Midterm Review

Operating Systems

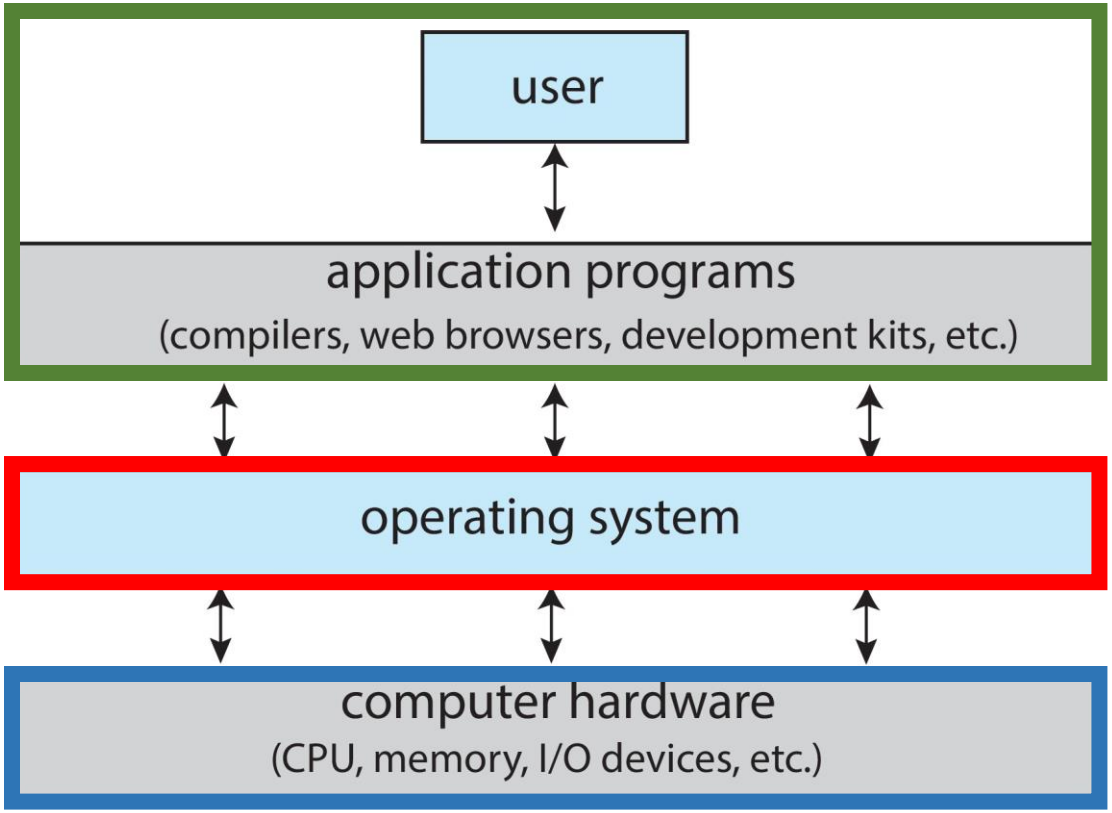
Yu-Ting Wu

Scope of the Midterm

- Chapter 1: Introduction
- Chapter 2: Operating system structure
- Chapter 3: Processes
- Chapter 4: Threads and concurrency
- Chapter 5: CPU scheduling

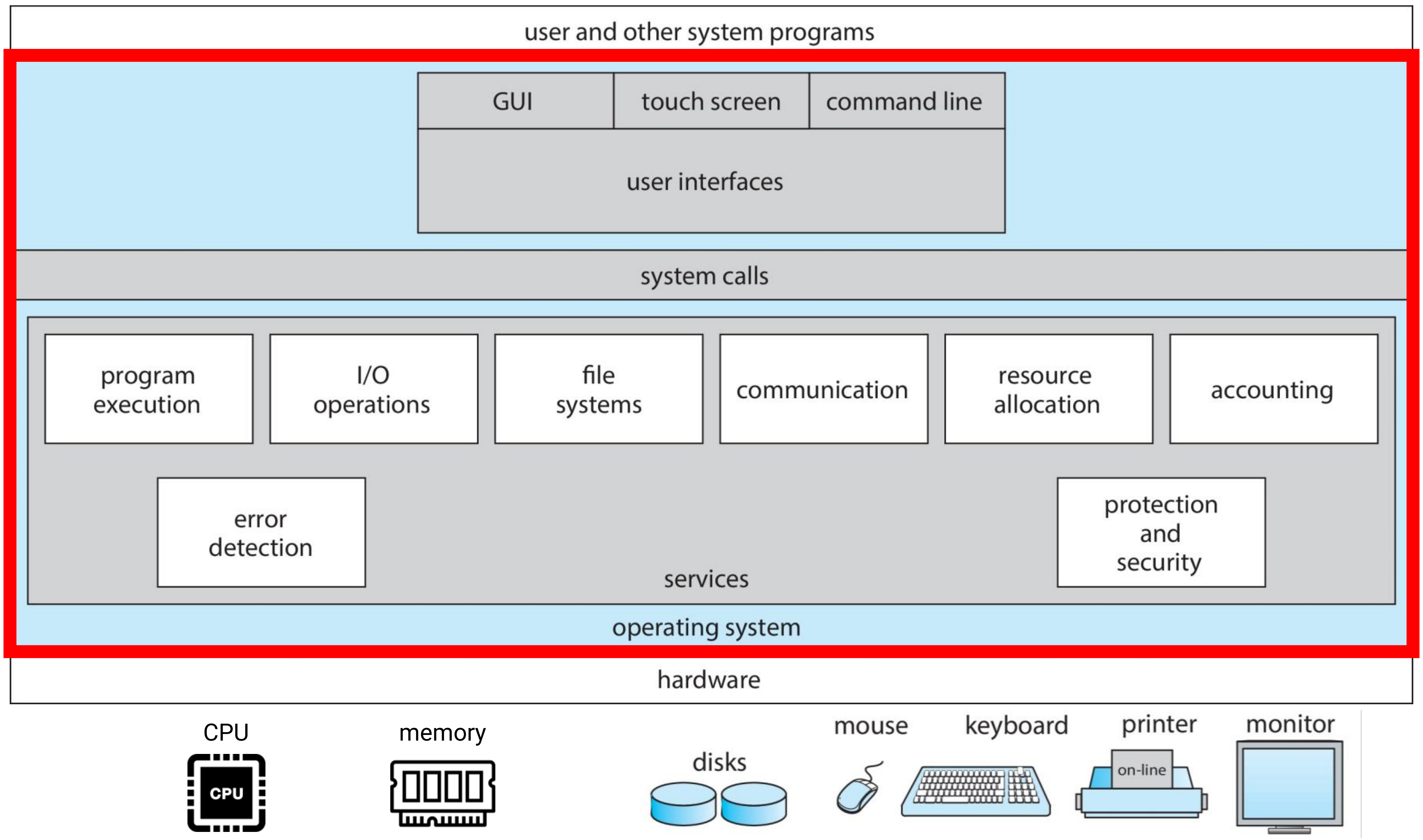
Overview

Goals of an OS

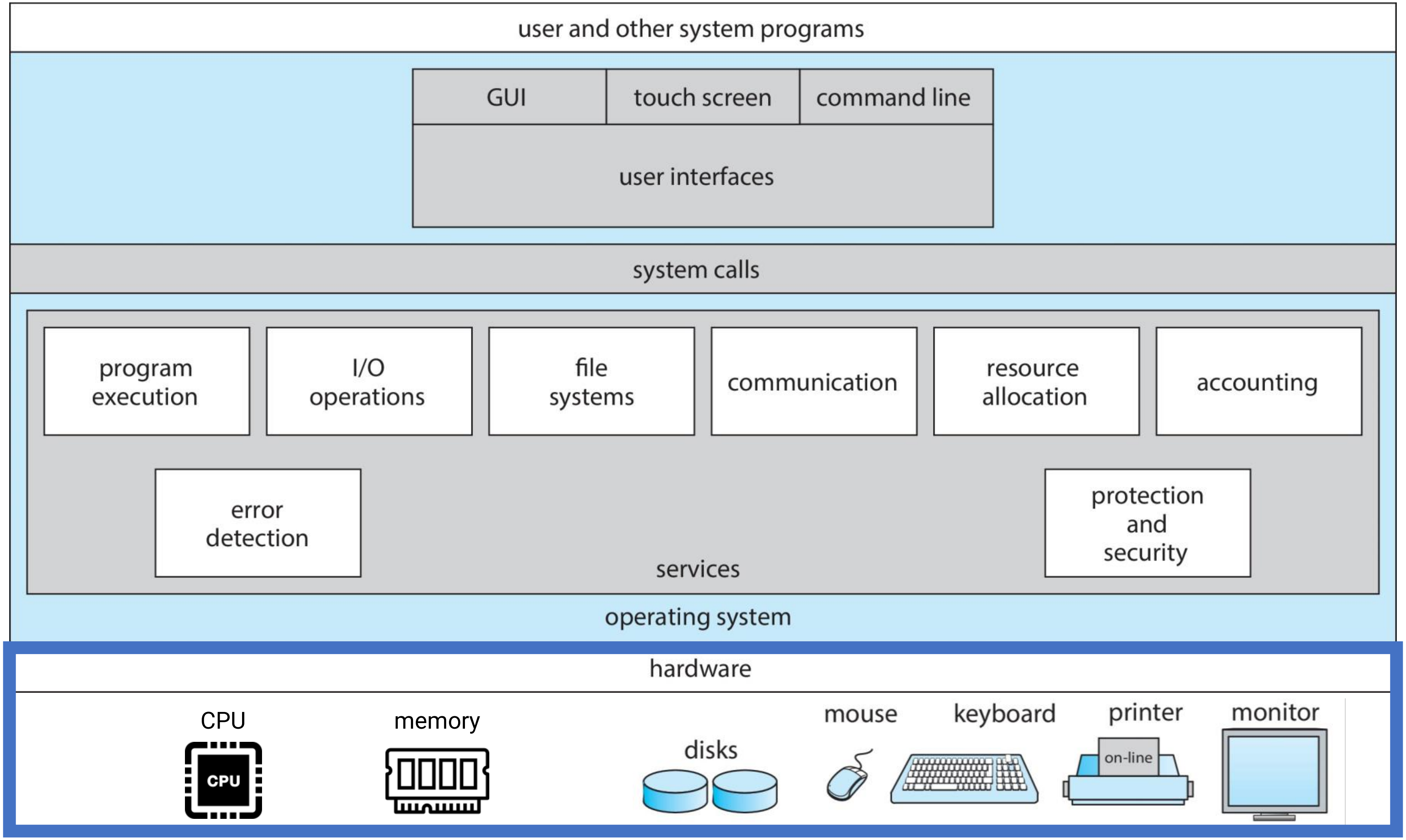


- **Control program**
  - Execute users' program
  - Make the computer system convenient to use
- **Resource allocator**
  - Use the computer hardware in an efficient manner

Operating System Services

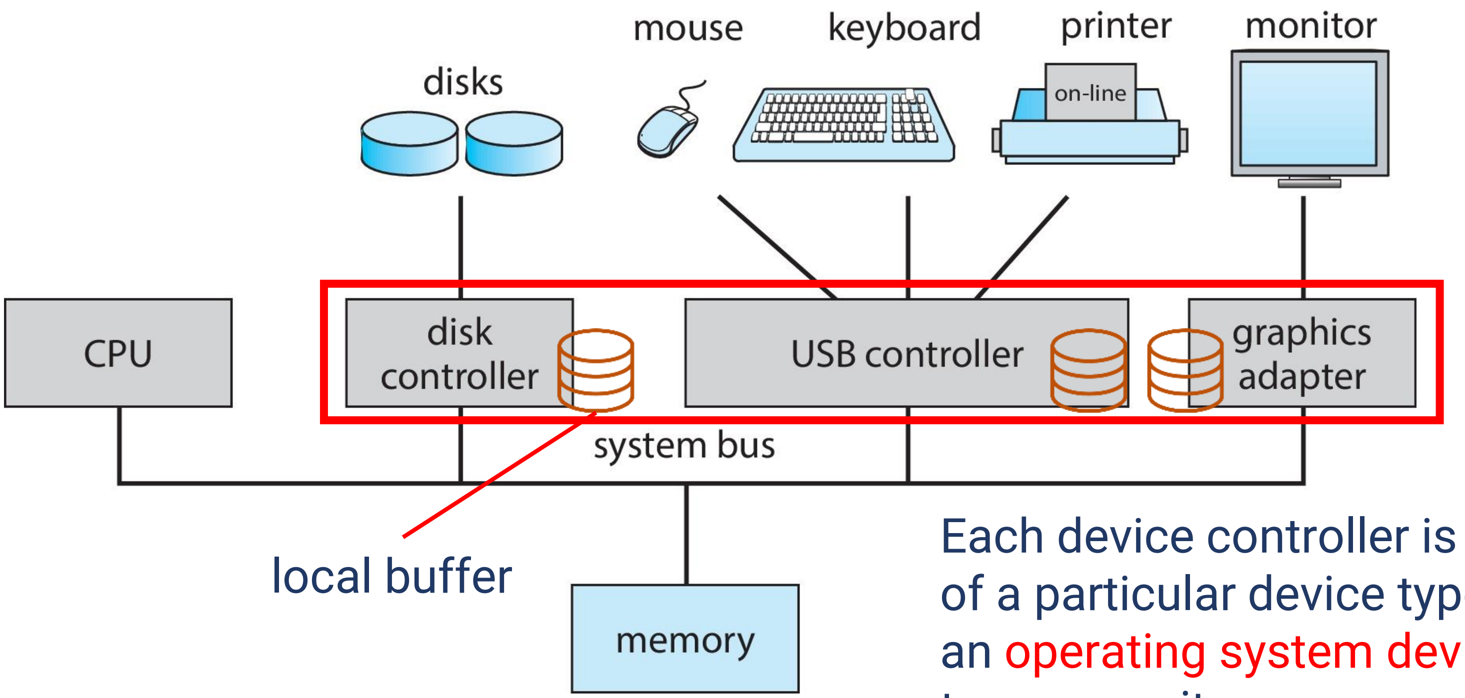


Computer System Organization



Computer System Operations

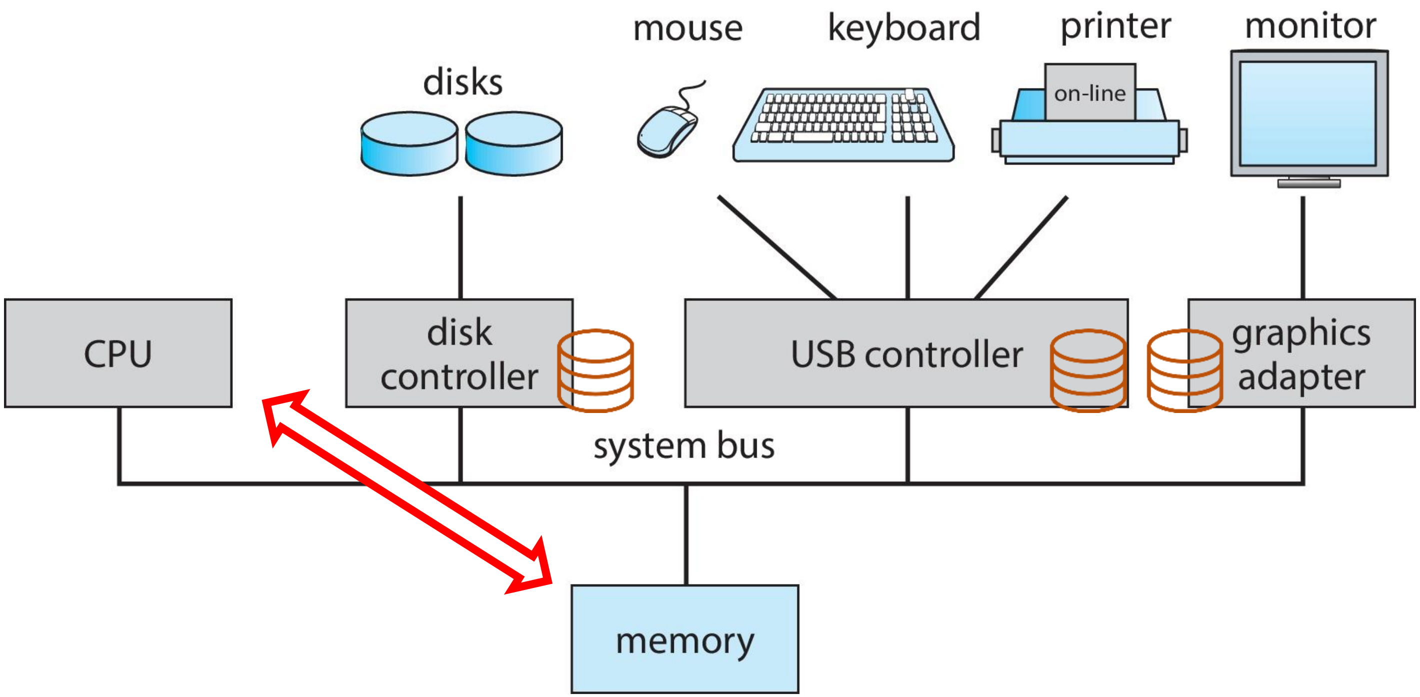
- CPU (or CPUs) and device controllers connect through common **bus**, which provides access to memory



Each device controller is in charge of a particular device type, and has an operating system device driver to manage it

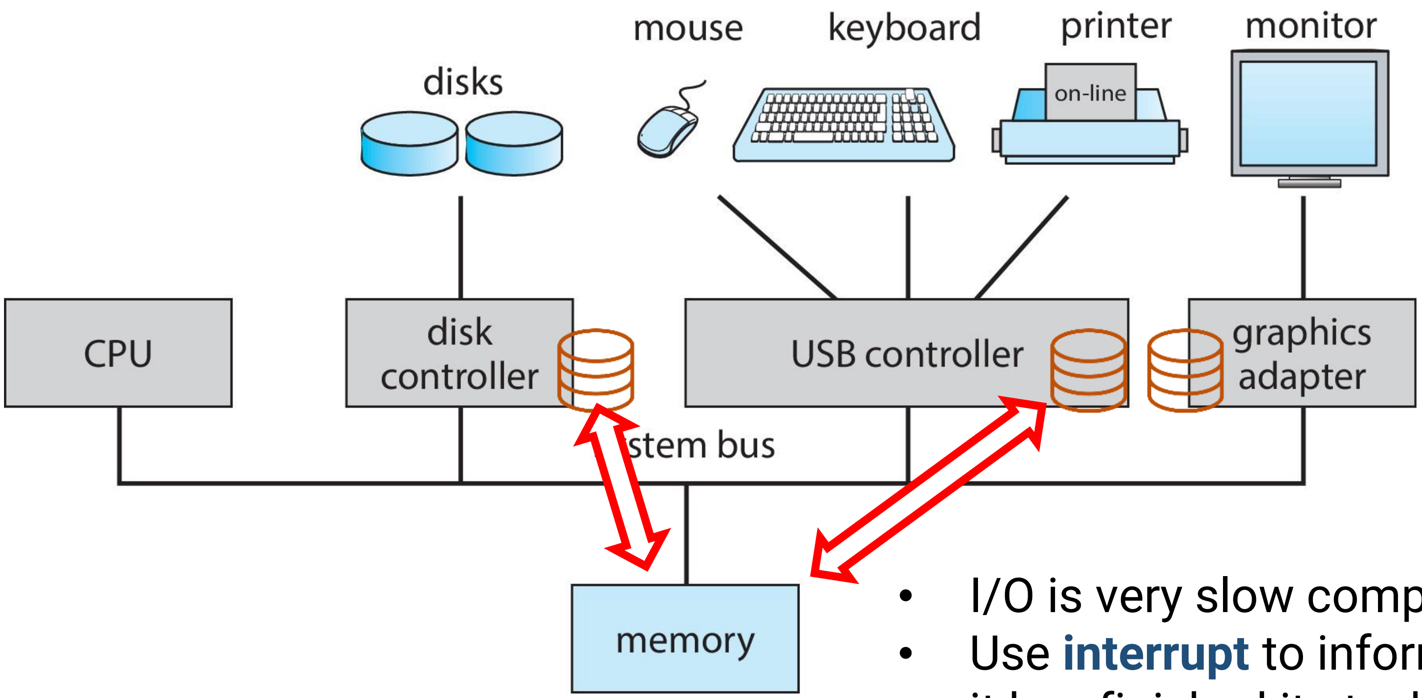
Computer System Operations (cont.)

- CPU moves data from/to main memory to/from local buffer for executing programs
  - Main memory is the **only** storage that CPU can access



Computer System Operations (cont.)

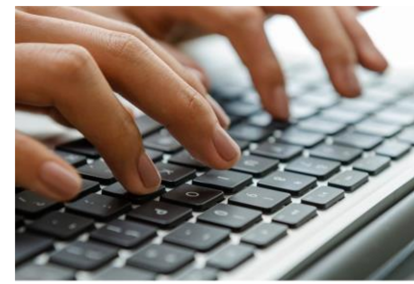
- I/O: from the device to local buffer of controller
  - **Read:** devices ➔ controller buffers ➔ memory
  - **Write:** memory ➔ controller buffers ➔ devices



- I/O is very slow compared to CPU
- Use **interrupt** to inform CPU that it has finished its task

Interrupt

- Interrupt provides a way to change the **flow of control** in the CPU
- **Hardware interrupt (signal)**
  - Service requests from one of the devices
    - Examples: keyboard, mouse click, etc.

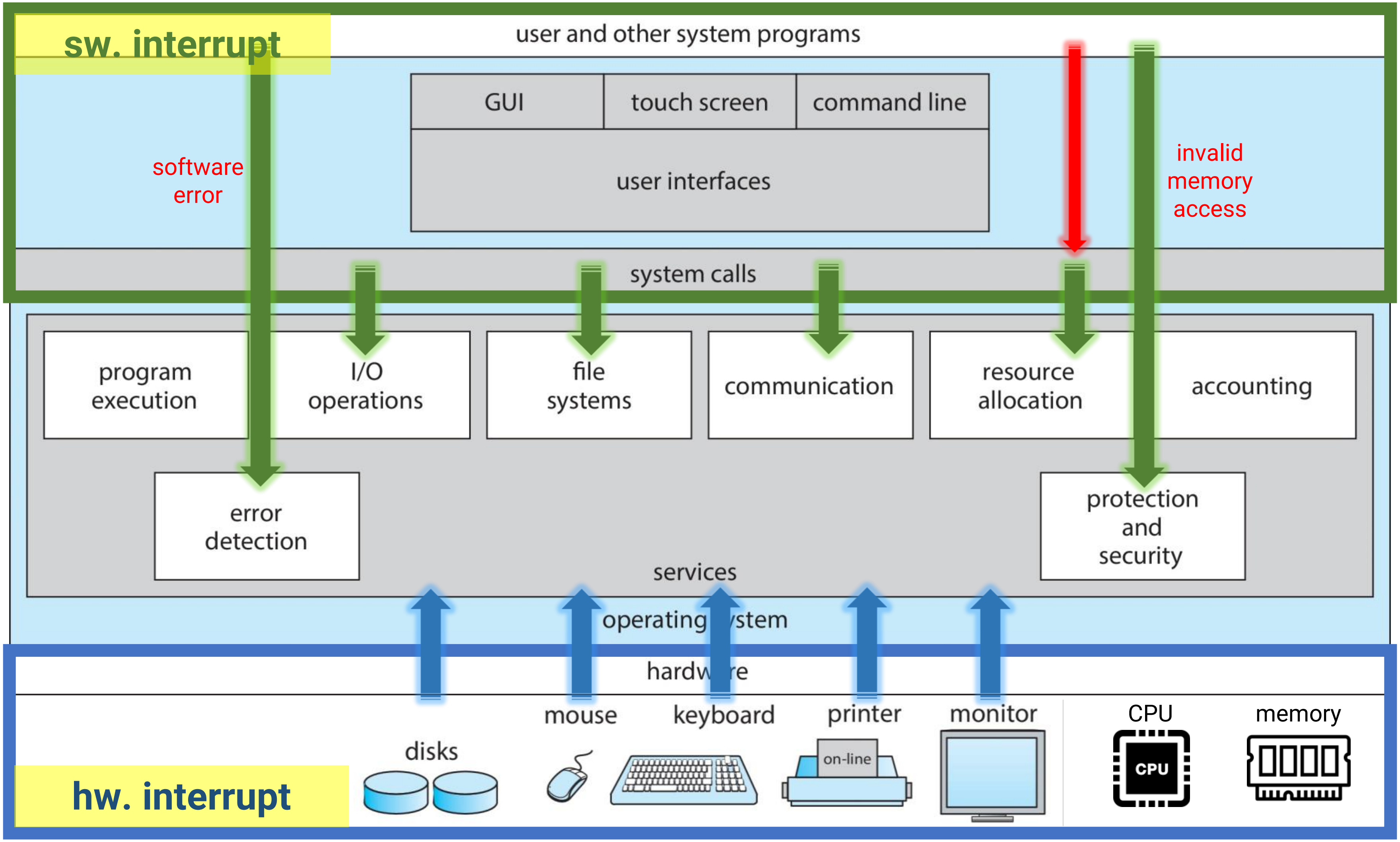


- **Software interrupt (trap)**
  - **Invalid memory access**
  - **Software error**
    - Example: division by zero
  - **System calls**
    - Request for system services

```
#include <stdio.h>

int main(int argc, const char * argv[]) {
    FILE* fp = fopen("test.txt", "r");
    if (fp) {
        printf("\nNot NULL");
    } else {
        printf("NULL");
    }
}
```

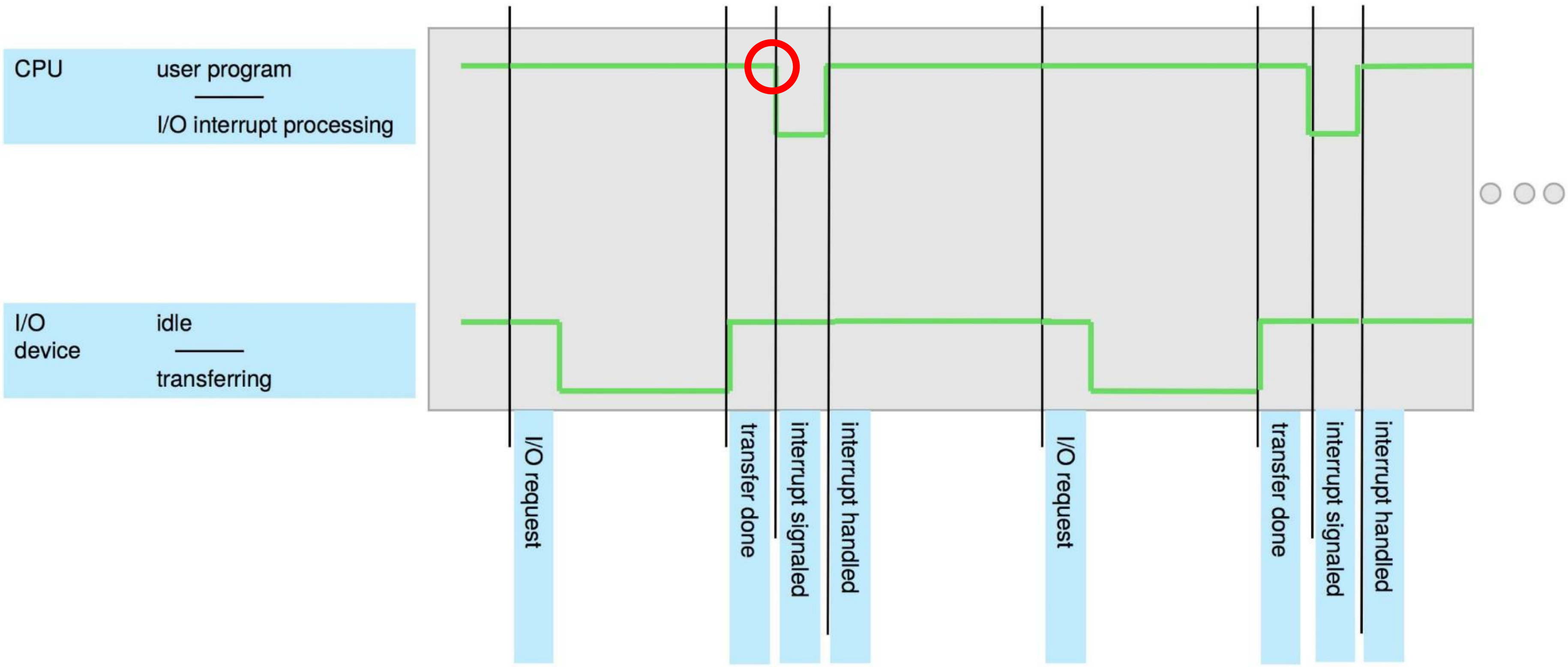
➔ system call (open)

Computer System Organization



Interrupt Timeline

- Must **save the address** of the interrupted instruction
  - Usually done by using **stacks**
- Transfer control to the **interrupt service routine**
  - Generic handler / interrupt vector / hybrid



A Brief Introduction to Hardware Architecture

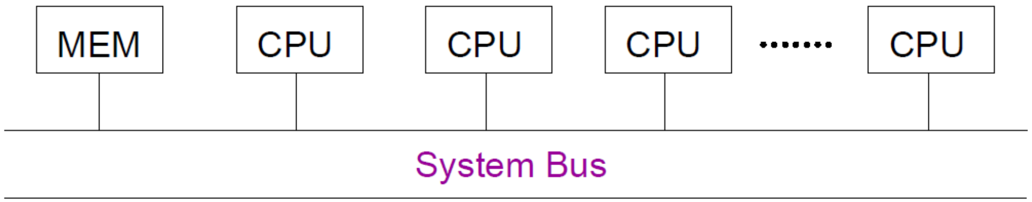
Processor

- **Single-processor system**
  - One main CPU per system
  - Example: earlier desktop or mobile device

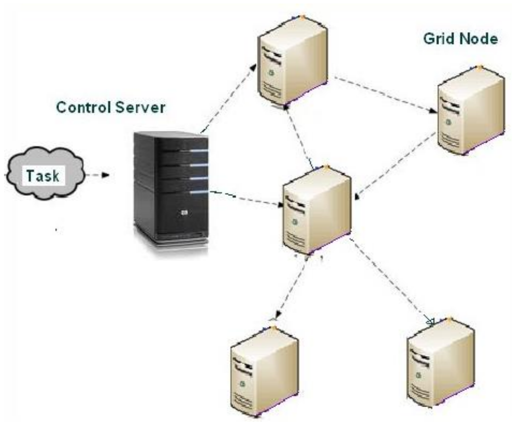
- **Multiple-processor system**
  - Multiple CPUs per system

Multi-Processor System

- **Tightly coupled**
  - More than one processor in close communication sharing bus, memory, and peripheral devices



- **Loosely coupled**
  - Otherwise (such as distributed systems)
  - Each machine has its own memory



Multi-Processor System (cont.)

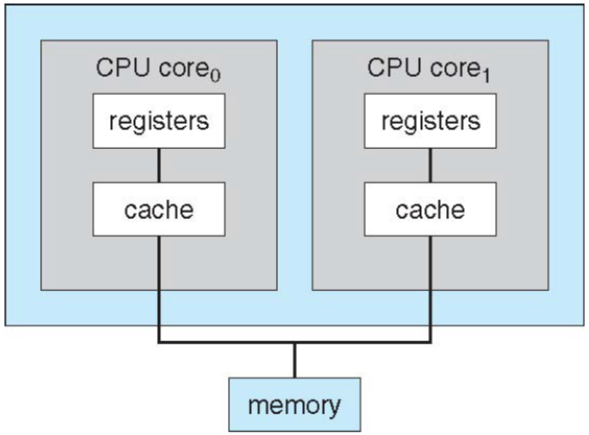
- **Symmetric model**
  - Each processor in the system runs an identical copy of the OS

- **Asymmetric model**
  - Master-slave fashion
  - Commonly seen in extremely large systems

Multi-Processor System (cont.)

- **Advantages of multi-processor systems**
  - **Speedup**: better throughput
  - **Lower cost**: building one small fast chip is very expensive
  - **More reliable**: Graceful degradation and fail soft

- The recent trend: from a fast single processor to lots of processors
  - **Multiple cores** over a single chip
  - Hyper-threading (logical core)

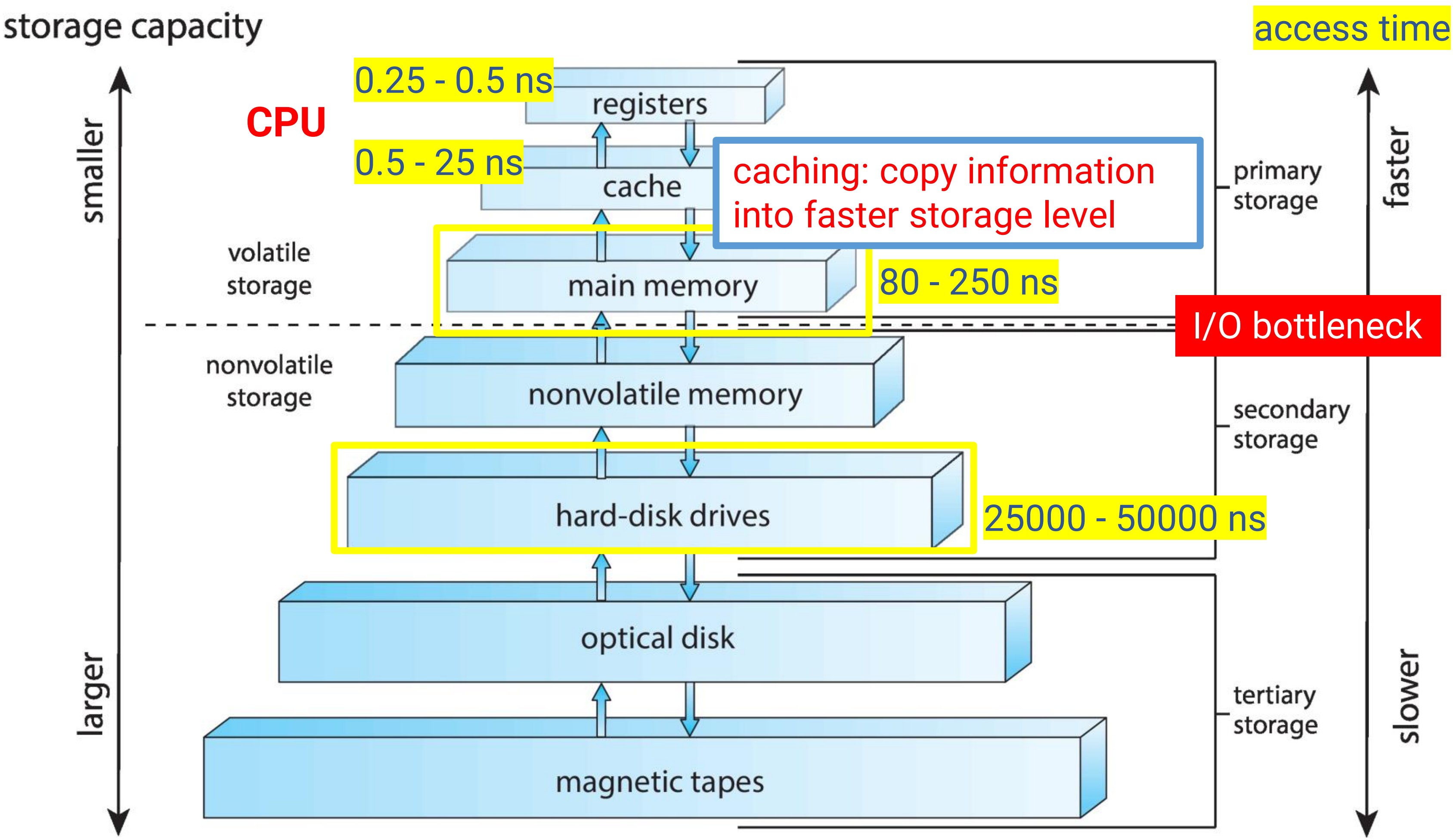


Storage Structure

- Organized in hierarchy based on
  - **Speed**
  - **Cost**
  - **Volatility**

- **Main memory**
  - The **only large storage media** that the CPU can access directly
  - Typically volatile
- **Secondary storage** (ex: HDD, USB sticks, CD, DVD, …)
  - Extension of main memory that provides large storage capacity
  - Typically nonvolatile

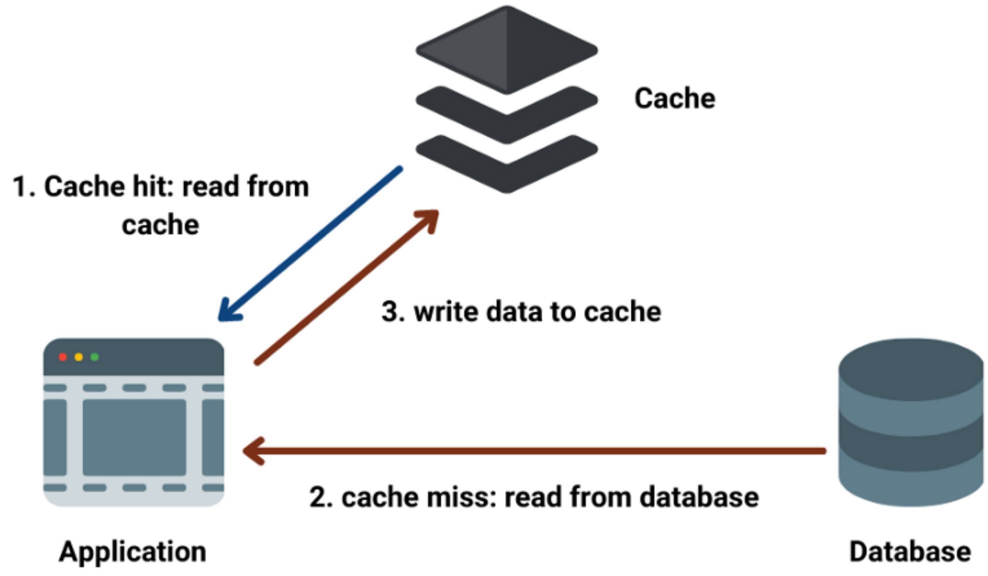
Storage Structure (cont.)



Caching

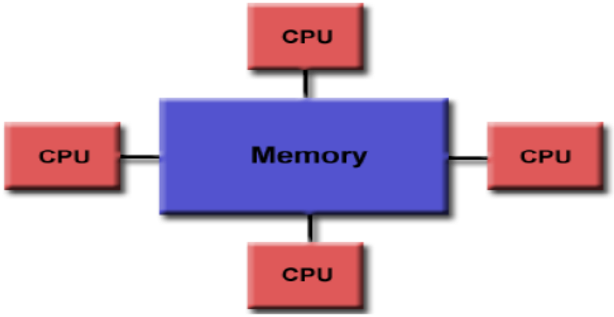
- Information is **copied** to a faster storage system on a **temporary** basis
- Assumption: data will be used again soon (locality)

- Cache management
  - Cache size
  - Replacement policy

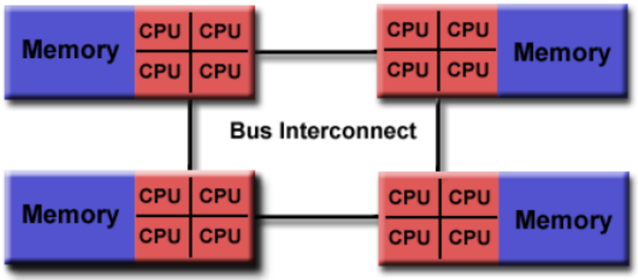


Memory Access Architecture

- **Uniform Memory Access (UMA)**
  - Most commonly represented today by **symmetric multi-processor (SMP)** machines
  - Equal access times to memory
  - Example: most commodity computers

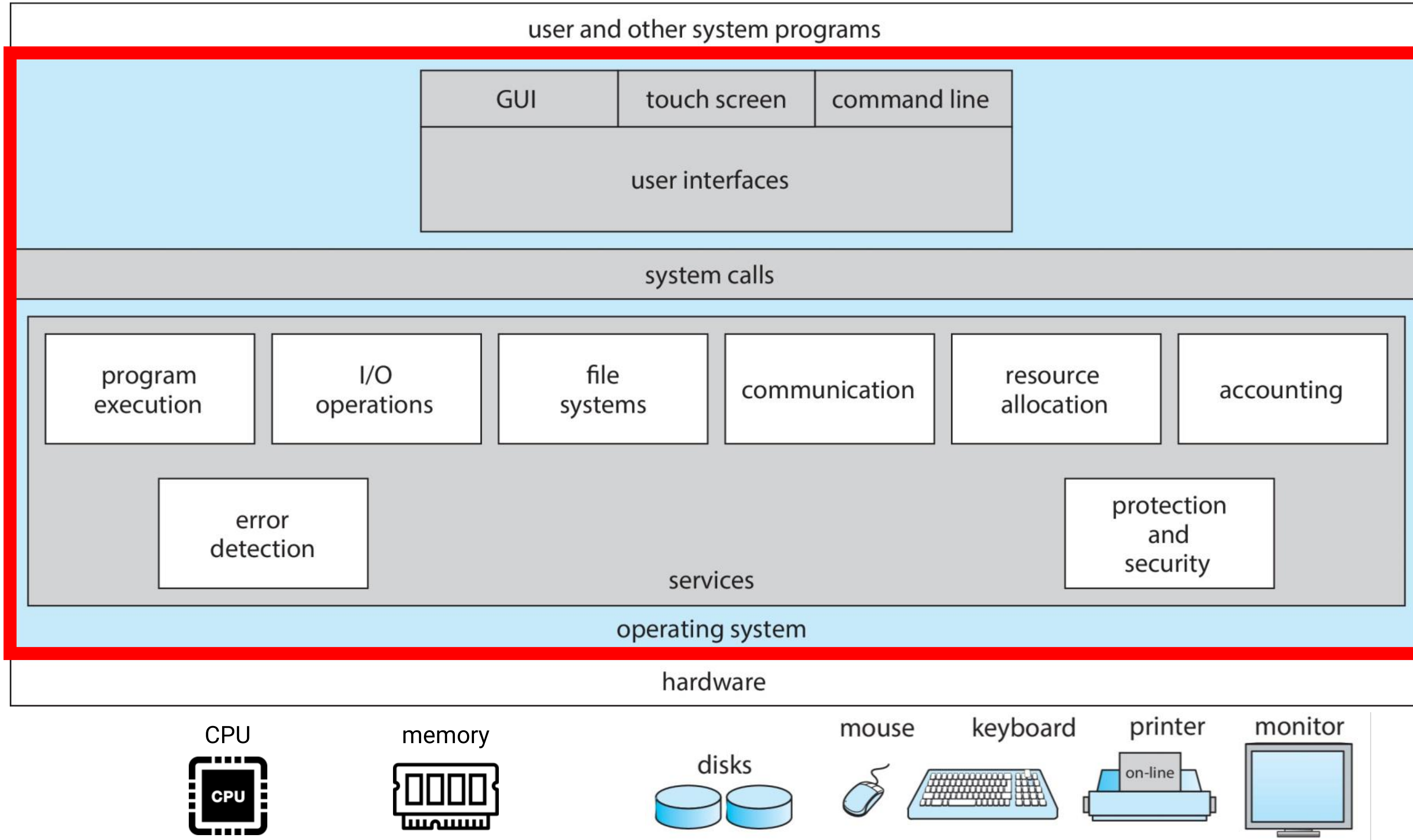


- **Non-Uniform Memory Access (NUMA)**
  - Often made by physically **linking two or more SMPs**
  - One SMP can directly access memory of another SMP
  - Memory access across link is slower
  - Example: IBM Blade server



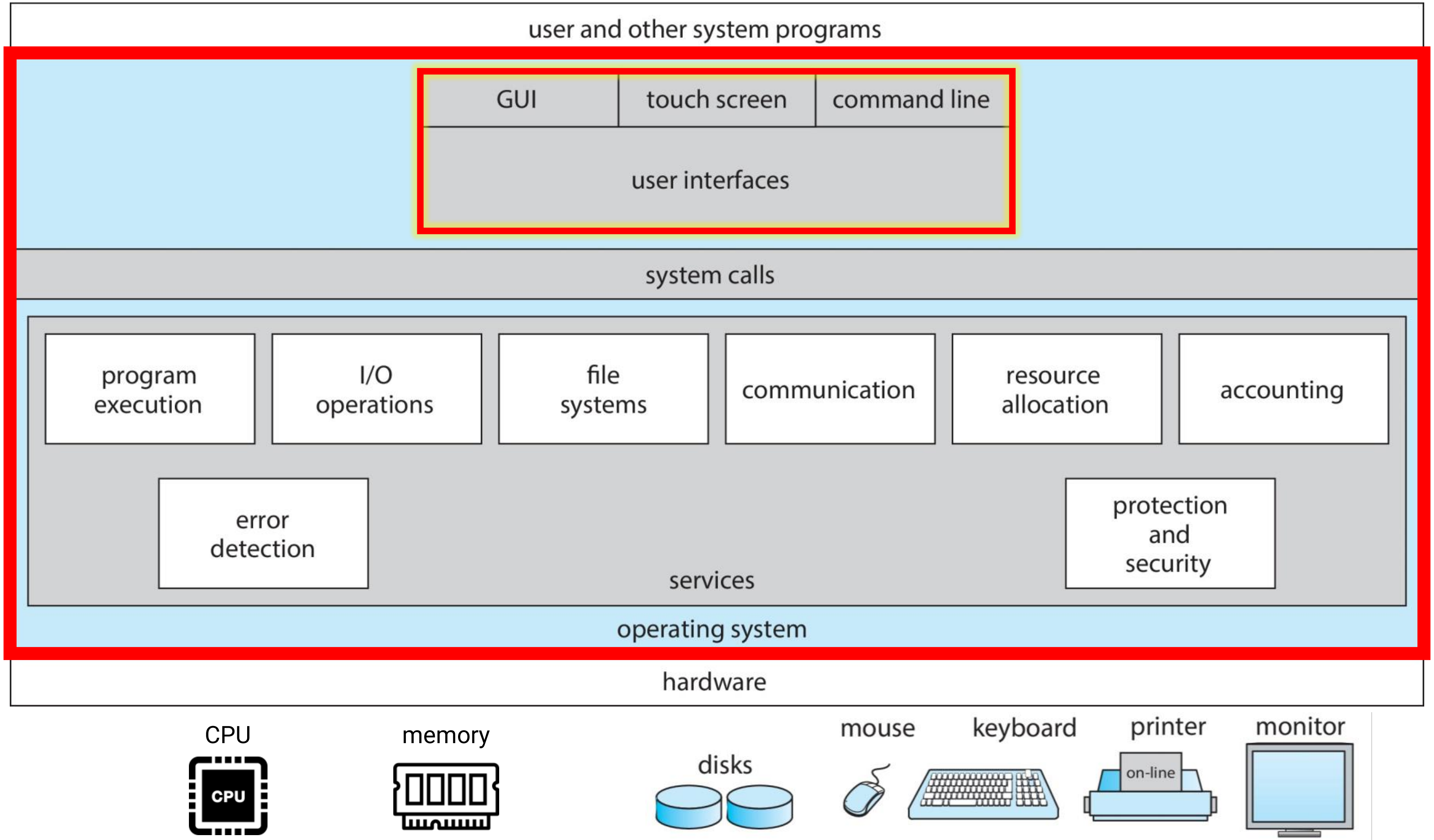
Operating System Services

Operating System Services



Operating System Service: User Interfaces

Operating System Services



User Interface

- **Command line interface (CLI)**
  - Fetch a command from user and execute it
  - Shell (command-line interpreter)
    - Ex: CSHELL, BASH
    - Allow to some modification based on user behavior and preference
- **Graphic user interface (GUI)**
  - Usually with mouse, keyboard, and monitor
  - Icons are used to represent files, directories, programs, etc.
  - Usually built on CLI

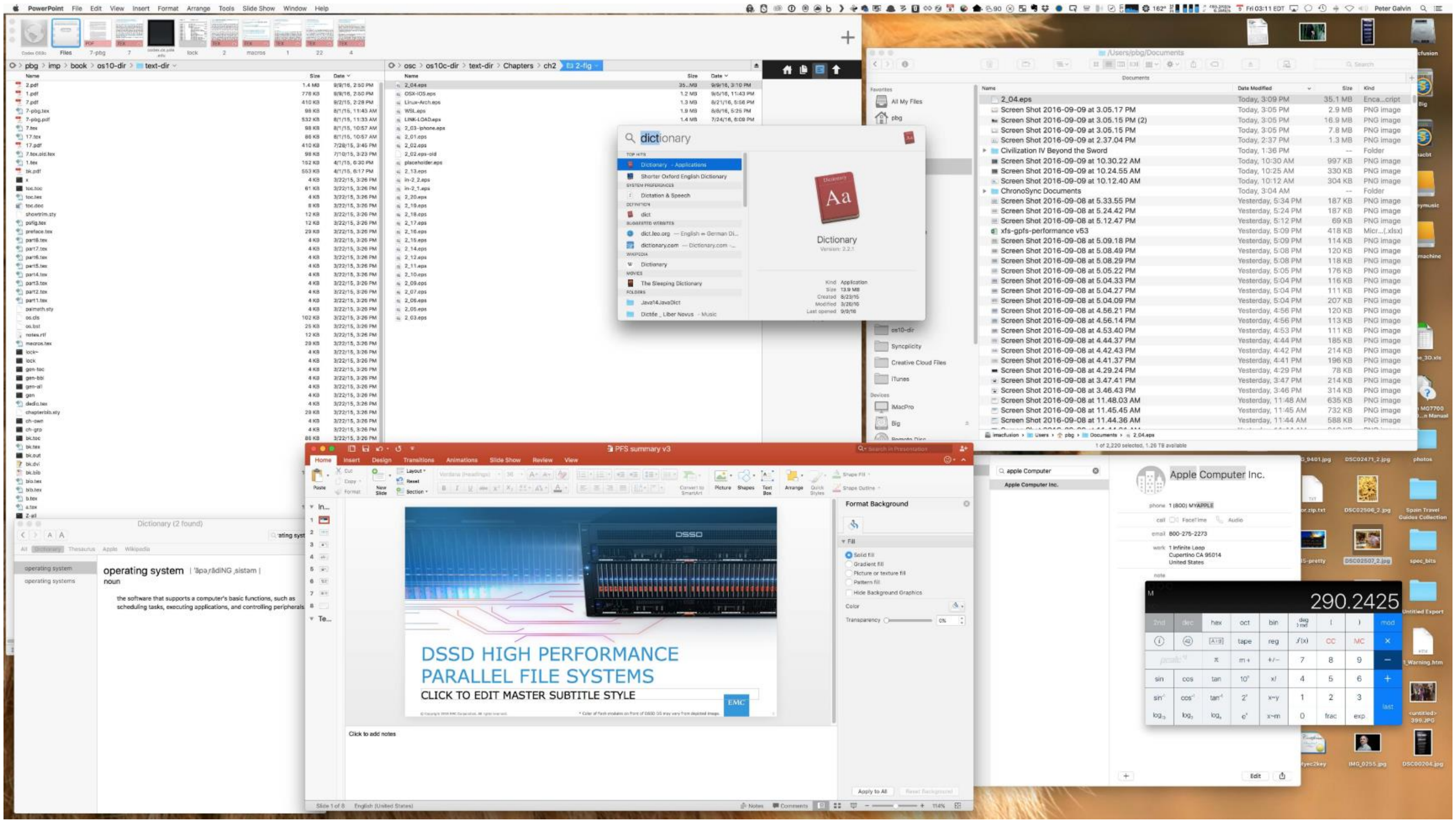
- Most systems have both CLI and GUI

Command Line Interface

```
1. root@r6181-d5-us01:~ (ssh)
root@r6181-d5-u...  ⌘1    ssh  ⌘2    root@r6181-d5-us01...  ⌘3

Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbg$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                       50G   19G   28G  41% /
tmpfs                 127G  520K  127G   1% /dev/shm
/dev/sda1             477M   71M  381M  16% /boot
/dev/dssd0000         1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                       12T  5.7T  6.4T  47% /mnt/orangefs
/dev/gpfs-test         23T  1.1T   22T   5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root     97653 11.2  6.6 42665344 17520636 ?   S<Ll Jul13 166:23 /usr/lpp/mmfs/bin/mmfsd
root     69849  6.6  0.0      0     0 ?        S    Jul12 181:54 [vpthread-1-1]
root     69850  6.4  0.0      0     0 ?        S    Jul12 177:42 [vpthread-1-2]
root      3829  3.0  0.0      0     0 ?        S    Jun27 730:04 [rp_thread 7:0]
root      3826  3.0  0.0      0     0 ?        S    Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-r-x------ 1 root root 20667161 Jun  3  2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```

Bourne Shell (default shell of UNIX ver. 7)

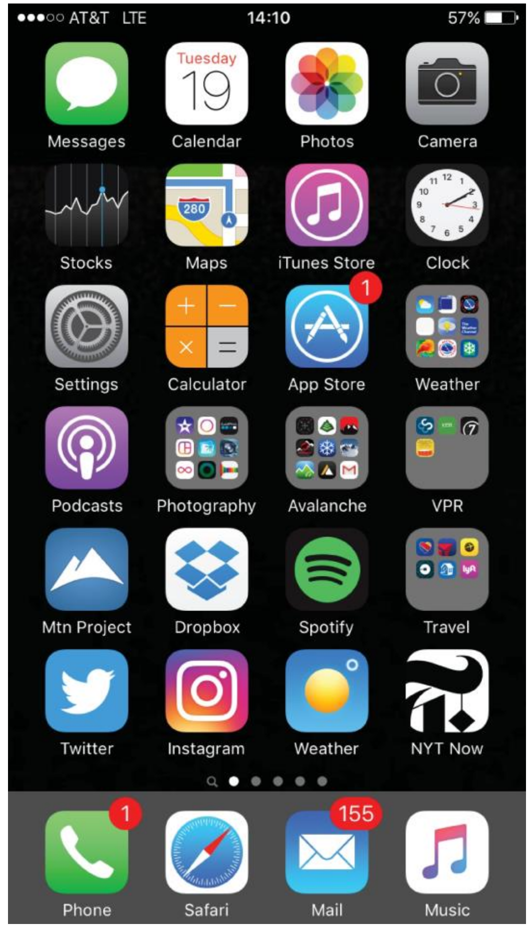
Graphic User Interface

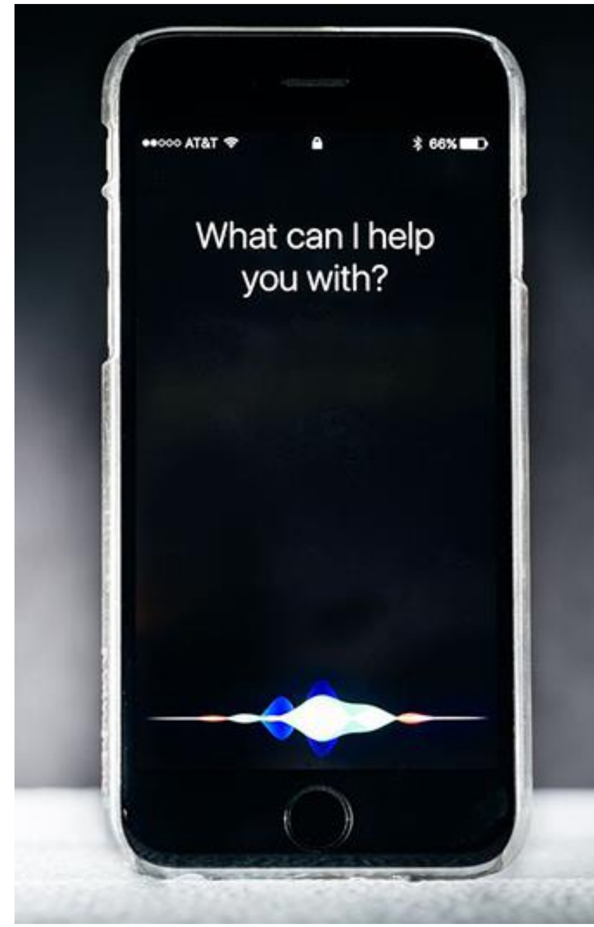


Mac OS X GUI

Other Interfaces

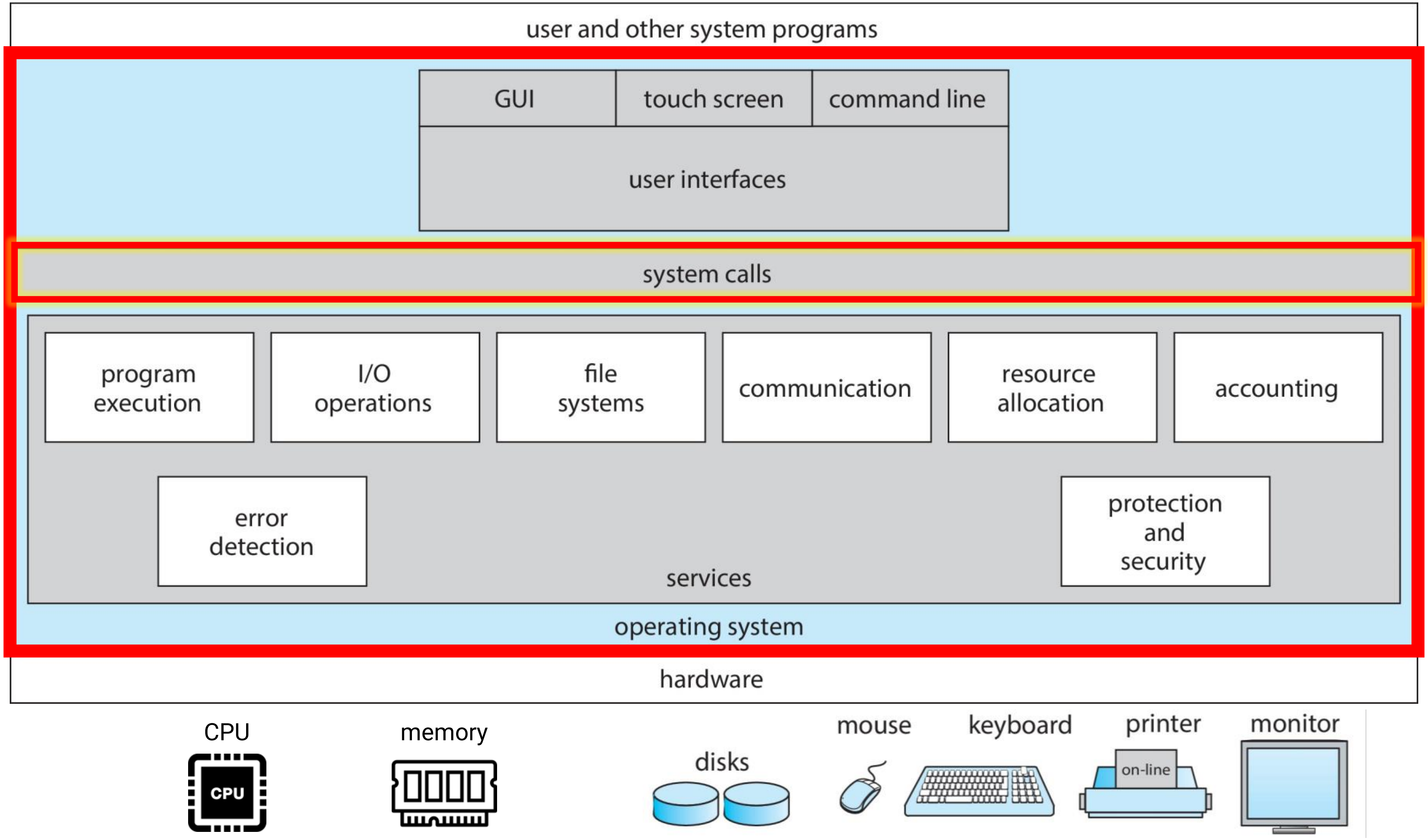
- Batch
- Touch-screen
- Voice control





System Calls and API

Operating System Services



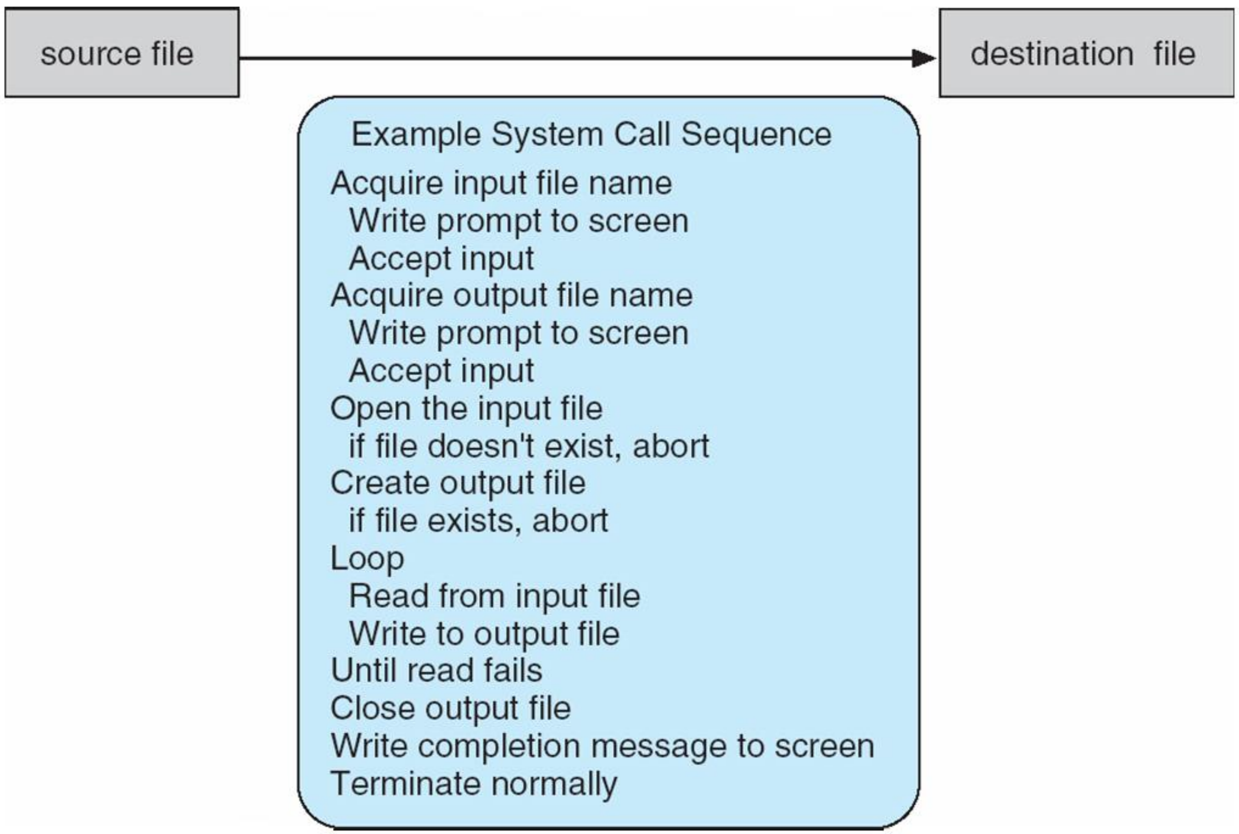
System Calls

- **Programming interface to the services provided by the OS**
  - An explicit request to the kernel made via **software interrupt**
  - Generally available as assembly-language instructions

- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use

System Calls (cont.)

- Example: a sequence of system calls for copying a file



System Calls (cont.)

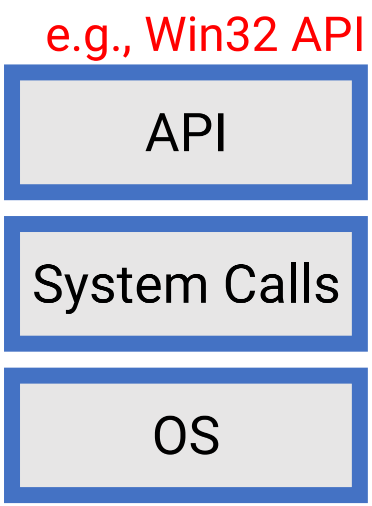
- Request OS services
  - **Process control**
    - End (normal exit) or abort (abnormal)
    - Load and execute
    - Create and terminate
    - Get or set attributes of process
    - Wait for a specific amount of time or an event
    - Memory dumping, profiling, tracing, allocate, and free
  - **File management**
    - Create and delete
    - Open and close
    - Read, write, and reposition
    - Get or set attributes
    - Operations for directories

System Calls (cont.)

- Request OS services (cont.)
  - **Device management**
    - Request or release
    - Logically attach or detach devices
  - **Information maintenance**
    - Get or set time or date
    - Get or set system data (e.g., maximum memory for a process)
  - **Communications**
    - Send and receive messages
    - Message passing or shared memory
  - **Protection**

Application Programming Interface (API)

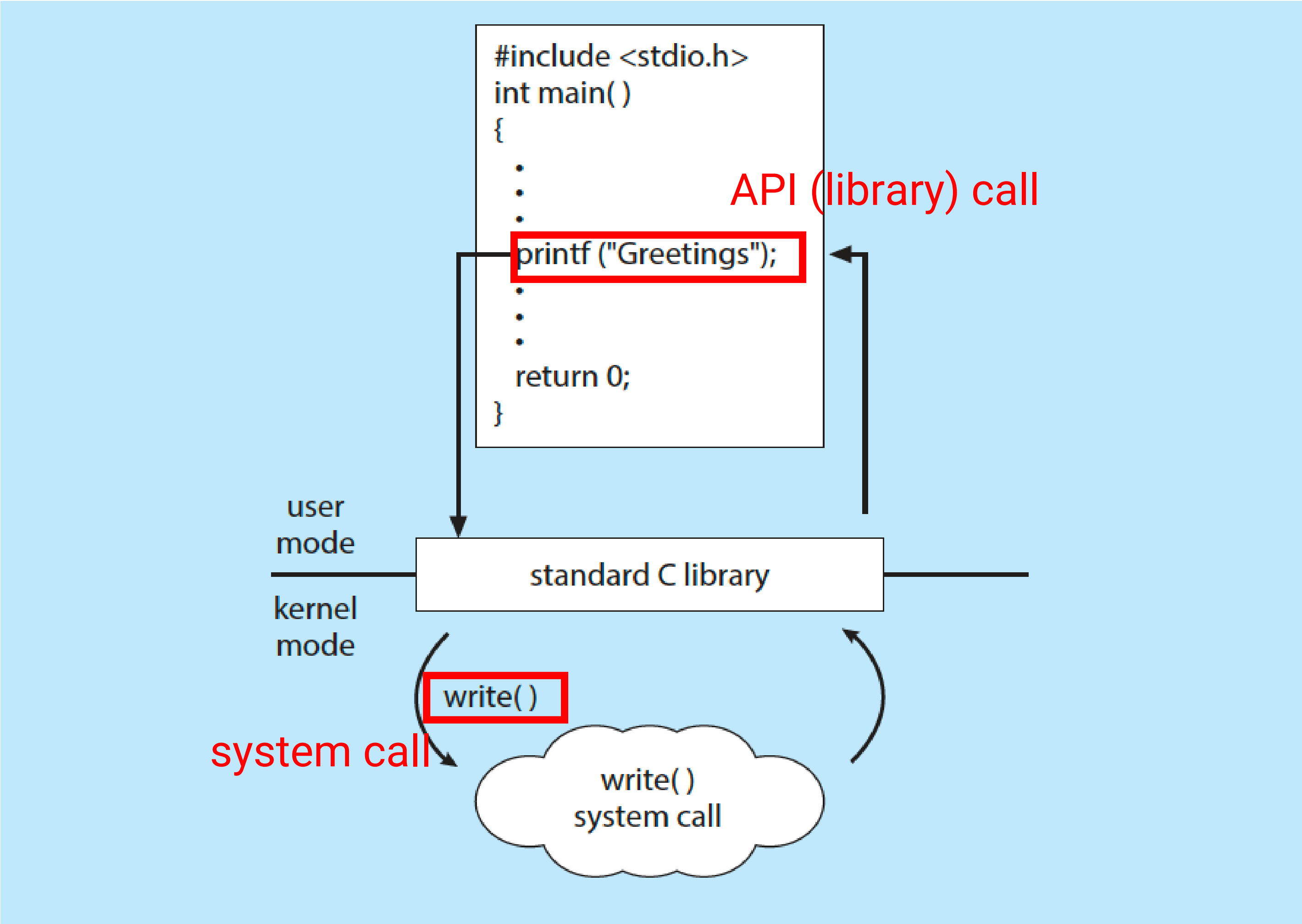
- An **encapsulation** of system calls for user programs
- Provide **portability**
- Usually implemented by high-level languages
  - C library, Java
- Could involve zero or multiple system calls
  - abs(): zero
  - fopen(): multiple
  - malloc(), free() ➔ brk()



API (cont.)

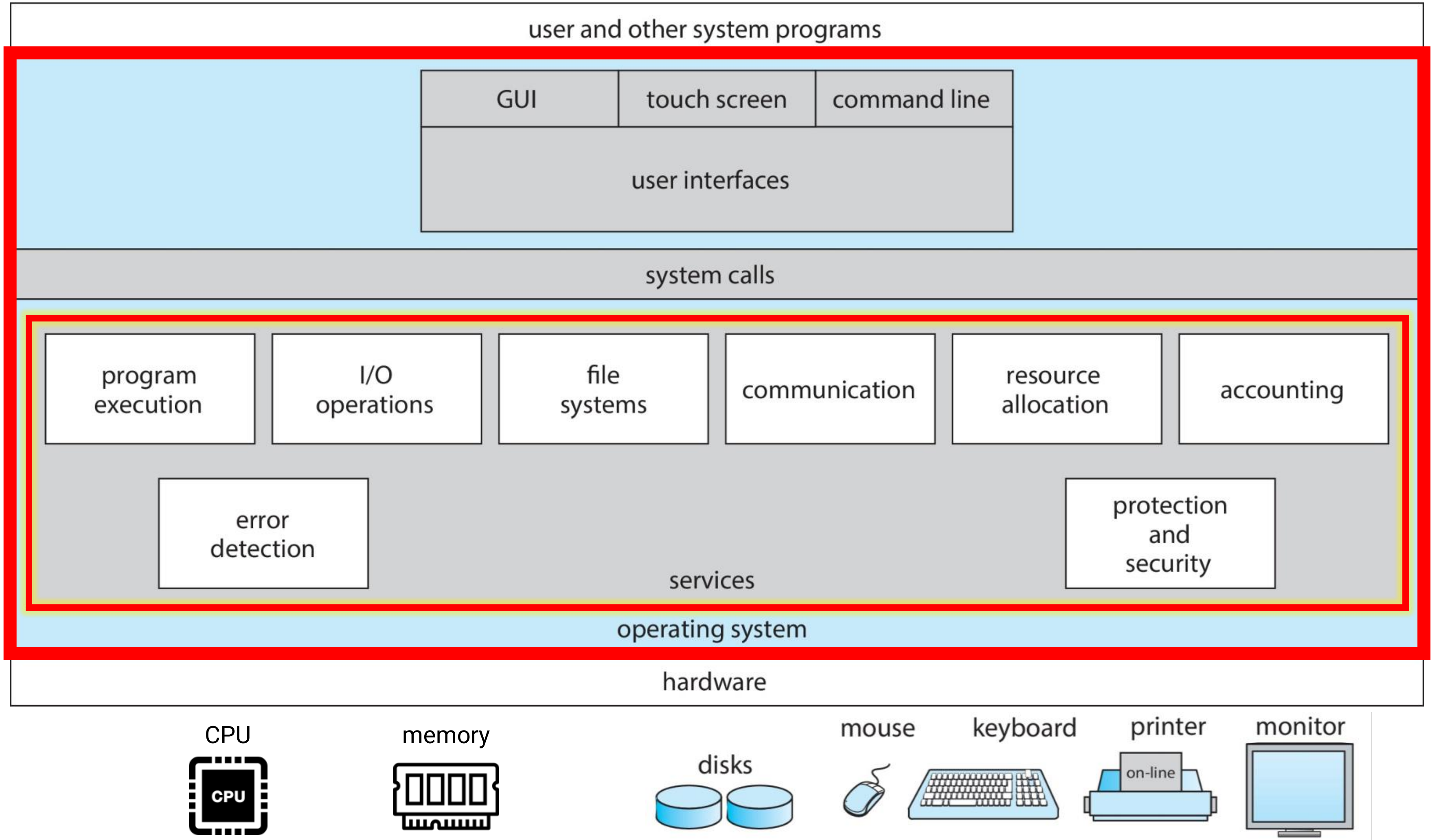
- Three most common APIs
  - **Win32 API**
    - For Microsoft Windows
    - https://en.wikipedia.org/wiki/Windows_API
    - https://docs.microsoft.com/zh-tw/windows/win32/apiindex/windows-api-list?redirectedfrom=MSDN
  - **POSIX API**
    - POSIX stands for **P**ortable **O**perating **S**ystem **I**nterface for Uni**x**
    - Used by Unix, Linux, and Max OS X
    - https://en.wikipedia.org/wiki/POSIX
  - **Java**
    - For Java virtual machine (JVM)

System Call and API



Operating System Design

Operating System Services

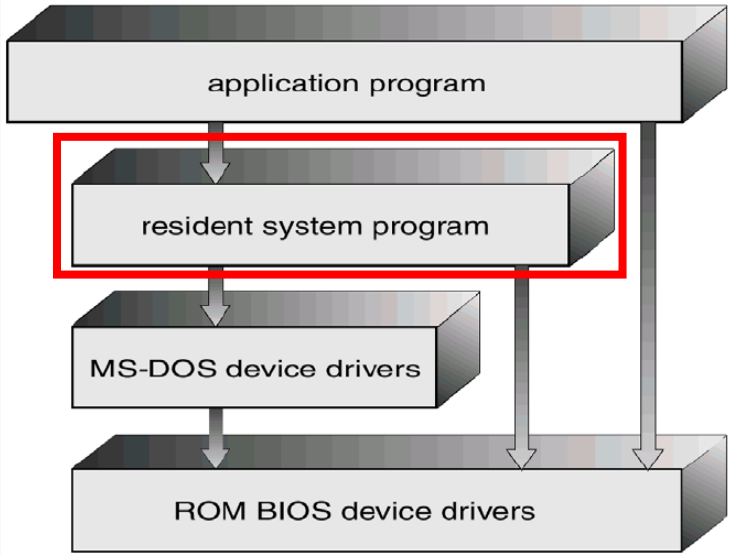


Overview of OS Structure

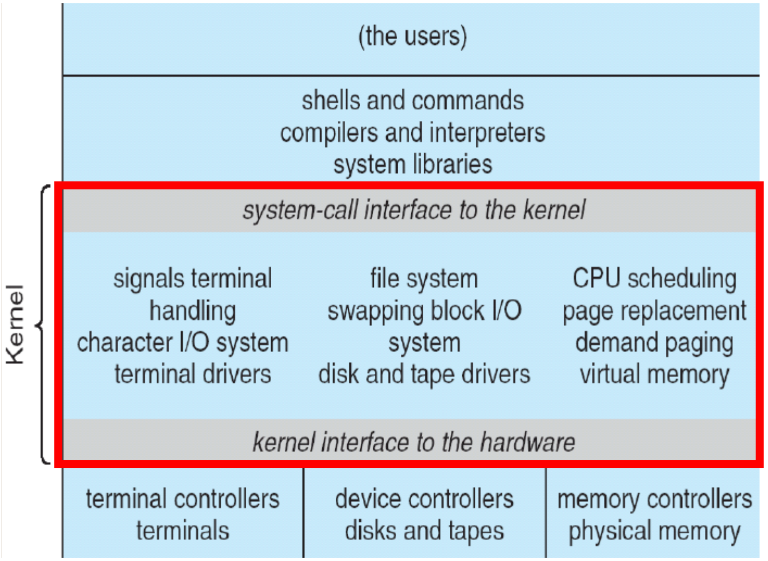
- Simple OS architecture
- Layer OS architecture
- Microkernel OS
- Modular OS architecture
- Hybrid systems
- Virtual machine

Simple OS Architecture

- Only one or two levels
- Drawbacks
    - Unsafe
    - Difficult to enhance



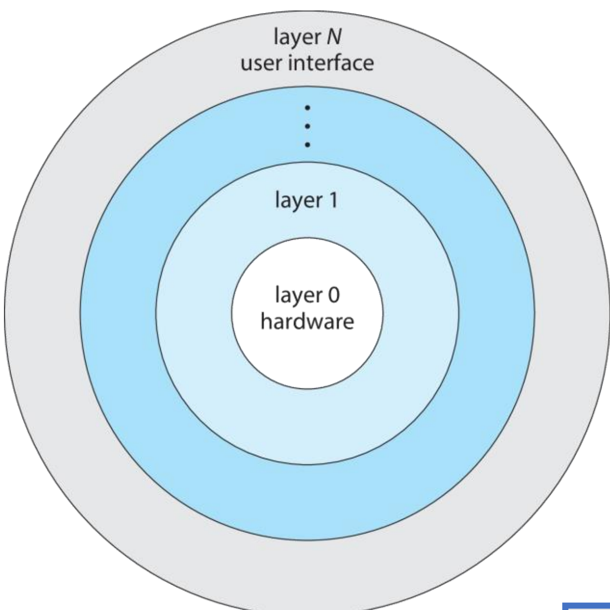
MS-DOS

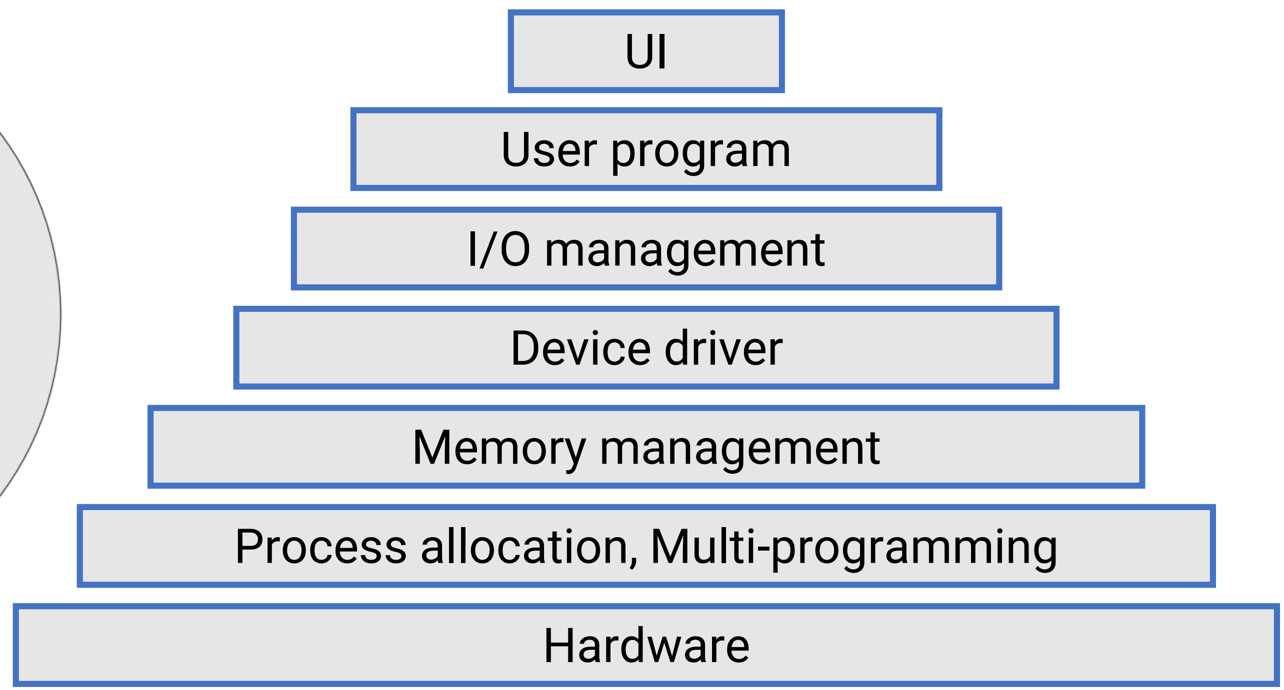


UNIX

Layered OS Architecture

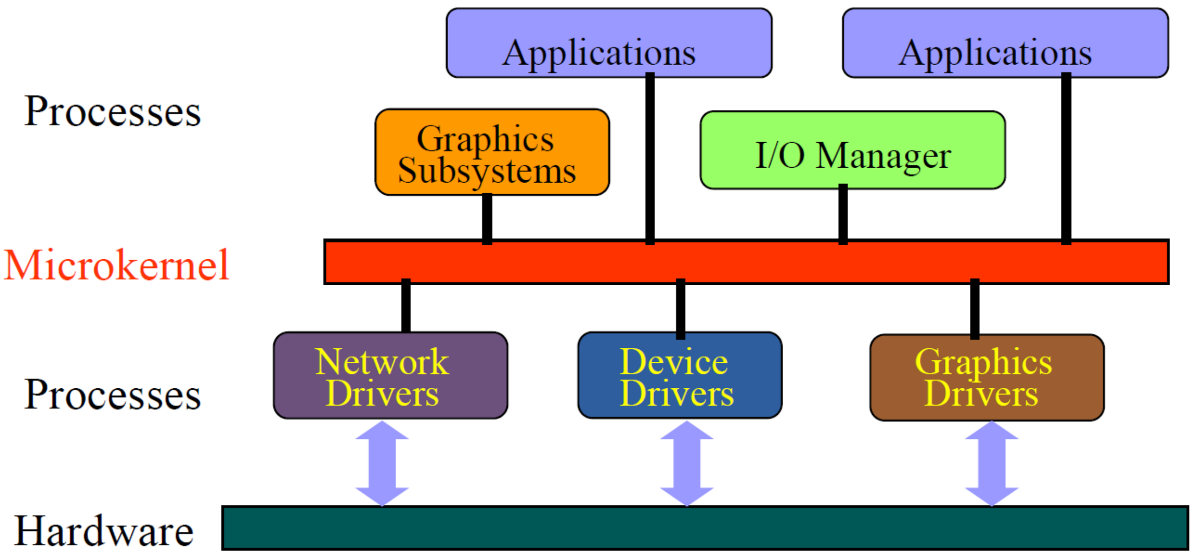
- Lower levels are independent of upper levels
- Pros: easier debugging and maintenance
- Cons: less efficient and difficult to define layers





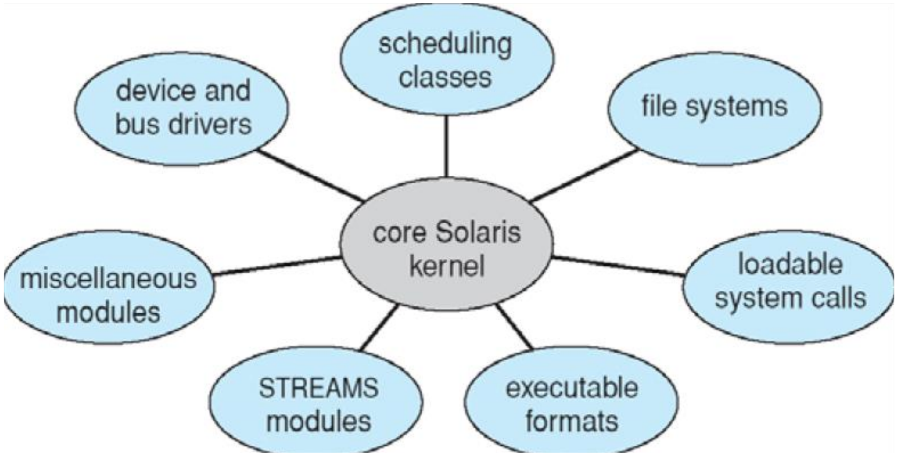
Microkernel OS

- **Kernel should be as small as possible**
  - Move most parts of the original kernels into user space
- Communication is provided by **message passing**
- Easier for extending and porting
- Slow



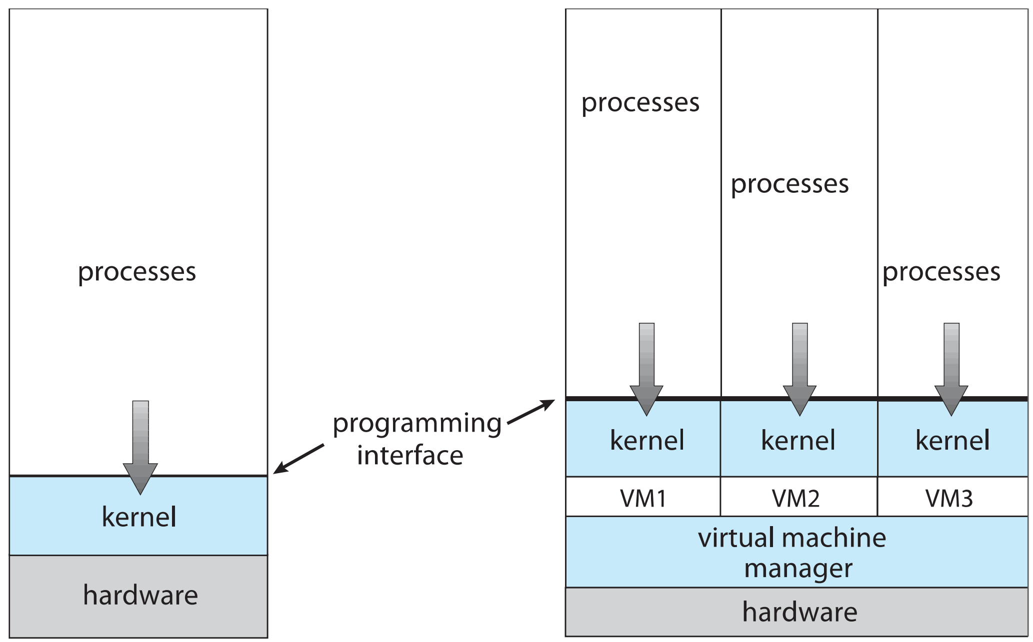
Modular OS Architecture

- **Employed by most modern OS**
  - Object-oriented approach
  - Each core component is separate
  - Each module talks to the others over known interfaces
  - Each module is loadable as needed **within the kernel**
- Similar to layers but with more flexibility



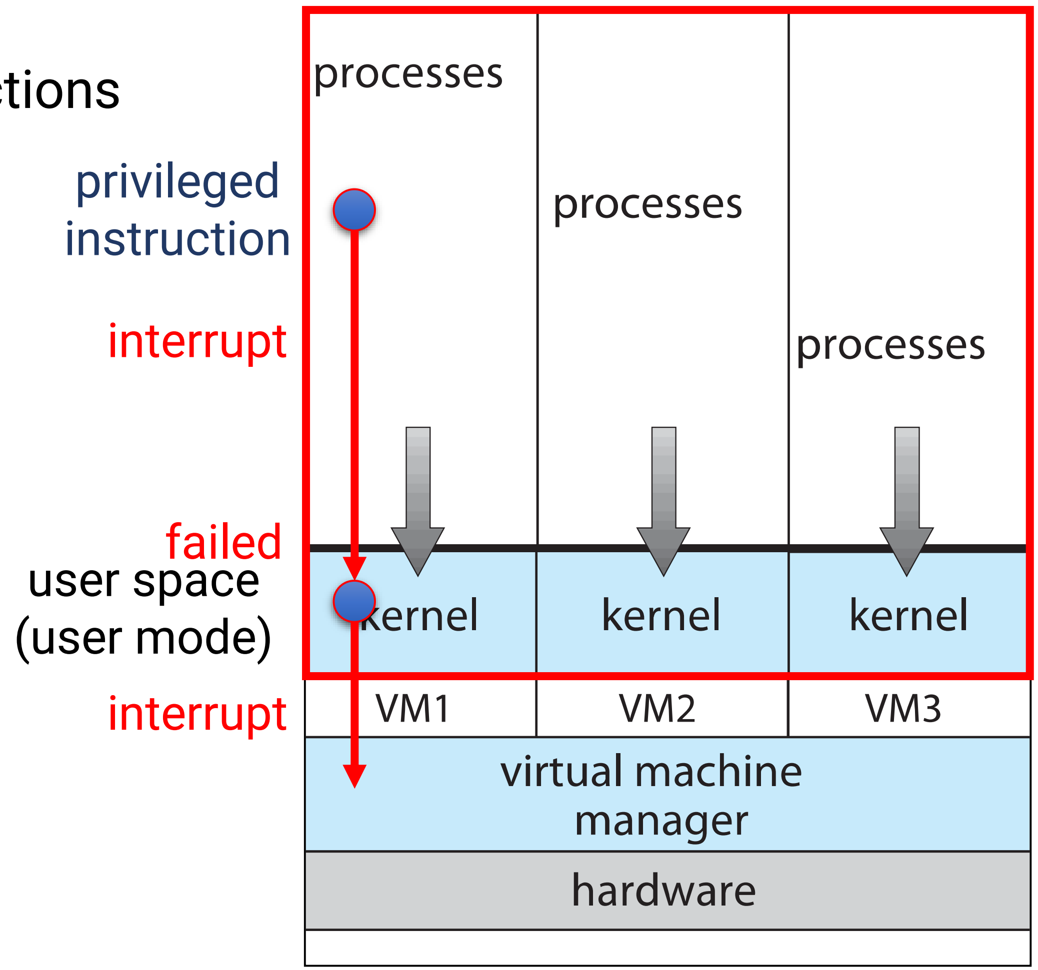
Virtual Machine

- Provide an interface that is identical to the underlying bare hardware
  - Each process is provided with a (virtual) copy of the underlying computer



Virtual Machine (cont.)

- **Challenges**
  - Privileged instructions

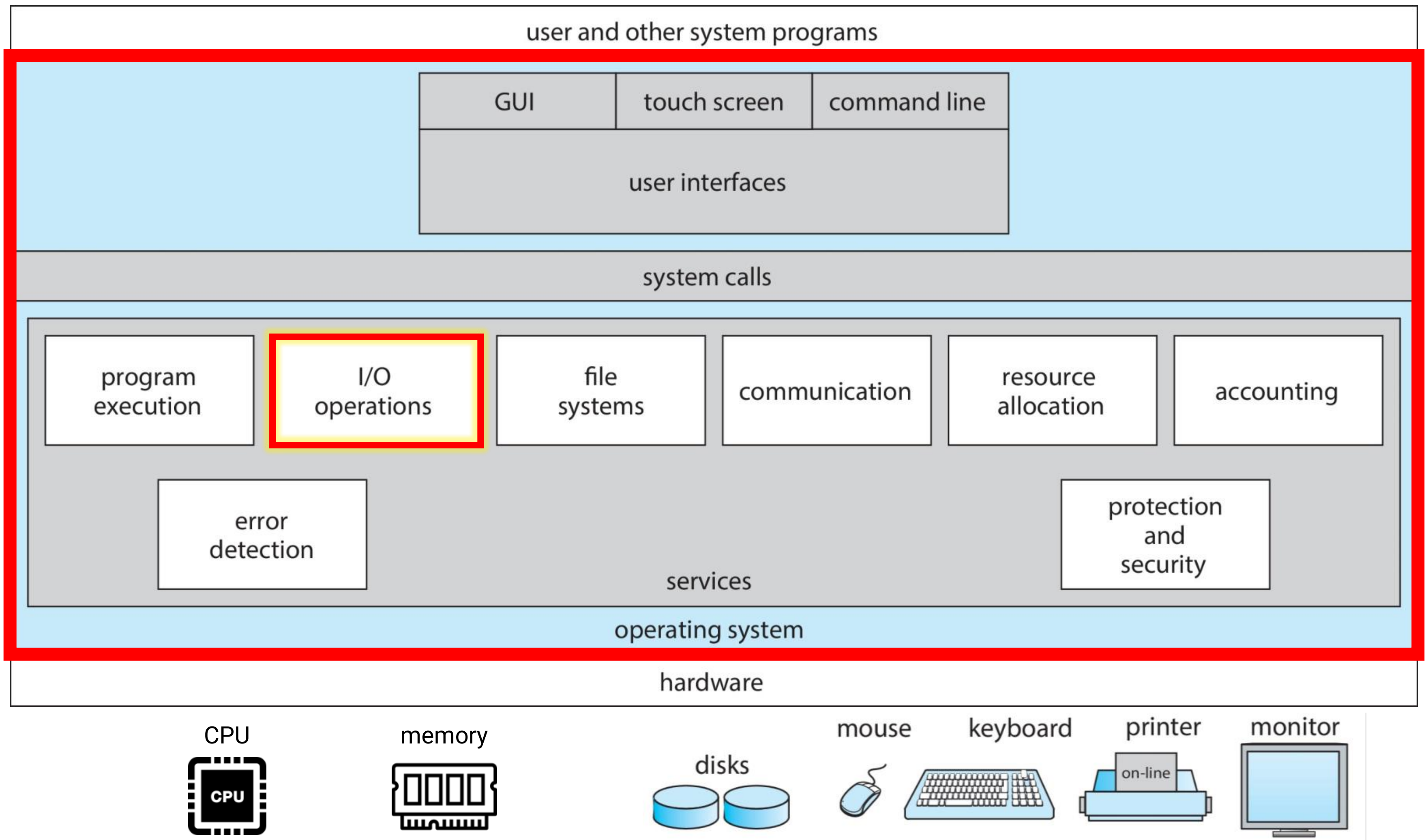


Virtual Machine (cont.)

- **Advantages**
  - Provide **complete protection** of system resources
  - Provide an approach to solve **system compatibility** problems
  - Provide a vehicle for **OS research and development**
  - Provide a mean for increasing **resource utilization** in cloud computing

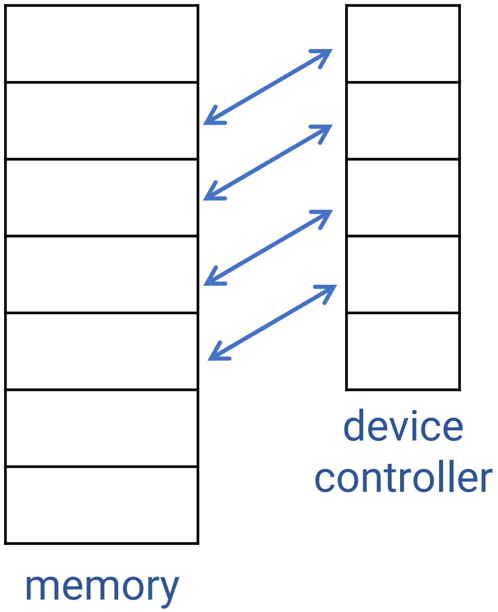
Operating System Service: I/O Operations

Operating System Services



Strategies for Handling I/O

- **Interrupted-based I/O**
  - For slow devices
  - OS is informed when jobs have been done
  - Example: disk
- **Programmed I/O (pulling)**
  - Keep asking if the jobs have been done
  - Example: network interface card
- **Memory-mapped I/O**
  - Frequently used or very fast devices
  - Special I/O instructions are used to move data between memory & device controller registers
  - Example: GPU

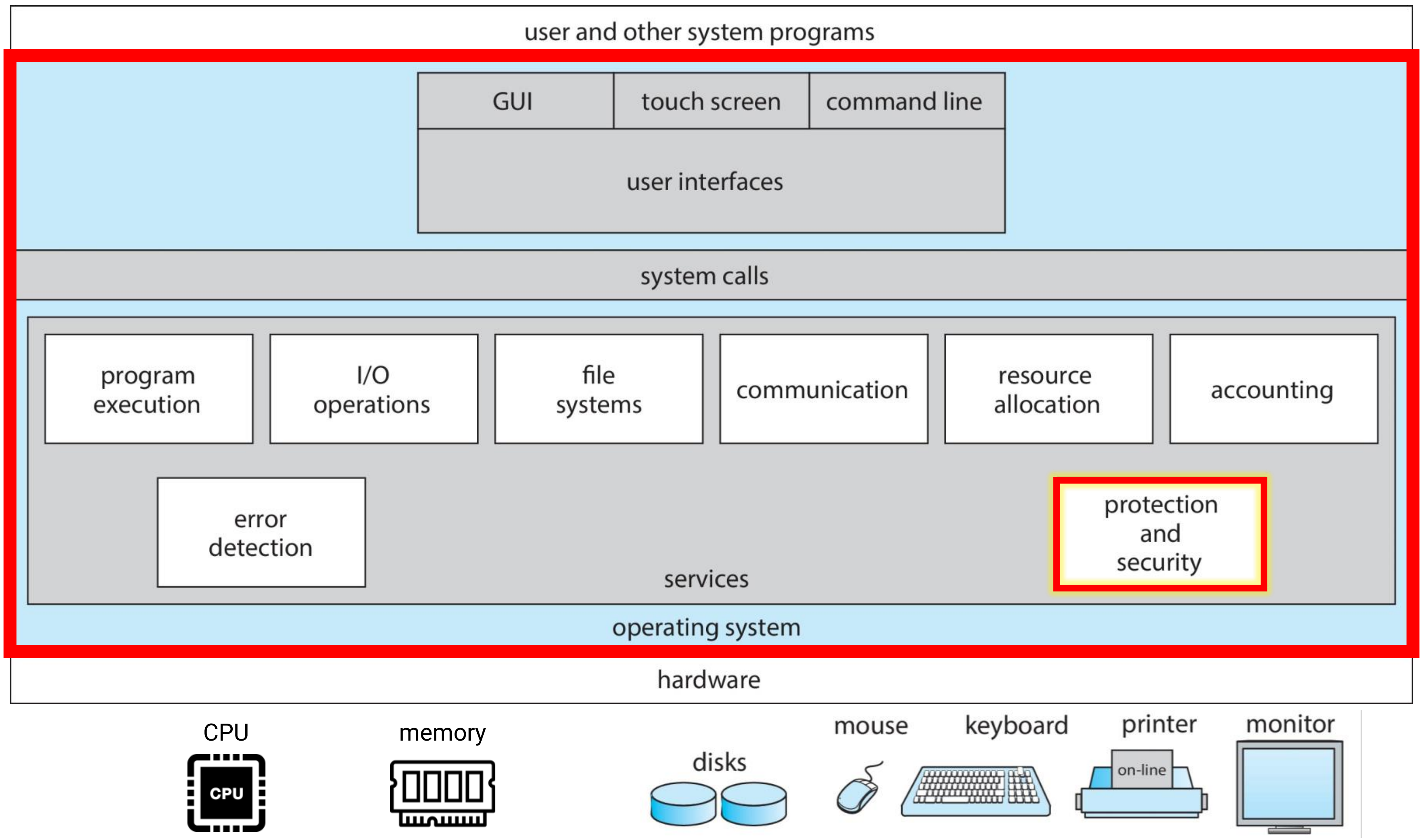


DMA: Direct Memory Access

- **Goal**
  - Device controller can transfer **blocks of data** from buffer storage to main memory **without CPU intervention**
  - Only one interrupt is generated per block (rather than per byte), thus avoiding CPU handling excessive interrupts
- **Procedure with DMA**
  - Execute the device driver to setup the registers of the DMA controller
  - DMA moves blocks of data between the memory and its own buffers
  - Transfer from its buffers to its devices
  - Interrupt the CPU when the job is done

Operating System Service: Protection and Security

Operating System Services

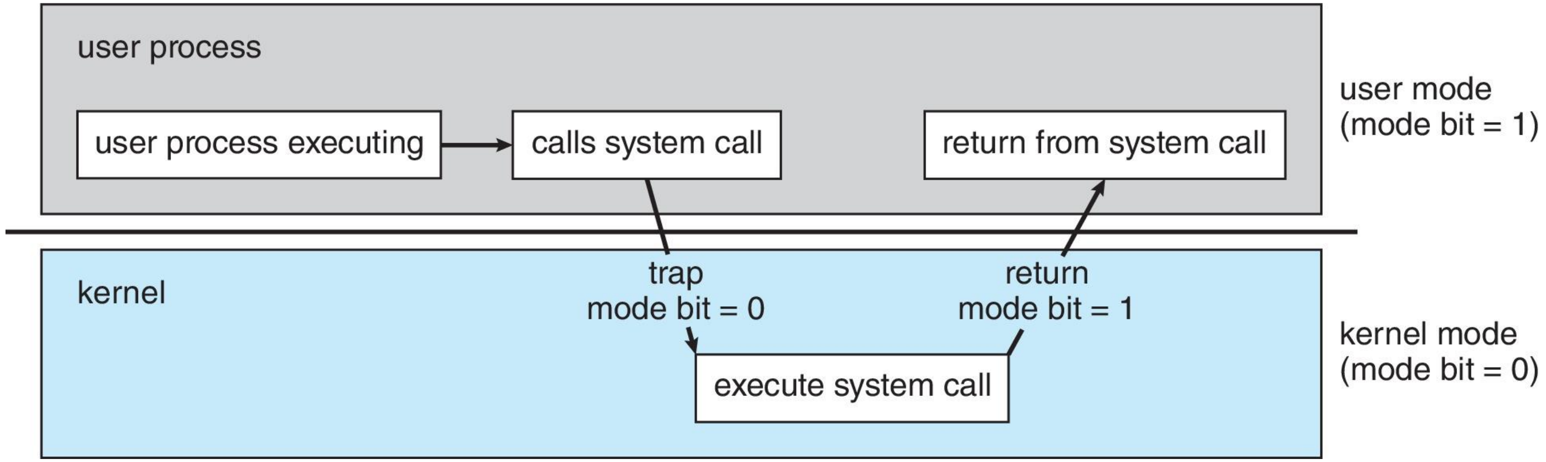


Protection and Security

- **Goal**
  - Prevent error and misuse
  - Resources are only allowed to be accessed by authorized processes
- **Protection**
  - Any mechanism for controlling the access of processes or users to the resources defined by the computer system
- **Security**
  - Defense of a system from external and internal attacks
  - Examples: viruses, denial of service, identity theft

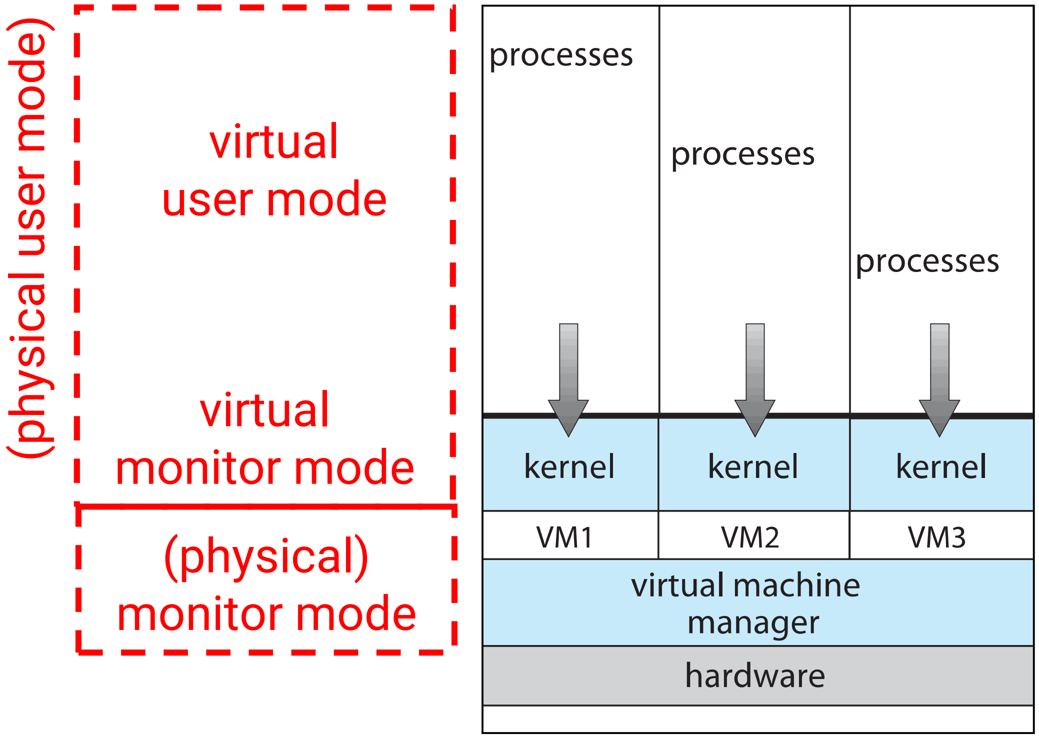
Protection

- **Dual-mode operations**
  - **User mode**
    - Executions except those after a trap or an interrupt occurs
  - **Monitor mode (system mode, privileged mode)**
    - Can execute all instructions including privileged ones (machine instructions that may cause harm)
  - Implemented by a **mode bit** and **system calls**



Protection (cont.)

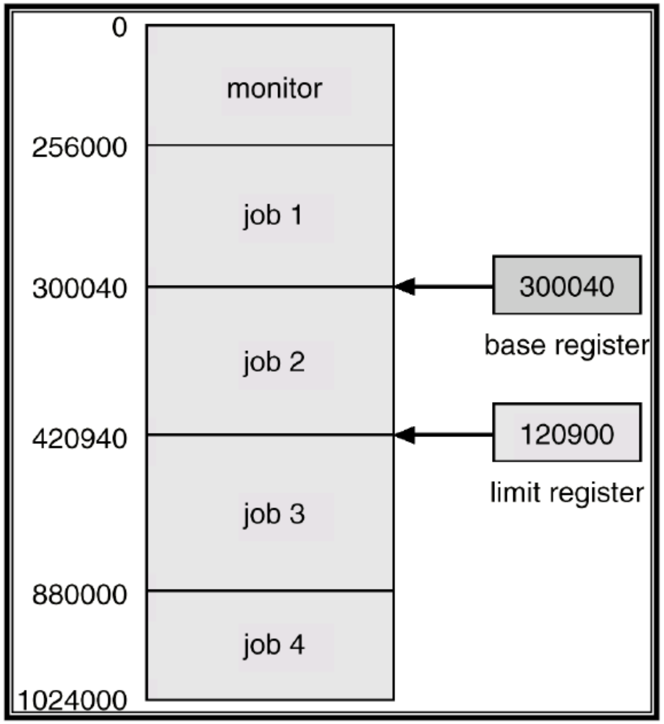
- **Virtual machine** has more than two modes

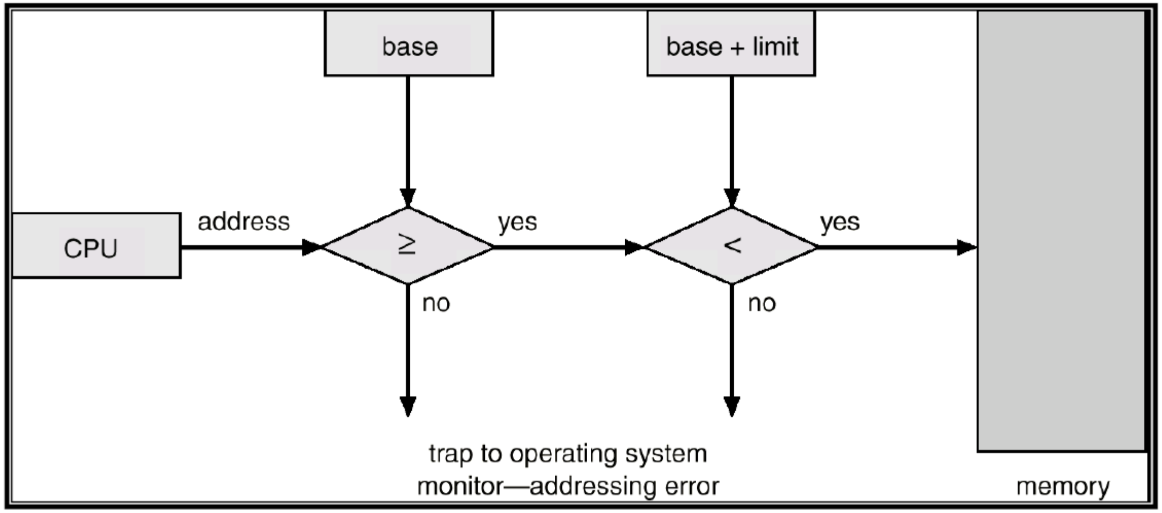


Protection (cont.)

- **I/O protection**
  - I/O devices are scarce resources, user programs must issue I/O through OS
  - Examples: fopen (open), gets (read), puts (write)
- **Memory protection**
  - Prevent a user program from modifying the code or data structures of either the OS or other users
  - Example: instructions to modify the memory space
- **CPU protection**
  - Prevent user programs from sucking up CPU power
  - Implement by **timers** and **time-sharing**
    - Need context switch

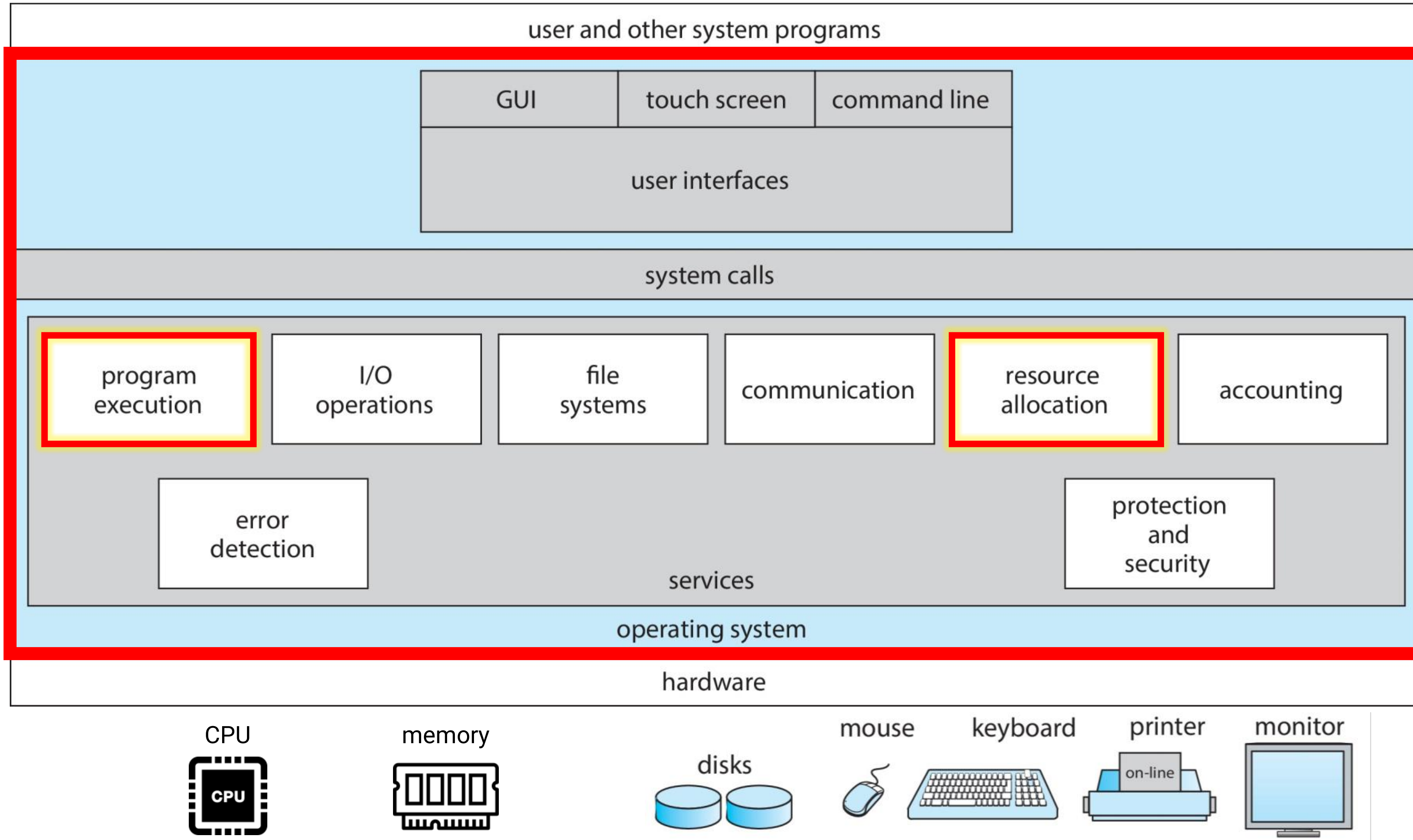
Protection (cont.)





Operating System Service:
Program Execution & Resource Allocation

Operating System Services



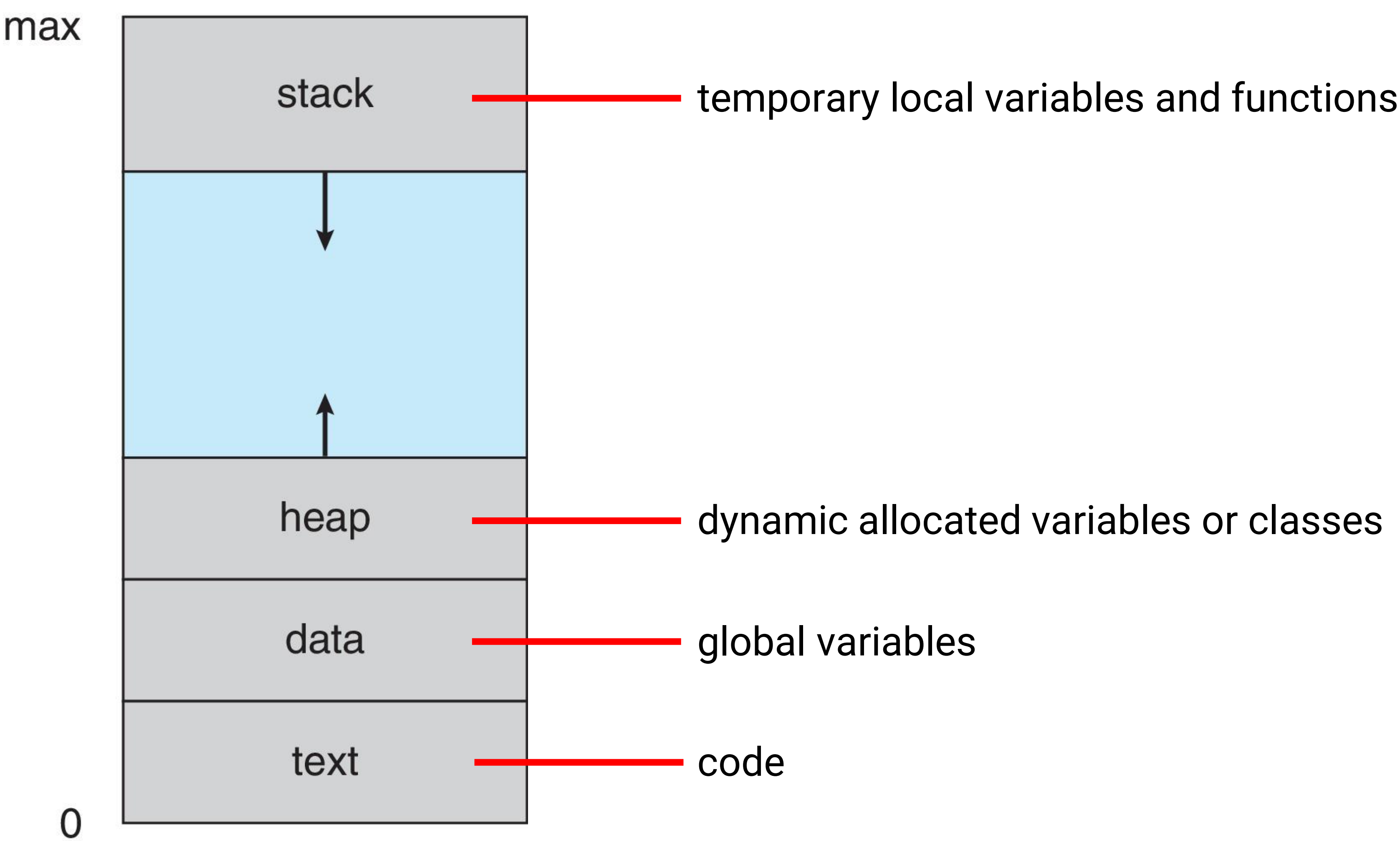
Process Concepts

Process Concepts

- An operating system concurrently executes a variety of programs
  - **Program: passive entity**, binary file stored **in disk**
  - **Process: active entity**, a running program **in memory**

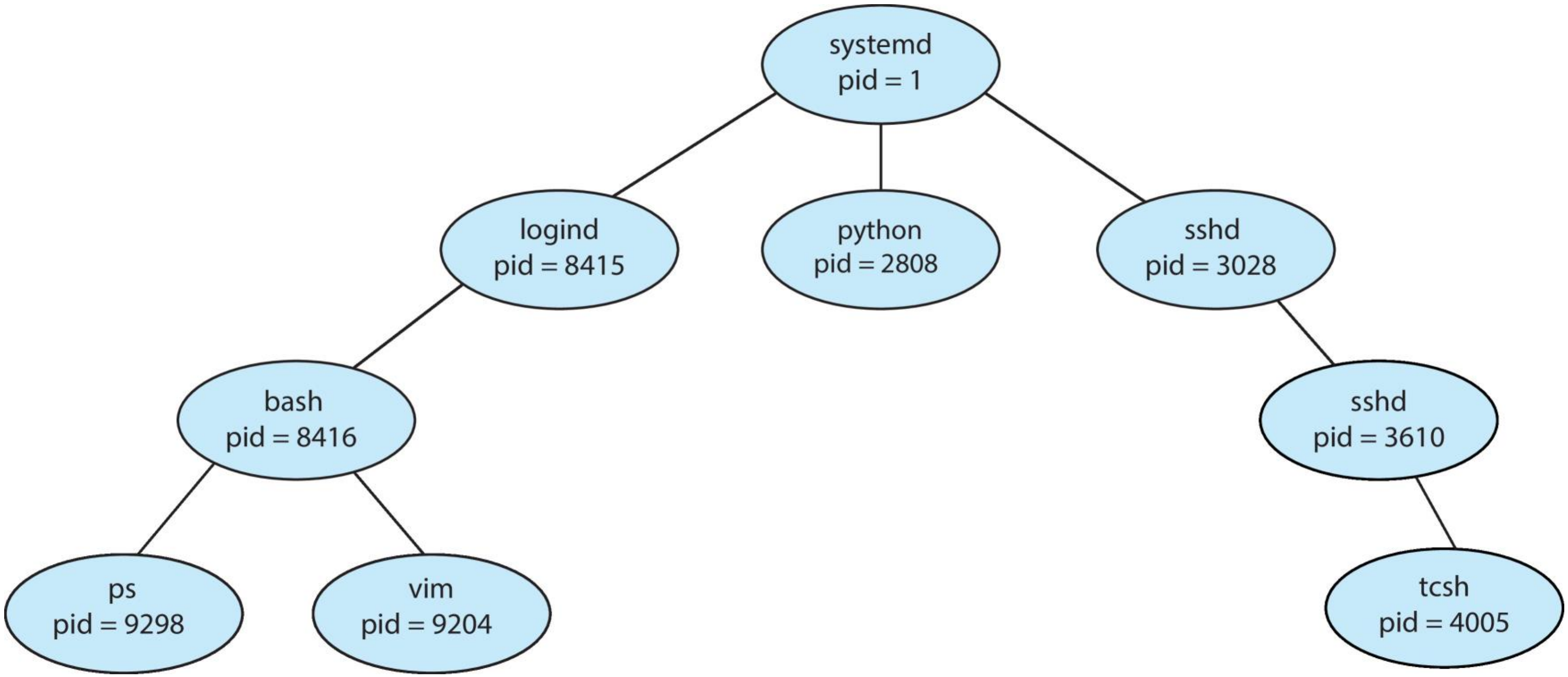
- A process includes
  - **Code segment** (text section)
  - **Data section:** global variables
  - **Stack:** temporary local variables and functions
  - **Heap:** dynamic allocated variables or classes
  - **Current activity** (e.g., program counter, register contents)
  - **Associated resources** (e.g., handlers of open files)

Process in Memory



Tree of Processes

- Each process is identified by a unique **processor identifier (pid)**



Process Creation

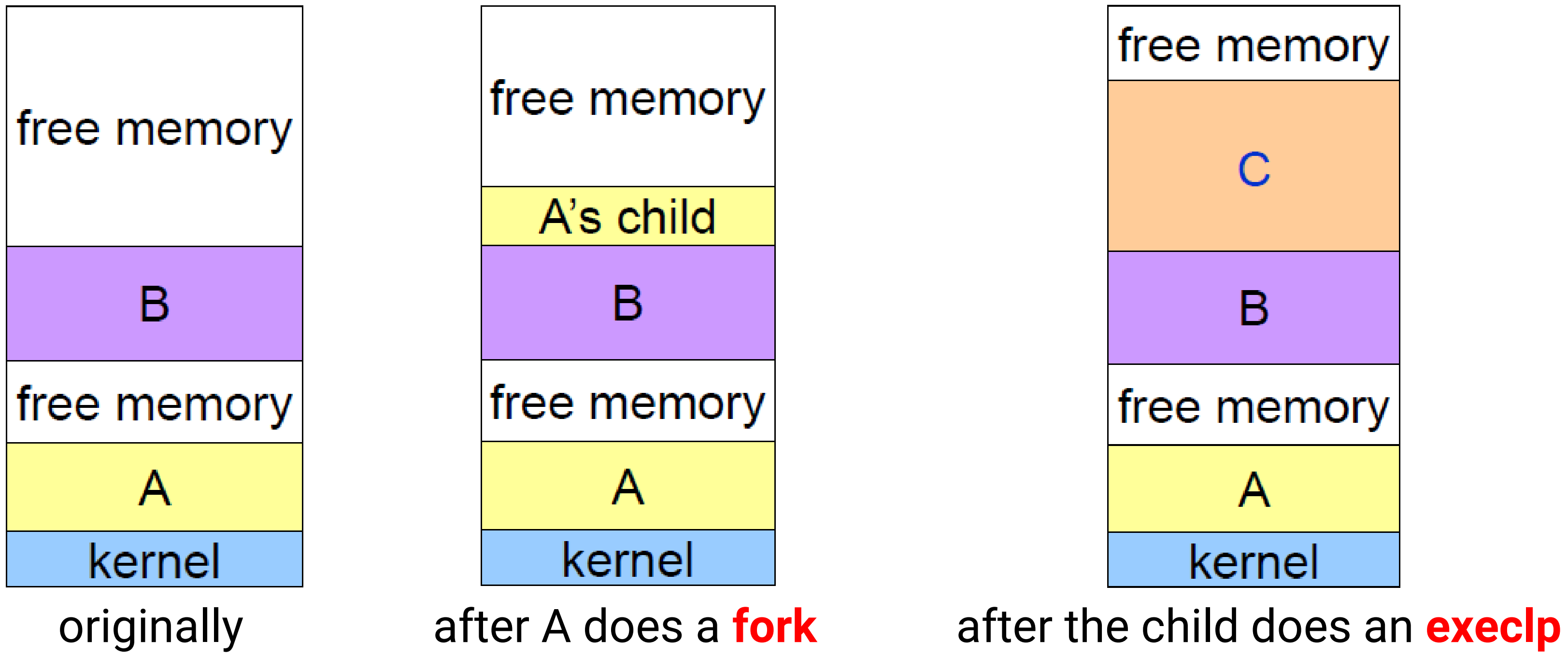
- **Resource sharing** (three possibilities)
  - Parent and child processes share **all** resources
  - Child process shares **subset** of parent's resources
  - Parent and child share **no** resources
- **Execution order** (two possibilities)
  - Parent and children execute **concurrently**
  - Parent **waits until children terminate**
- **Address space** (two possibilities)
  - Children **duplicate** of parent, communication via sharing variables
  - Child **has a program loaded into it**, communication via message passing

UNIX / Linux Process Creation

- **fork system call**
  - Create a new (child) process
  - The new process **duplicates** the address space of its parent
  - Child and parent **execute concurrently** after fork
  - Child: return value of fork is 0
  - Parent: return value of fork is PID of the child process
- **execlp system call**
  - Load a new binary file into memory
  - Destroy the old code
- **wait system call**
  - The parent **waits** for one of its child processes to complete

UNIX / Linux Process Creation (cont.)

- Memory space of fork()
  - Old implementation: A's child is an exact copy of parent
  - Current implementation: use **copy-on-write** technique to store differences in A's child address space



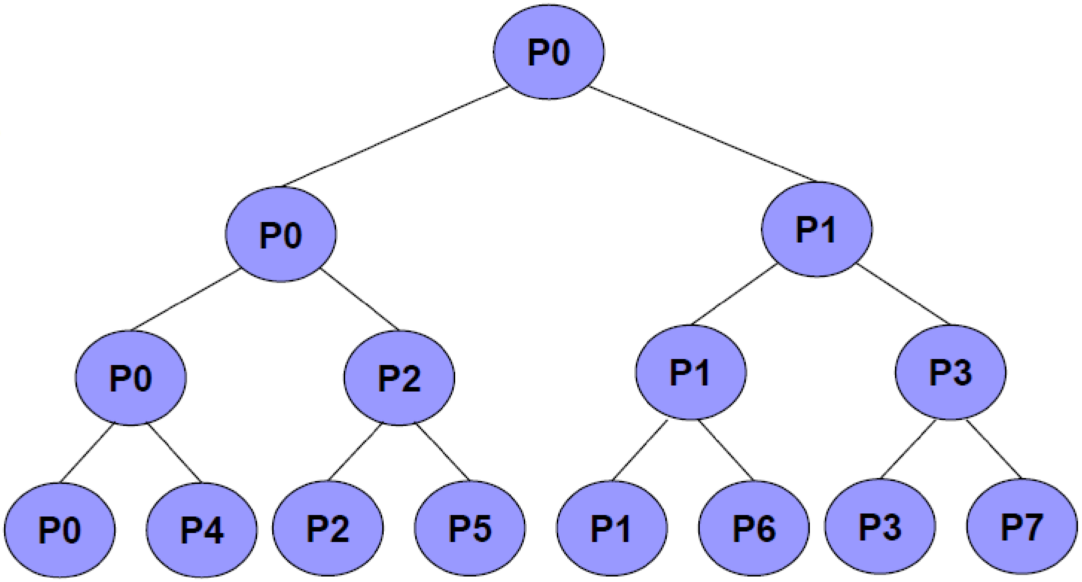
UNIX / Linux Example

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
    }
    else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
    }

    return 0;
}
```

UNIX / Linux Example Quiz

- How many processors are created?

```
#include <stdio.h>
#include <unistd..h>
int main()
{
    for (int i = 0; i < 3 ; i++)
            fork();
    return 0;
}
```



Process Termination

- Terminate when the last statement is executed or **exit()** is called
  - Return status data from child to parent
  - All resources of the process, including physical and virtual memory, open files, I/O buffers, are deallocated by the OS

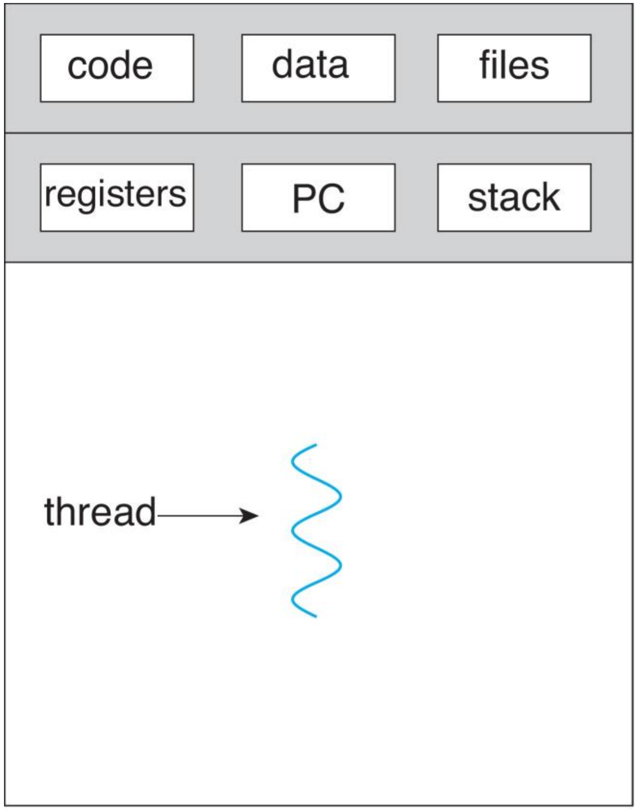
- Parent may terminate execution of children processes by specifying its PID (**abort**)
  - Children has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting, and the OS does not allow a child to continue if its parent terminates

Thread Concepts

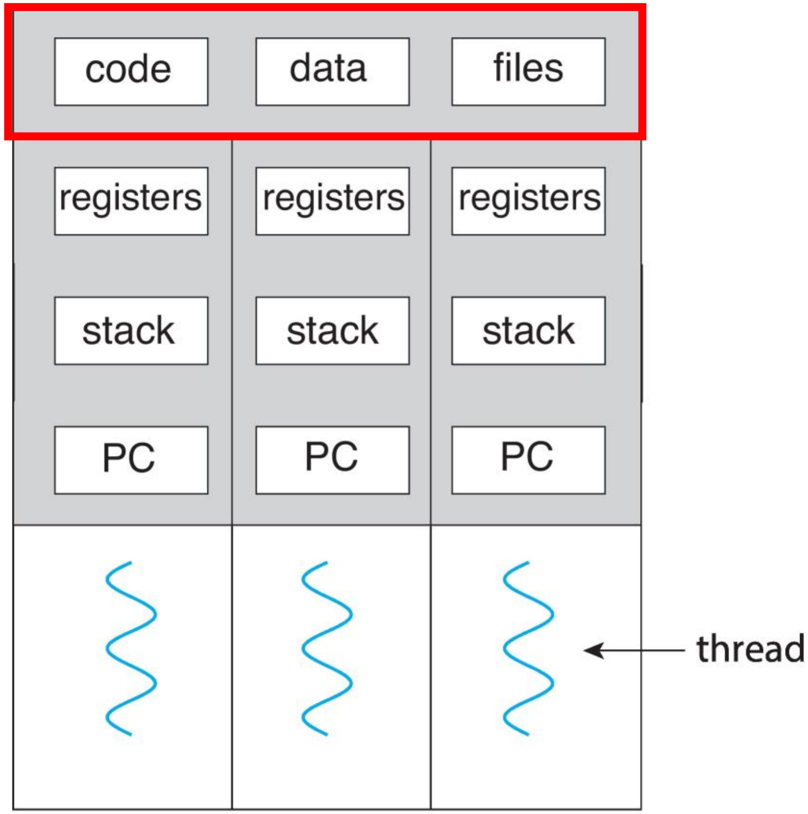
Thread Concepts

- A thread is a **lightweight** process
  - Basic unit of CPU utilization
- All threads belonging to the same process **share**
  - Code section
  - Data section (including dynamic allocated variables)
  - OS resources (e.g., open files)
- **But each thread has its own** (thread control block)
  - Thread ID
  - Program counter
  - Register set
  - Stack

Thread Concept (cont.)



single-threaded process



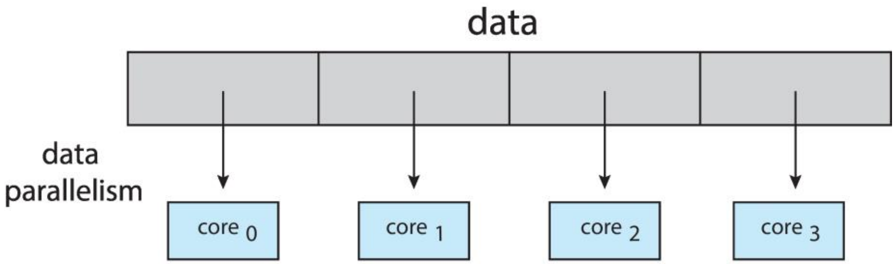
multithreaded process

Benefits of Threads

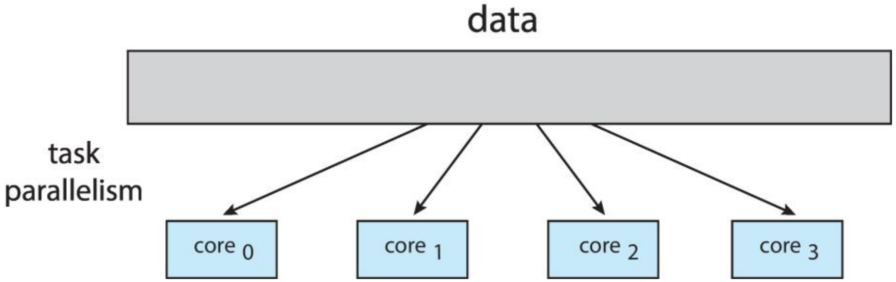
- **Responsiveness**
  - Allow continued running of a process even if part of it is blocked or is performing a lengthy operation
- **Resource sharing**
  - Thread share resources of process
- **Scalability**
  - Take advantage of multicore architectures
- **Economy**
  - Thread creation is cheaper than process creation
    - Ex: creating a thread is 30x cheaper than a process in Solaris
  - Thread switching has lower overhead than context switching

Parallelism

- **Data parallelism**
  - Distribute subsets of the same data across multiple cores
  - Same operation on each



- **Task parallelism**
  - Distribute threads across cores, each thread performing unique operation



Amdahl's Law

- Identify performance gains from adding additional cores to an application that has both serial and parallel components
- Assume ***S*** is the serial portion and the system has ***N*** processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

  - Example: 75% parallel / 25% serial, moving from 1 to 2 cores results in a speedup of 1.6 times
  - As *N* approaches infinity, speedup approaches *1/S*

User Threads and Kernel Threads

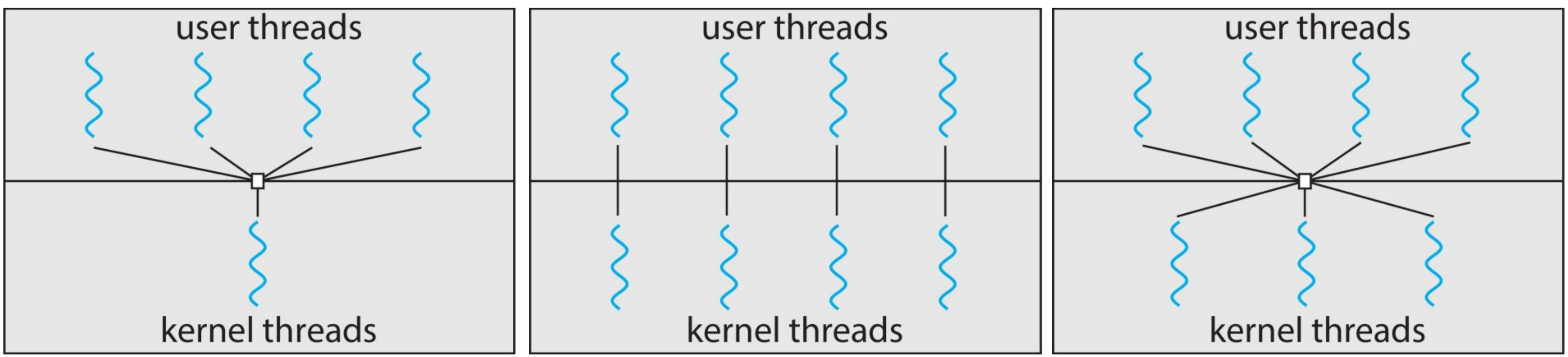
- **User threads**
  - Thread management done by user-level threads library
  - Ex: POSIX Pthreads, Win32 threads, Java threads

- **Kernel threads**
  - Supported by kernel (OS) directly
  - Ex: Windows 2000 (NT), Solaris, Linux, Tru64 UNIX

- Programmers create **user threads** using thread APIs and then the threads are bounded to **kernel threads** by the OS

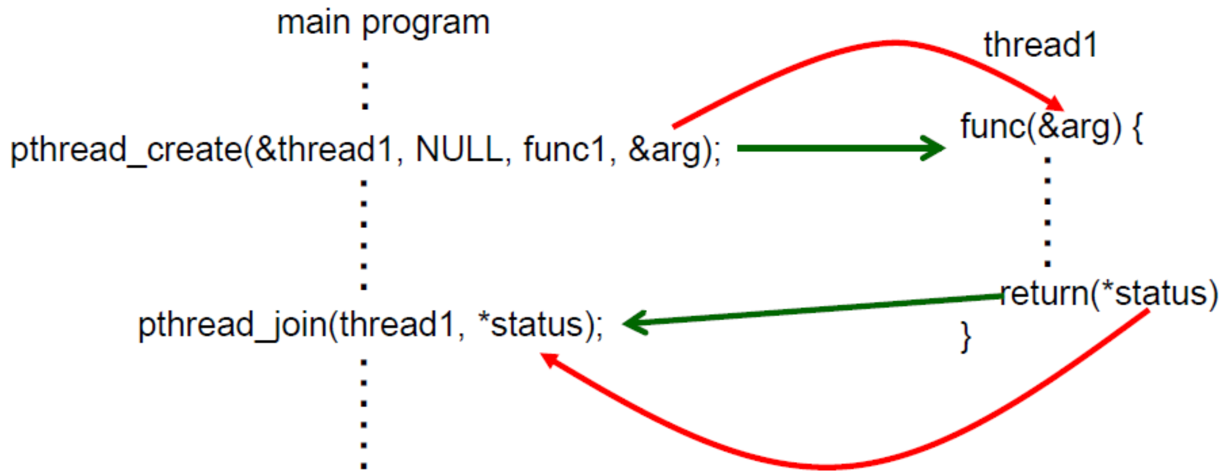
Multi-Threading Models

- **Many-to-One**
- **One-to-One**
- **Many-to-Many**



Pthread Creation

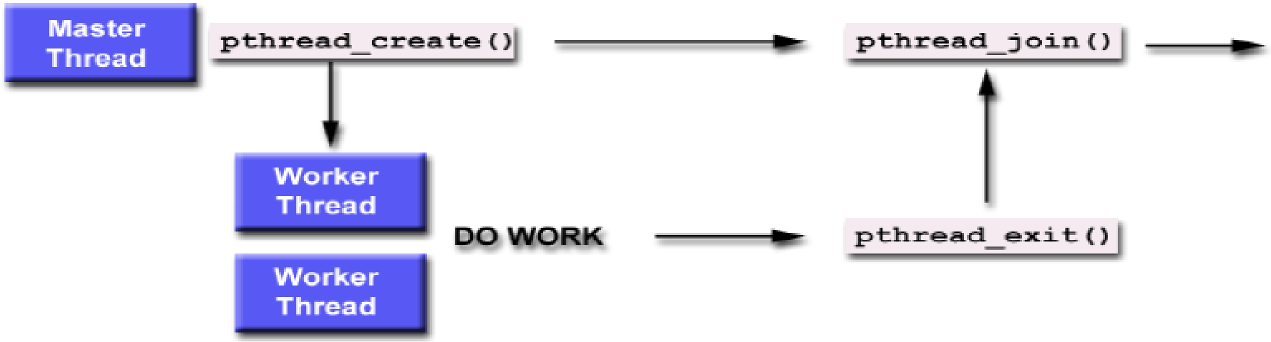
- Library call: ***pthread_create(thread, attr, routine, arg)***
  - *thread*: an unique identifier (token) for the new thread
  - *attr*: used to set thread attributes
  - *routine*: the routine that the thread will execute once it is created
  - *arg*: a single argument that may be passed to routine



Pthread Joining and Detaching

- Library call: ***pthread_join(threadId, status)***
  - Blocks until the specified thread (threadId) terminates
  - One way to accomplish synchronization between threads
  - Example:

    for (int i = 0; i < n ; ++i) pthread_join(thread[i], NULL);
- Library call: ***pthread_detach(threadId)***
  - Once a thread is detached, it can never be joined
  - Detach a thread could free some system resources



Linux Kernel Threads

- Linux implements threads by **extending** processes
- **The fork system call**
  - Create a new process and a copy of the associated data of the parent process
- **The clone system call**
  - Create a new process and a link that points to the associated data of the parent process

Linux Threads (cont.)

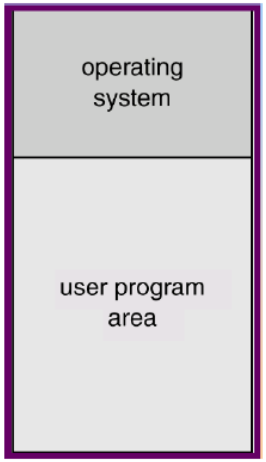
- A set of **flags** is used in the clone system call for indication of the level of the sharing
  - None of the flags is set ➔ clone = fork
  - All flats are set ➔ parent and child share everything

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

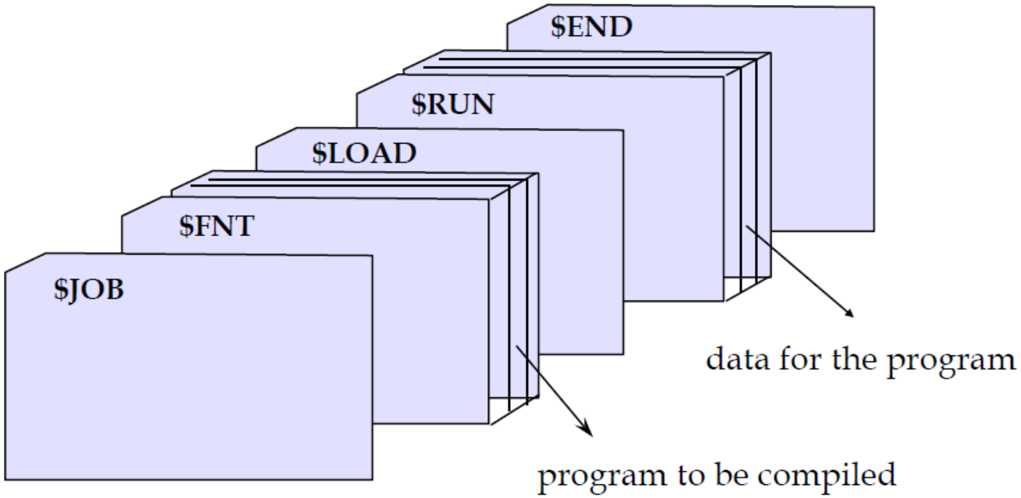
Process Execution Strategy

Simple Batch System

- Workflow
    - Users submit data (program, data, control card)
    - Operator sort jobs with similar requirement
    - OS simply transfer control from one job to the next one
        - Resident monitor: automatically transfer control from one job the next

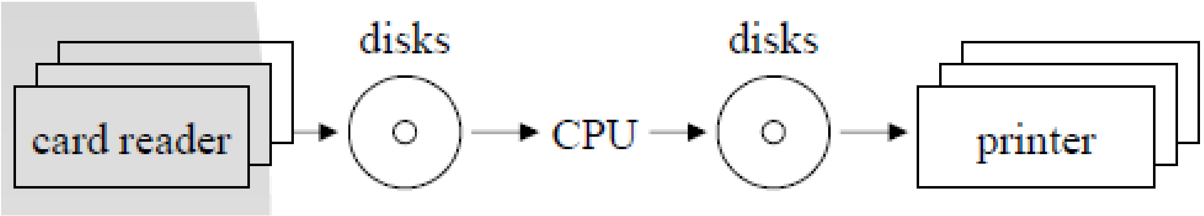


Memory layout



Simple Batch System (cont.)

- **Problem of batch systems**
  - One job at a time ➔ multi-programming
  - No interaction between users and jobs ➔ time sharing
  - CPU is often idle
- **Spooling** (Simultaneous Peripheral Operation On-Line)
  - Replace sequential-access devices with random-access devices (disks)



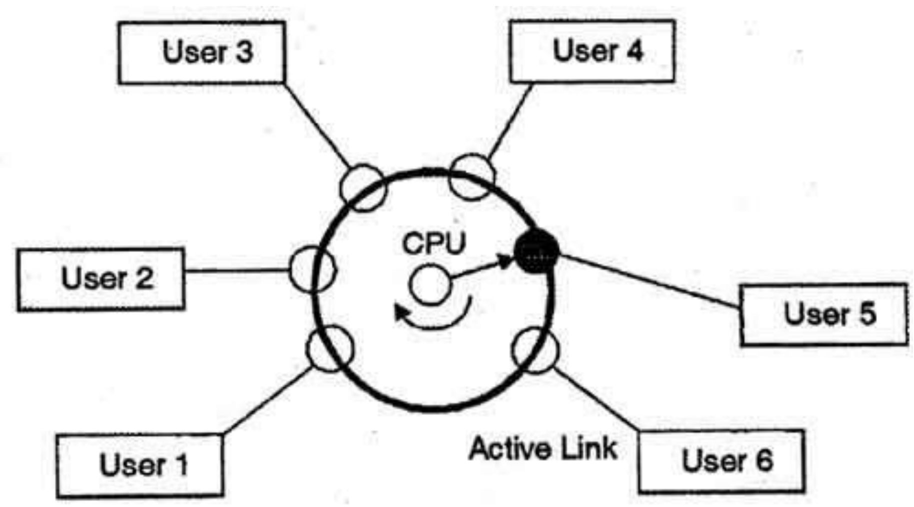
  - Ex: printer
    - Data for printing is first written into a directory as files, then printed by the printer

Multi-Programming

- Single user cannot always keep CPU and I/O devices busy
    - Even with spooling, disk I/O is still too slow compared to CPU and memory
- **Put multiple programs in memory**
- **OS organizes jobs** so that the CPU always has one to execute
    - When job has to wait (e.g., for I/O), OS switches to another job according to a **scheduling algorithm**
    - Increase CPU utilization

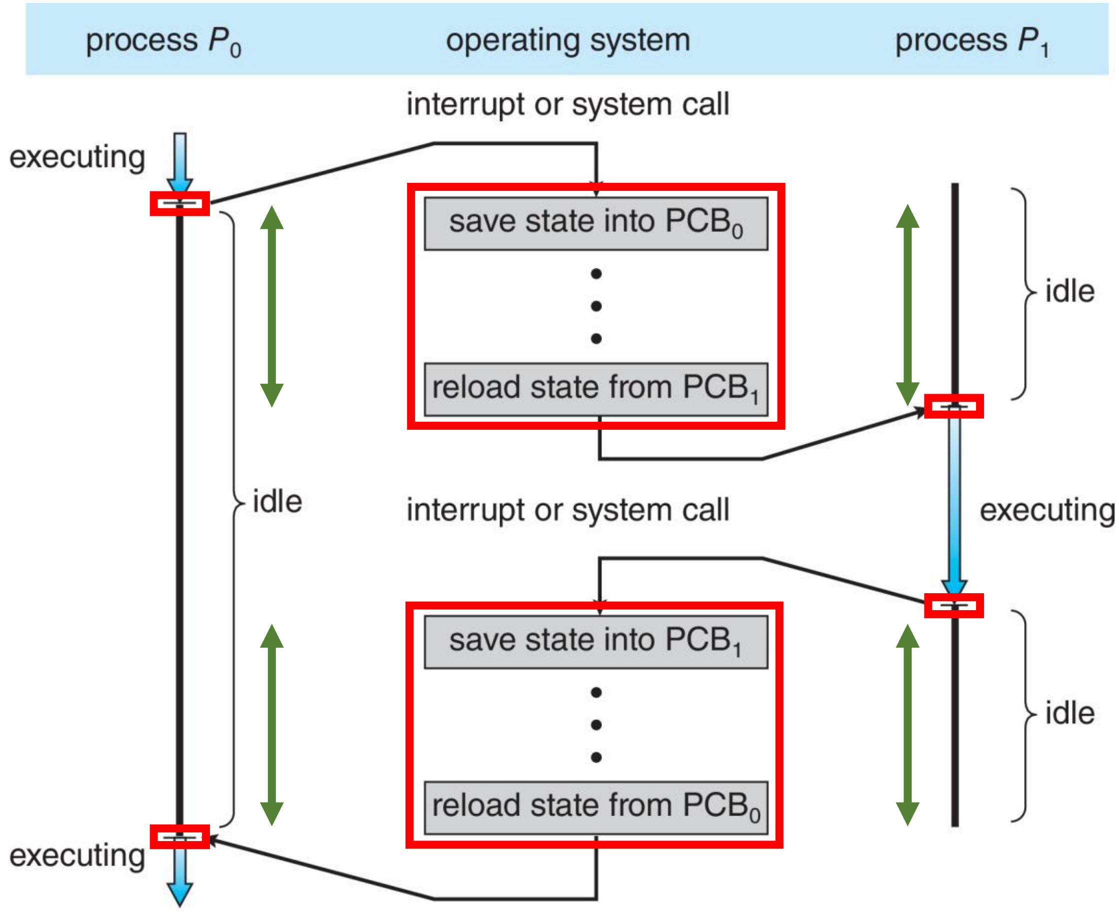
Time-Sharing (Multi-Tasking)

- CPU switches jobs frequently so that users can interact with each job while it is running
  - A logical extension of multi-programming
  - **Interactivity** !
    - Response time should be less than 1 sec.



Context Switch

- Occurs when the CPU switches from one process to another

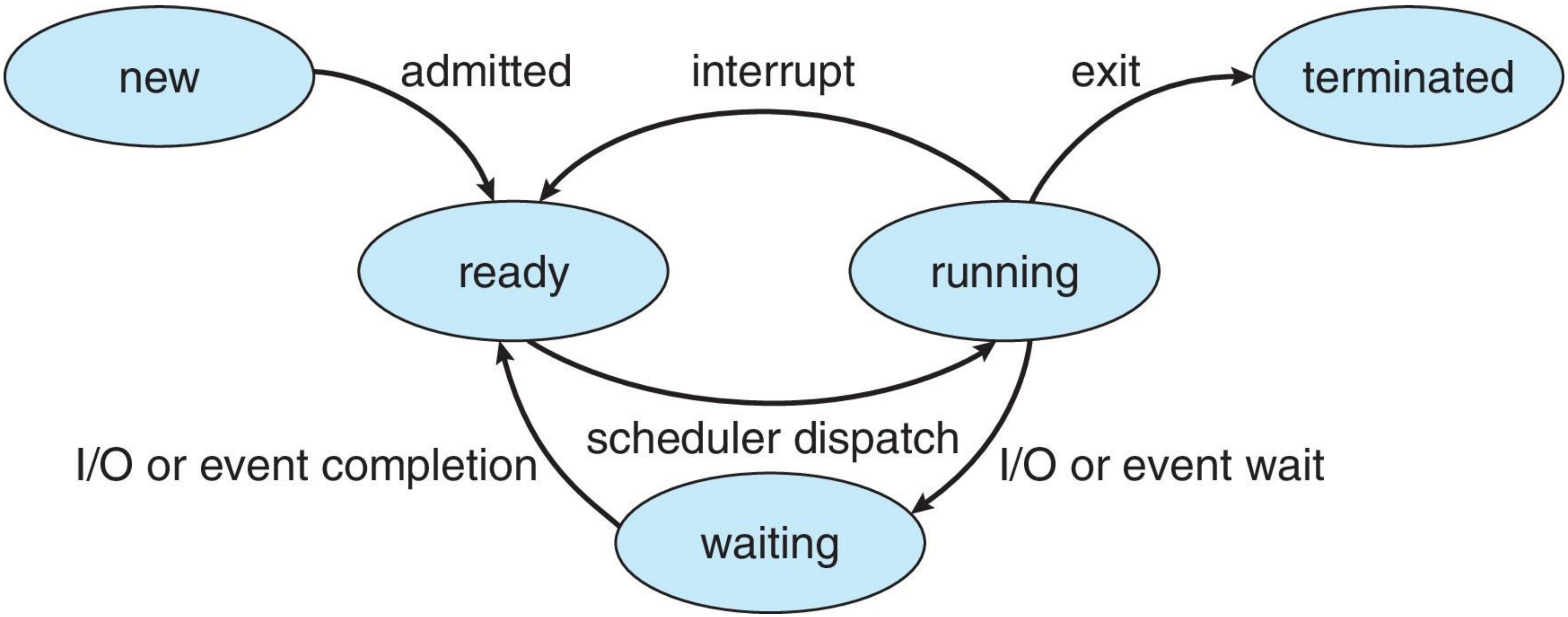


Process Scheduling Overview

Process State

- Types of states
  - **New**
    - The process is being created
  - **Ready**
    - The process is in the memory waiting to be assigned to a processor
  - **Running**
    - The process whose instructions are being executed by CPU
  - **Waiting**
    - The process is waiting for events to occur
  - **Terminated**
    - The process has finished execution

Process State (cont.)

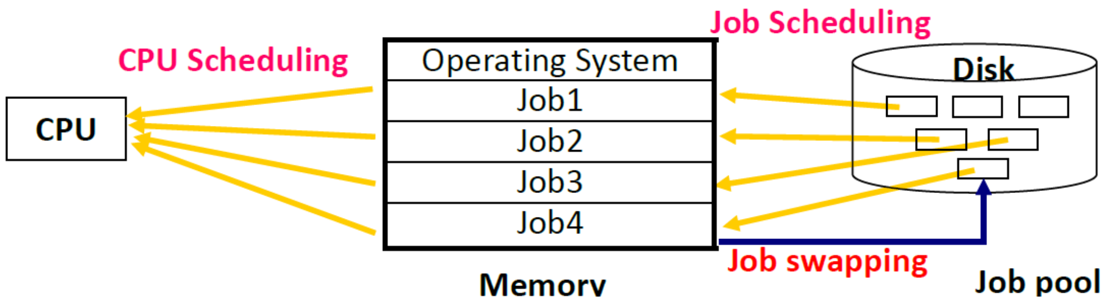


Only one process is running on any processor at any instant

However, many processes may be ready or waiting (put into a queue)

Process Schedulers

- **Short-term scheduler (CPU scheduler)**
  - Select which process should be executed and allocated CPU **(Ready state ➔ Running state)**
- **Long-term scheduler (job scheduler)**
  - Select which processes should be loaded into memory and brought into the ready queue **(New state ➔ Ready state)**
- **Medium-term scheduler**
  - Select which processes should be swapped in/out memory **(Ready state ➔ Waiting state)**



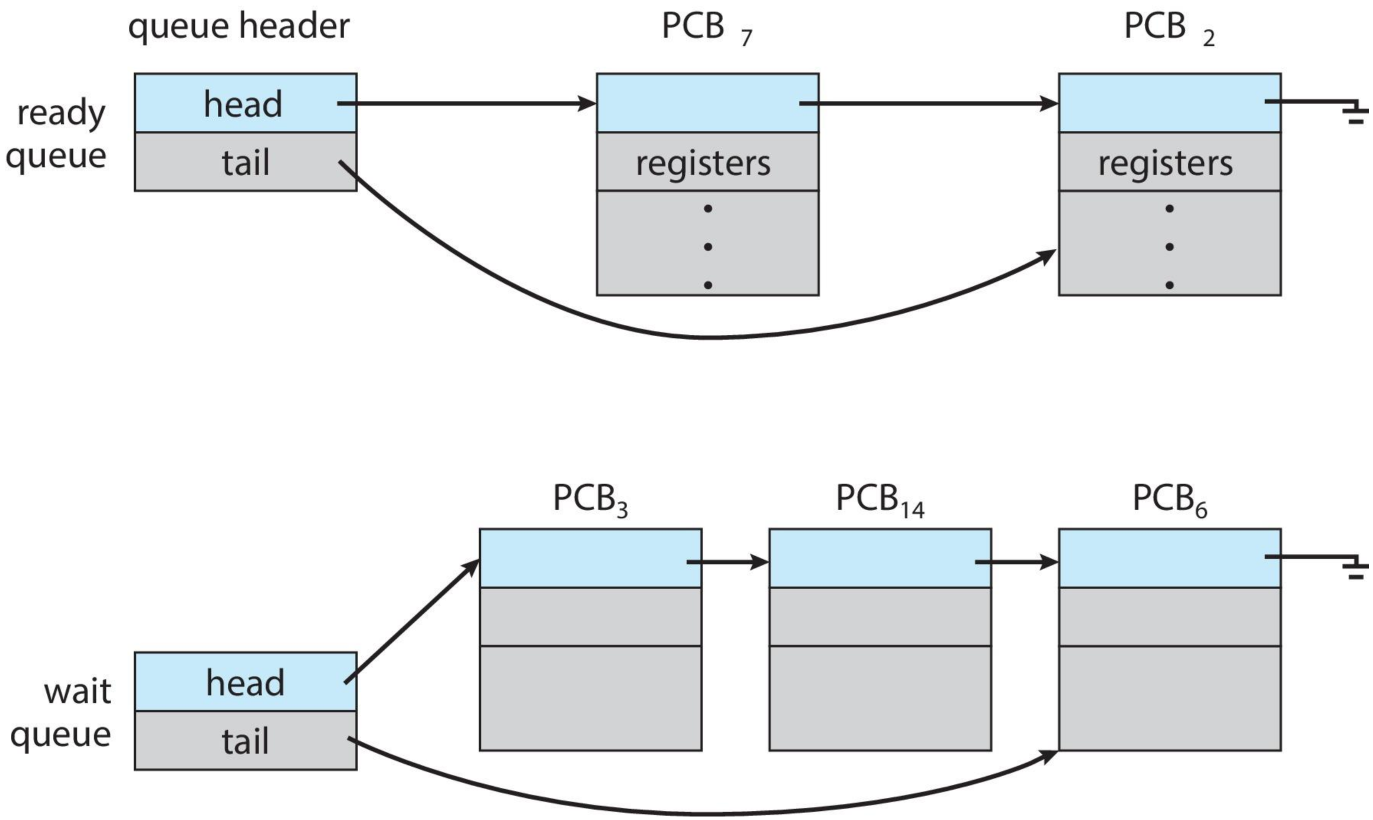
Process Scheduling Queues

- Maintain **scheduling queues** of processes
  - **Job queue (New state)**
    - Set of all processes in the system
  - **Ready queue (Ready state)**
    - Set of all processes residing in main memory
    - Ready and waiting to execute
  - **Waiting queue (Wait State)**
    - Set of processes waiting for an event (e.g., I/O)

  - Processes migrate among the various queues

Process Scheduling Queues (cont.)

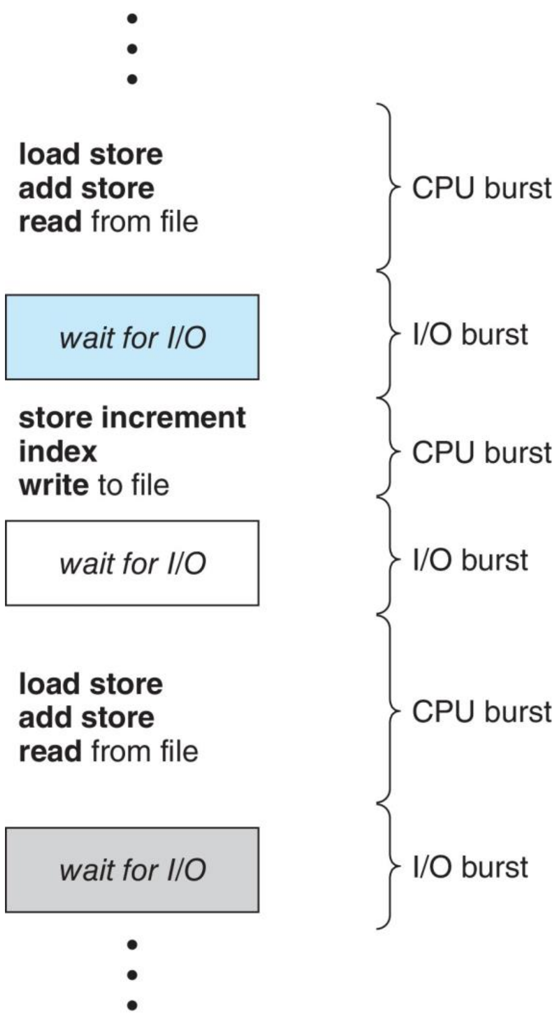
- Ready queue and waiting queue



CPU Scheduling Algorithms

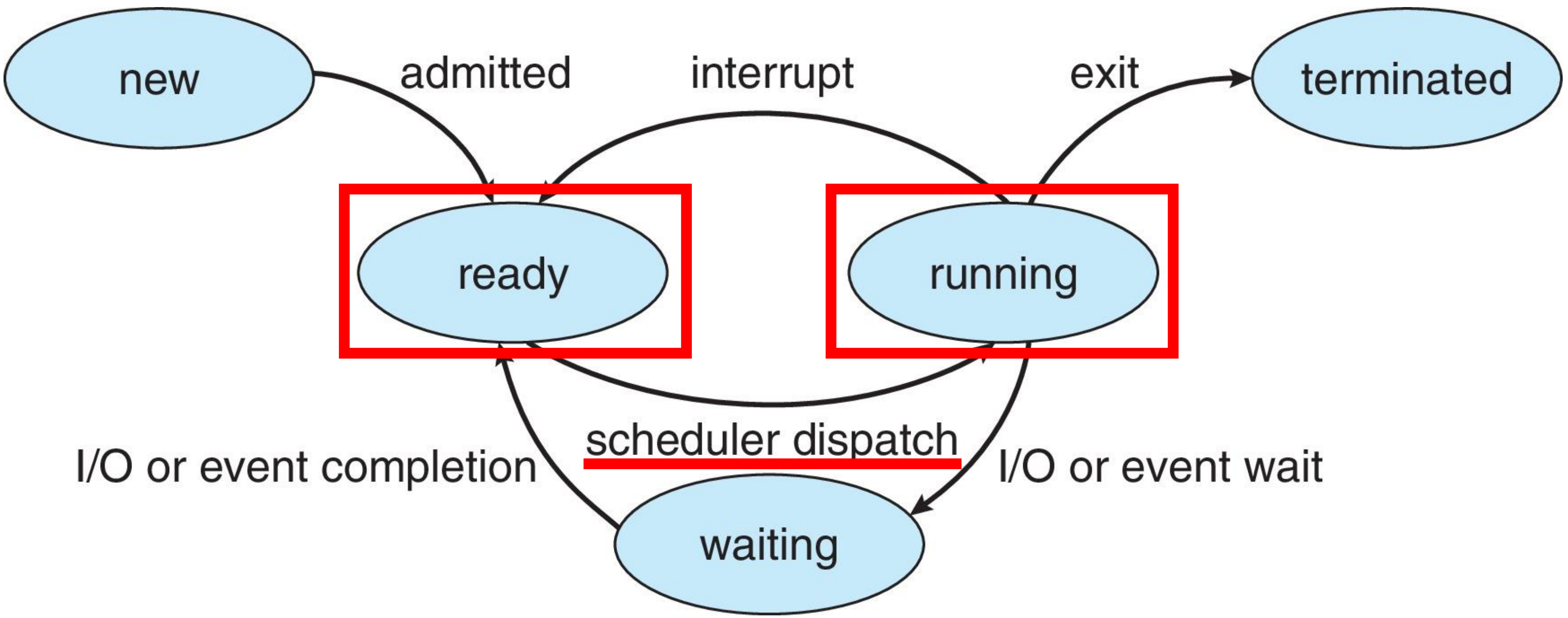
Basic Concepts

- **CPU-I/O burst cycle**
  - Process execution consists of a cycle of **CPU execution (CPU burst)** and **I/O wait (I/O burst)**
    - Generally, there is a large number of short CPU bursts, and a small number of long CPU bursts
    - An I/O-bound program would typically has many very short CPU bursts
    - A CPU-bound program might have a few long CPU bursts



CPU Scheduler

- Selects process from ready queue to execute
  - Allocate a CPU for the selected process



Preemptive vs Non-preemptive

- CPU scheduling decisions may take place when a process
  - Switches from running to waiting state
  - Switches from running to ready state
  - Switches from waiting to ready
  - Terminates
- **Non-preemptive scheduling**
  - Scheduling under 1 and 4 (no choice in terms of scheduling)
  - The process keeps the CPU until it is **terminated** or **switched to the waiting state**
- **Preemptive scheduling**
  - Scheduling under all cases
  - E.g., Windows 95 and subsequent versions, Mac OS X

Scheduling Criteria

- **CPU utilization**
  - Theoretically 0% ~ 100%
  - Real systems: 40% (light) ~ 90% (heavy)
- **Throughput** — system view
  - Number of completed processes per time unit
- **Turnaround time**
  - Submission ~ completion
- **Waiting time**
  - Total waiting time in the ready queue
- **Response time**
  - Submission ~ the first response is produced

single job view

Algorithms

- First-Come, First-Served (FCFS) scheduling
- Shortest-Job-First (SJF) scheduling
- Priority scheduling
- Round-Robin scheduling
- Multi-level queue scheduling
- Multi-level feedback queue scheduling

FCFS Scheduling

- Process (burst time) in arriving order
  - P1 (24), P2 (3), P3 (3)
- The Gantt Chart of the schedule

  - Waiting time: P1 = 0, P2 = 24, P3 = 27
  - Average Waiting Time (AWT): (0 + 24 + 27) / 3 = 17
  - **Convoy effect**
    - Short processes behind a long process

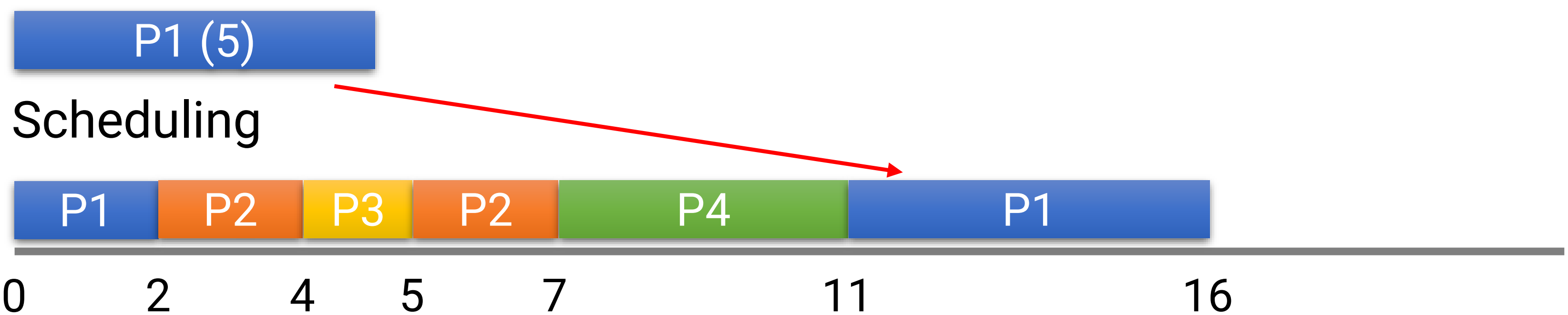
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
- A process with shortest burst length gets the CPU first
- SJF provides the **minimum** (**optimal**) **average waiting time**
- Two schemes
  - **Non-preemptive**
    - Once the CPU is given to a process, it cannot be preempted until its completion
  - **Preemptive**
    - If a new process arrives with shorter burst length, preemption happens

Preemptive SJF Example

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Ready queue at t = 11



- **Wait time = completion time - arrival time - run time**
- **AWT** = [(16-0-7) + (7-2-4) + (5-4-1) + (11-5-4)] / 4 = 3
- **Response time**: P1 = 0, P2 = 0, P3 = 0, P4 = 2

Approximate Shortest-Job-First (SJF)

- **SJF difficulty**
  - No way to know length of the next CPU burst
- **Approximate SJF**
  - The next burst can be predicted as an exponential average of the measured length of previous CPU bursts

$$\tau_{n+1} = \alpha \underbrace{t_n}_{\text{new one}} + (1-\alpha)\underbrace{\tau_n}_{\text{history}}$$

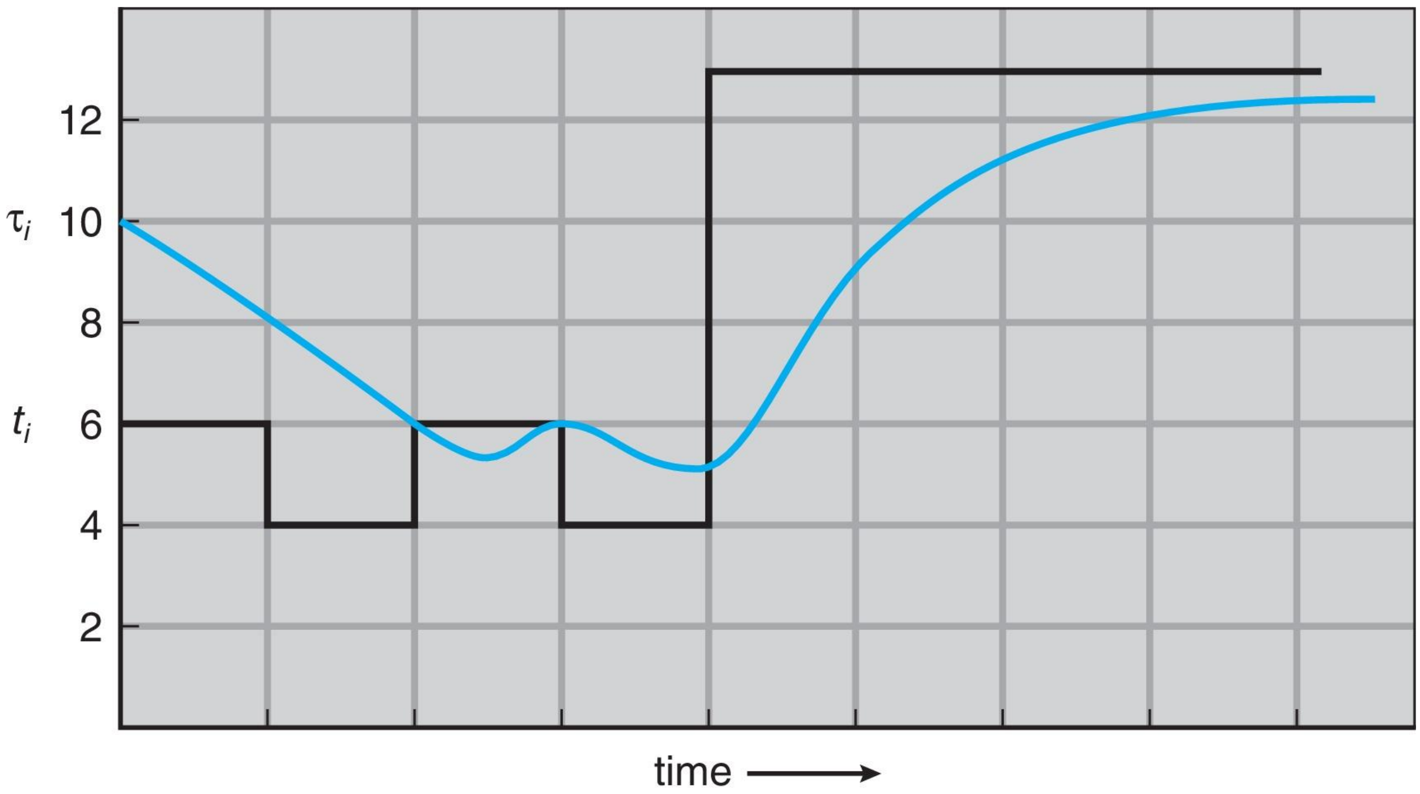
$$= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2} + \ldots$$

Example:

$\alpha = 1/2$

$$= (\frac{1}{2})t_n + (\frac{1}{2})^2 t_{n-1} + (\frac{1}{2})^3 t_{n-2}$$

Exponential Prediction of Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Priority Scheduling

- A **priority number** is associated with each process
- The CPU is allocated to the highest priority process
  - Preemptive
  - Non-preemptive
- **SJF is a priority scheduling** where priority is the predicted next CPU burst time
- Problem: **starvation**
  - Low priority processes never execute
  - Example: IBM 7094 shutdown at 1973, a 1967-process never run
  - Solution: **aging**
    - As time progresses, increase the priority of processes

Round-Robin (RR) Scheduling

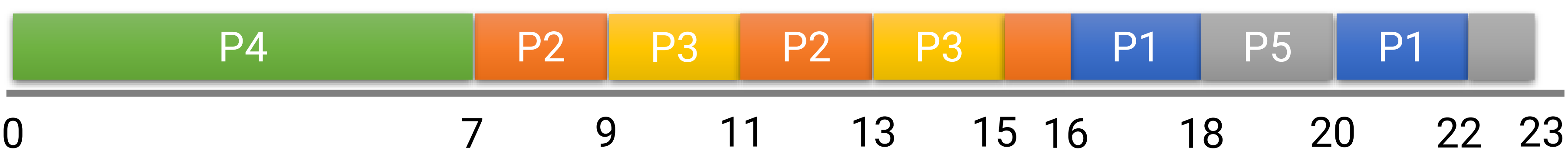
- Each process gets a small unit of CPU time (**time quantum**, TQ), usually 10 ~ 100 ms
  - Context switch time usually < 10 microseconds
- After TQ elapsed, process is **preempted** and **added to the end of the ready queue**
- If there are *n* processes in the ready queue and the time quantum is *q*, each process gets *1/n* of the CPU time (q time units)
  - No process waits more than *(n-1)q* time units
- Performance
  - TQ large ➔ FCFS
  - TQ small ➔ (context switch) overhead increases

Priority Scheduling with Round-Robin

Process	Burst Time	Priority
P1	4	3
P2	5	2
P3	4	2
P4	7	1
P5	3	3

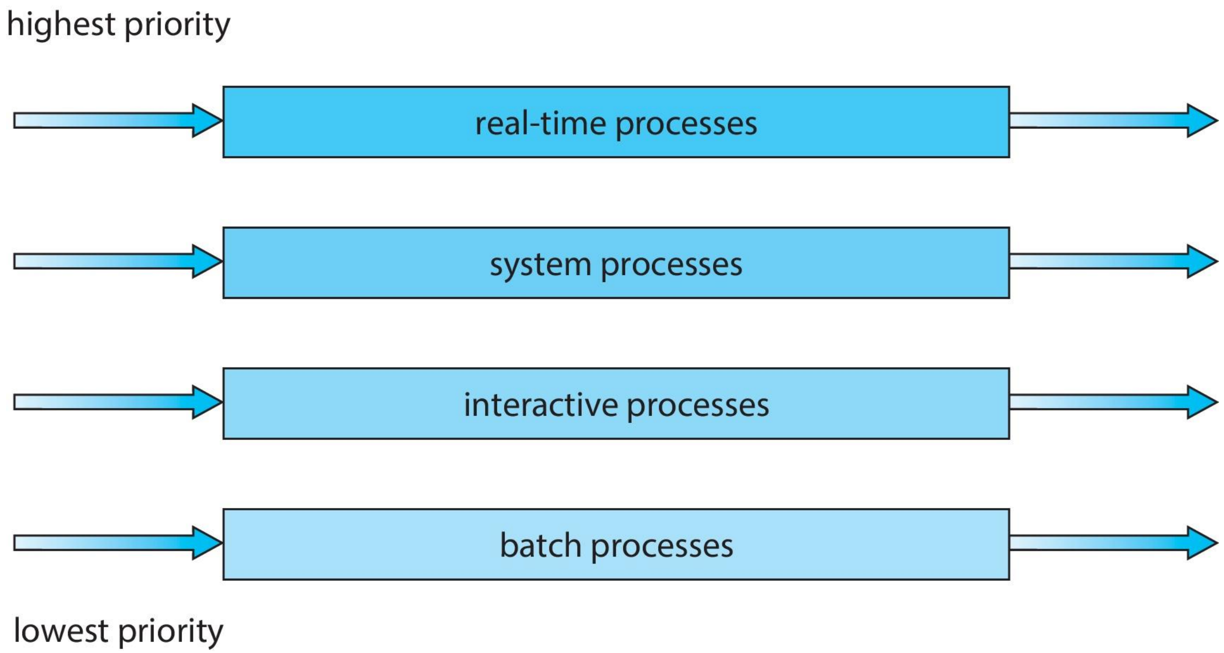
- Run the process with the highest priority
- Processes with the same priority run round-robin

Gantt Chart (TQ = 2)



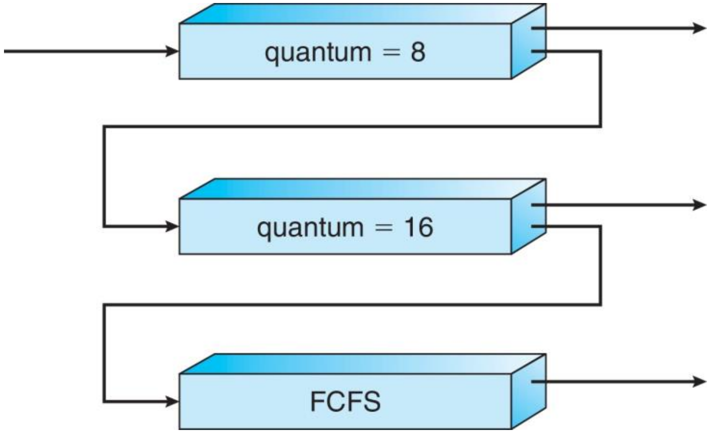
Multi-level Queue Scheduling

- Ready queue is partitioned into separate queues
- Each queue has its own scheduling algorithm
- Scheduling must be done **between** queues
  - Time slice: each queue gets a certain amount of CPU

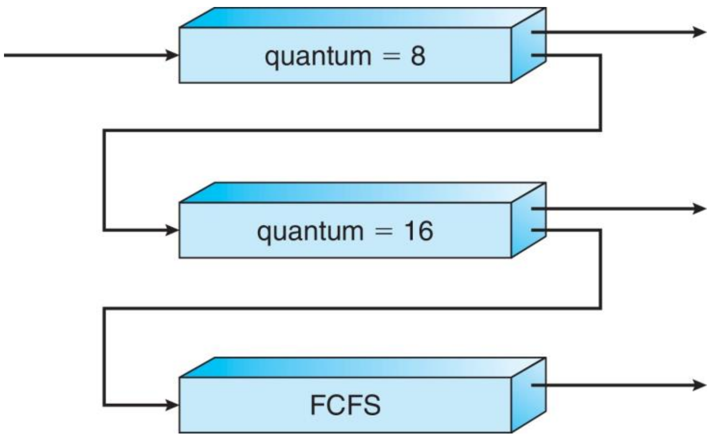


Multi-level Feedback Queue Scheduling

- A process can move between the various queues
  - Aging must be implemented
- Idea: separate processes according to the characteristic of their CPU burst
  - **I/O-bound** and **interactive processes** in **higher priority** queue ➔ short CPU burst
  - **CPU-bound** processes in **lower priority** queue ➔ long CPU burst



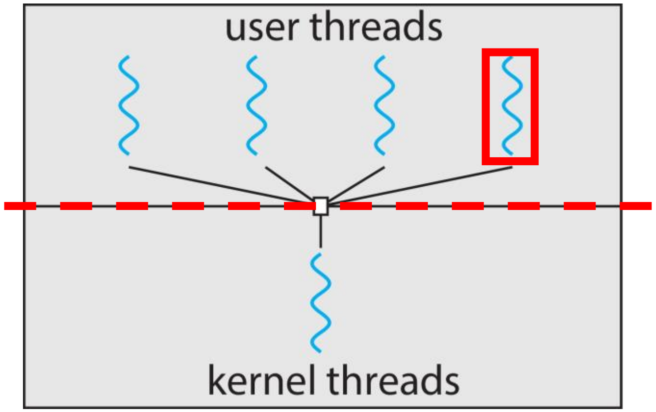
Multi-level Feedback Queue (cont.)



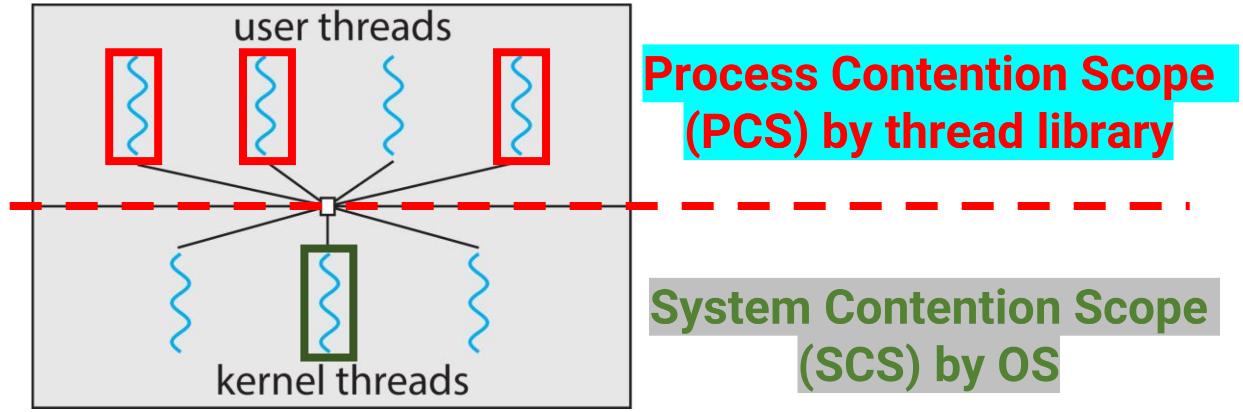
- Three queues
  - Q0: RR with TQ 8 ms.
  - Q1: RR with TQ 16 ms.
  - Q2: FCFS
- Scheduling
  - A new process enters queue Q0 which is served in RR
    - When it gains CPU, the process receives 8 ms
    - If it does not finish in 8 ms., the process is moved to queue Q1
  - At Q1, job is again served in RR and receives 16 additional ms.
    - If it still does not complete, it is preempted and moved to queue Q2

Thread Scheduling

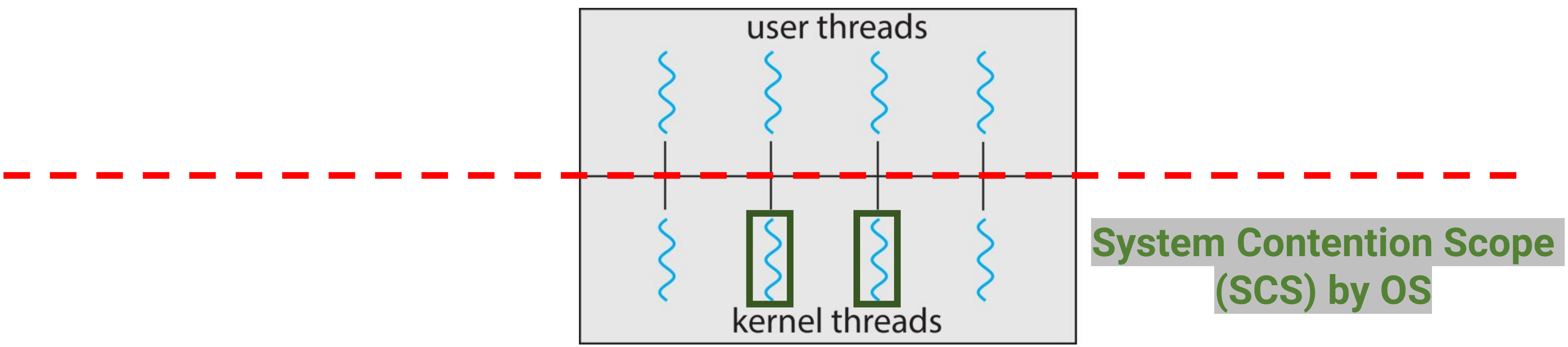
Many-to-One



Many-to-Many



One-to-One

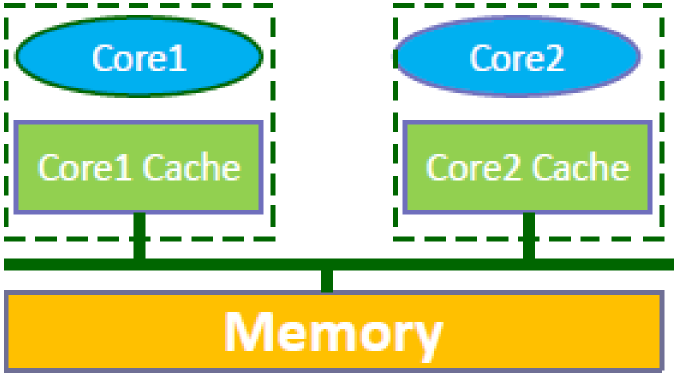


Special Scheduling Issues

Processor Affinity

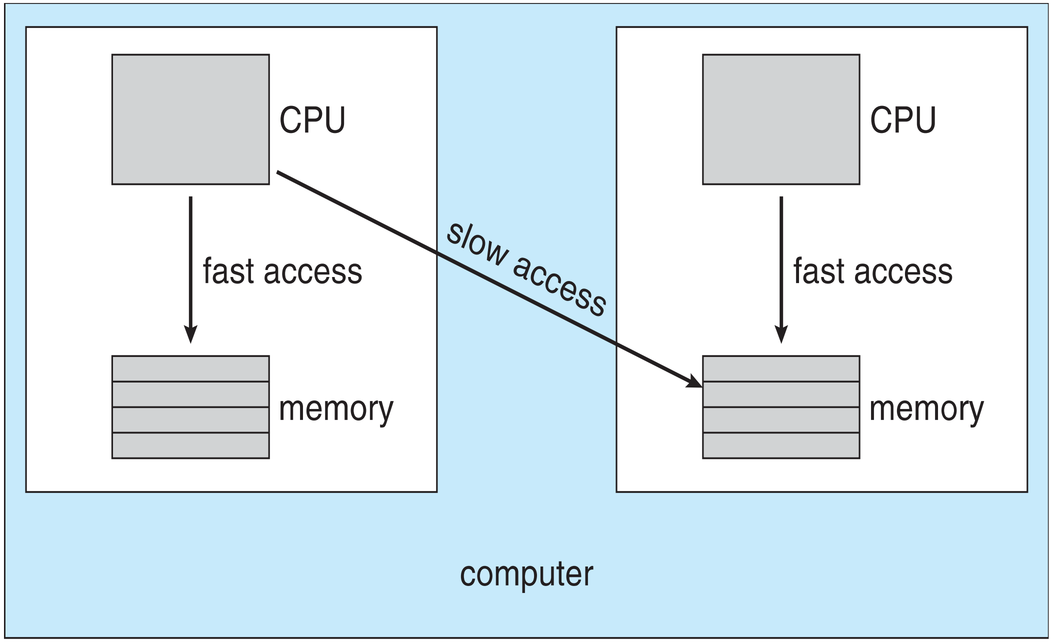
- **Processor affinity**
  - A process has an affinity for the processor on which it is currently running
    - A process populates its recent used data in cache
    - Cache invalidation and repopulation has high cost

- Solution
  - **Soft affinity**
    - Possible to migrate to other processors
  - **Hard affinity**
    - Not to migrate to other processor



NUMA and CPU Scheduling

- Occurs in systems containing combined CPU and memory boards
- CPU scheduler and memory-placement works together



Load Balancing

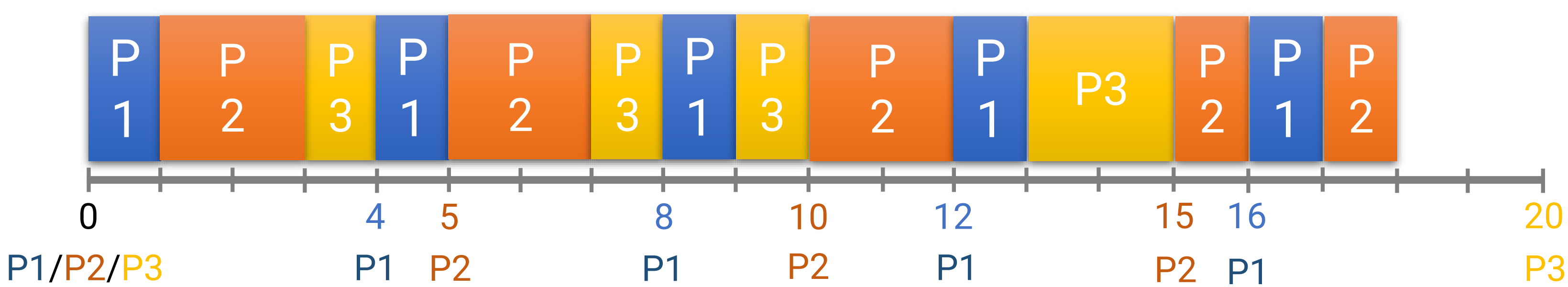
- Keep the workload evenly distributed across all processors
  - Only necessary on systems where each processor has its own private queue of eligible processes to execute
- **Two strategies**
  - **Push migration**
    - Move (push) processes from overloaded to idle or less-busy processor
  - **Pull migration**
    - Idle processor pulls a waiting task from a busy processor
  - Often implemented in parallel
- Load balancing often counteracts the benefits of processor affinity

Real-time Scheduling Algorithms

- Must support **preemptive**, **priority-based** scheduling
  - But only guarantees **soft real-time**
- **Description**
  - T1 = (0, 4, 10) == (Ready, Execution, Period)
  - T2 = (1, 2, 4)
- **Rate-Monotonic (RM) algorithm**
  - **Shorter period ➔ high priority**
  - Fixed-priority real-time system scheduling algorithm
- **Earliest-deadline-first (EDF) algorithm**
  - **Earlier deadline ➔ higher priority**
  - Dynamic priority algorithm

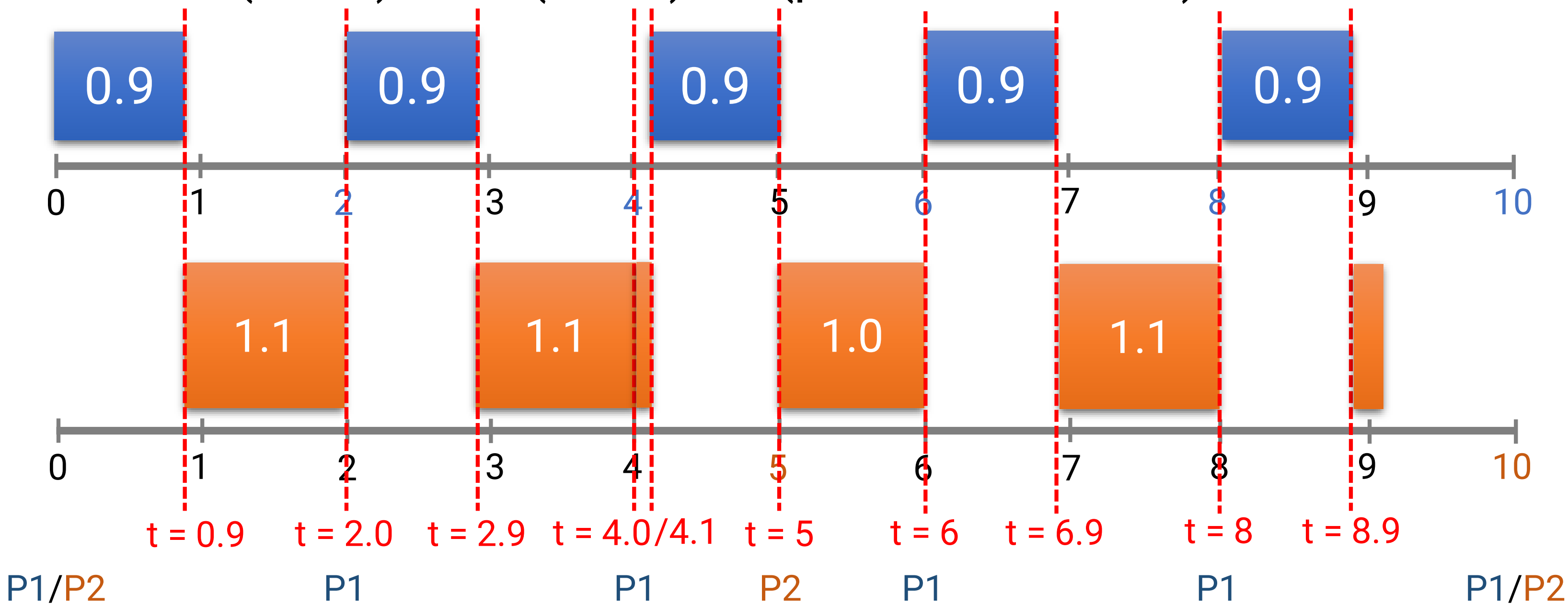
Rate-Monotonic (RM) Scheduler

- **Fixed-priority scheduling**
  - All jobs of the same task have same priority
  - The task's priority is fixed
- **The shorter period, the higher priority**
- Example (Preempted)
  - T1 = (4, 1), T2 = (5, 2), T3 = (20, 5) for (period, execution)
    - Period: 4 < 5 < 20
    - Priority: T1 > T2 > T3



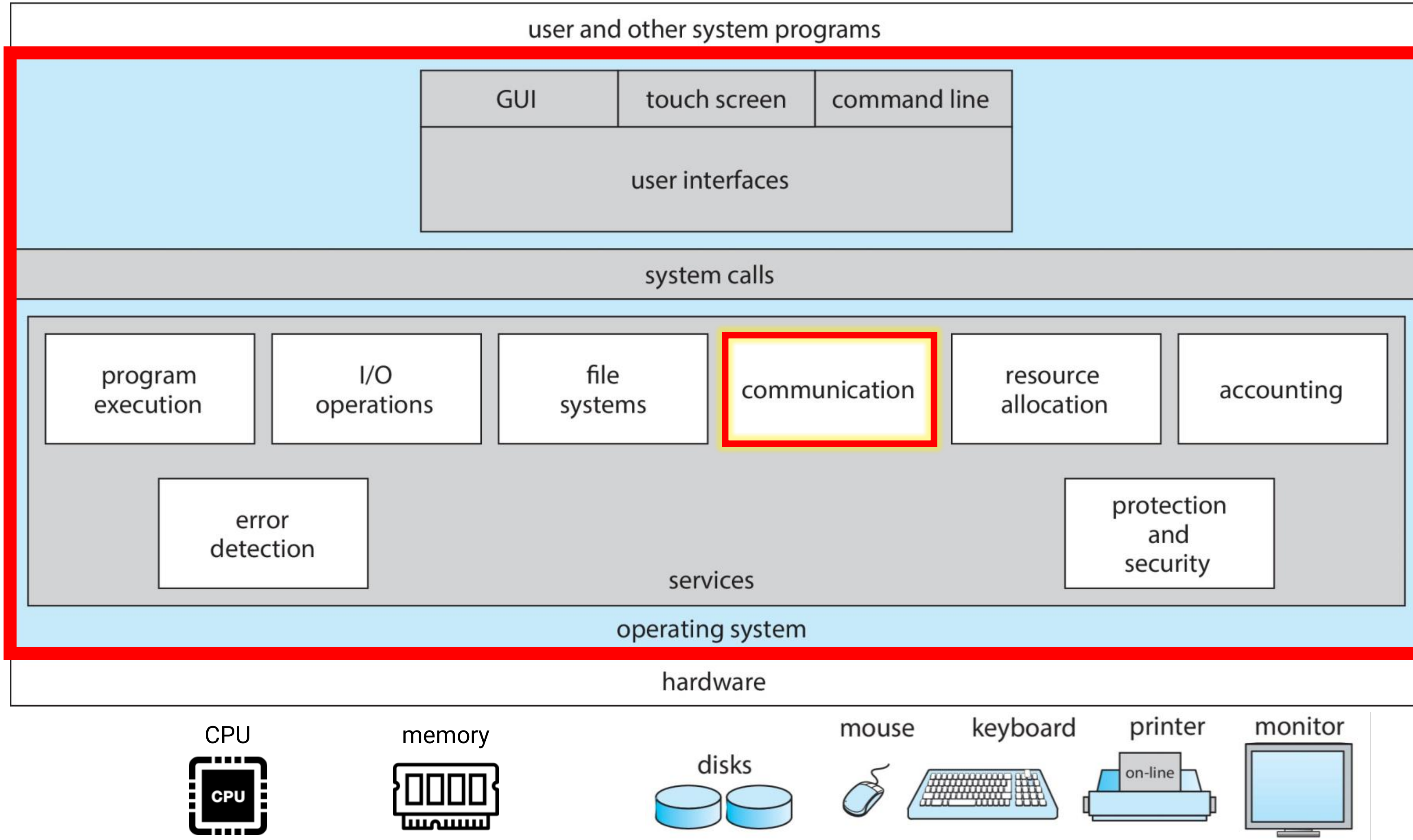
Early Deadline First (EDF) Scheduler

- **Dynamic-priority scheduler**
  - Task's priority is not fixed
  - Task's priority is determined by deadline
- Example (preempted)
  - T1 = (2, 0.9), T2 = (5, 2.3) for (period, execution)



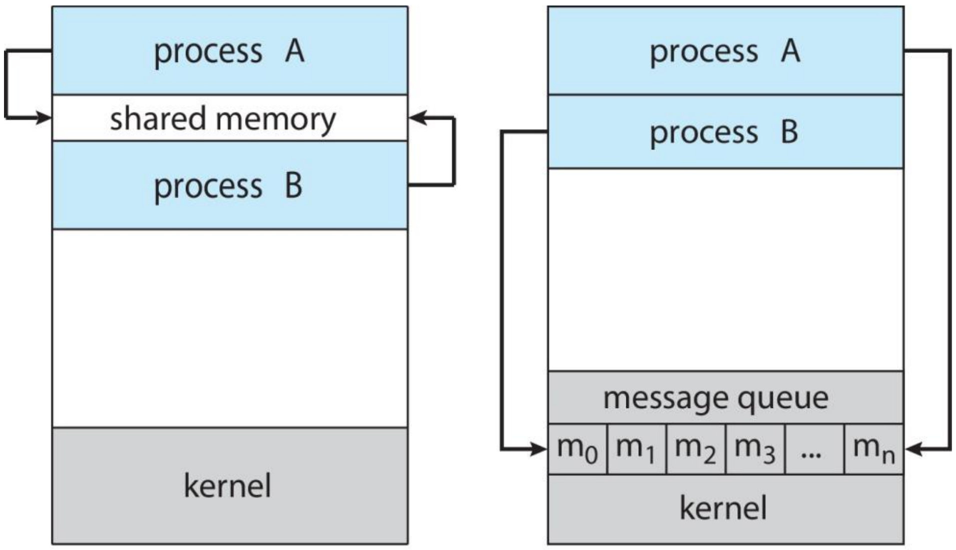
Operating System Service: Inter-Process Communication

Operating System Services



Communication Methods

- **Shared memory**
  - Require more careful **user synchronization**
  - Implemented by memory access (faster)
  - Use memory address to access data
- **Message passing**
  - No conflict: more efficient for small data
  - Use send/recv message
  - Implement by system call (slower)



shared memory message passing

Message Passing

Message Passing System

- Mechanism for processes to **communicate** and **synchronize** their actions
- IPC provides two operations
  - **Send** (message)
  - **Receive** (message)

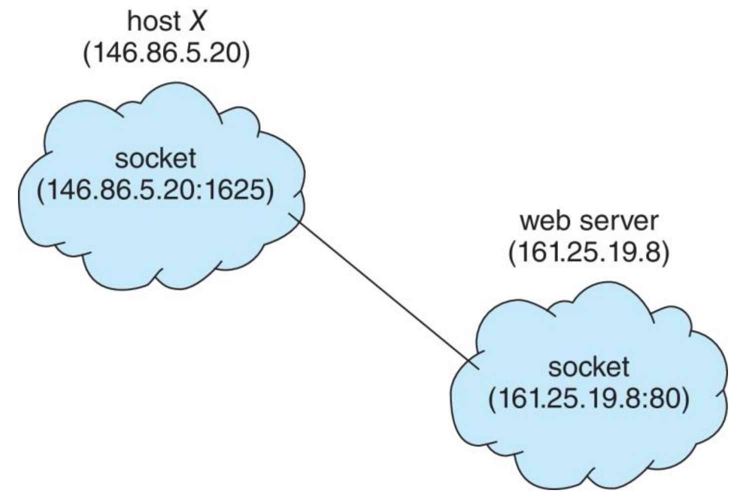
- To communicate, processes need to
  - Establish a **communication link**
  - Exchange a message via send/receive

Message Passing System (cont.)

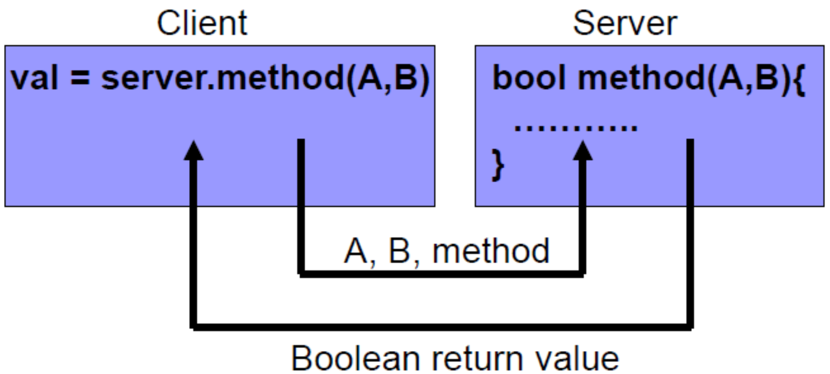
- Implementation of communication link
  - **Physical**
    - HW bus
    - Network
  - **Logical (properties of the link)**
    - **Direct or indirect communication**
    - Symmetric or asymmetric communication
    - **Blocking or non-blocking**
    - Automatic or explicit buffering
    - Send by copy or send by reference
    - Fixed-sized or variable-sized messages

Message Passing Methods

- **Sockets**
  - A network connection identified by **IP** and **port**
  - Exchange **unstructured stream of bytes**



- **Remote Procedure Calls**
  - Cause a procedure to execute in another address space
  - **Parameters** and **return values** are passed by messages



Blocking and Non-Blocking

- Messages passing may be either **blocking** or **non-blocking**
- **Blocking (synchronous)**
  - **Blocking send**: sender is blocked until the message is received by receiver or by the mailbox
  - **Blocking receive**: receiver is blocking until the message is available
- **Non-blocking (asynchronous)**
  - **Non-blocking send**: sender sends the message and resumes operation
  - **Non-blocking receive**: receiver receives a valid message or a null

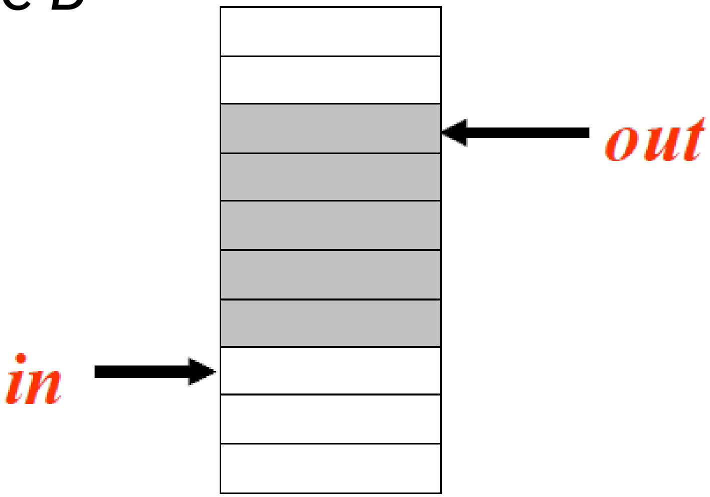
Shared Memory

Shared Memory

- Processes are responsible for
  - **Establishing a region of shared memory** (ask OS for help)
    - Typically, the created shared-memory regions resides in the address of the process creating the shared memory segment
    - Participating processes must agree to remove memory access constraint from OS
  - **Determining the form of the data and the location**
  - **Synchronization**: ensuring data are not written simultaneously by processes

Consumer and Producer

- **Producer** process produces information that is consumed by a **Consumer** process
- Buffer as a circular array with size *B*
  - Next free: *in*
  - First available: *out*
  - Empty: *in = out*
  - Full: *(in + 1) % B = out*



- The solution allows at most (B - 1) item in the buffer
  - Otherwise, cannot tell the buffer is empty or full

That's All !