue, and linked onto another (ReadyQ → RunningQ, RunningQ → WaitingQ, …)



- At any time, there are many processes, each in its own state
    - Ready, running, waiting
- When a process is running, its hardware state is inside the CPU (**context switch**)
    - PC, SP, registers, etc.
- When the OS stops running a process (puts it in the ready or waiting state), it saves the registers' values in the process's PCB from the CPU
    - When the OS puts the process back in the running state, it loads the hardware register values from that process's PCB into the CPU

Parts of State Switching

- **Process Dispatch**: The act of selecting a process from the ready queue and giving it CPU time is called a process dispatch
- Context Switch: part of the process dispatch that specifically deals with:
    - Saving the hardware state of the currently executing process to its PCB
    - Loading the hardware state of the next process to be executed from its PCB into the CPU registers and resuming its execution



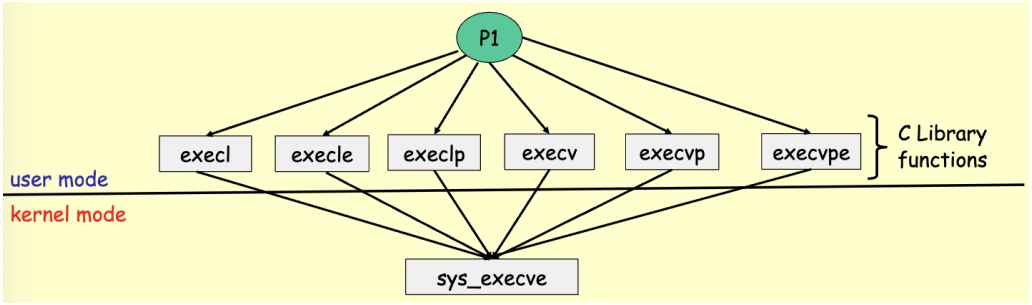
Process Creation

- One process can create another process, where the creator is called the parent and the created process is called the child
- When the child is created, the parent may either wait for the child to finish (via wait/waitpid system call), or it may continue in parallel
- In POSIX OSs, a process is created through fork() system call
    - Clone of the parent process

- Creates and initializes a new PCB with a new pid
- Creates a new address space with a copy of the entire contents of the address space of the parent
- Places new PCB on the ready queue
- Value returns once in the parent process (returns child's PID) and once in the child process (returns 0)

Executing Another Process

- We want to be able to load and execute a completely new executable file from the disk
- How? **int execve**(char *prog, char *argv[ ], char *envp[ ])
  - Stops the current process & loads program **prog** into memory
  - Initializes the hardware execution context, args for new program
  - Starts executing the new program, the first statement to execute being main
  - **Note**: execve system call does **NOT** create a new process! It simply overrides an existing process with the instructions of a new executable program
- You can only access execve through C exec* library calls



- Example: **execl**("/bin/ls", "ls", "-l", NULL);
  - **execl**("./ex0", "ex0", "arg0", "arg1", "arg2", NULL);
- To run a new program, we do **NOT** wants to overload the program on top of the current process
  - Instead use fork() + execve():
    - Create new child process, then load the new program on top of that child

Process Termination

- A process terminates by calling the exit() system call
  - OS deallocates process's resources (memory, open files, sockets, …)
  - The process then enters the "zombie" state
  - A "zombie" process is **NOT** active or running but still has a PCB
  - The PCB stores the exit status and minimal information about the process
  - A zombie process's PCB remains in the kernel until the parent cleans it up by calling wait()/waitpid()

```
for (int i=0; i<4; i++){
    if (fork() == 0){
        // Child process
        printf("Child: Pare
        exit(i);
```

```c
while (1){
    int exitStatus;
    int childPid = waitpid(-1, &exitStatus, 0);
    if (childPid < 0) break; // no more zombies
    printf("Cleaned up child process %d. exitStatus: %d\n", childPid, WEXITSTATUS(exitStatus));
```

Topic 3 - IPC

Inter-Process Communication (**IPC**)

- Two or more processes that run in the same machine (under the control of the same OS) communicate with each other to share or exchange data
- Why communicate?
    - Processes cooperating in a process chain
        - The output of one process is sent to another
    - Processes cooperating in a parallel application
        - Several processes working on different parts of a big problem in parallel
    - Client-Server communication
        - A client process asking for a service from a server process

IPC Alternatives

- Shared memory
    - OS allows different processes to share a portion of the memory by having each process map that memory to their address spaces
- Message Passing
    - Exchange data via a logical data channel (link)
    - Need to use send/receive or write/read operations
    - **Pipes**, message queues, sockets

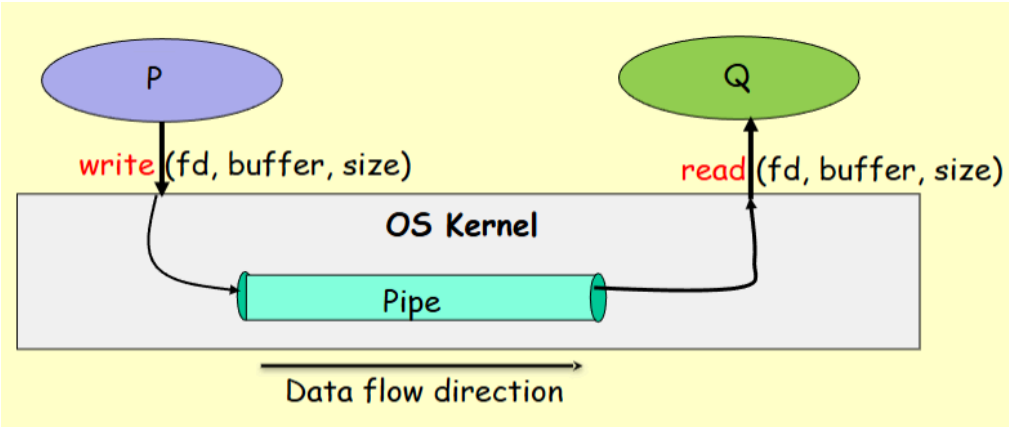
Shared Memory

- OS allows different processes to share a portion of the memory by having each process map that memory to their address spaces
    - All shared data will be kept in the shared memory, and all processes sharing that memory segment can access them
- Must explicitly be deleted when no longer needed using msgctl system call with the IPC_RMID command

Message Passing (Pipes)

- If process 1 and 2 want to communicate, they need to:
    - Establish a logical communication channel or link between them
    - Exchange messages via send/receive (write/read) operations
- Pipes:
    - Byte stream - message boundaries are NOT preserved
    - Unidirectional
    - Usually, two related processes

- Data flows from a write process to a reader process
- No message boundaries; data is read as a continuous stream of bytes
- Pipes exist for as long as the processes are running



- Read end: fd[0]
- Write end: fd[1]
- When coding, be sure to close either end when not in use
    - close(fd[0]) (close read end)
    - close(fd[1]) (close write end)

```c
int pipefd[2];
pipe(pipefd);

printf("Pipe descriptors: [0]: %d, [1]: %d\n"

if (fork() == 0){
    // Child. Will send messages to the parer
    close(pipefd[0]);  // Close the read-end

    char str[BUFFER_SIZE];
    while (1){
        memset(str, 0, BUFFER_SIZE);
        printf("Child: Enter a message: ");
        fgets(str, BUFFER_SIZE, stdin);
        str[strlen(str)-1] = '\0';        //
        if (strlen(str) == 0) continue;

        // Put the message in the pipe
        write(pipefd[1], str, strlen(str));
```

```c
/* Parent: Simply reads the message from the pipe &
close(pipefd[1]);  // Close the write end of the pi

while (1){
    memset(str, 0, BUFFER_SIZE);

    // Read the message from the pipe
    int len = read(pipefd[0], str, BUFFER_SIZE);
    str[len] = '\0';  // Mark the end of the string

    printf("--->Parent received: [%s]\n", str);
    if (strcmp(str, "quit") == 0) break;
} /* end-while */
```

- In Linux, it's possible to combine several commands together into a command chain using pipes(pipe operator: |)

Practice Problems

1. Why is it necessary for two processes to explicitly allocate shared memory to share a data structure such as an array?

This is necessary because two separate processes **do not** have access to each other's memory space. Thus, in order to share a data structure, such as an array, they need a memory space that is accessible to both processes. That is why they must explicitly allocate shared memory.

2. What is a pipe? What is it used for?

A pipe is a unidirectional message-passing link between processes. It is used to share data between two processes without allocating space for shared memory. This is done by assigning one process to be a read process and one to be a write process, where the write sends messages through the pipe and to the read process, where it is read.

3. What is a message queue

4. Assume you want to create 2 child processes, C1 and C2. You want the standard output of C1 to go to the standard input of C2. Write a very simple code to create child processes C1, C2 and tie them up using a pipe. C1 will then send the message “Hello\n” to its standard output, which C2 will read from its standard input and print it on the screen. After both children terminate, the parent prints “Parent done...\n” on the screen and terminates. You may assume the existence of a system call "pipe(int fd[2])" that creates a pipe and returns 2 descriptors. fd[0] will be the read end of the pipe, fd[1] will be the write end of the pipe. Also assume the existence of another system call “dup2(int fd1, int fd2)”, which makes fd2 same as fd1, i.e., copies the contents of fd1 over fd2. Use fork() system call to create a child process.

```
int pid2;
if((pid2 = fork()) == 0){ //C2
        close(fd[1]) // close write end
        char[6] message;
        read(fd[0], message, 6);
        write(1, message, 6);
        exit(0);
}
int pid1;
if((pid1 = fork()) == 0){ //C1
        close(fd[0]); // close read end
        write(fd[1], “Hello\n”, 6);
        exit(0);
}
waitpid(pid1);
```

```
printf(“Parent done…”);
return 0;
```

5. Implement a program that creates a child process and sends the message “Hello\n” to the child process over a pipe. The child must read the message coming from the pipe, print it out and then terminate. Make sure parent P waits for the child to terminate before printing “Parent done...\n” and terminating itself. You may assume the existence of a system call "pipe(int fd[2])" that creates a pipe and returns 2 descriptors. fd[0] will be the read end of the pipe, fd[1] will be the write end of the pipe. You can use waitpid(int pid) system call to wait for the termination of a process with “pid”. Use fork() system call to create a child process.

```
int fd[2];
pipe(fd);
Int pid;
if((pid = fork()) == 0){ // C1
        close(fd[1]);
        char[6] message;
        read(fd[0], message, 6);
        printf(message);
        exit(0);
}
close(fd[0]);
write(fd[1], “Hello\n”, 6);
close(fd[1]);
waitpid(pid, null, 0);
printf(“Parent done…\n”);
return 0;
```

6. Consider implementing a program P that creates two child processes, C1 and C2, with the following constraints: P creates 2 children and sends “Hello\n” to C1, which receives this message and sends it over to C2, which simply prints it on the screen. To achieve this, you must create two pipes, one between P and C1 and another between C1 and C2. You may assume the existence of a system call "pipe(int fd[2])" that creates a pipe and returns 2 descriptors. fd[0] will be the read end of the pipe, fd[1] will be the write end of the pipe. Make sure parent P waits for all children to terminate before printing “Parent done...\n” and terminating itself. You can use waitpid(int pid) system call to wait for the termination of a process with “pid”. Use fork() system call to create a child process.

```
int fd2[2];
pipe(fd2);

int pid2;
if((pid2 = fork()) == 0){ // C2
        close(fd2[1]);
        char[6] message;
        read(fd2[0], message, 6);
        write(1, message, 6);
        exit(0);
}
close(fd2[0]);
int fd1[2];
pipe(fd1);
int pid1;
if((pid1 = fork()) == 0){ //C1
        char[6] message;
        read(fd1[0], message, 6);
        write(fd2[1], message, 6);
        exit(0);
}
close(fd1[0]);
write(fd1[1], “Hello\n”, 6);
waitpid(pid1, null, 0;
printf(“Done”);
return 0;
```