Introduction to the Operating System

Dr. Animesh Chaturvedi

Assistant Professor: IIIT Dharwad

Young Researcher: Heidelberg Laureate Forum

Postdoc: King's College London & The Alan Turing Institute

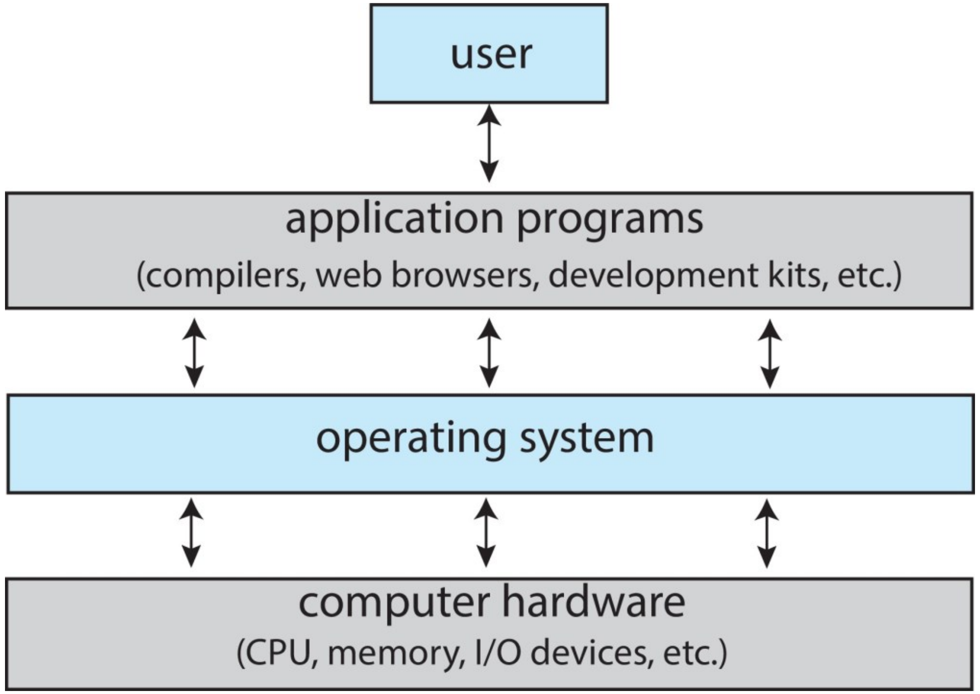
PhD: IIT Indore MTech: IIITDM Jabalpur

Introduction to the Operating System

- What is an Operating System (OS)?
- Operating system goals (PC Motherboard)
- OS manages Program execution
- OS manages CPU
- OS manages memory (Storage-Device Hierarchy)
- OS manages devices
- OS History
- Operating System Services

What is an Operating System (OS)?

- Program, Middleware, and Process between user programs and computing hardware
- Manages hardware:
  - CPU,
  - main memory,
  - IO devices
    - disk
    - network card
    - Mouse
    - keyboard etc.



Operating system goals

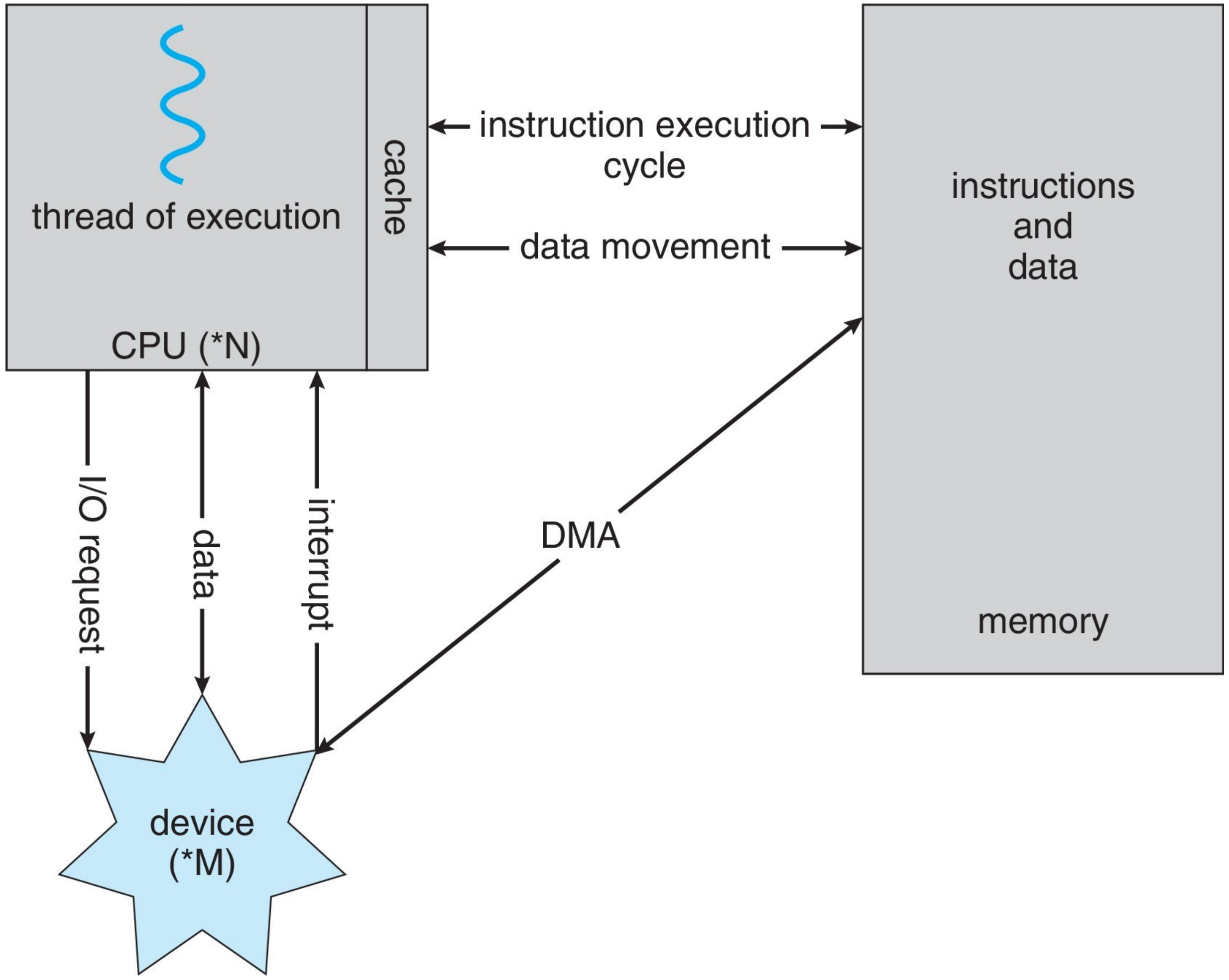
- OS manages program memory – Loads program executable (code, data) from disk to memory
- OS manages CPU – Initializes program counter (PC) and other registers to begin execution
- OS manages external devices – Read/write files from disk.
- A program that acts as an intermediary between a user of a computer and the computer hardware
- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

OS manages Program execution

- A compiler translates high level programs into an executable (".c" to "a.out")
- The exe contains instructions that the CPU can understand, and data of the program (all numbered with addresses)
- Instructions run on CPU: hardware implements an instruction set architecture (ISA)
- CPU also consists of a few registers, e.g.,
  - Pointer to current instruction (program counter or PC)
  - Operands of instructions, memory addresses
- To run an exe, CPU – fetches instruction pointed at by PC from memory
  - loads data required by the instructions into registers
  - decodes and executes the instruction
  - stores results to memory
- Most recently used instructions and data are in CPU caches for faster access

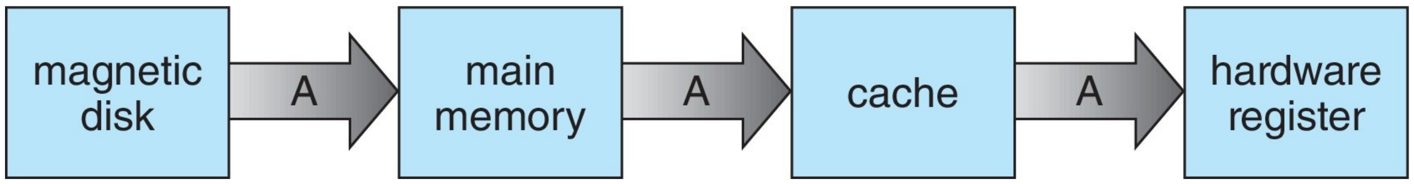
OS manages CPU

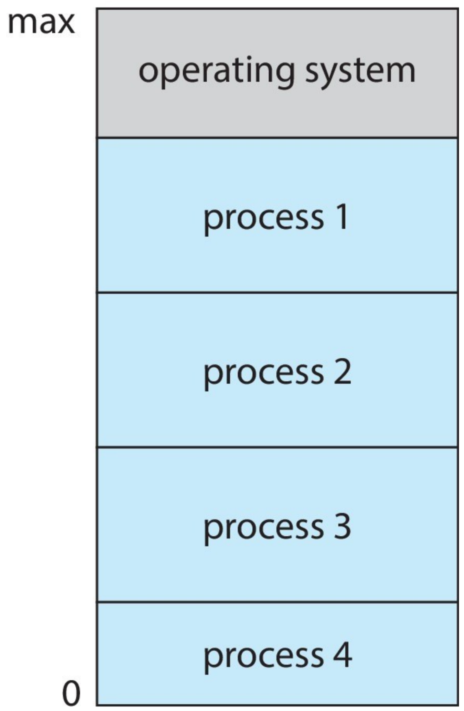
- OS provides the process abstraction
  - Process: a running program
  - OS creates and manages processes.
- Each process has the illusion of having the complete CPU, i.e., OS virtualizes CPU
- Timeshares CPU between processes
- Enables coordination between processes



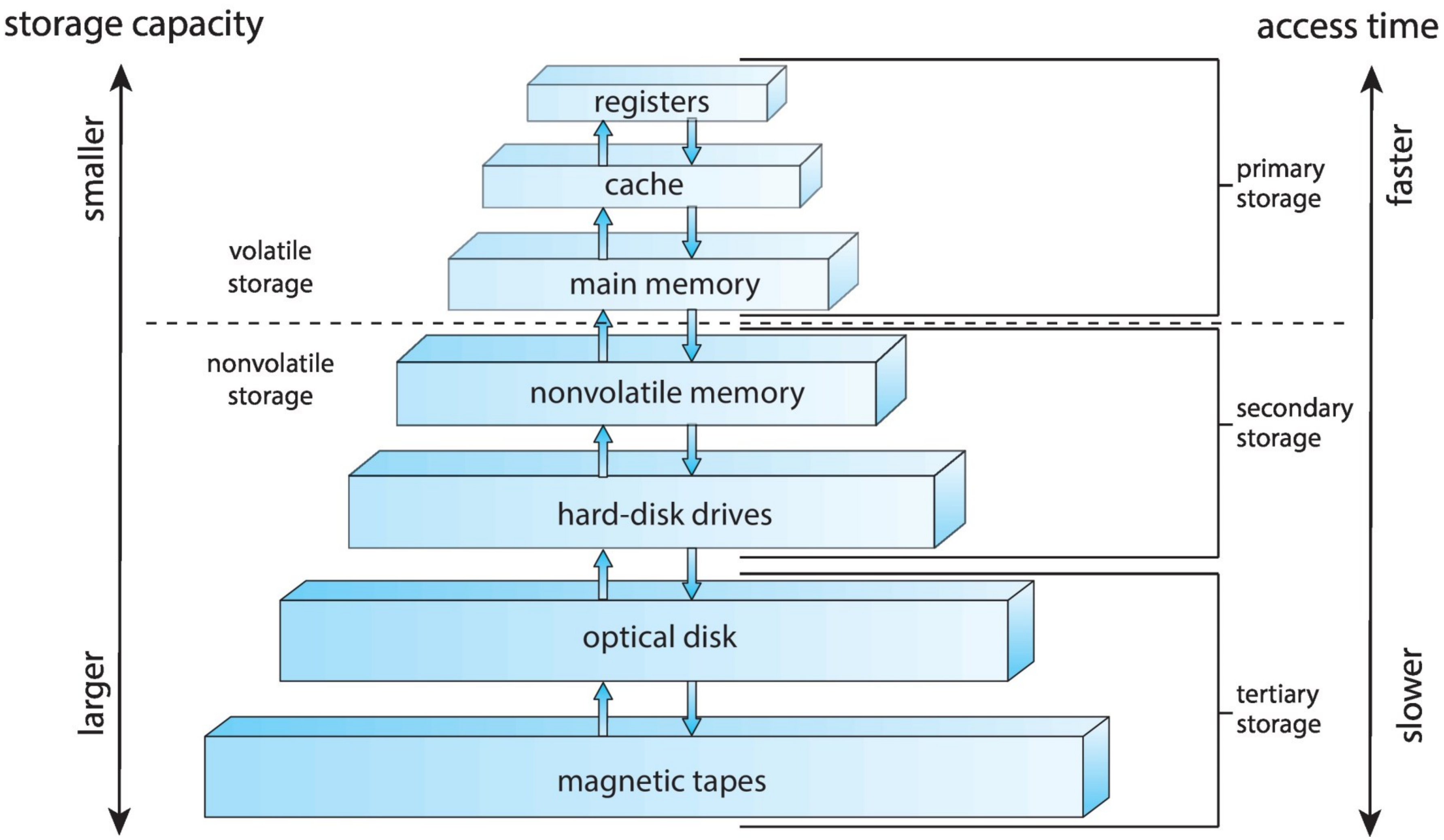
OS manages memory

- OS manages the memory of the process: code, data, stack, heap etc
- Each process thinks it has a dedicated memory space for itself, numbers code and data starting from 0 (virtual addresses)
- OS abstracts out the details of the actual placement in memory, translates from virtual addresses to actual physical addresses





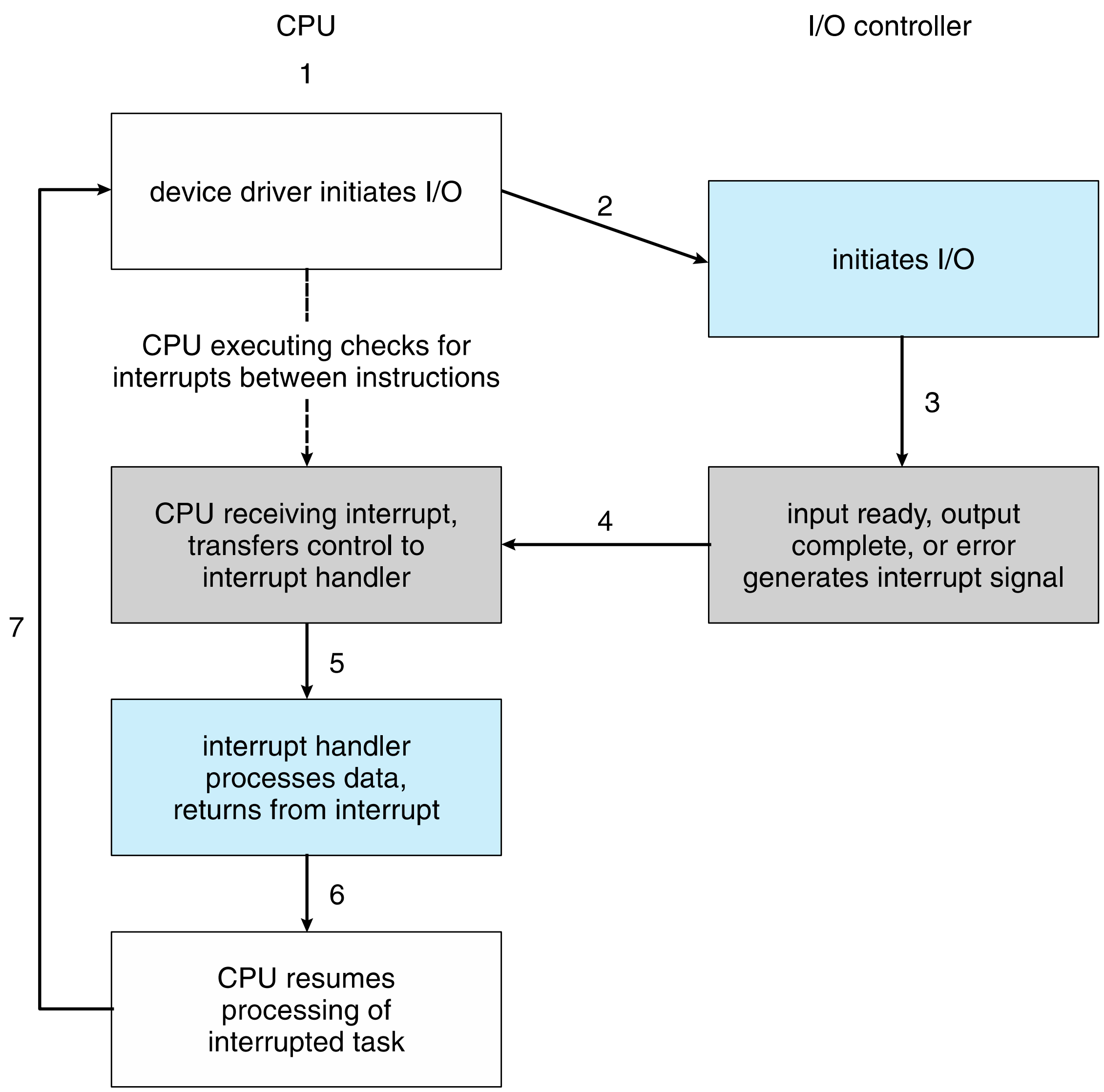
OS manages memory



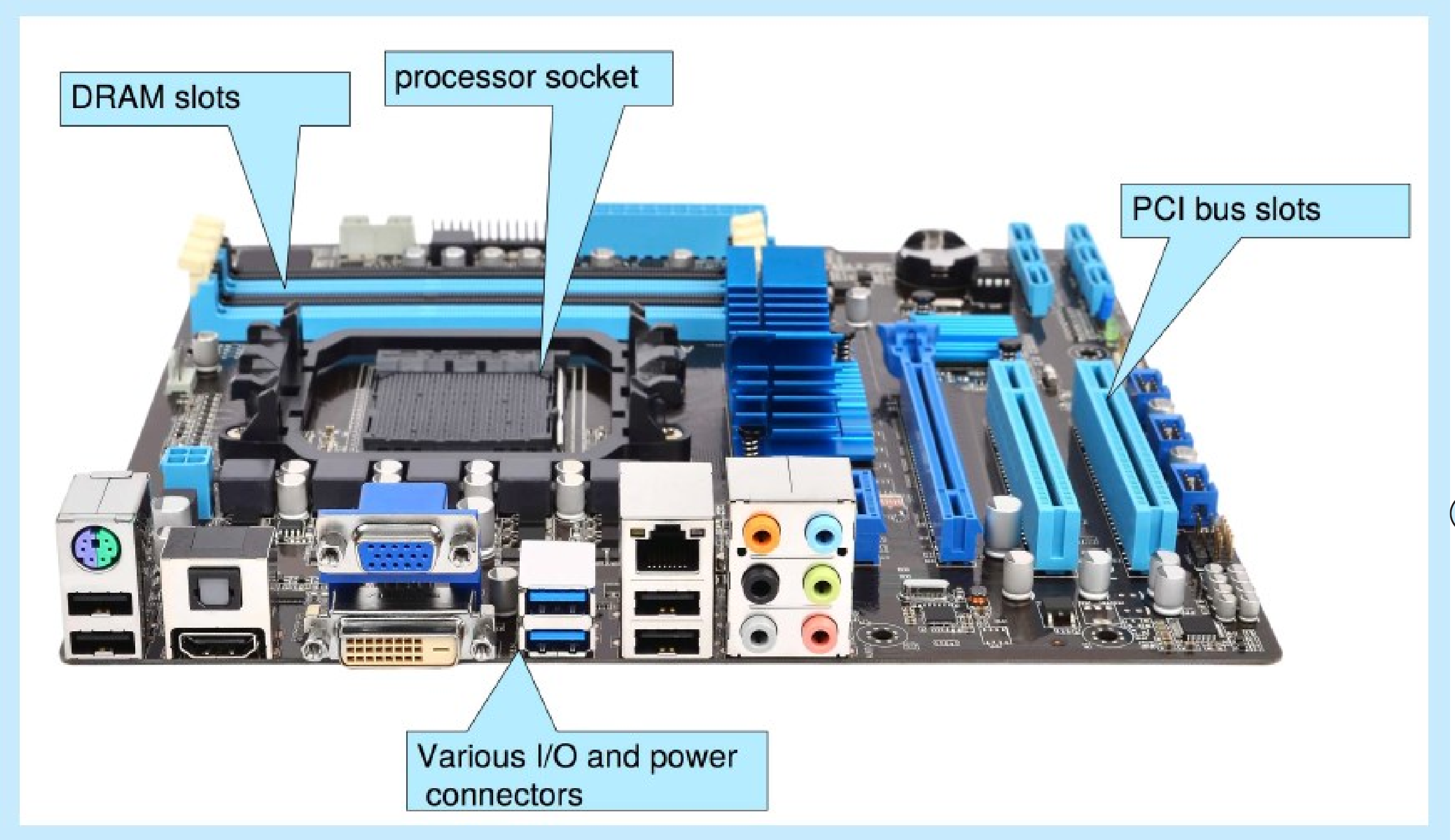
Storage-Device Hierarchy

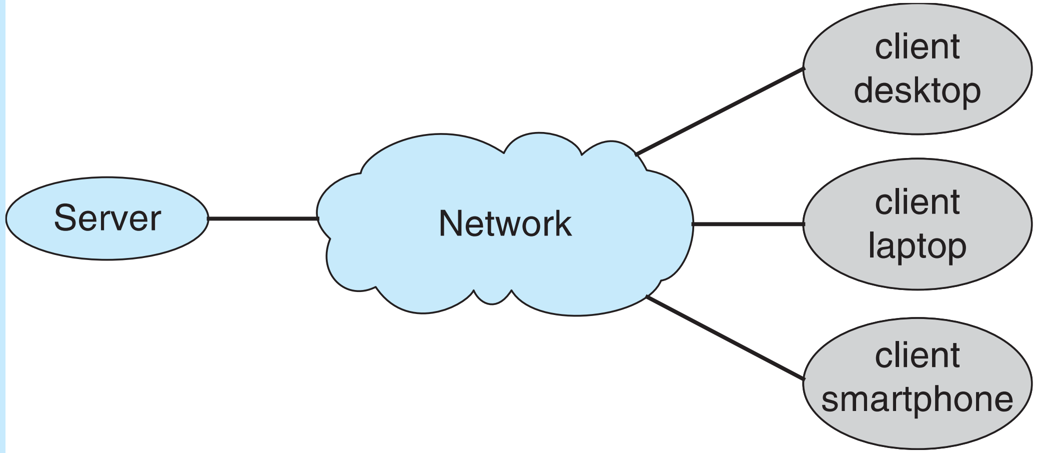
OS manages devices

- OS has code to manage disk, network card, and other external devices: device drivers.
- Device driver talks the language of the hardware devices
  - Issues instructions to devices (fetch data from a file)
  - Responds to interrupt events from devices (user has pressed a key on keyboard).
- Persistent data organized as a filesystem on disk

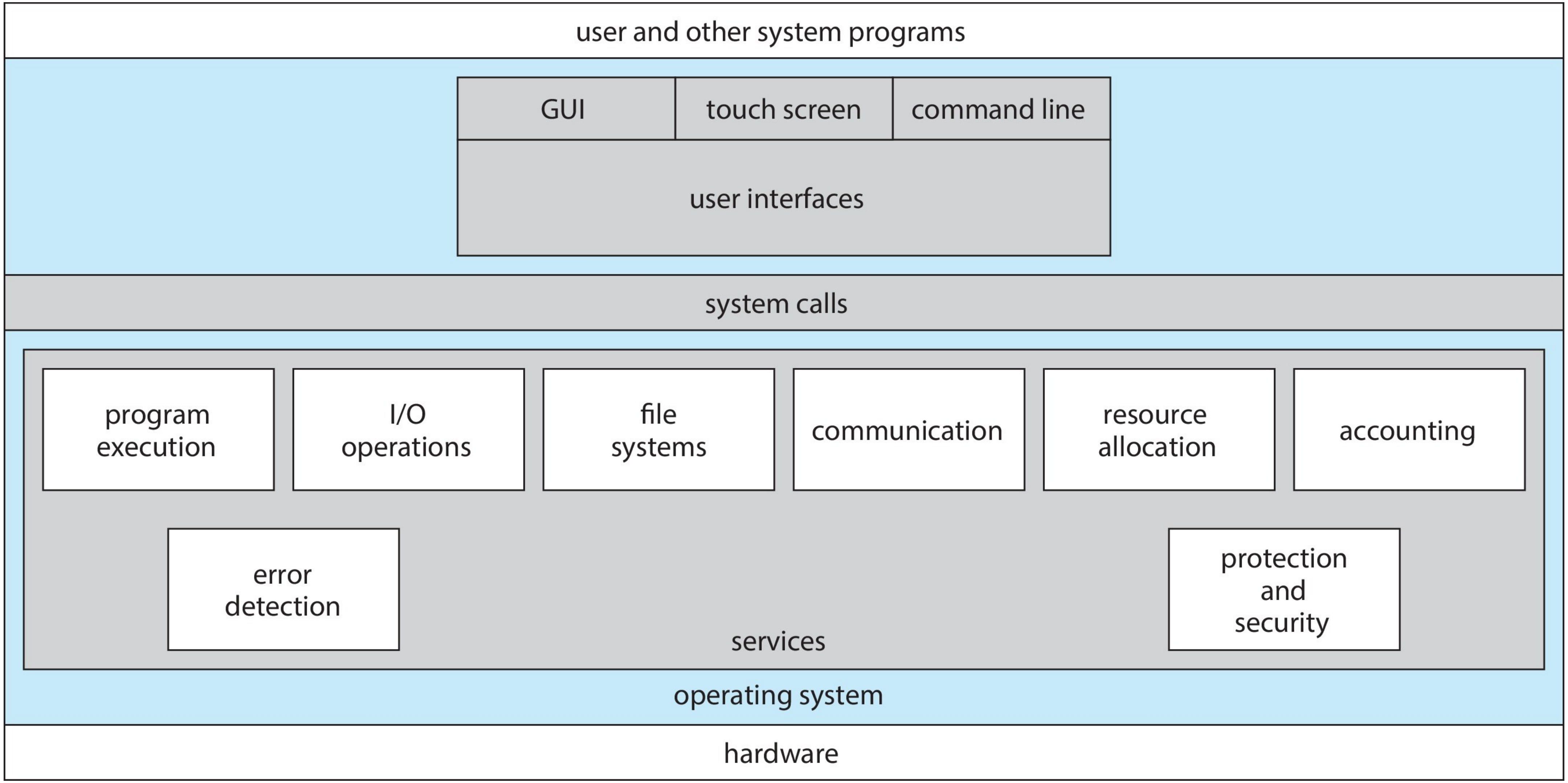


PC Motherboard and Client Server





Operating System Services



OS History

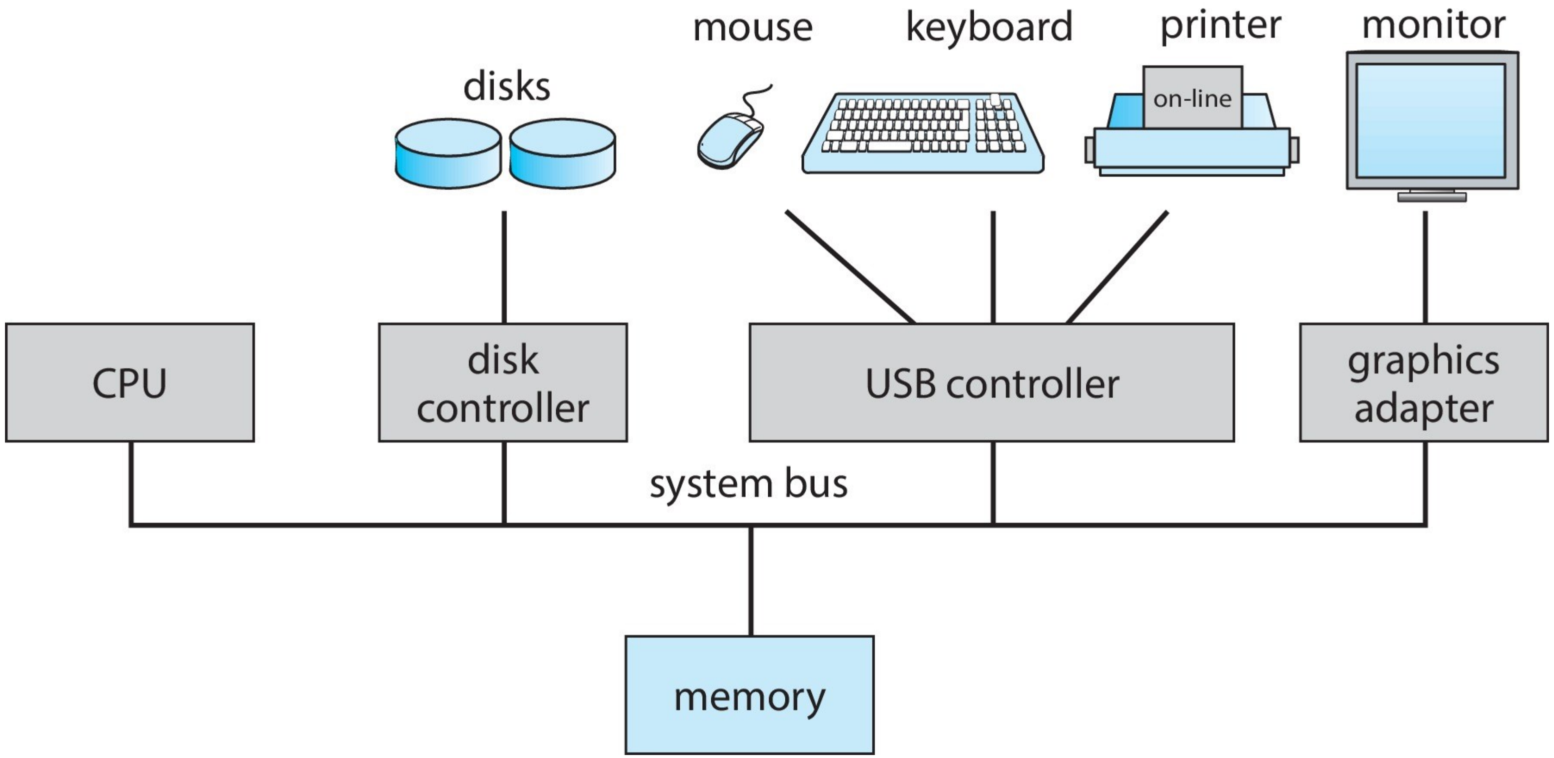
- Started out as a library to provide common functionality across programs
- Later, evolved from procedure call to system call: what's the difference?
- When a system call is made to run OS code, the CPU executes at a higher privilege level.
- Evolved from running a single program to multiple processes concurrently
- Convenience, abstraction of hardware resources for user programs.
- Efficiency of usage of CPU, memory, etc.
- Isolation between multiple processes

Program and Process in OS

- Computer System Organization
- System, Application, Middleware, Bootstrap
- Application Programming Interface (API)
- System Calls to copy contents between files
- Program and Process
- Function call, Parent process, and Child Process

Computer System Organization

- Computer-system operation
  - One or more CPUs, device controllers connect through common **bus** providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles

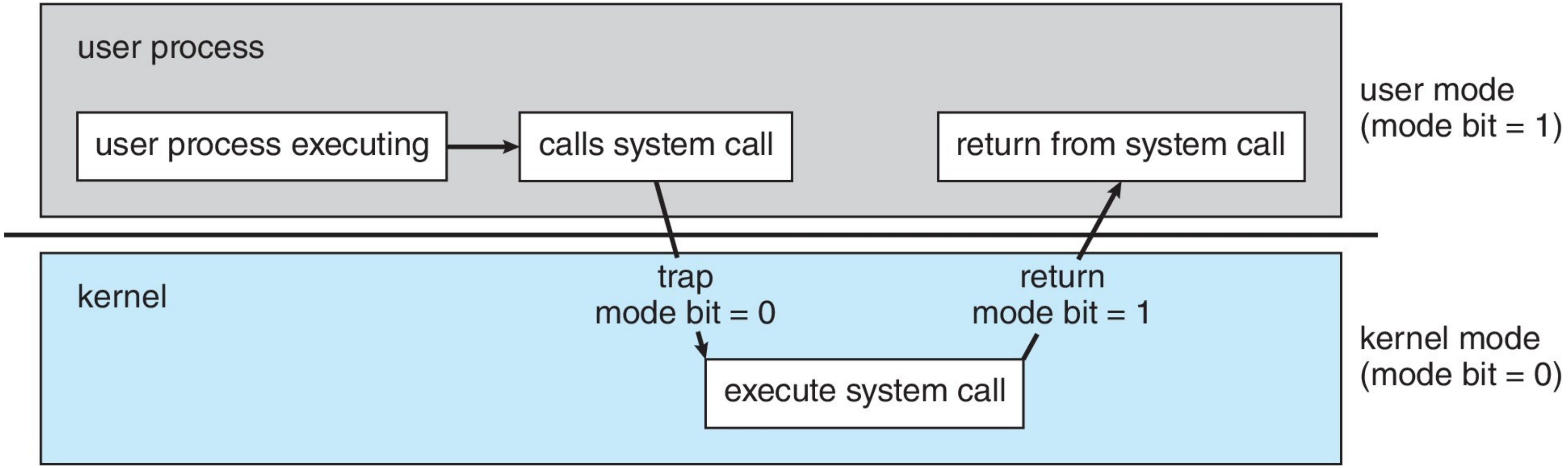


System, Application, Middleware, Bootstrap

- Everything else is either
  - A ***system program*** (ships with the operating system, but not part of the kernel) , or
  - An ***application program***, all programs not associated with the operating system
- A ***middleware*** is a set of software frameworks that provide additional services to application developers such as databases, multimedia, graphics.
- ***Bootstrap program*** is loaded at power-up or reboot
  - Typically stored in ROM or EPROM, generally known as ***firmware***
  - Initializes all aspects of system
  - Loads operating system kernel and starts execution

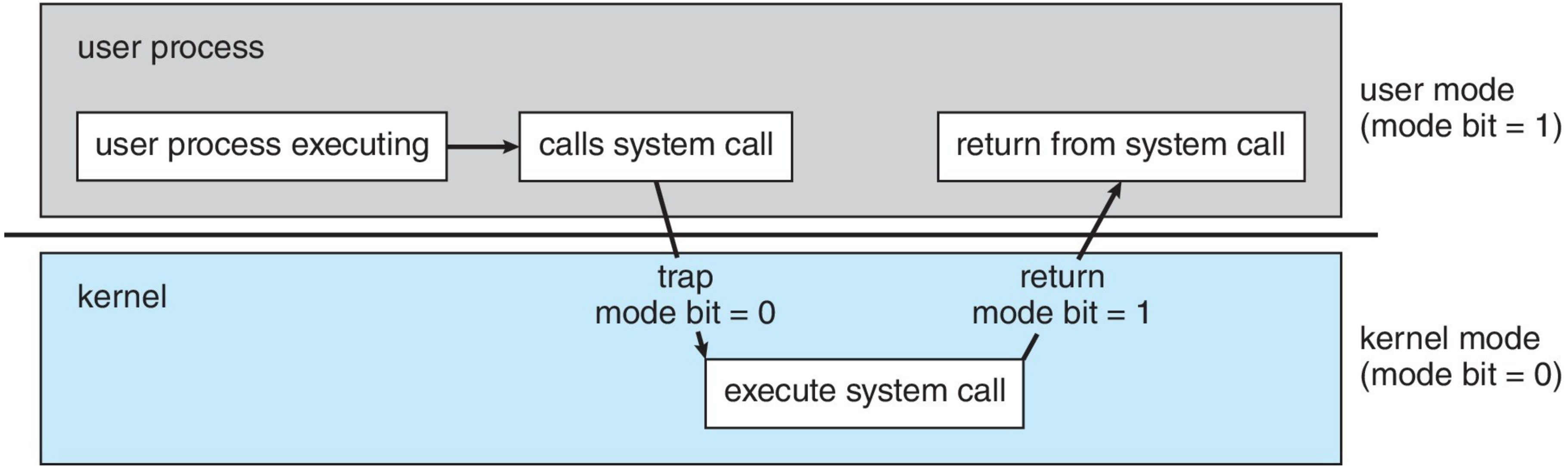
Application Programming Interface (API)

- Request for operating system service – **system call**
- Functions available to write user programs
- API provided by OS is a set of "System Calls"
  - Function call into OS code that runs at a higher privilege level of the CPU
  - Sensitive operations (e.g., access to hardware) are allowed to a higher privilege level
  - "Blocking" system calls cause the process to be blocked



Application Programming Interface (API)

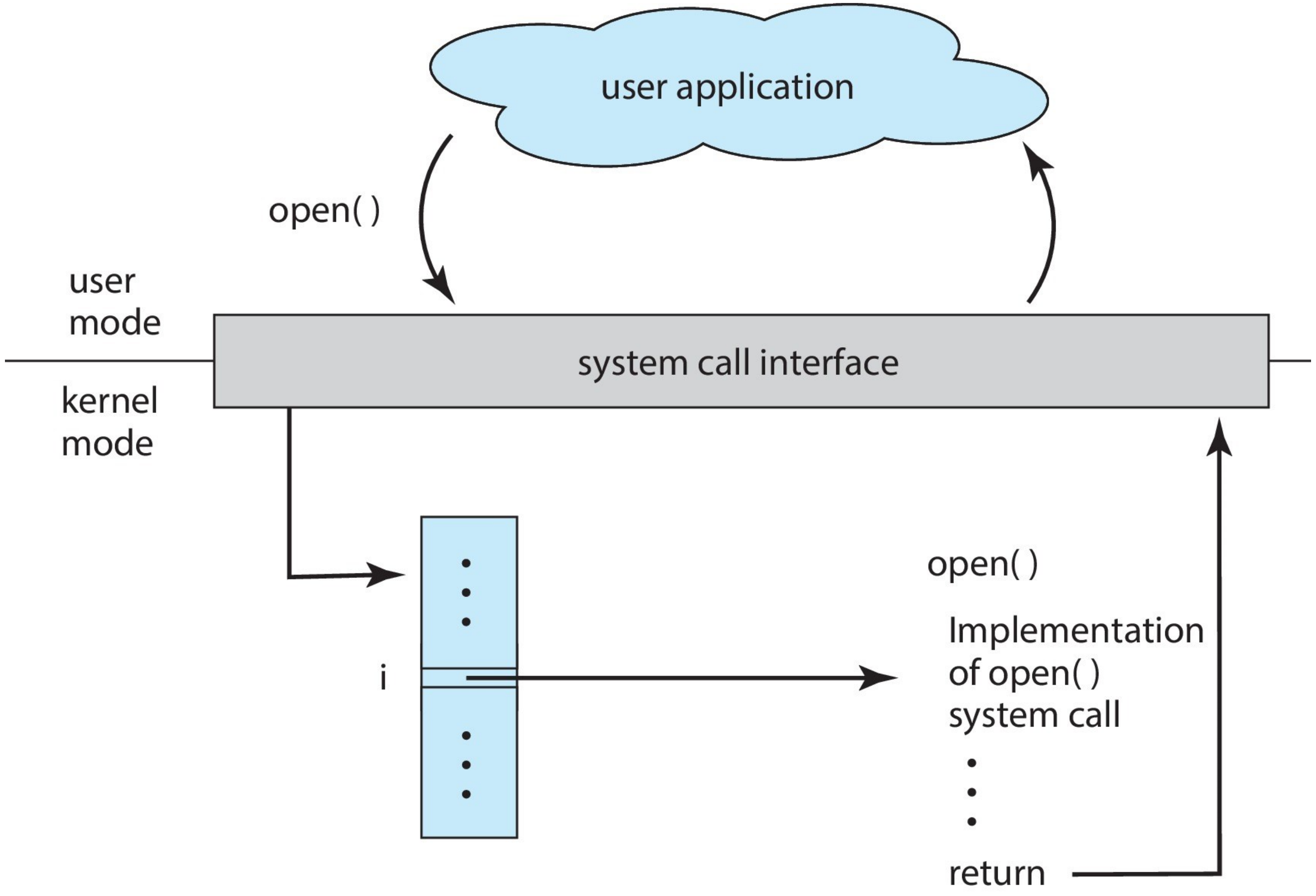
- CPU hardware has multiple privilege levels:
  - One to run user code: user mode
  - One to run OS code like system calls: kernel mode
  - Some instructions execute only in kernel mode
- Kernel does not trust user stack and user provided addresses
  - Kernel creates a separate kernel stack and Interrupt Descriptor Table (IDT)

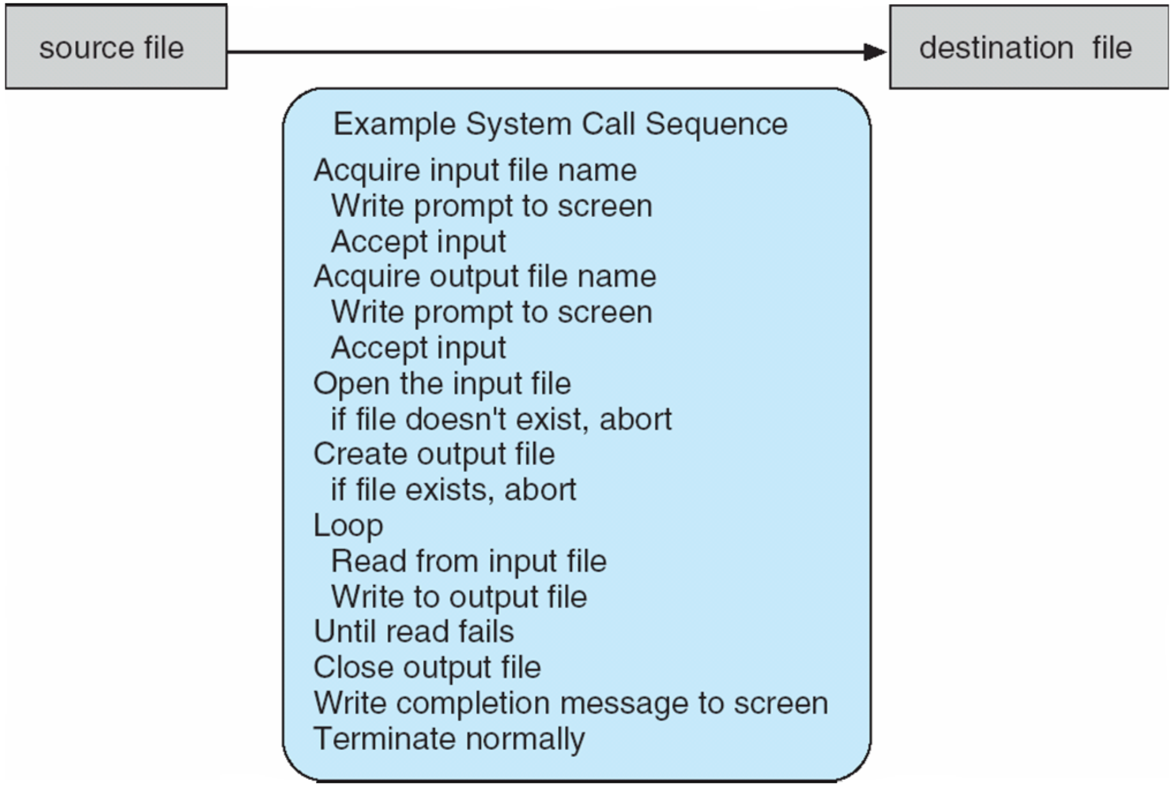


Application Programming Interface (API)

- POSIX API: a standard set of system calls that an OS must implement
  - open, read, write, close, wait, exec, fork, exit, and kill
  - fork() creates a new child process
  - exec() makes a process execute a given executable
  - exit() terminates a process
  - wait() causes a parent to block until child terminates
- Program language libraries hide the details of invoking system calls
  - C program → libraries → system calls

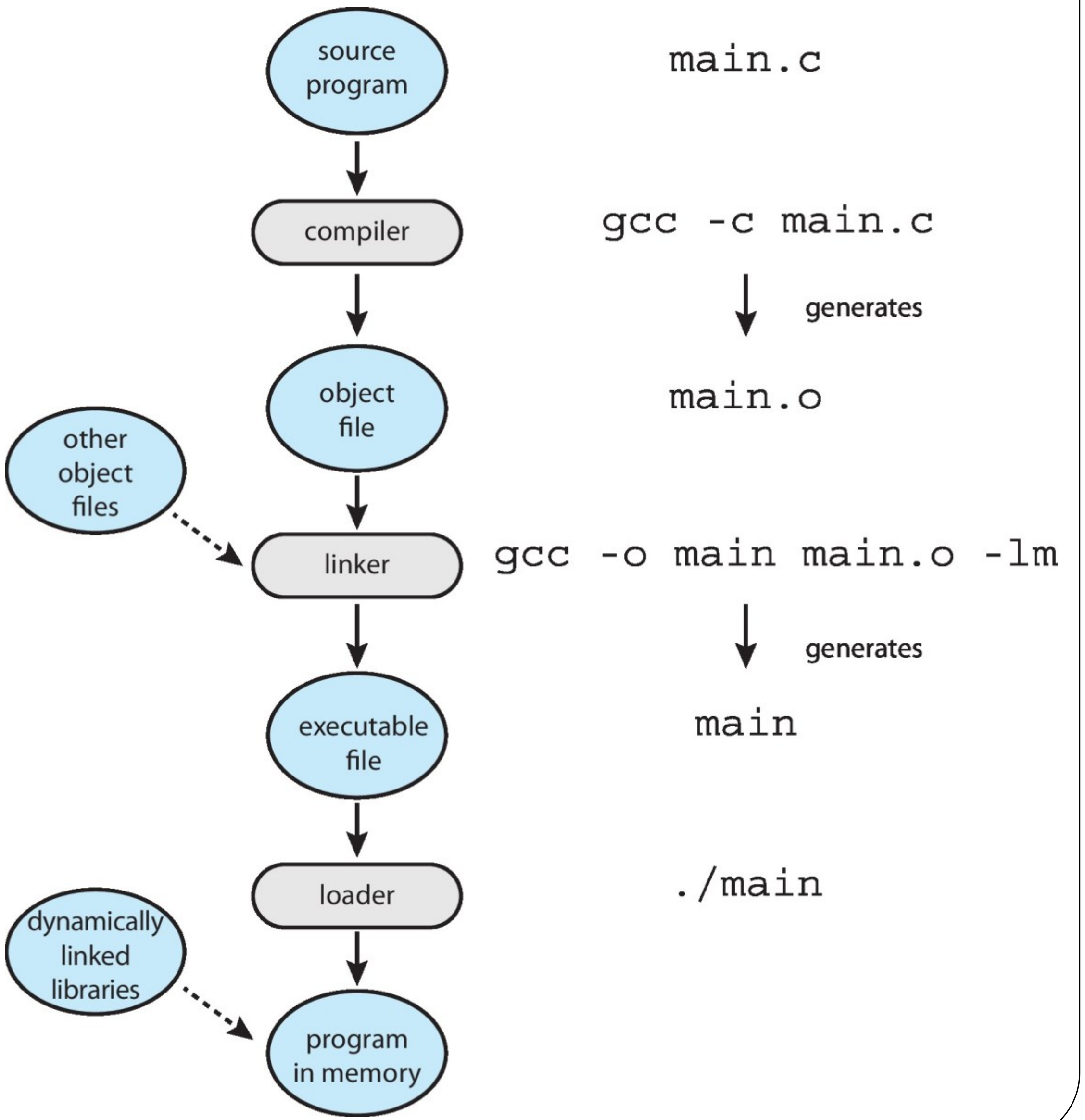
System Calls to copy contents between files





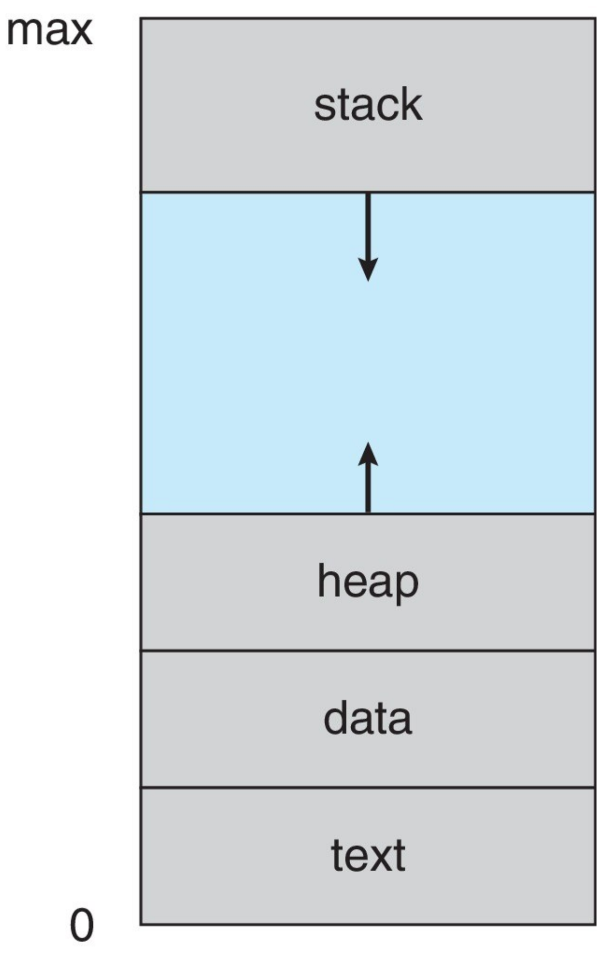
Program and Process

- When you run an exe file, the OS creates a process = a running program
- OS timeshares CPU across multiple processes: virtualizes CPU
- OS has a CPU scheduler that picks one of the many active processes to execute on a CPU
  - Policy: which process to run
  - Mechanism: how to "context switch" between processes



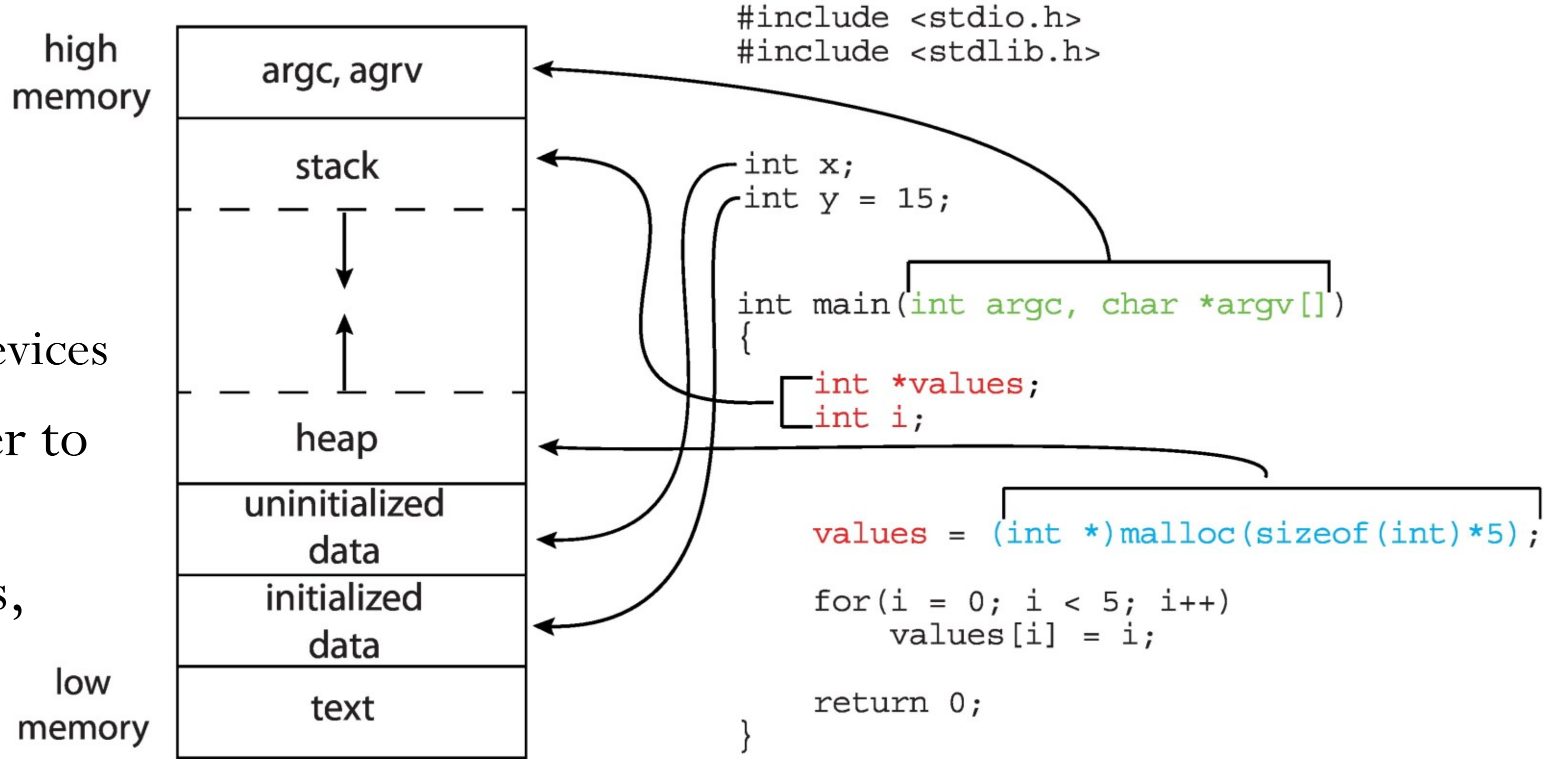
Program and Process

- OS allocates memory and creates memory image
  - Loads code, data from disk exe
  - Creates runtime stack, heap
  - Opens basic files – STD IN, OUT, ERR
  - Initializes CPU registers – PC points to first instruction
- Memory image
  - Code & data (static)
  - Stack and heap (dynamic)



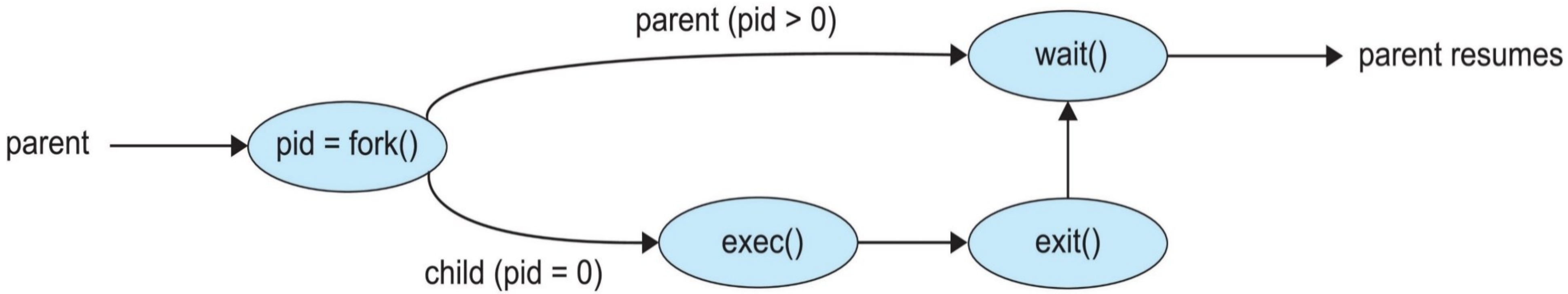
Program and Process

- A unique identifier (PID)
- CPU context: registers
  - Program counter
  - Current operands
  - Stack pointer
- File descriptors
  - Pointers to open files and devices
- Points CPU program counter to current instruction – Other registers may store operands, return values etc.



Function call, Parent process, and Child Process

- **Function call** is different from Parent and Child Processes
- A function call translates to a jump instruction
  - pushed Stack Frame containing old values of PC (return value, function arguments etc.) about the callee function to stack
  - then PC updated to new value of called function
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)



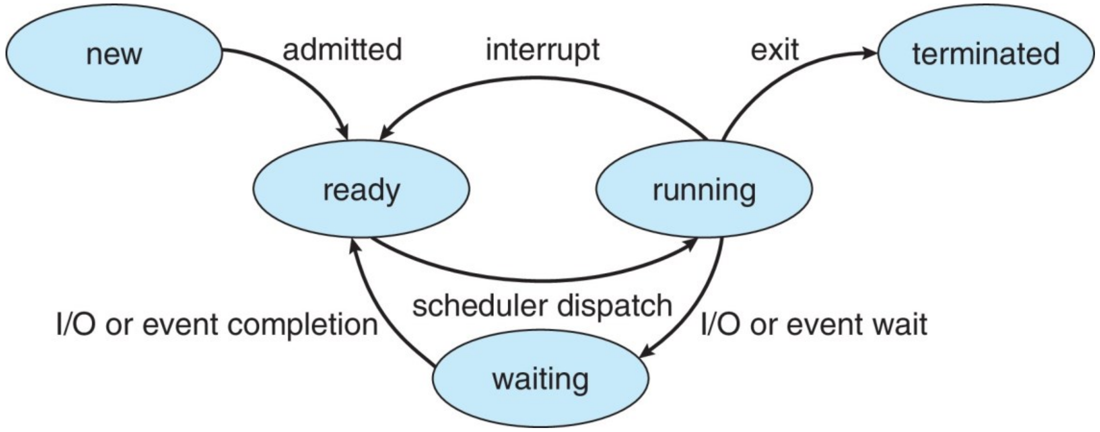
CPU and Process Scheduling

- Process states
- Process Control Block (PCB)
- CPU Switch From Process to Process
- Mechanism of Context Switch (Dispatcher)
- CPU Scheduler optimization
- OS scheduler
- Scheduling policy (FCFS, SJF, SRTF, RR, Priority, and Real-time)

Process states

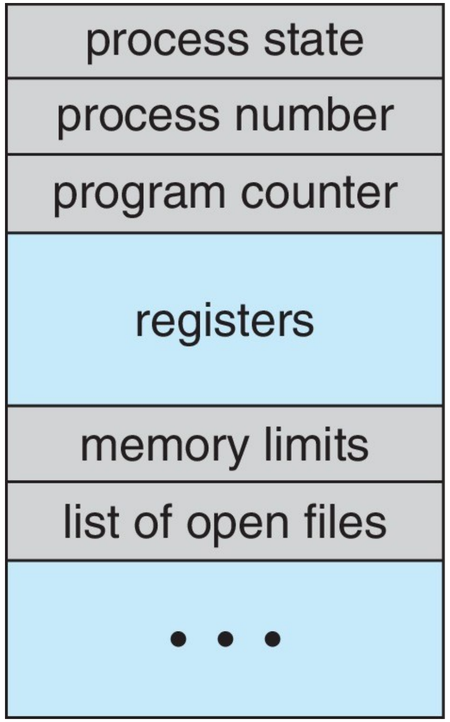
As a process executes, it changes **state**

- **New**: The process is being created
- **Running**: Instructions are being executed
- **Waiting/Blocked**: The process is waiting for some event to occur
  - Example: Disk issues an interrupt when data is ready
- **Ready**: The process is waiting to be assigned to a processor
- **Terminated**: The process has finished execution



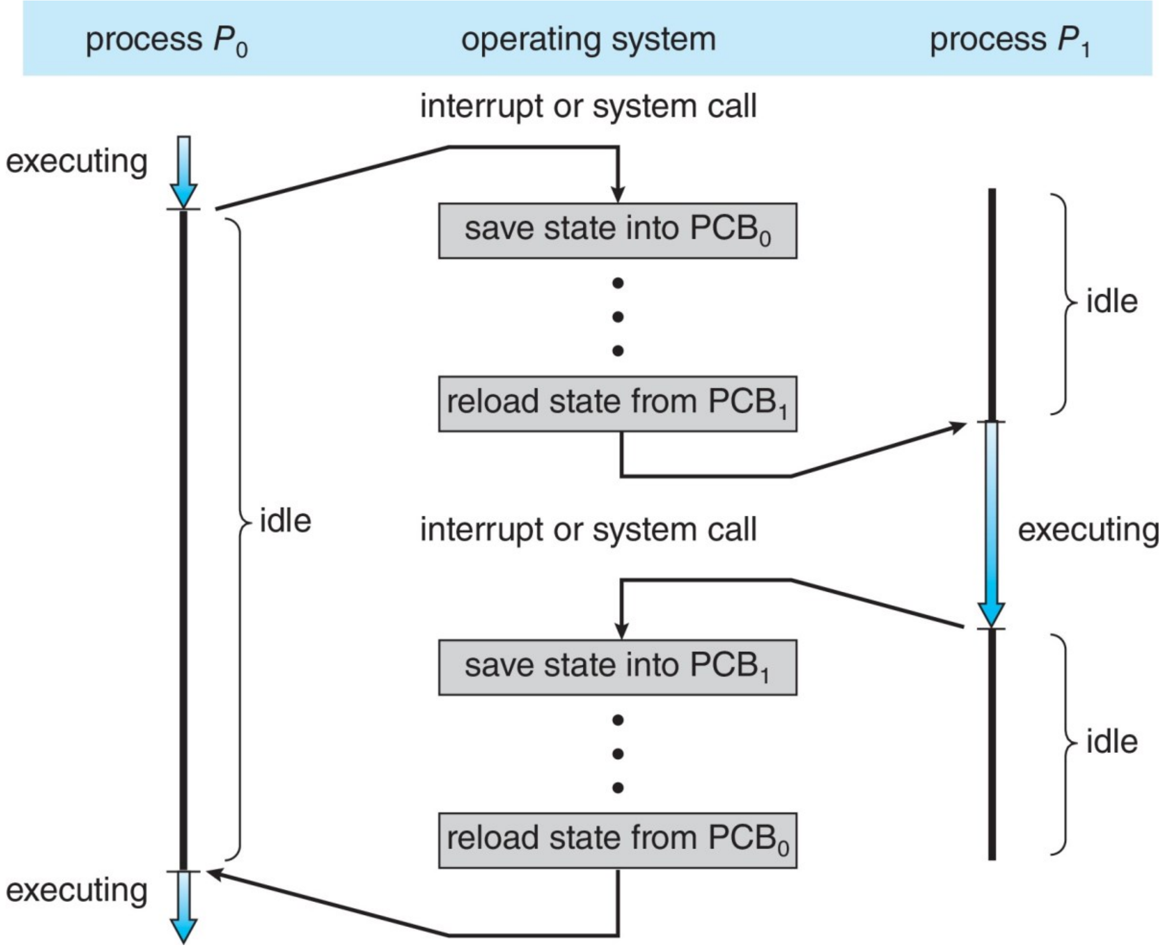
Process Control Block (PCB)

- Information associated with each process(also called **task control block**)
- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information
  - memory allocated to the process
- Accounting information –
  - CPU used,
  - clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



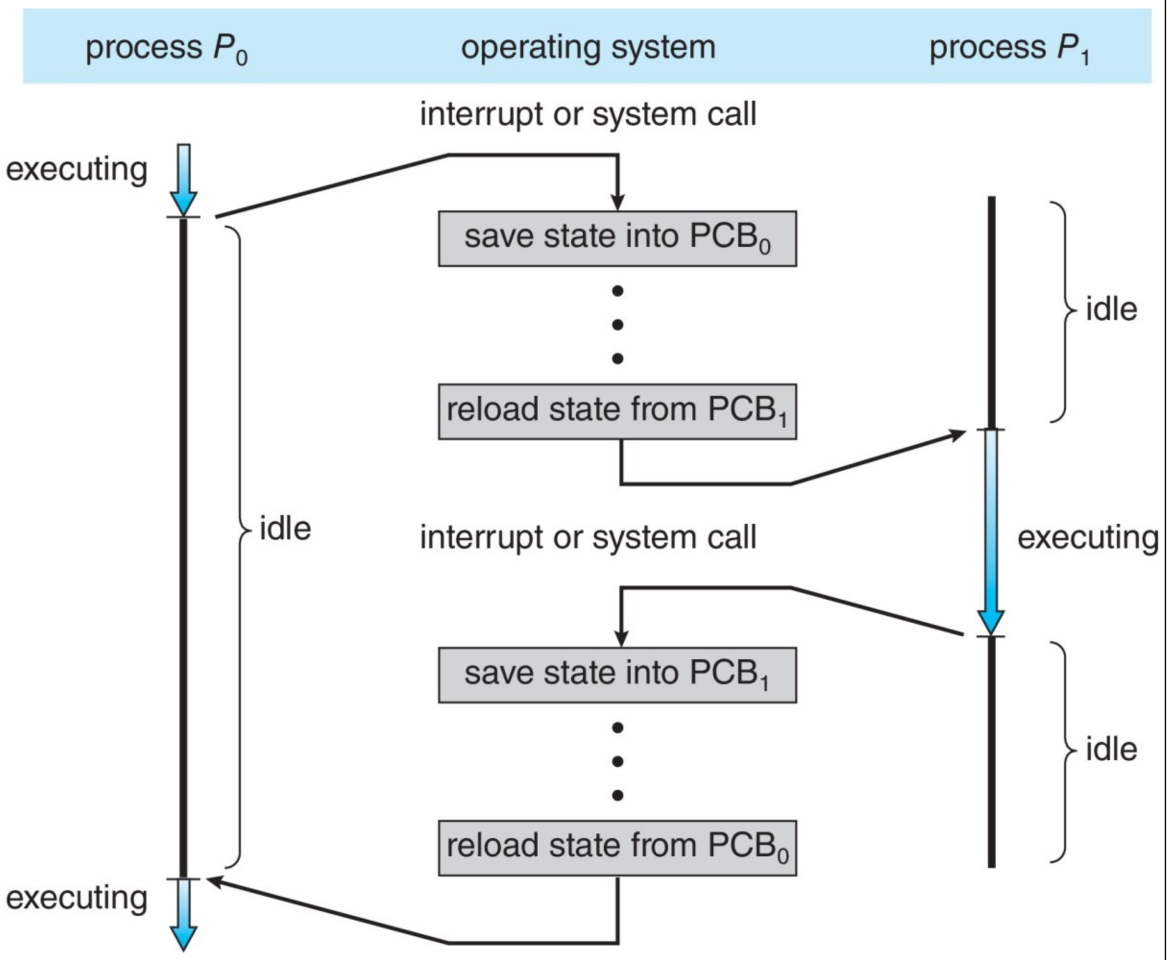
CPU Switch From Process to Process

- A **context switch** occurs when the CPU switches from one process to another.
- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process mentioned in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
- Time dependent on hardware support



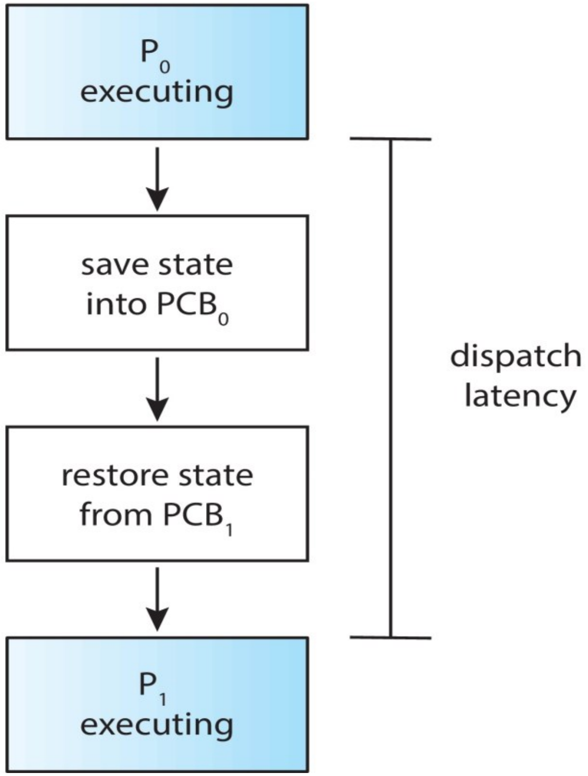
Mechanism of Context Switch

- Example: process P_0 has moved from user to kernel mode, OS decides it must switch from P_0 to P_1
- Save context (PC, registers, kernel stack pointer) of P_0 on kernel stack
- Switch SP to kernel stack of P_1
- Restore context from P_1's kernel stack
- OS already saved registers on P_1's kernel stack, when it switched out P_1 in the past
- Now, CPU is running P_1 in kernel mode, then CPU switch to user mode of P_1



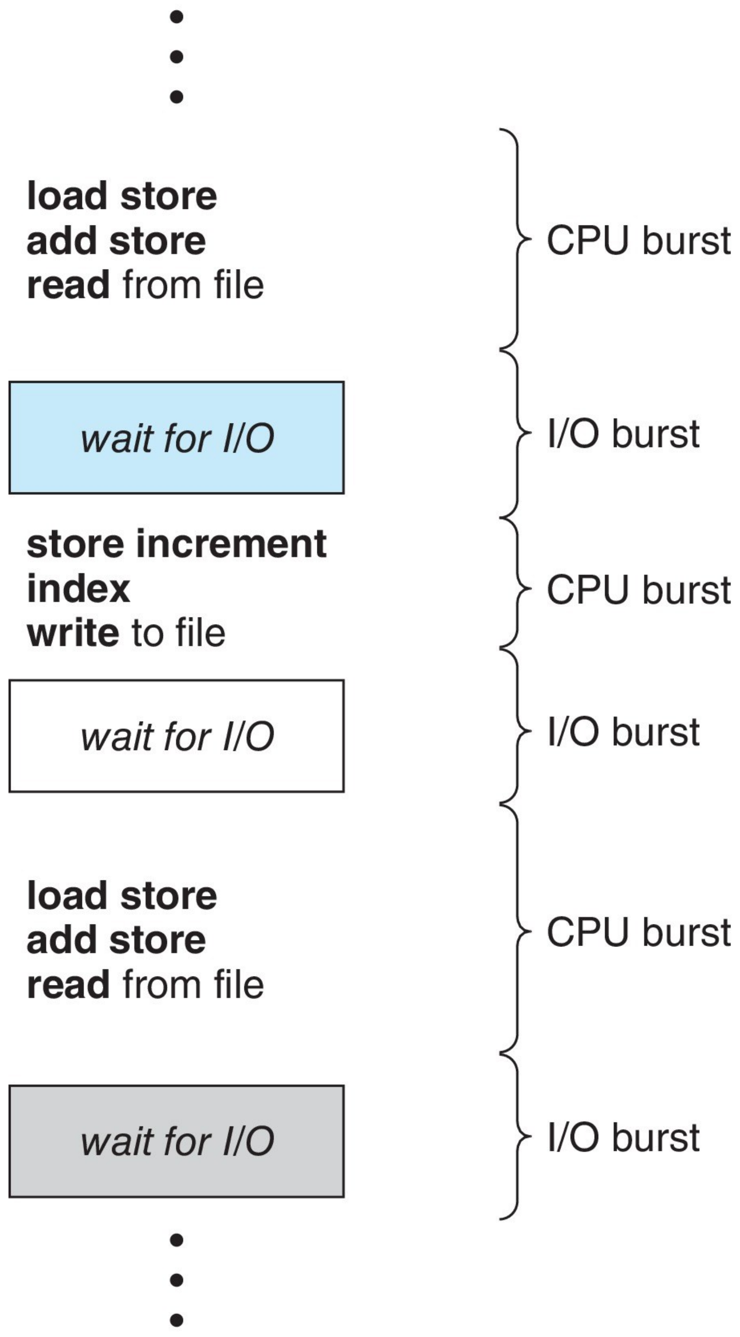
Mechanism of Context Switch (Dispatcher)

- Dispatcher module gives control of the CPU to the process selected by the CPU scheduler; this involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running



CPU Scheduler optimization

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates



OS scheduler

- OS scheduler has two parts
  - Policy to pick which process to run
  - Mechanism to switch to that process
- **Non preemptive (cooperative) schedulers**: once the CPU has been allocated to a process, the process keeps the CPU
  - Switch only if process blocked or terminated
- **Preemptive (non-cooperative):** schedulers can switch even when process is ready to continue
  - CPU generates periodic timer interrupt
  - After servicing interrupt, OS checks if the current process has run for too long

Scheduling policy

- On context switch, which process to run next, from set of ready processes?
- OS scheduler schedules the CPU requests (bursts) of processes
  - CPU burst = the CPU time used by a process in a continuous stretch
  - If a process comes back after I/O wait, it counts as a fresh CPU burst
- Optimize
  - Maximize (utilization = fraction of time CPU is used)
  - Minimize average (turnaround time = time from process arrival to completion)
  - Minimize average (response time = time from process arrival to first scheduling)
  - Fairness: all processes must be treated equally
  - Minimize overhead: run process long to reduce context switch (~1 microsecond)

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
- Turnaround times tend to be high

First- Come, First-Served (FCFS) Scheduling

- Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

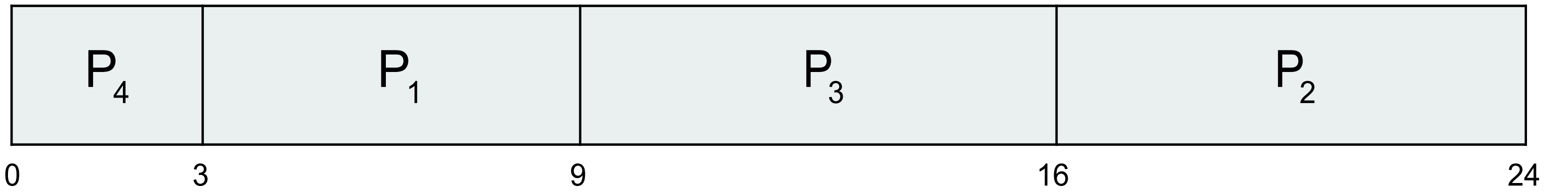
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF or Shortest Job Next (SJN) is optimal – gives minimum average waiting time for a given set of processes
- Preemptive version called **shortest-remaining-time-first**
- How do we determine the length of the next CPU burst?
  - Could ask the user
  - Estimate
- Provably optimal when all processes arrive together.
- SJF is non- preemptive, so short jobs can still get stuck behind long ones.

Shortest-Job-First (SJF) Scheduling

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Shortest Remaining Time First Scheduling

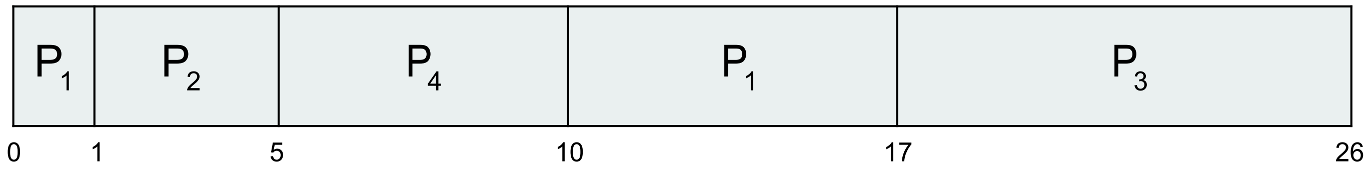
- Preemptive version of SJF
- Whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJF algorithm.
- Is SRT more "optimal" than SJF in terms of the minimum average waiting time for a given set of processes?
- Also called Shortest Time-to-Completion First (STCF)
- Preemptive scheduler
- Preempts running task if time left is more than that of new arrival

Shortest Remaining Time First Scheduling

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5

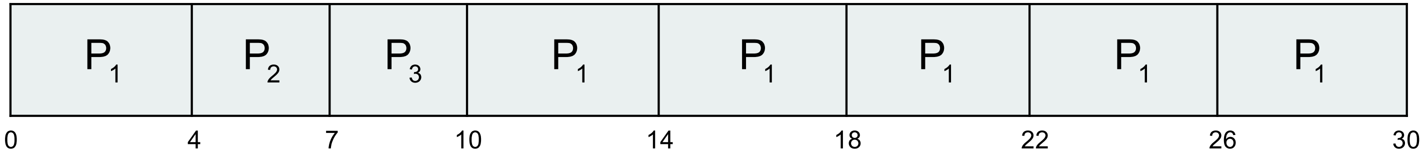
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- Every process executes for a fixed quantum slice
- Preemptive
- Good for response time and fairness
- Bad for turnaround time
- Timer interrupts every quantum to schedule next process
- Performance
  - q large $\Rightarrow$ FIFO (FCFS)
  - q small $\Rightarrow$ RR
- Slice big enough q with respect to context switch, otherwise overhead is too high

Round Robin (RR) with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

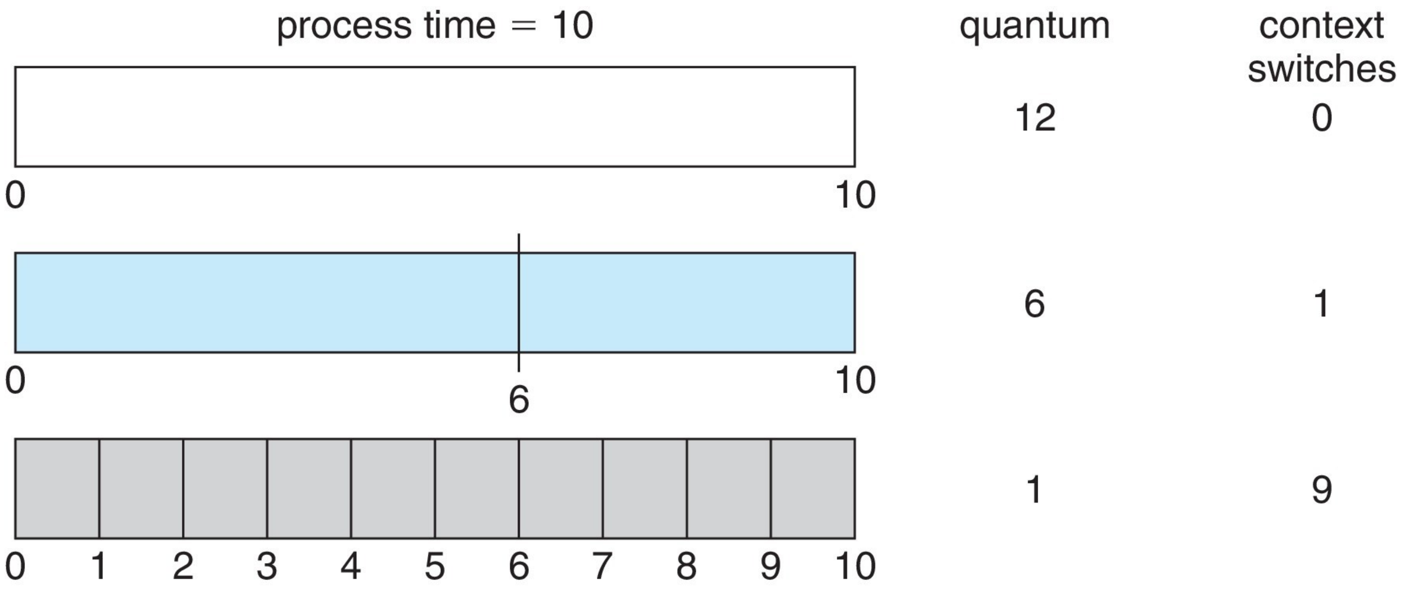
- The Gantt chart is:



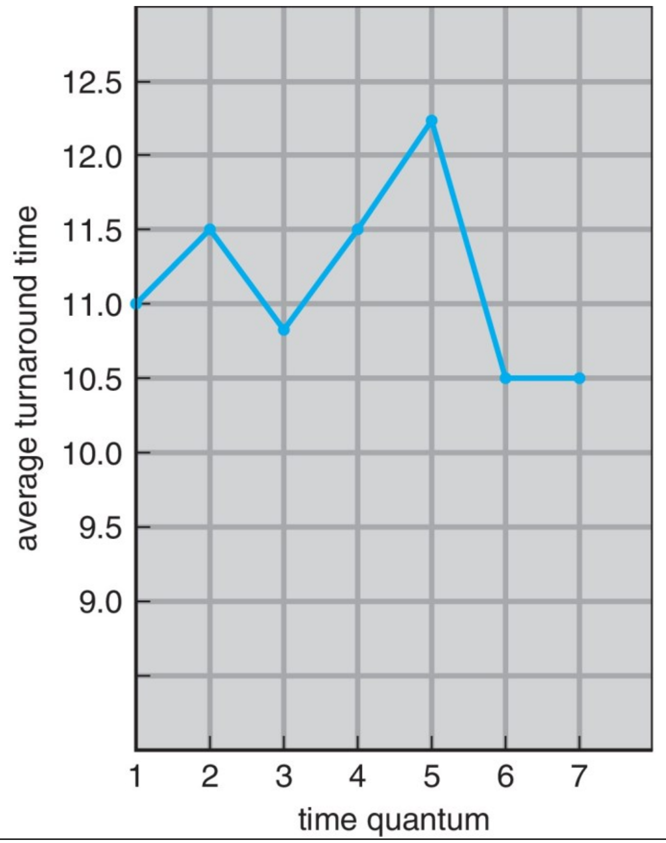
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

Round Robin (RR) with varying Time Quantum

- Time Quantum and Context Switch Time



- Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

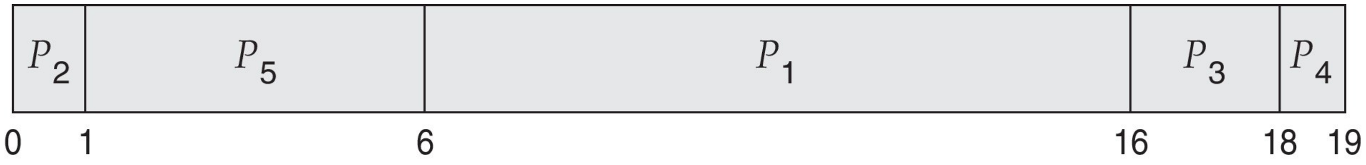
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Non-preemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart

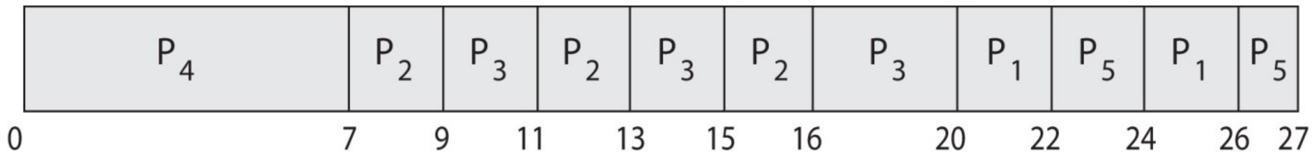


- Average waiting time = 8.2

Priority Scheduling v/s Round-Robin

- Run the process with the highest priority. Processes with the same priority run round-robin
- Example:

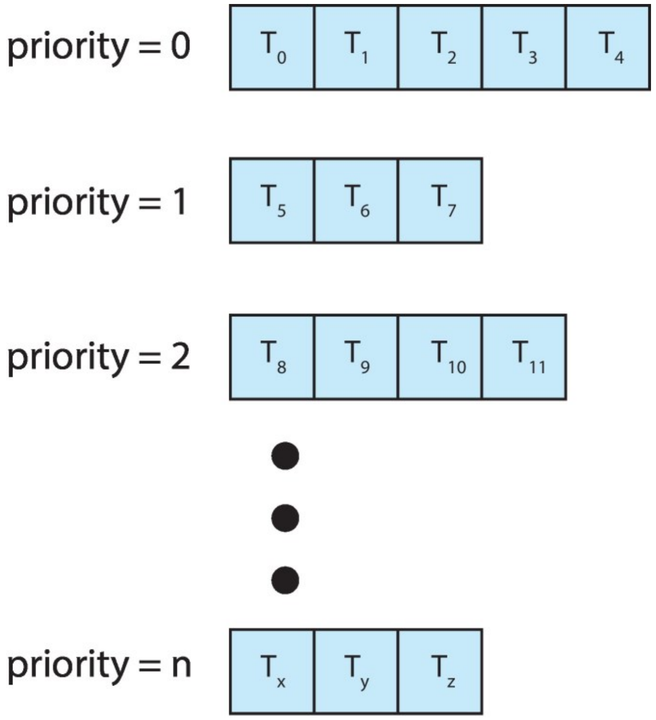
Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3



- Gantt Chart with time quantum = 2

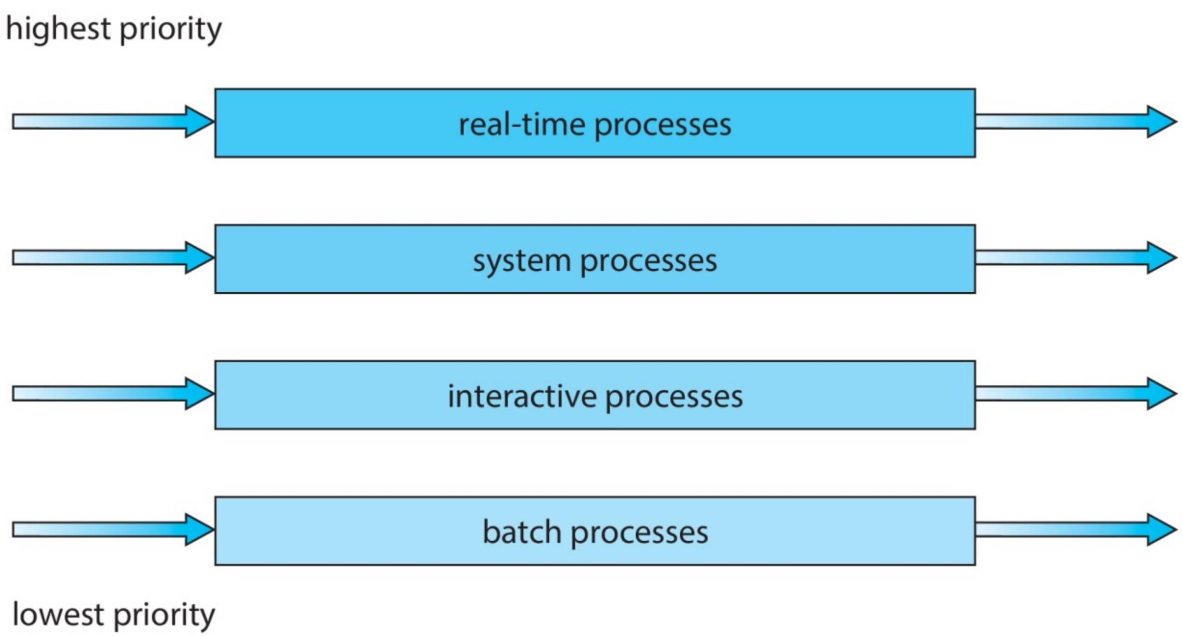
Multilevel Queue

- The ready queue consists of multiple queues
- Multilevel queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine which queue a process will enter when that process needs service
  - Scheduling among the queues
- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



Priority Multilevel Queue

- Prioritization based upon process type



Real-Time CPU Scheduling

- Real schedulers are more complex
- Can present obvious challenges
- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- **Hard real-time systems – task must be serviced by its deadline**

References

- Mythili Vutukur. Lectures on Operating Systems, Department of Computer Science and Engineering, IIT Bombay, https://www.cse.iitb.ac.in/~mythili/os/
- Avi Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating System Concepts (Tenth Edition). https://www.os-book.com/OS10/slide-dir/index.html

ขอบคุณ

Thai

Grazie

Italian

תודה רבה

Hebrew

धन्यवादः

Sanskrit

ಧನ್ಯವಾದಗಳು

Kannada

Ευχαριστώ

Greek

Thank You

English

Gracias

Spanish

Спасибо

Russian

Obrigado

Portuguese

شكرا

Arabic

https://sites.google.com/site/animeshchaturvedi07

Merci

French

多謝

Traditional Chinese

धन्यवाद

Hindi

Danke

German

நன்றி

Tamil

Tamil

多谢

Simplified Chinese

ありがとうございました

Japanese

감사합니다

Korean