Operating Systems Concepts

Midterm Review

CS 4375, Fall 2025

Instructor: MD Armanuzzaman (***Arman***)

marmanuzzaman@utep.edu

October 13, 2025

Summary

- xv6 scheduling
    - Default round robin scheduling of xv6
    - Understand the code base
- Implement priority scheduling in xv6
    - Tasks in homework 3
- Implement aging policy

Agenda

- Midterm:
    - Logistics
    - Topics
    - Review

Midterm Exam

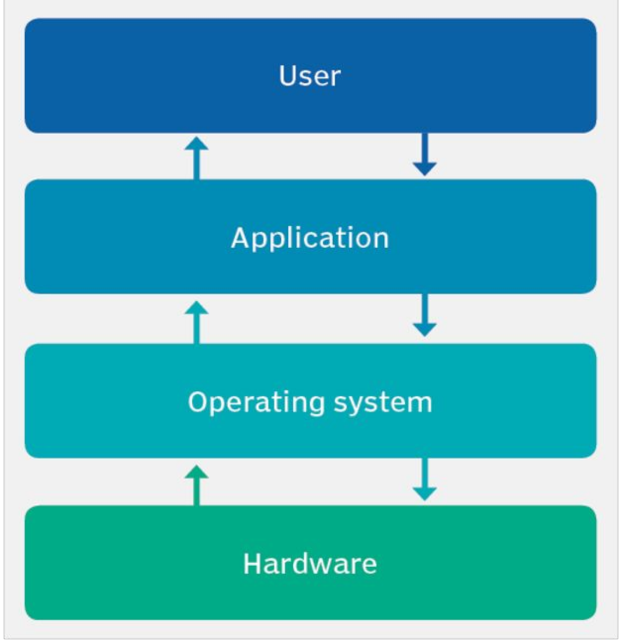
- Wednesday, October 15th
- During class time
- Handwritten one page note allowed
  - No other notes allowed
- Question pattern
  - MCQ
  - Written
- Total points **150**
- 1 hour

Topics

- Introduction to OS
- OS structure
- Processes
- Threads
- System Calls and Context Switches
- CPU Scheduling
- Address Translation

Introduction to OS

- An operating system (OS) is the systems software that **interfaces** application programs with the underlying hardware
- Applications make requests for services through a **defined APIs**
- The OS **manages** and **interfaces** to underlying hardware (e.g., **processors, memory, storage devices, network interfaces**) so that applications don't need to know about hardware details
- The OS **launches** and **manages** every application, including multiple processes or threads.



Types of Operating Systems

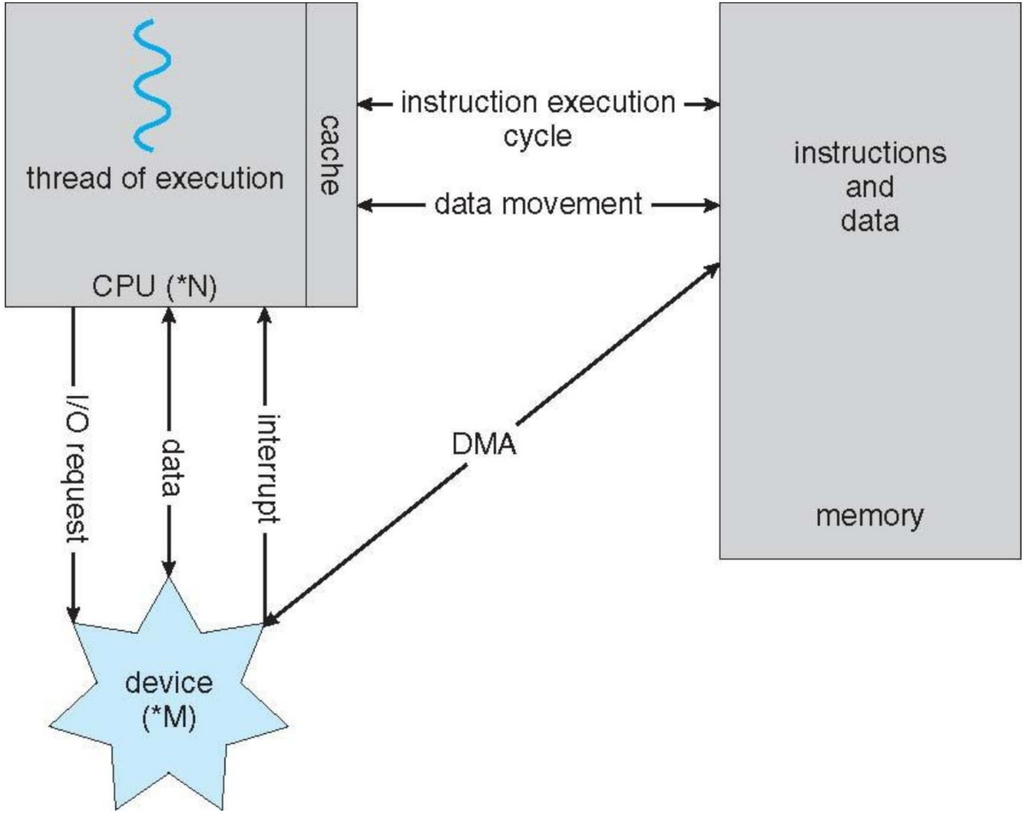
- General-purpose operating system
  - Run multiple applications in broad range of hardware
  - Windows, MacOS, Linux
- Mobile operating system
  - Efficient performance and resource usage and fast response time
  - Apple iOS, Google Android
- Embedded operating system
  - Usually provided on a chip that is incorporated into the device
  - ATMs, IoT devices, medical devices
  - Embedded Linux
- Real-time operating system
  - Respond quickly and predictably under time constraints
  - FreeRTOS, zephyr

Execution Steps Simplified

- Fetch instruction at PC
- Decode the instruction
- Execute (possibly using registers and modifying them)
- Write possible result to registers/memory
- Increase PC to point to the next instruction

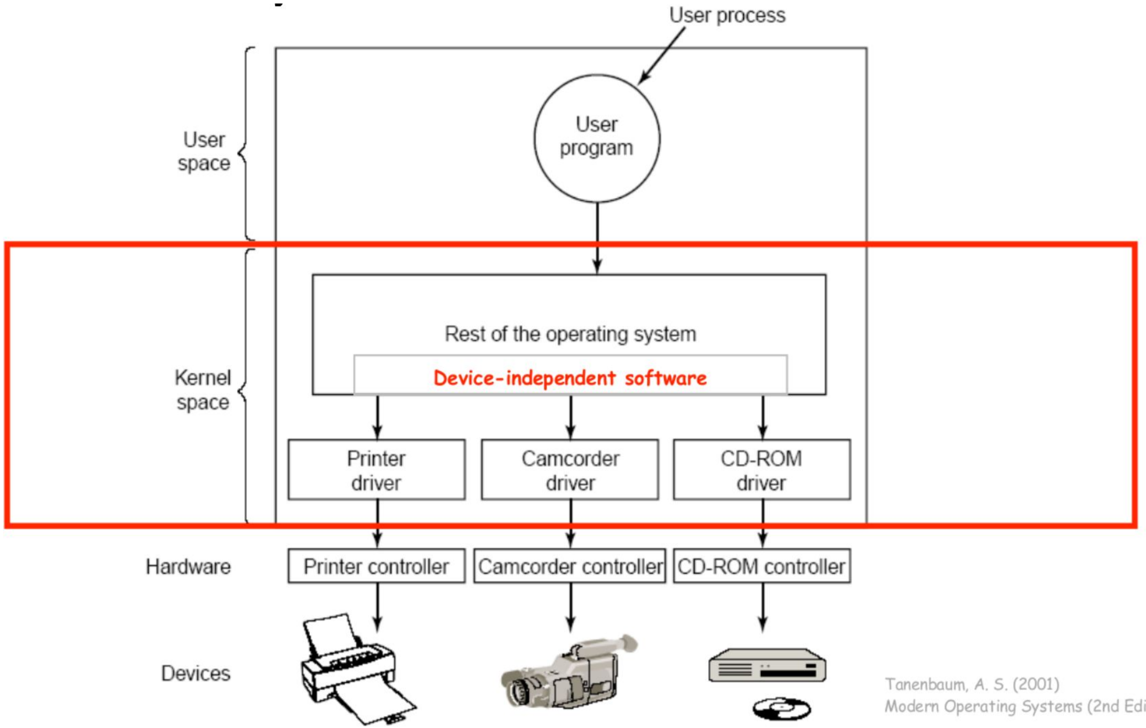
REPEAT!

Modern Computer Architecture



I/O Management

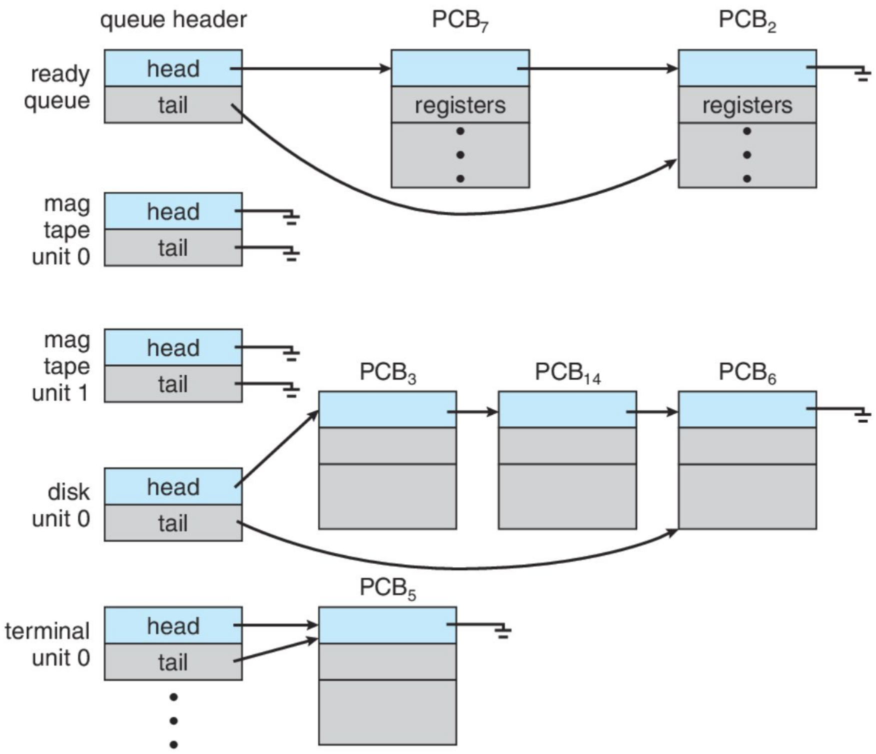
- Layers of the I/O subsystem



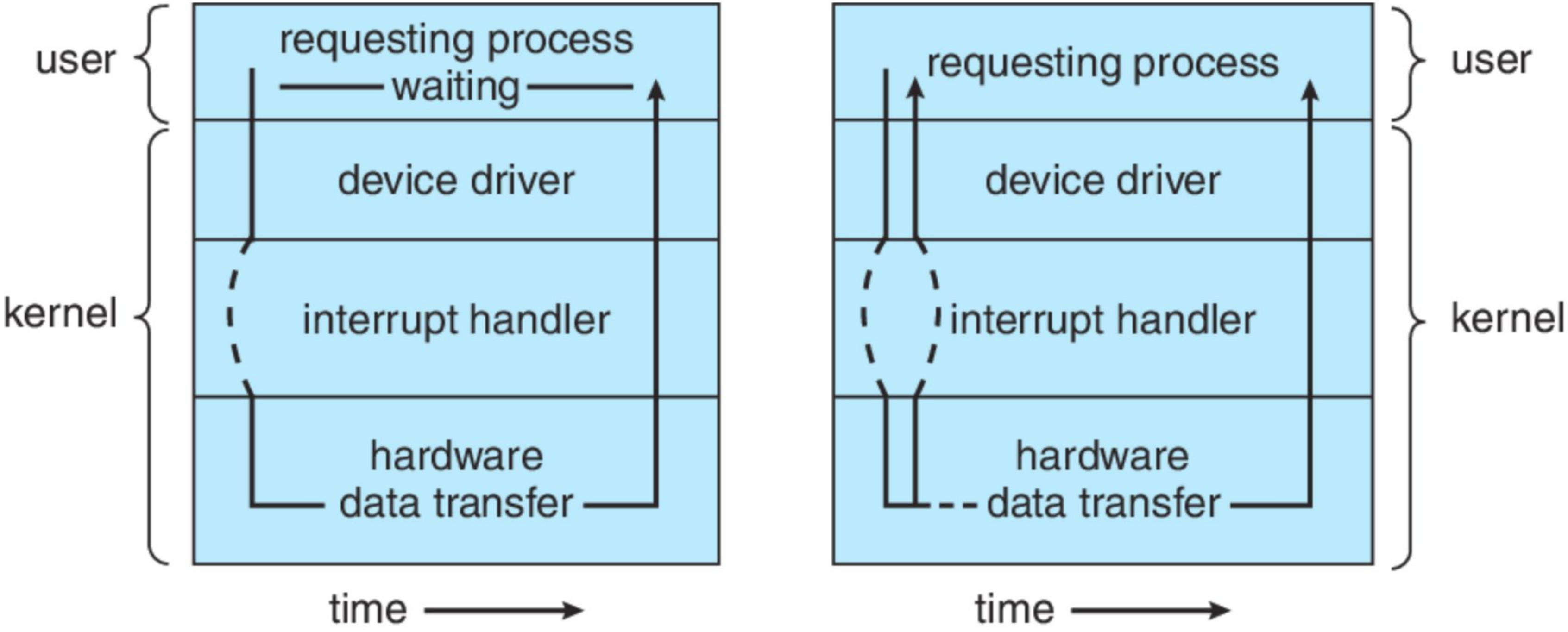
Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

I/O Management

- I/O Device Queues

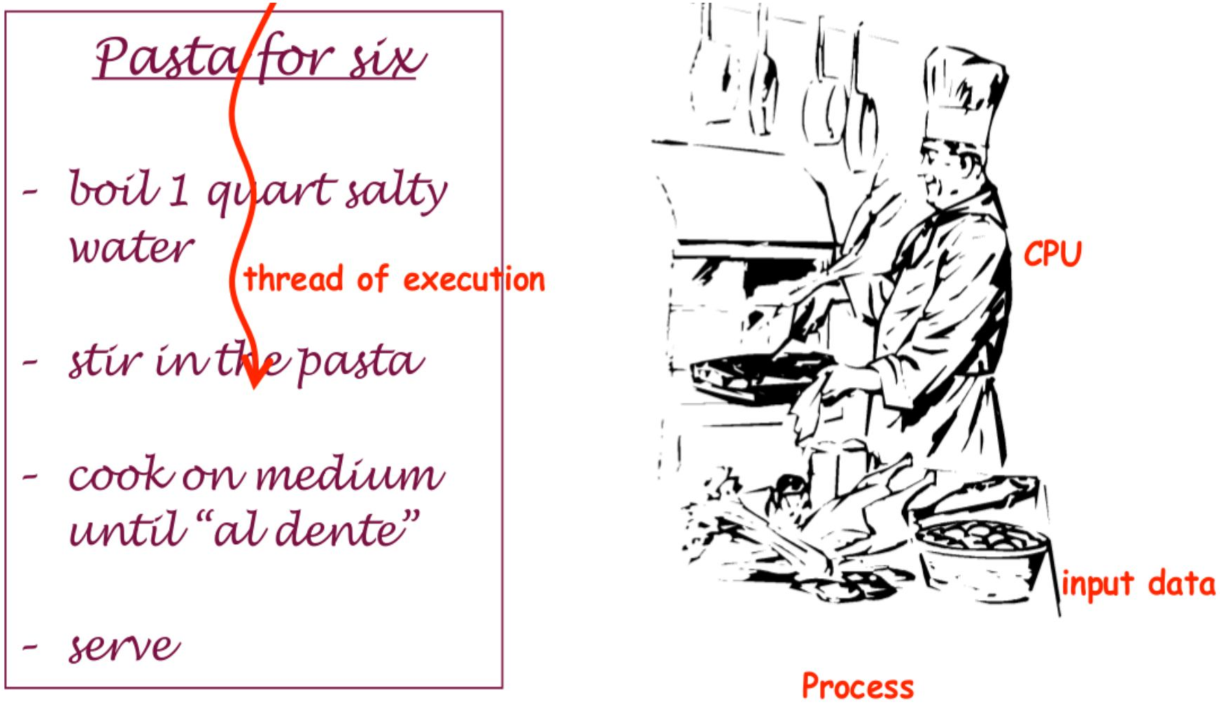


I/O Management: Two Methods



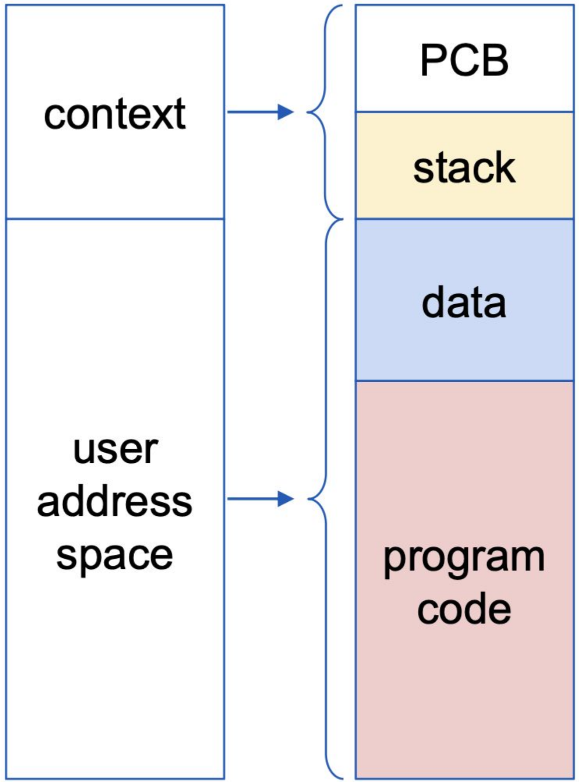
Processes Concept

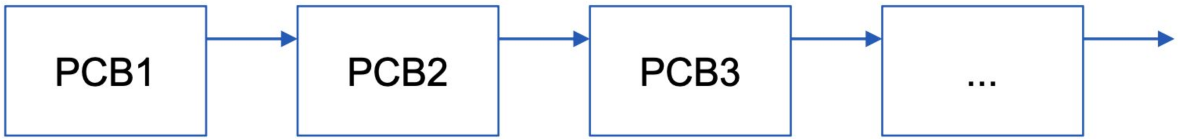
- A process is a program in execution.



Process Control Block (PCB)

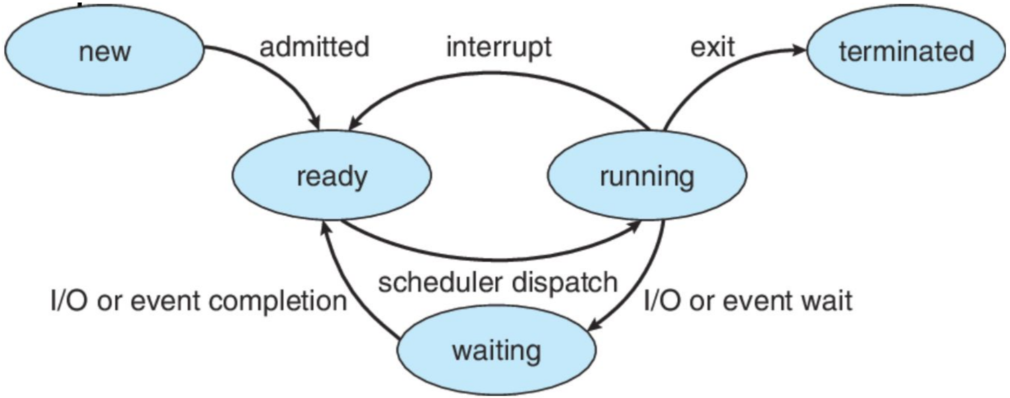
- PCB is included in the context, along with the stack
- PCB is a "snapshot" that contains all necessary and sufficient data to restart a process where it left off (ID, state, CPU registers, etc.)
- PCB is one entry in the OS's process table (array or linked list)





Process State

- A process changes its state during execution:
  - **New**: The process is being created
  - **Ready**: The process is waiting to be assigned to a processor
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Terminated**: The process has finished execution



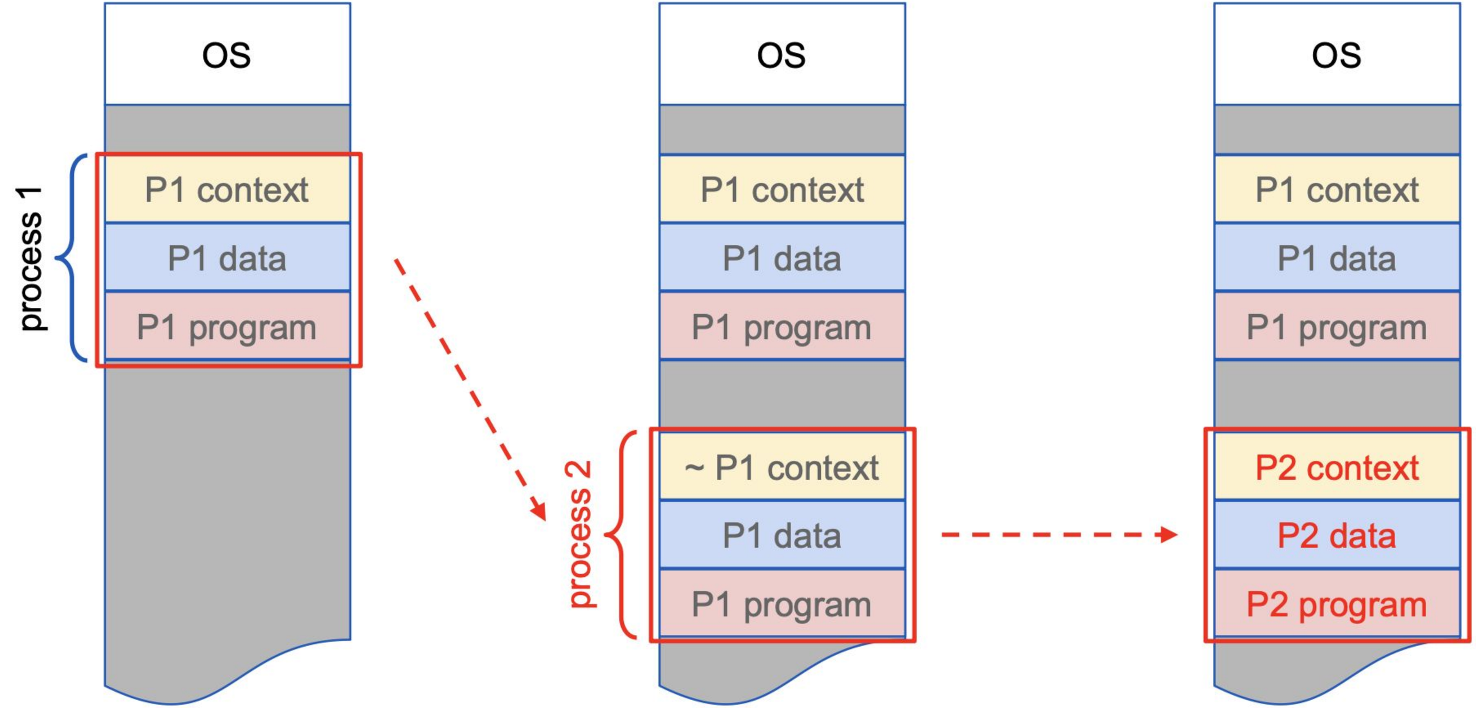
Process Creation

1. Clone child process

2. Replace child's image

```
pid = fork()
```

```
execvp(name, …)
```



Fork Example

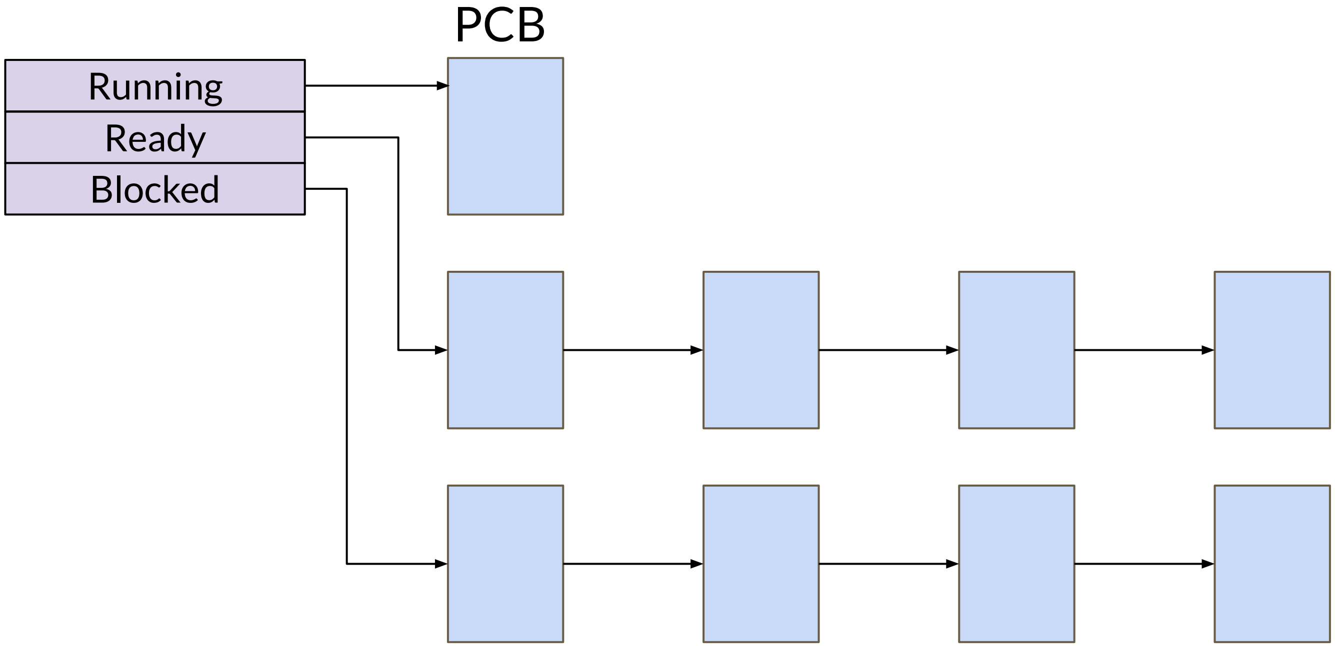
```c
#include <stdio.h>
#include <unistd.h>

int main()
{
    fork();
    fork();
    fork();
    printf("Process pid is %d\n", getpid());
    return 0;
}
```

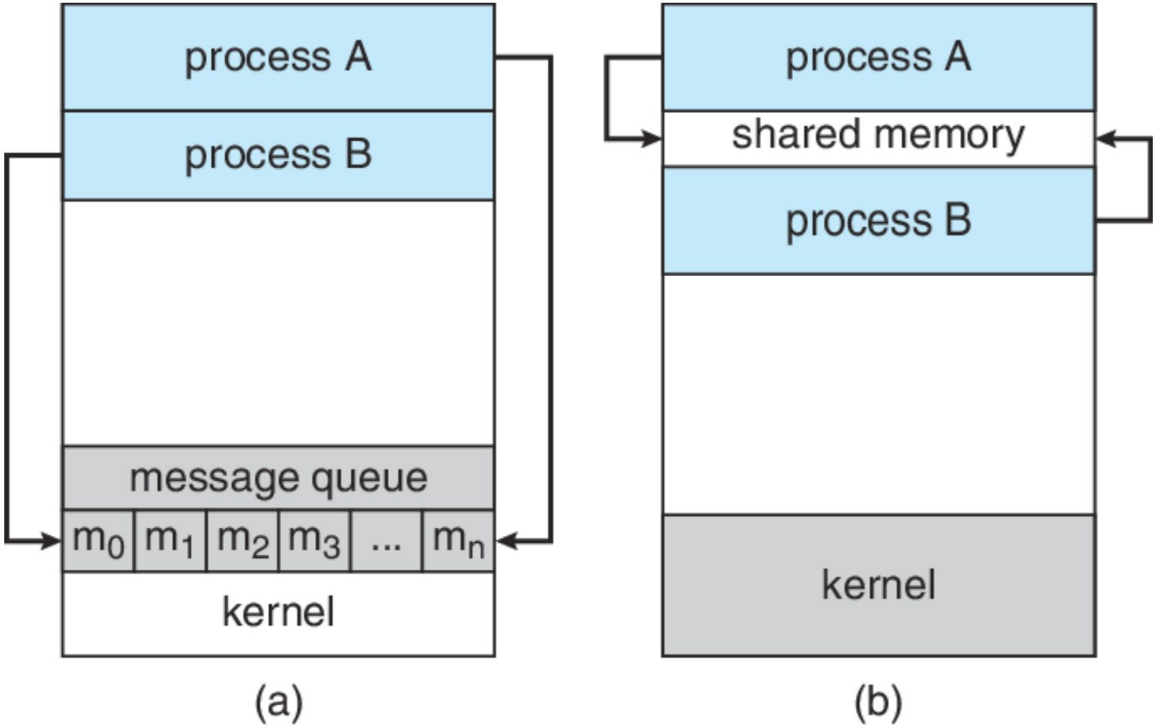
- **How many lines of output will this code produce?**

Process Queues

The process table can be split into per-state queues: PCBs can be linked together if they contain a pointer field



Communications Models

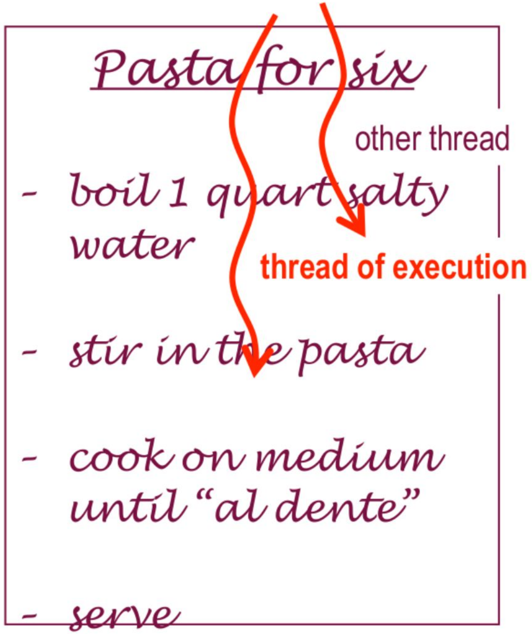


Message Passing

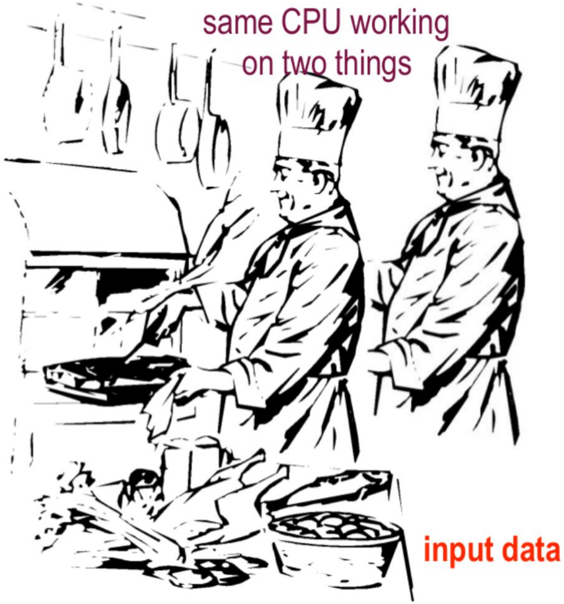
Shared Memory

Multithreading

The execution part is a “thread” that can be multiplied



Program

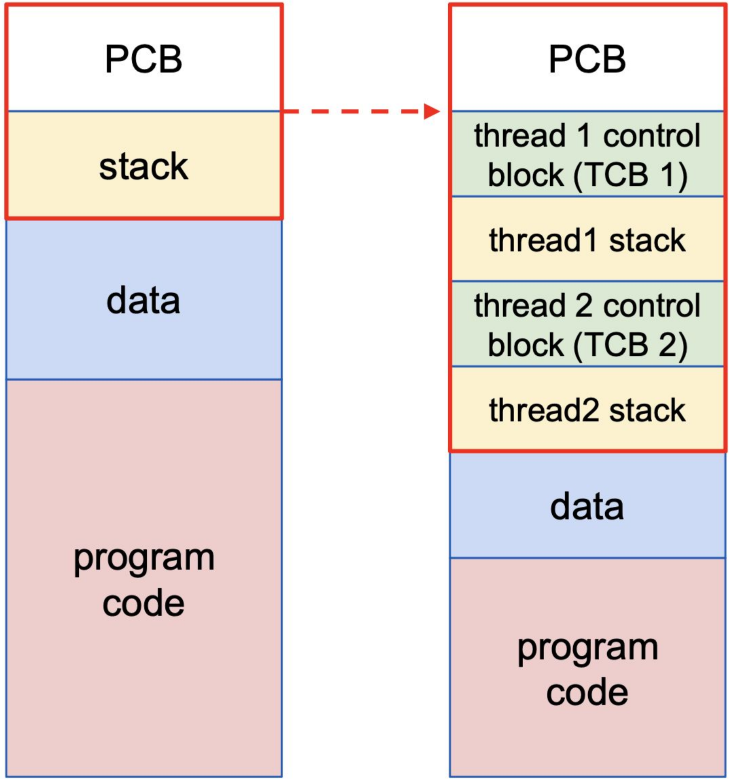


Process

New Process Description Model

Multithreading requires changes in the process description model.

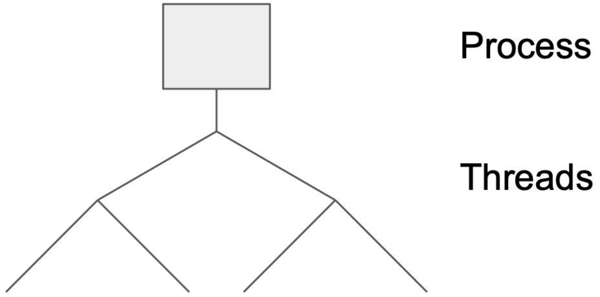
- Each thread of execution receives its own control block and stack.
  - Own execution state (“Running”, “Blocked”, etc.)
  - Own copy of CPU registers
  - Own execution history (stack)
- The process keeps a global control block listing resources currently used



Multiprocessing Model

Thread spawning

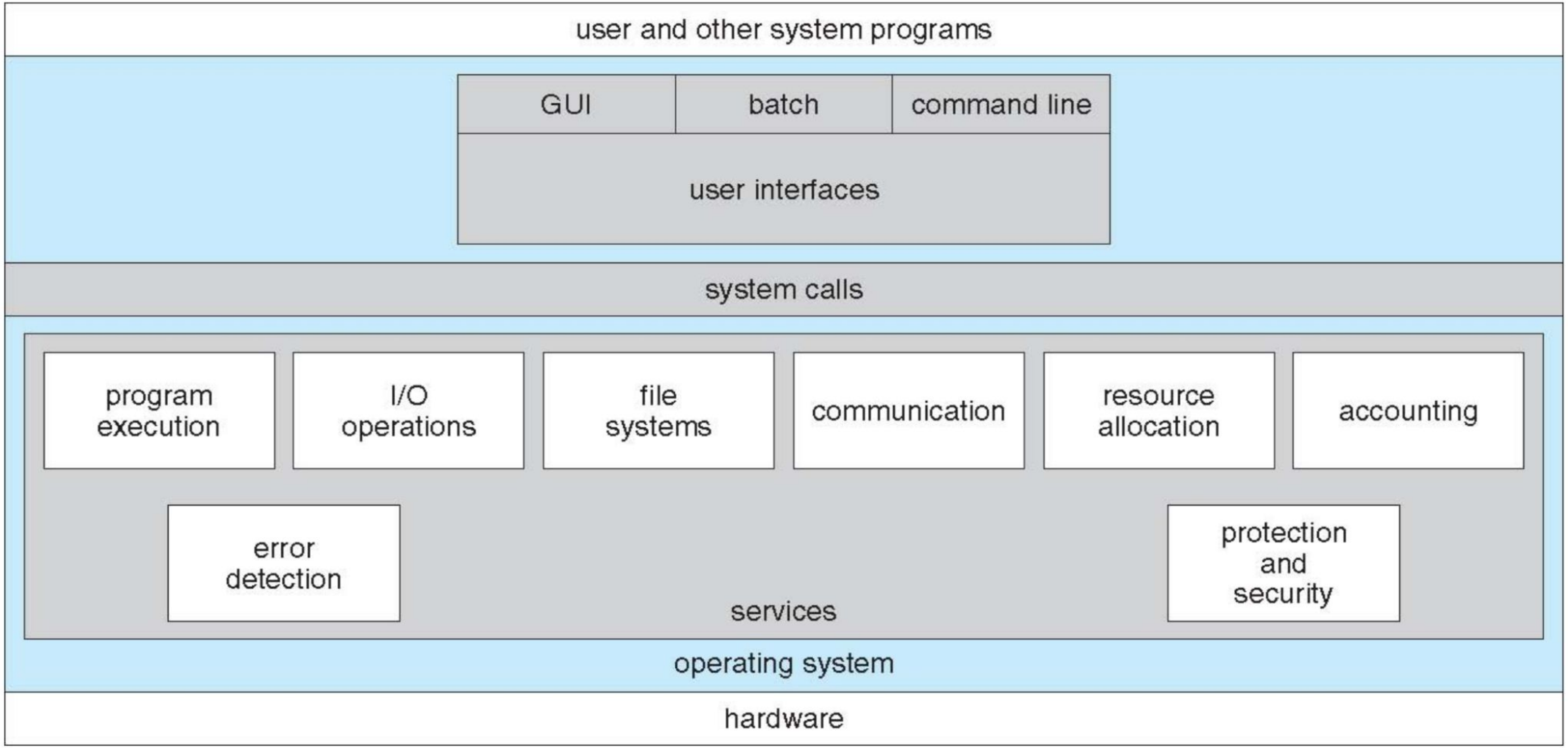
- Threads are created *within* processes, and belonging to processes
- All the threads created within one process, share the resources of the process including the address space
- Scheduling is performed on per-thread basis
- Threads have a similar lifecycle as the processes and will be managed mainly in the same way



Threads vs Processes

- A common terminology:
  - Heavyweight Process = Process
  - **Lightweight Process = Thread**
- Advantages:
  - Much quicker to create a thread than a process
    - spawning a new thread only involves allocating a new stack and a new CPU state block
  - Much quicker to switch between threads than to switch between processes
  - Threads share data easily
- Disadvantages:
  - Processes are more flexible
    - They don’t have to run on the same processor
  - No security between threads: One thread can stomp on another thread's data
  - For threads which are supported by user thread package instead of the kernel:
    - If one thread blocks all threads in task block

Operating System Services



Operating System Services

- Virtualize the CPU
  - **Performance**
    - Avoid overhead
    - Maximize CPU usage
  - **Control**
    - Provide performance while retaining control
    - CPU time, I/O resources, etc.

Solution

- OS
  - **System calls**
    - Allows the user program to request privileged operation
  - **Trap table**
    - Kernel boots first (privileged)
    - Sets up the **trap table**
    - Informs the hardware about trap handlers (Privileged instructions)
    - Hardware remembers until next reboot

Allowing user processes to jump to anywhere in the kernel is dangerous!!

Limited Direct Execution Protocol

OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember address of... syscall handler

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from-trap**		
	restore regs from kernel stack move to user mode jump to main	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs from kernel stack move to user mode jump to PC after trap	
		... return from main **trap** (via `exit()`)
Free memory of process Remove from process list		

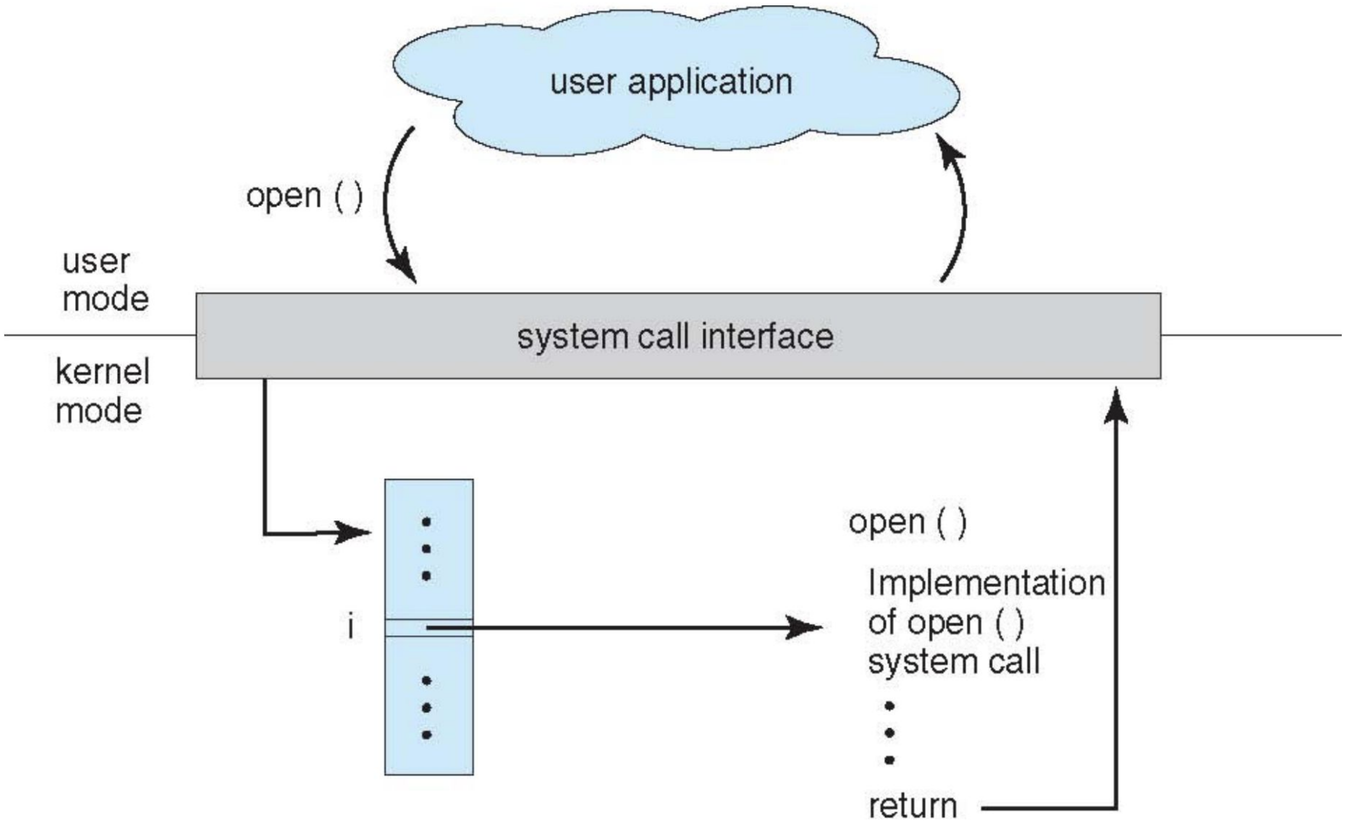
Phase 1

Phase 2

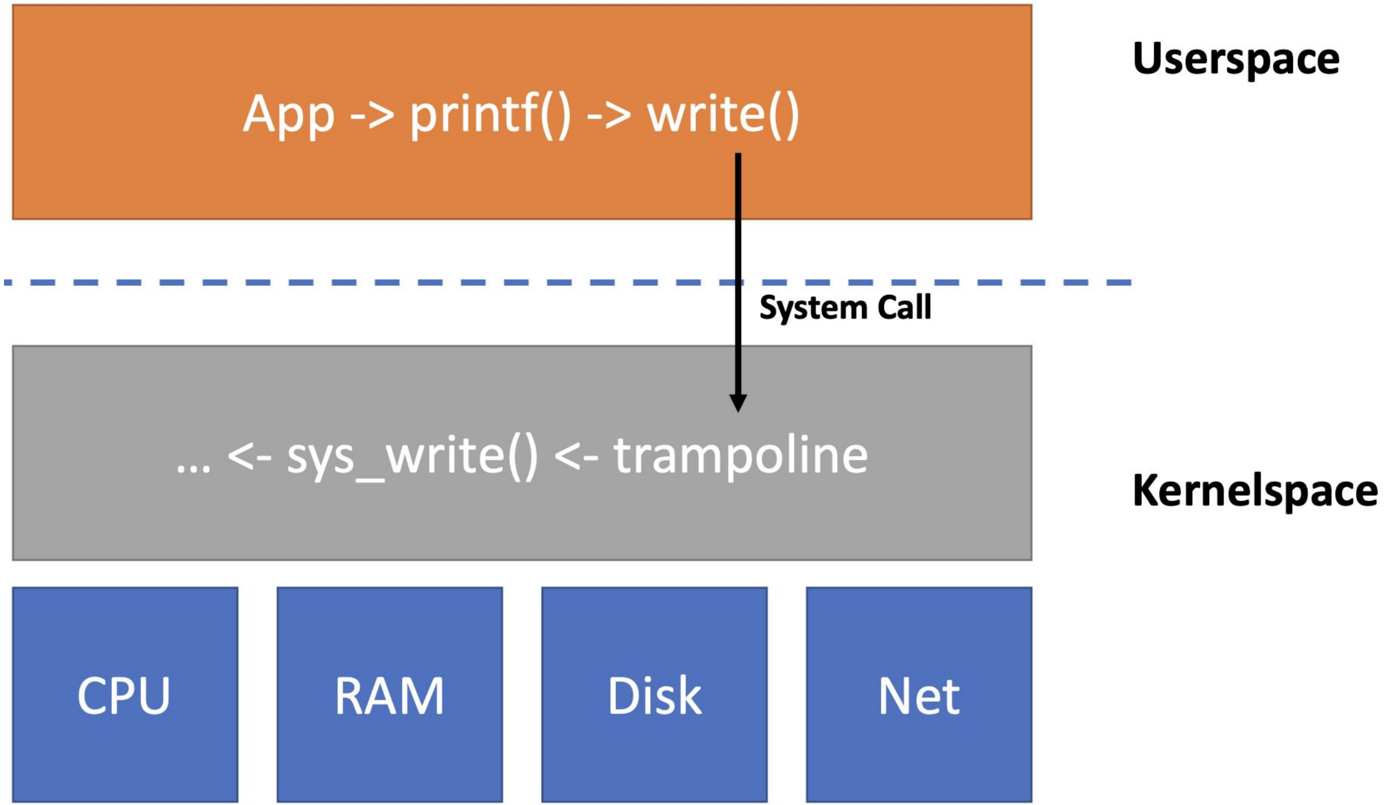
System Calls

- On modern operating systems, processes do not talk to hardware directly, but must go through OS
- **System Call:** request from a process for OS to do some kind of work on its behalf
- **Application Programming Interface (API):** interface provided by OS, usually in a high-level language (C or C++), that is easier to work with than raw system calls
    - POSIX for macOS, Linux, and other Unix-like, accessible through libc.so

System Call Interface



System calls

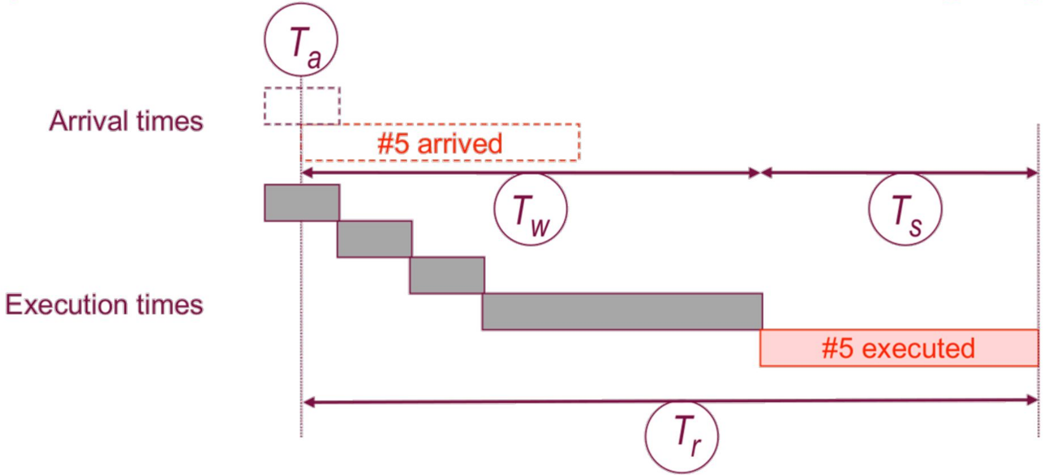


Scheduling Criteria

- CPU Utilization: Keep the CPU as busy as possible — **Maximize**
- Throughput: # of processes that complete their execution per time unit — **Maximize**
- Response Time: Amount of time it takes from when a request was submitted until the first response (not output) is produced (for time-sharing environment) — **Minimize**
- Waiting Time: Total amount of time a process has been waiting in the ready queue — **Minimize**
- Turnaround Time: Amount of time passed to finish execution of a particular process i.e. execution(service) time + waiting time — **Minimize**

Scheduling Metrics

- T_a - **Arrival time:** Time the process became "READY" [again]
- T_w - **Waiting time:** Time spent waiting for the CPU
- T_s - **Service time:** Time spent executing in the CPU
- T_r - **Turnaround time:** Time spent waiting and executing = $T_w + T_s$

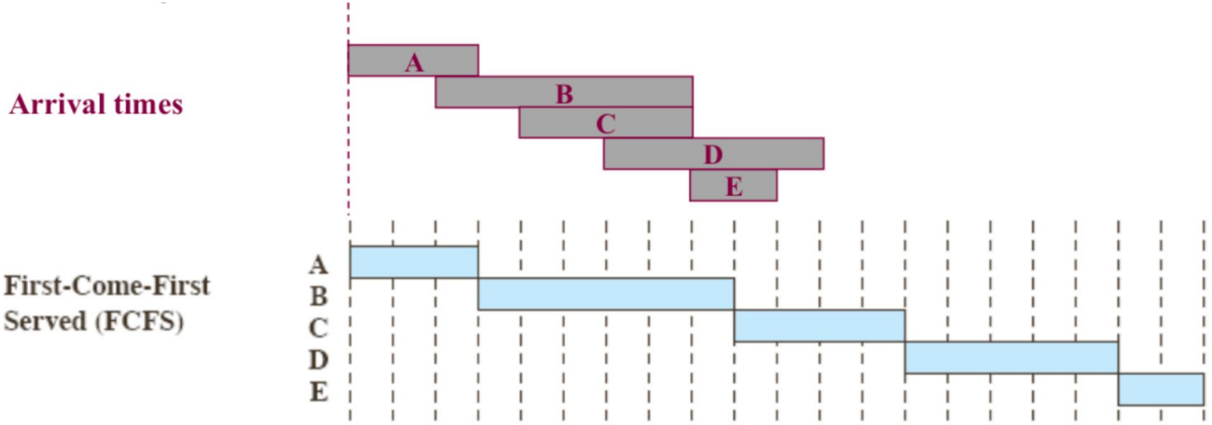


$T_r / T_s = 2.5$

Scheduling: First-Come, First-Served (FCFS)

- Processes are assigned the CPU in the order they request it
- When the running process blocks, the first “READY” process is selected
- When a process gets “READY”, it is put at the end of the queue

FCFS Scheduling Policy

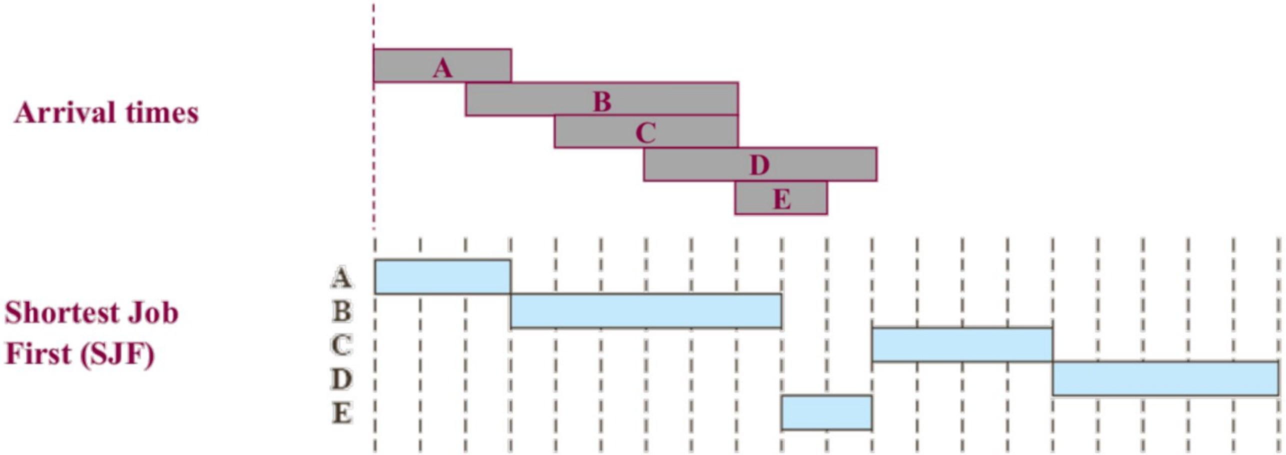


FCFS		A	B	C	D	E	Mean
	Finish Time	3	9	13	18	20	
	Turnaround Time (T_r)	3	7	9	12	12	8.60
	T_r/T_s	1.00	1.17	2.25	2.40	6.00	2.56

Scheduling: Non-Preemptive SJF

- Among several equally important "READY" jobs (or CPU bursts), the scheduler picks the one that will finish the earliest

SJF Scheduling Policy

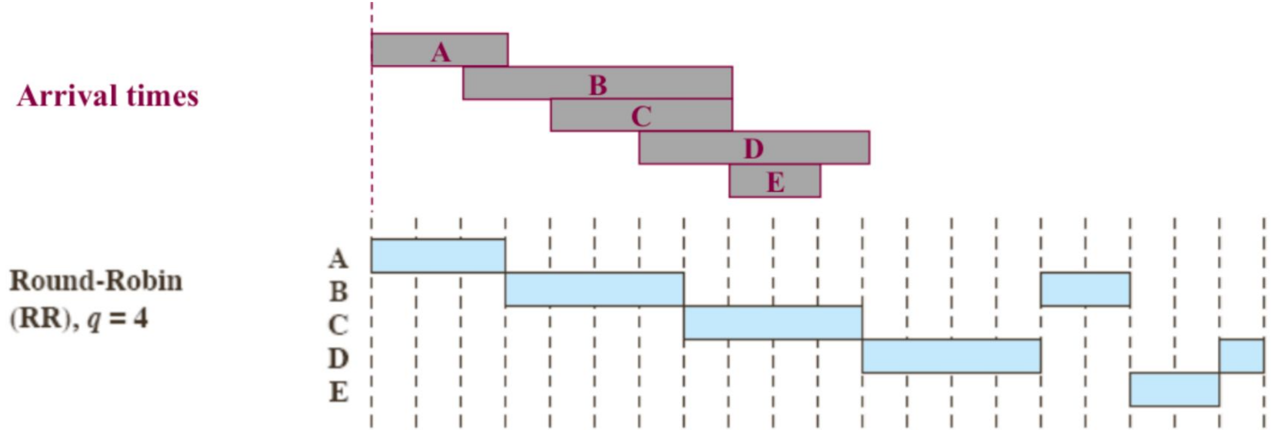


SJF		A	B	C	D	E	Mean
	Finish Time	3	9	15	20	11	
	Turnaround Time (T_r)	3	7	11	14	3	7.60
	T_r/T_s	1.00	1.17	2.75	2.80	1.50	1.84

Scheduling: Round-Robin (RR)

- A crucial parameter is the quantum **q** (~10-100ms)
  - **q** should be large compared to context switch latency (~10μs)
  - **q** should be less than the longest CPU burst, or RR degenerates to FCFS

RR (q = 4) Scheduling Policy



RR $q = 4$		A	B	C	D	E	Mean
	Finish Time	3	17	11	20	19	
	Turnaround Time (T_r)	3	15	7	14	11	10.00
	T_r/T_s	1.00	2.5	1.75	2.80	5.50	2.71

Comparison: Round-Robin

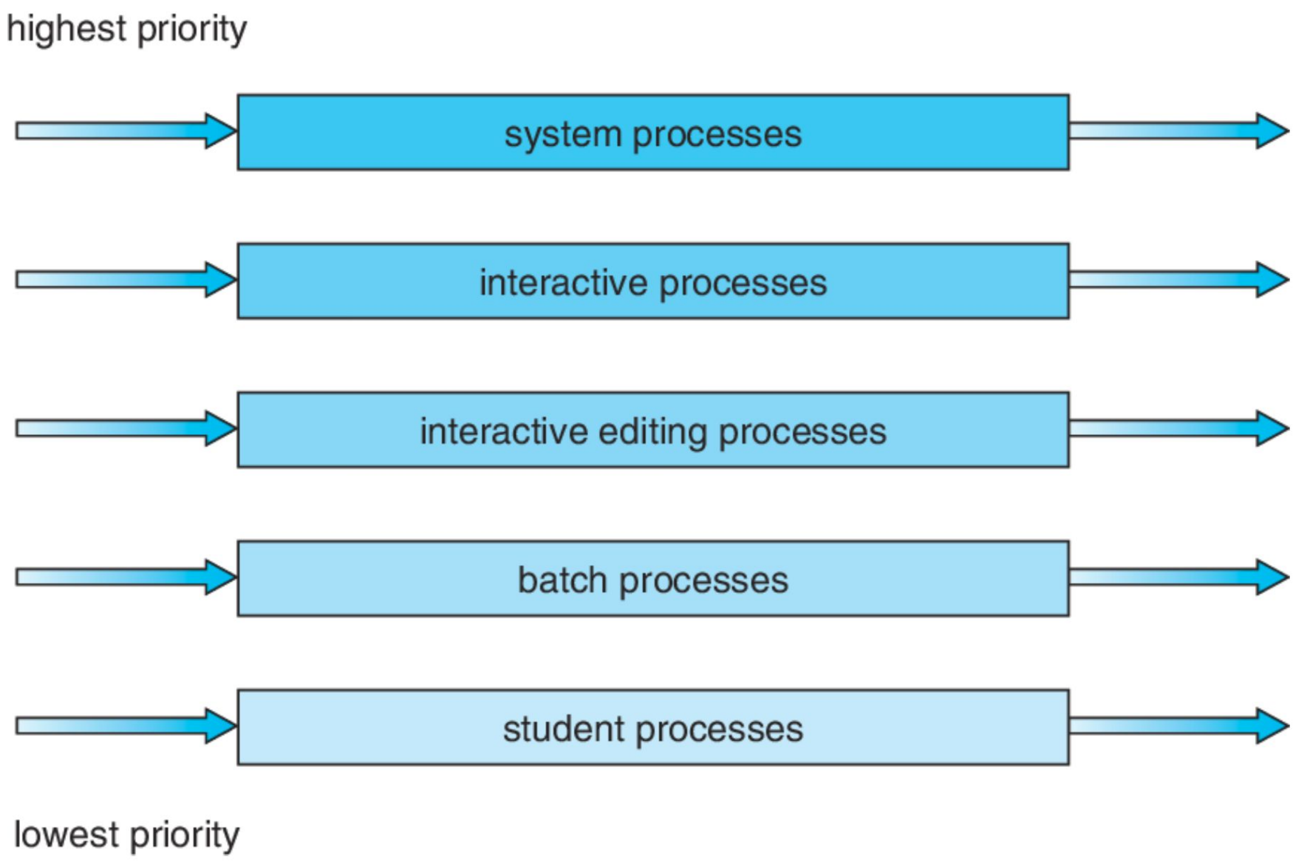
- PROS:
    - Great for timesharing
        - No starvation
    - Does not require prior knowledge of CPU burst times
    - Generally reduces average response time
- CONS:
    - What if all jobs are almost the same length
        - Increases the turnaround time
    - How to set the “best” time quantum?
        - If too small, it increases context-switch overhead
        - If too large, response time degrades

Exercise

Process	Arrival Time	Burst Time	Priority
P_1	9	3	5
P_2	6	4	4
P_3	8	5	3
P_4	7	9	2
P_5	5	7	1

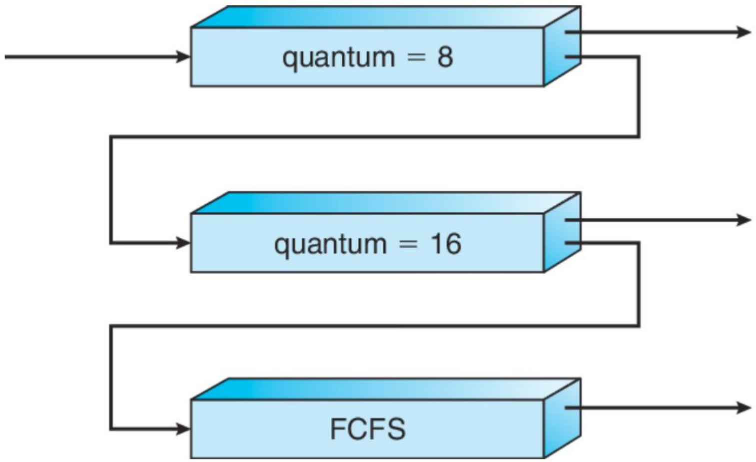
- Consider Shortest-Remaining-Time-First (SRTF) scheduling policy is used.
    a. Draw the Gantt chart
    b. Find the response time, waiting time, and turnaround time for each process.

Multilevel Queue Scheduler



Example of Multilevel Feedback Queue Scheduler

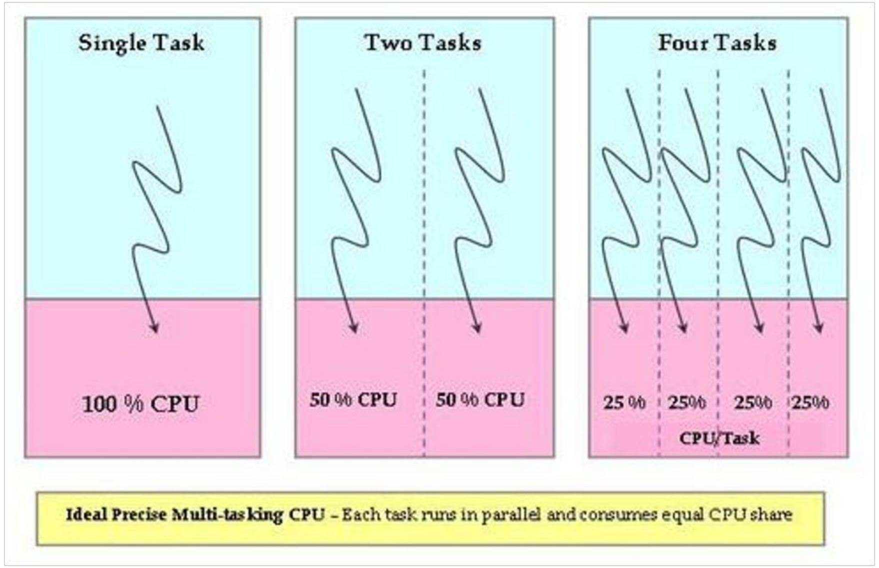
- Three queues:
  - Q_0 - RR with q = 8 ms
  - Q_1 - RR with q = 16 ms
  - Q_2 - FCFS



- Scheduling
  - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, **job is moved to queue Q_1**
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is **preempted and moved to queue Q_2**

Proportional Share Scheduling

- Try to guarantee that each process gets a certain percentage of the CPU time
  - Over some period of time, sometimes called the scheduling interval
  - “Fair share” may depend on process’s priority

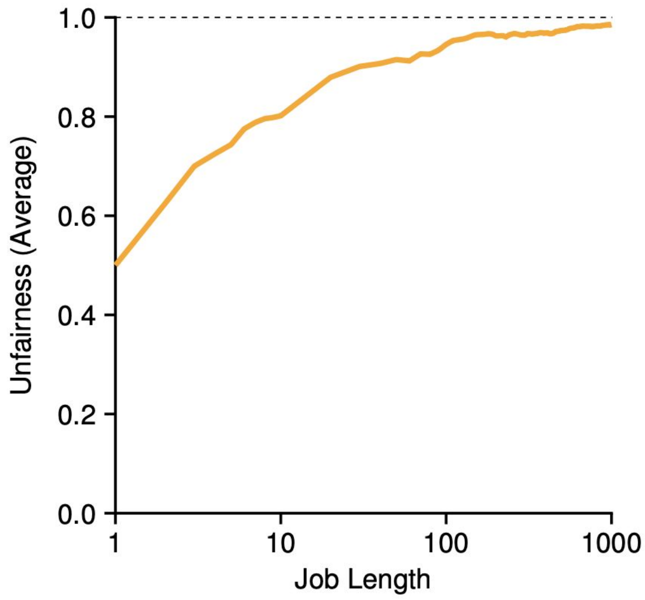


Lottery Scheduling

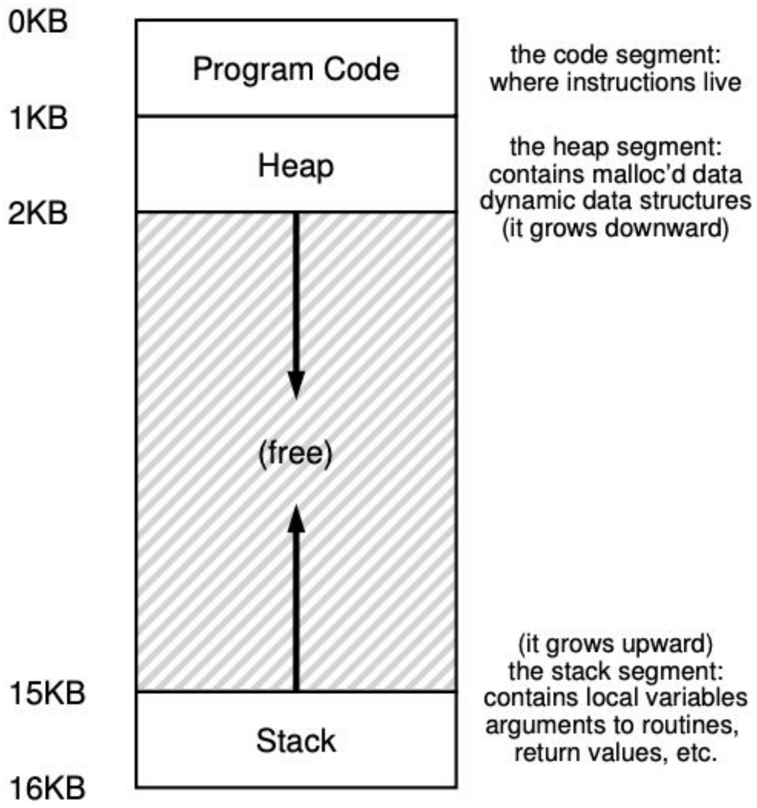
- Tickets assigned to a process represent its share
  - e.g., 10 out of 100 tickets process should get 10% of the CPU time
- Scheduling is probabilistic, using a random number generator
  - What problem does this raise?
- What about processes that do I/O?
- Ticket currency allows local authority (e.g., user) flexibility in allocating tickets
  - e.g., can do ticket inflation within local group of processes that trust each other
- Efficient implementation

Lottery Scheduling: Problems

- How To Assign Tickets?
  - Open problem
- Unfairness:
  - Two jobs with short lengths
  - Unfairness metric = A_rutime/B_runtime
- Not Deterministic

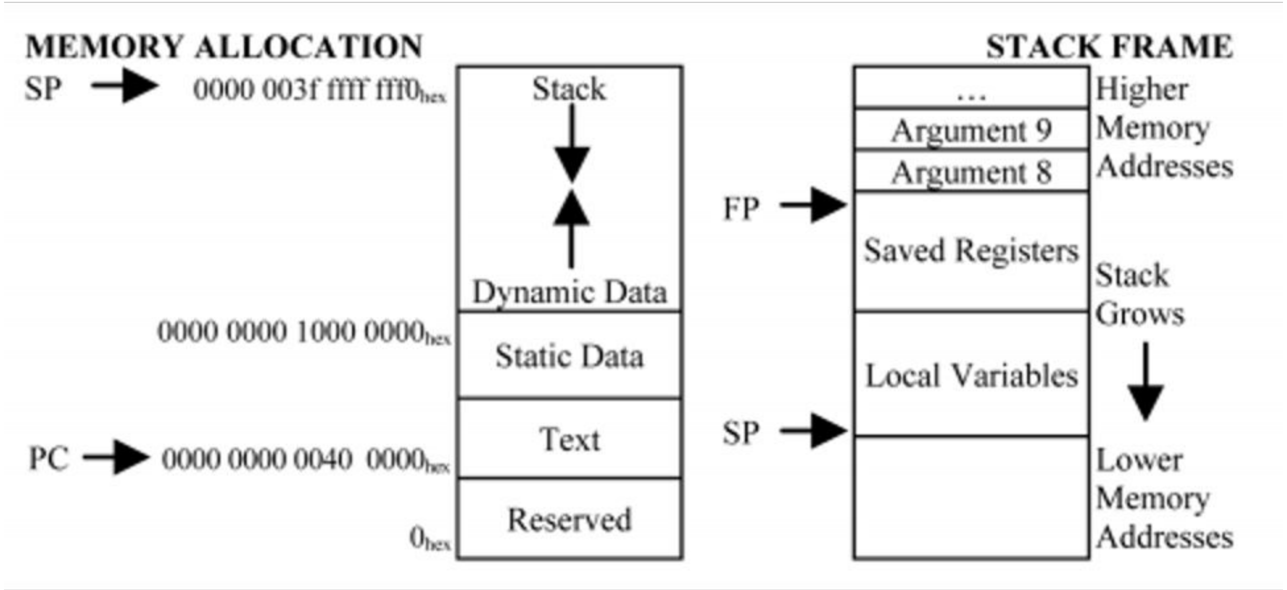


Program Address Space



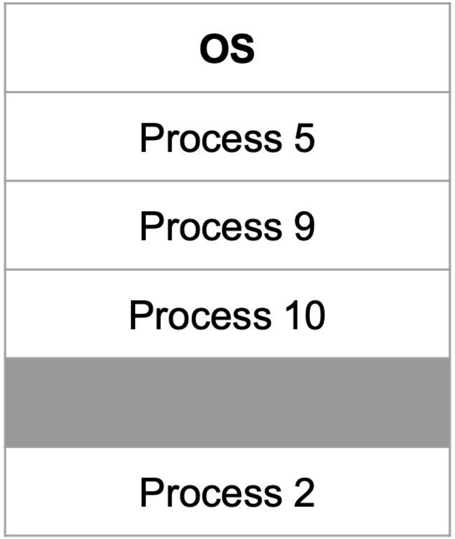
Program Address Space (RISC-V)

- RISC-V Linux 39-bit Address Space



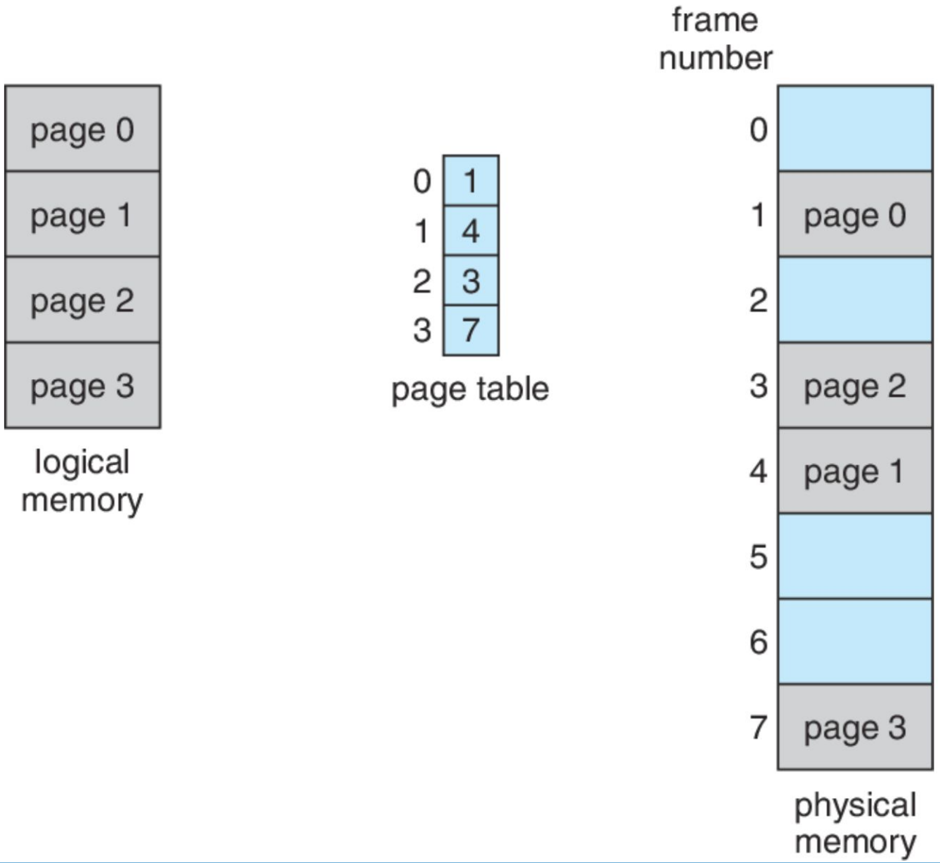
Memory Allocation

- Fixed-partition allocation
  - Divide memory into fixed-size partitions
  - Each partition contains exactly one process
  - The degree of multiprogramming is bound by the number of partitions
  - When a process terminates, the partition becomes available for other processes



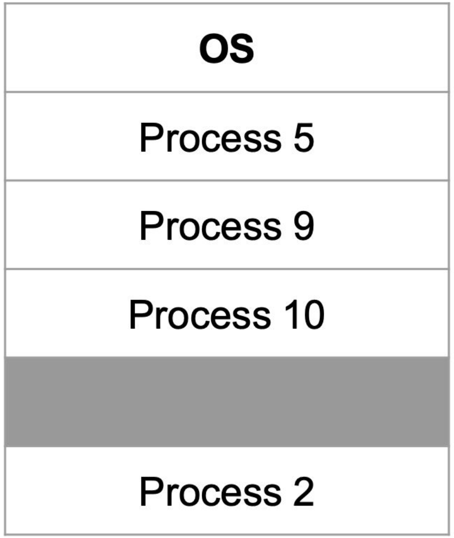
- May lead to Internal Fragmentation– allocated memory may be slightly larger than **requested** memory; this size difference is memory **internal** to a partition, but not being **used**

Paging Example

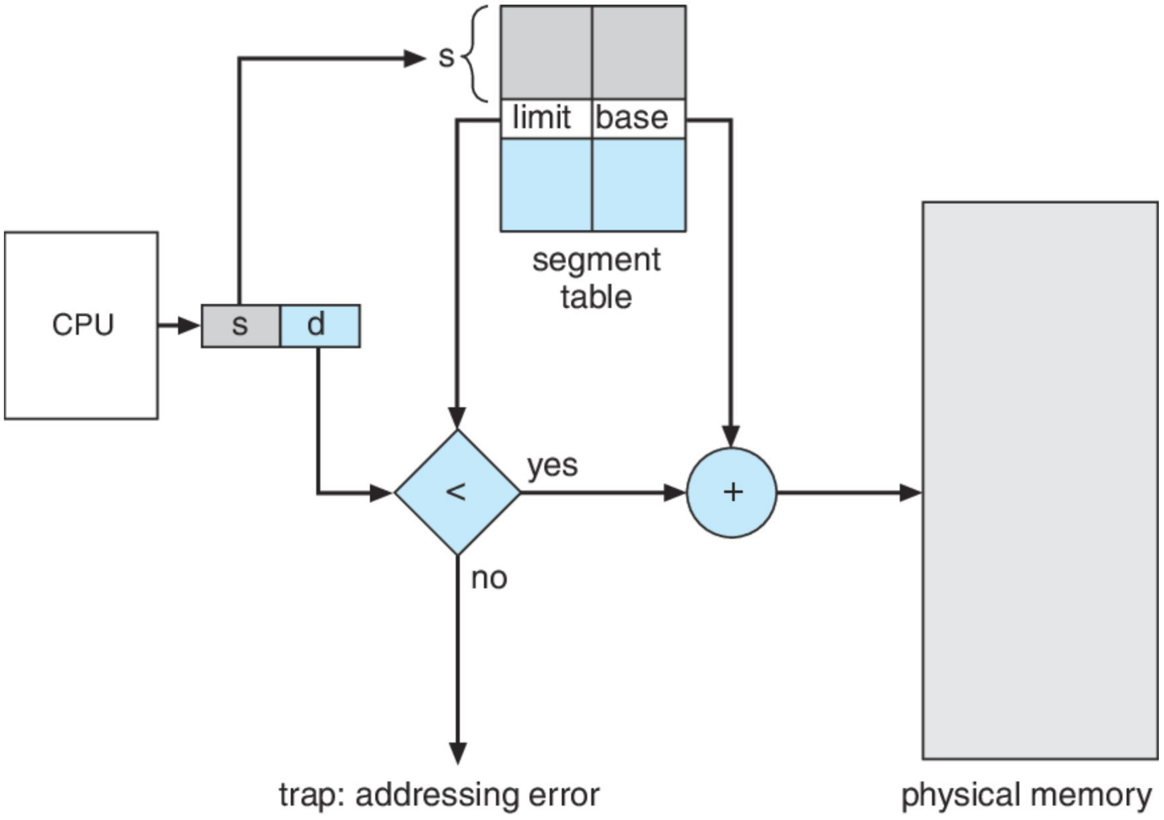


Memory Allocation

- Variable-partition Scheme (Dynamic)
  - When a process arrives, search for a hole large enough for this process
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - Allocate only as much memory as needed
  - Operating system maintains information about:
    - allocated partitions
    - free partitions (hole)



Segmentation: Address Translation Architecture



Paging vs Segmentation

Paging	Segmentation
Fixed-size division	Variable-size division
Managed by OS	Managed by compiler
Fragmentation: Internal	Fragmentation: External
Speed: Faster	Speed: Slower
Unit size by hardware	Unit size by user/programmer
Invisible to the user	Visible to the user

Modern OSes implement a hybrid approach called "**segmentation with paging**"

Midterm Exam

- Wednesday, October 15th
- During class time
- Handwritten one page note allowed
  - No other notes allowed
- Question pattern
  - MCQ
  - Written
- Total points **150**
- 1 hour