INDIAN INSTITUTE OF INFORMATION TECHNOLOGY

Operating Systems and Computer Networks

Dr. Animesh Chaturvedi

Assistant Professor: **IIIT Dharwad**

Young Researcher: **Heidelberg Laureate Forum**

and **Pingala Interaction in Computing**

Young Scientist: **Lindau Nobel Laureate Meetings**

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

PhD: **IIT Indore** MTech: **IIITDM Jabalpur**

LINDAU NOBEL LAUREATE MEETINGS

HEIDELBERG LAUREATE FORUM

KING'S College LONDON

The Alan Turing Institute

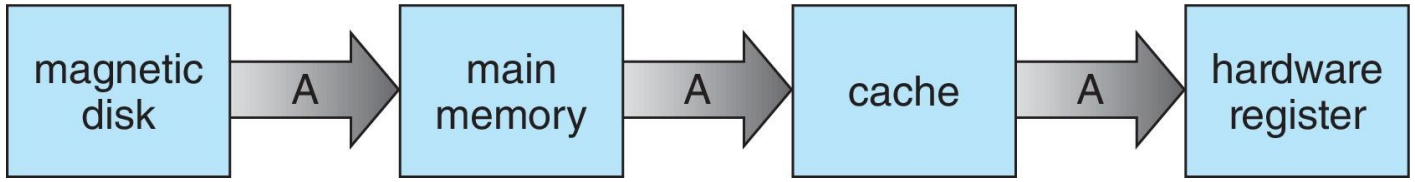
Indian Institute of Technology Indore
भारतीय प्रौद्योगिकी संस्थान इंदौर

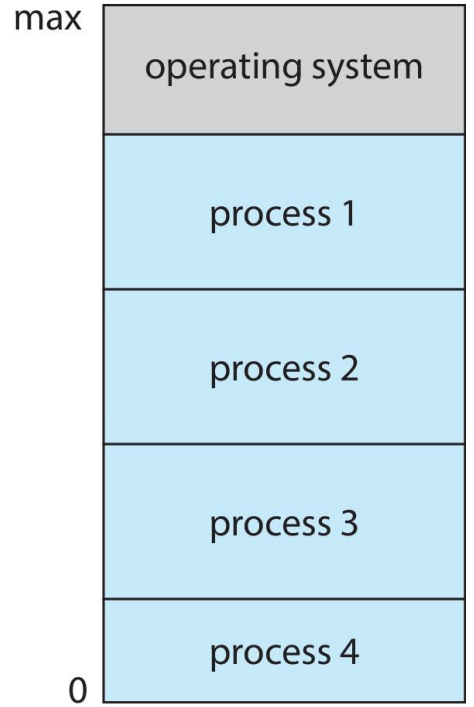
PDPM
Indian Institute of Information Technology, Design and Manufacturing, Jabalpur

Operating Systems

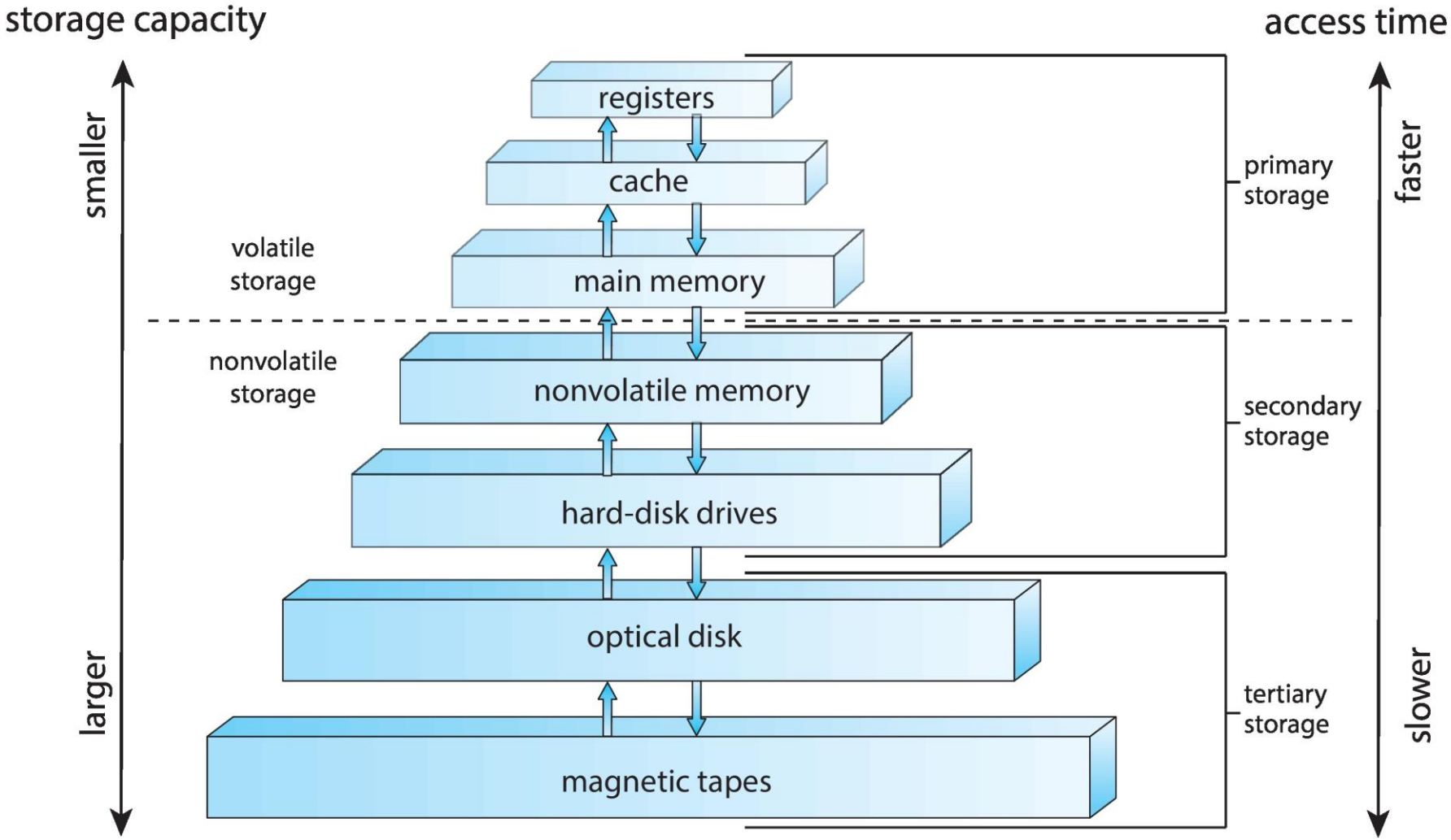
OS manages memory

- 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





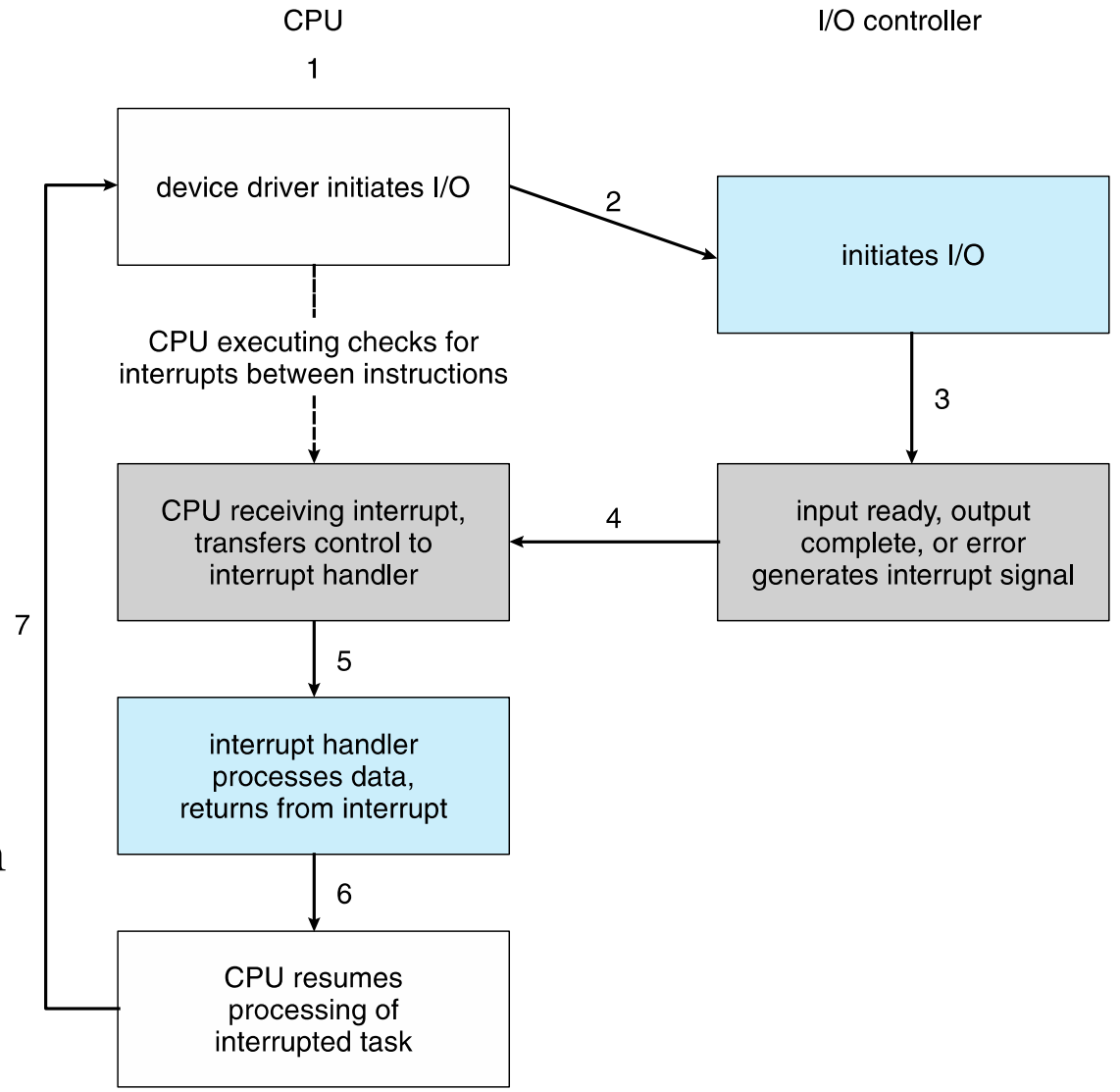
OS manages memory



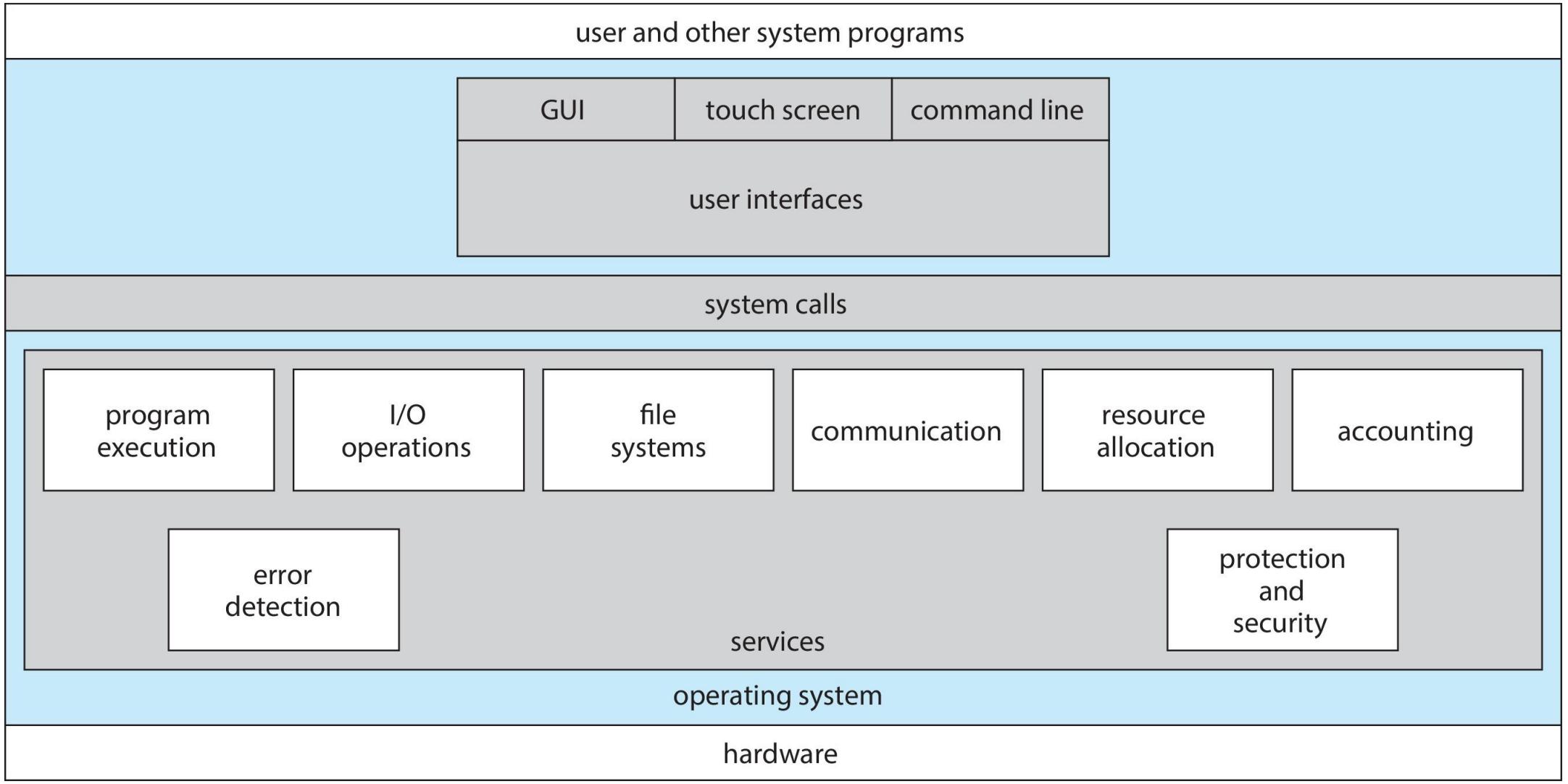
Storage-Device Hierarchy

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

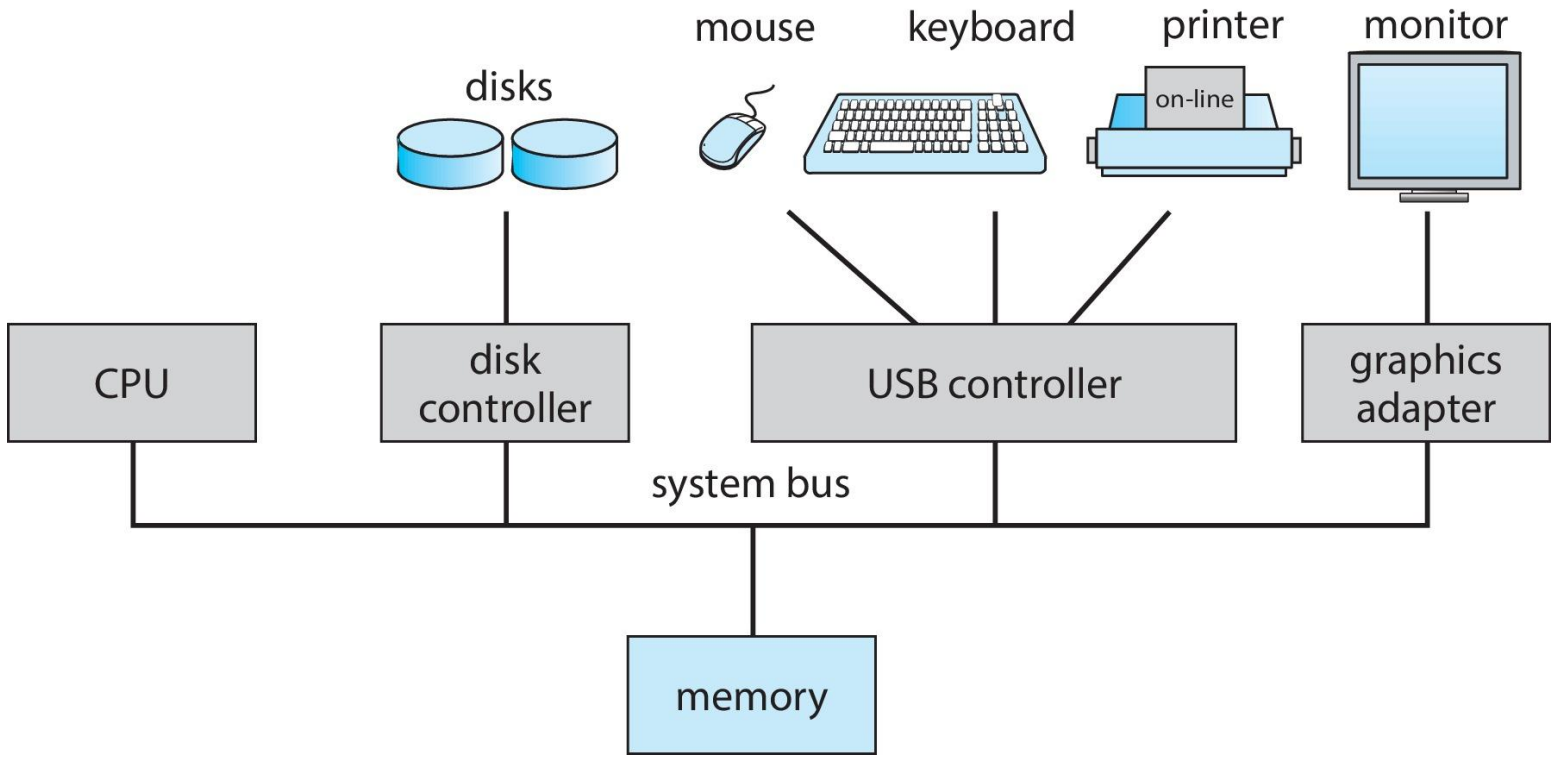


Operating System Services



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

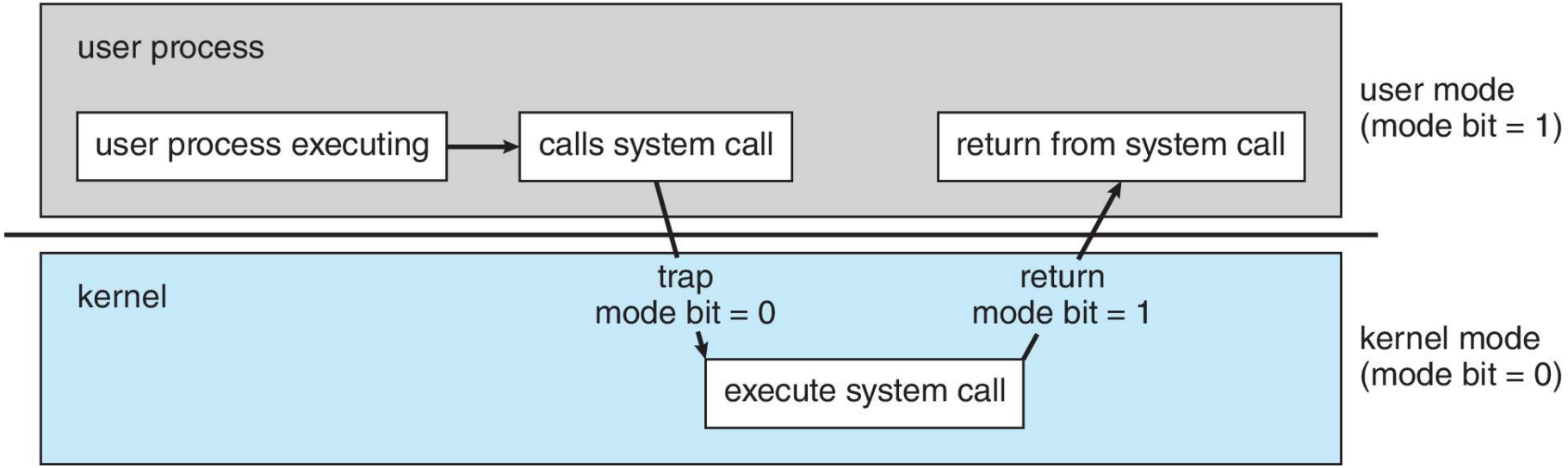


System, Application, Middleware, Bootstrap

- 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

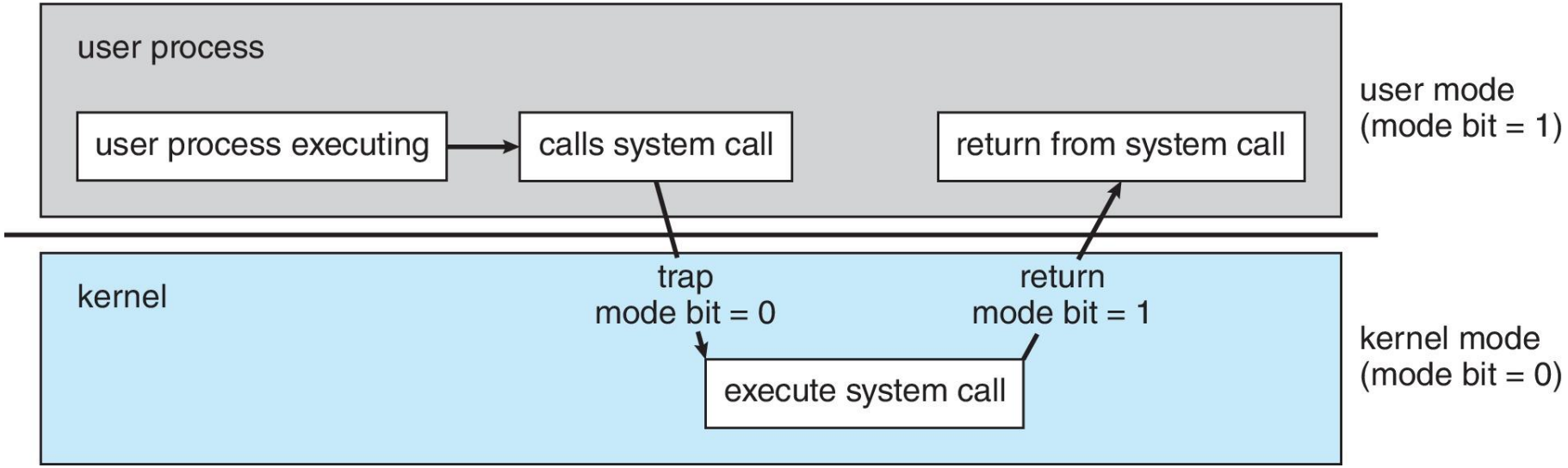
Application Programming Interface (API)

- 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)

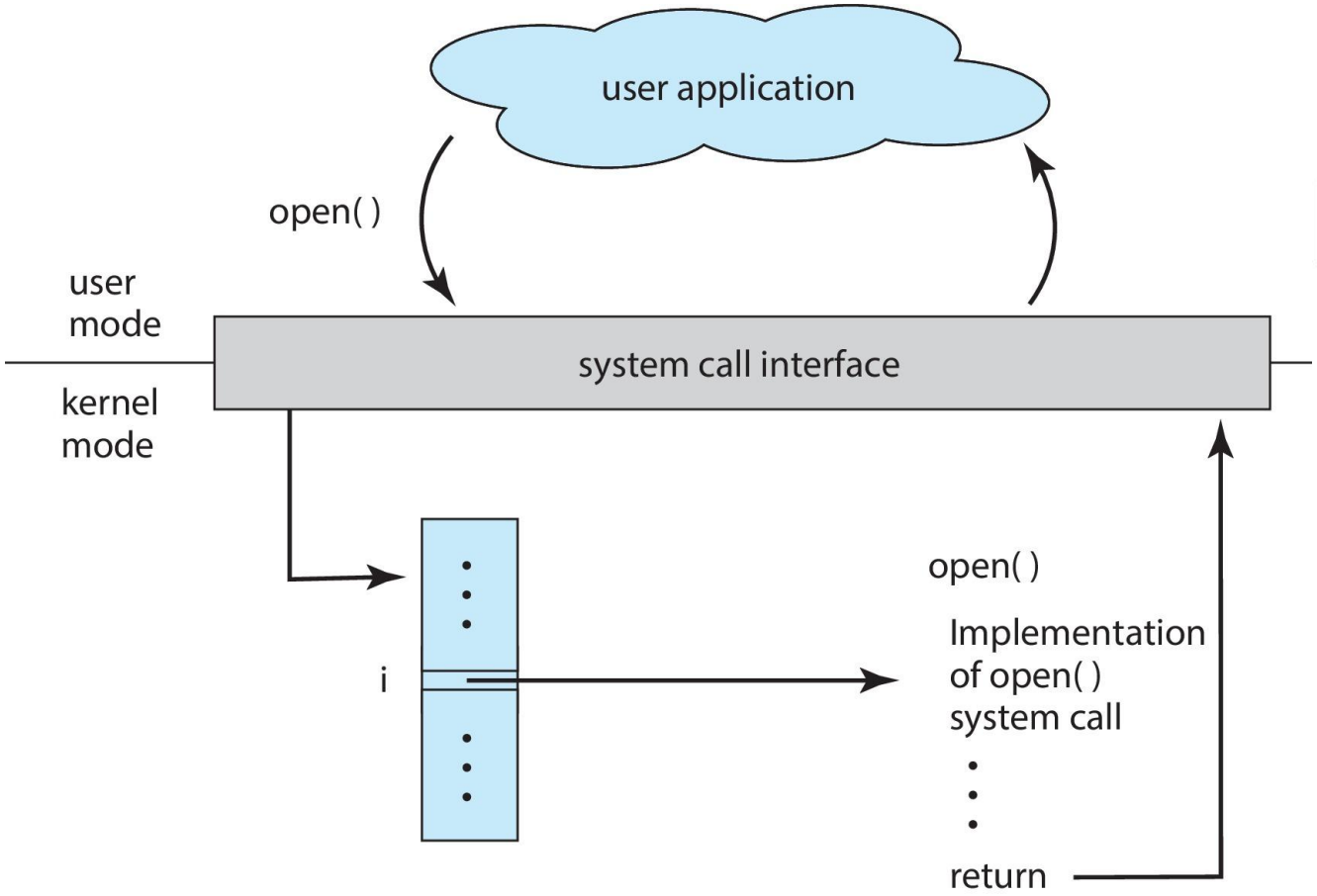
- CPU hardware has multiple privilege levels:
  - One to run user code: user mode
  - One to run OS code like system calls: kernel mode
  - Some instructions execute only in kernel mode
- Kernel does not trust user stack and user provided addresses
  - Kernel creates a separate kernel stack and Interrupt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

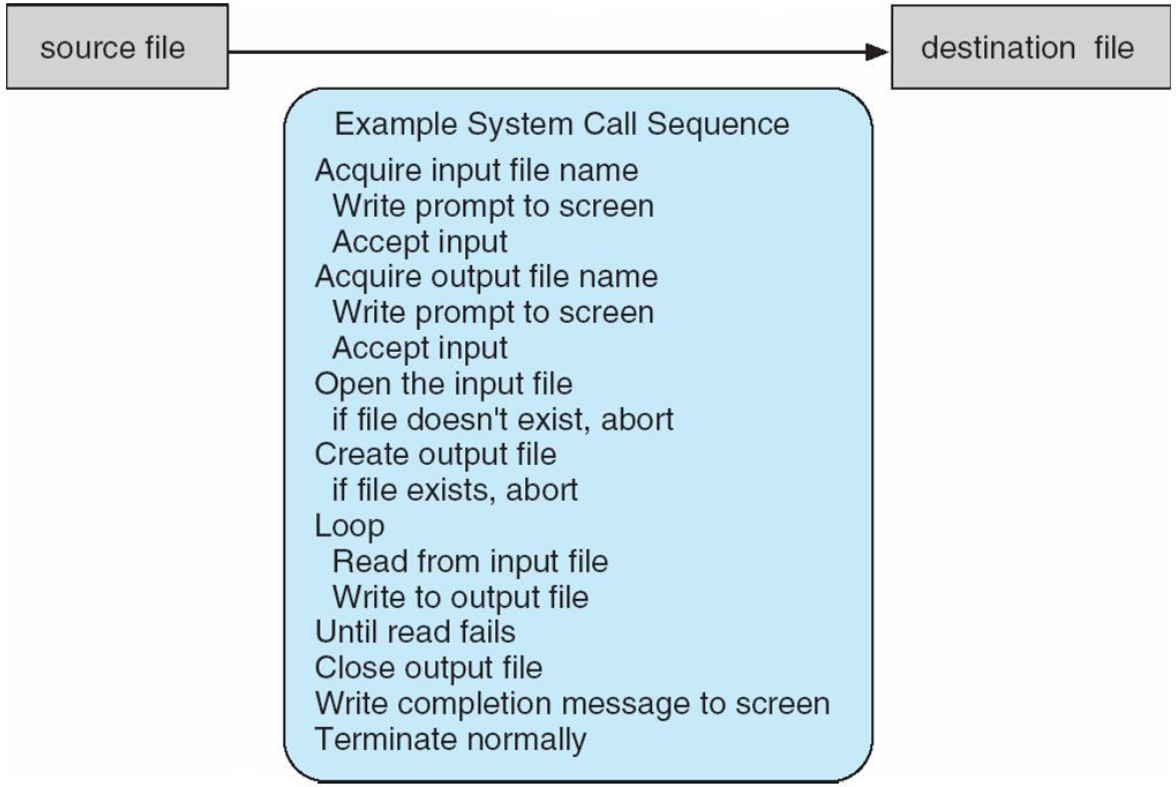


Application Programming Interface (API)

- 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

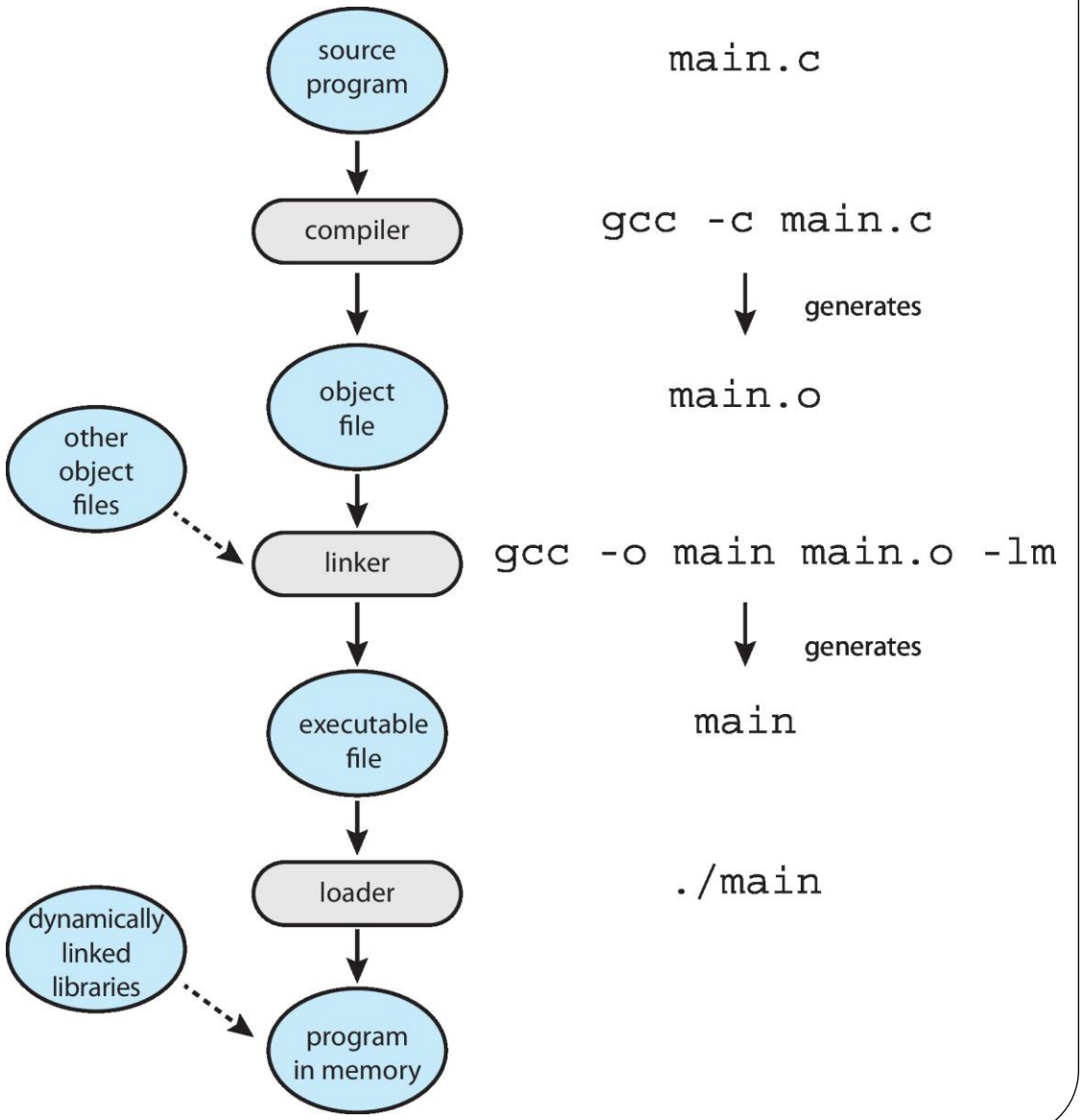
System Calls to copy contents between files





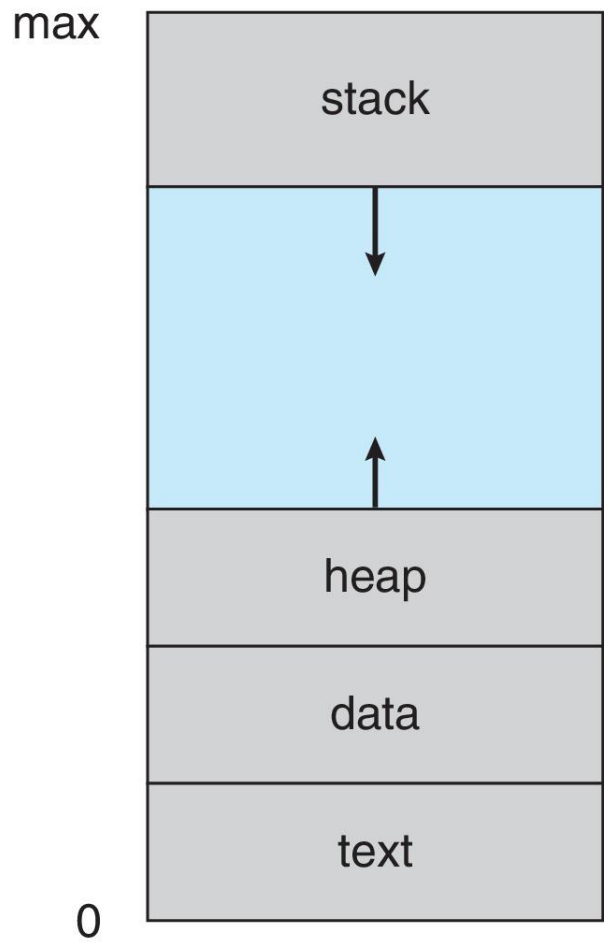
Program and Process

- 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



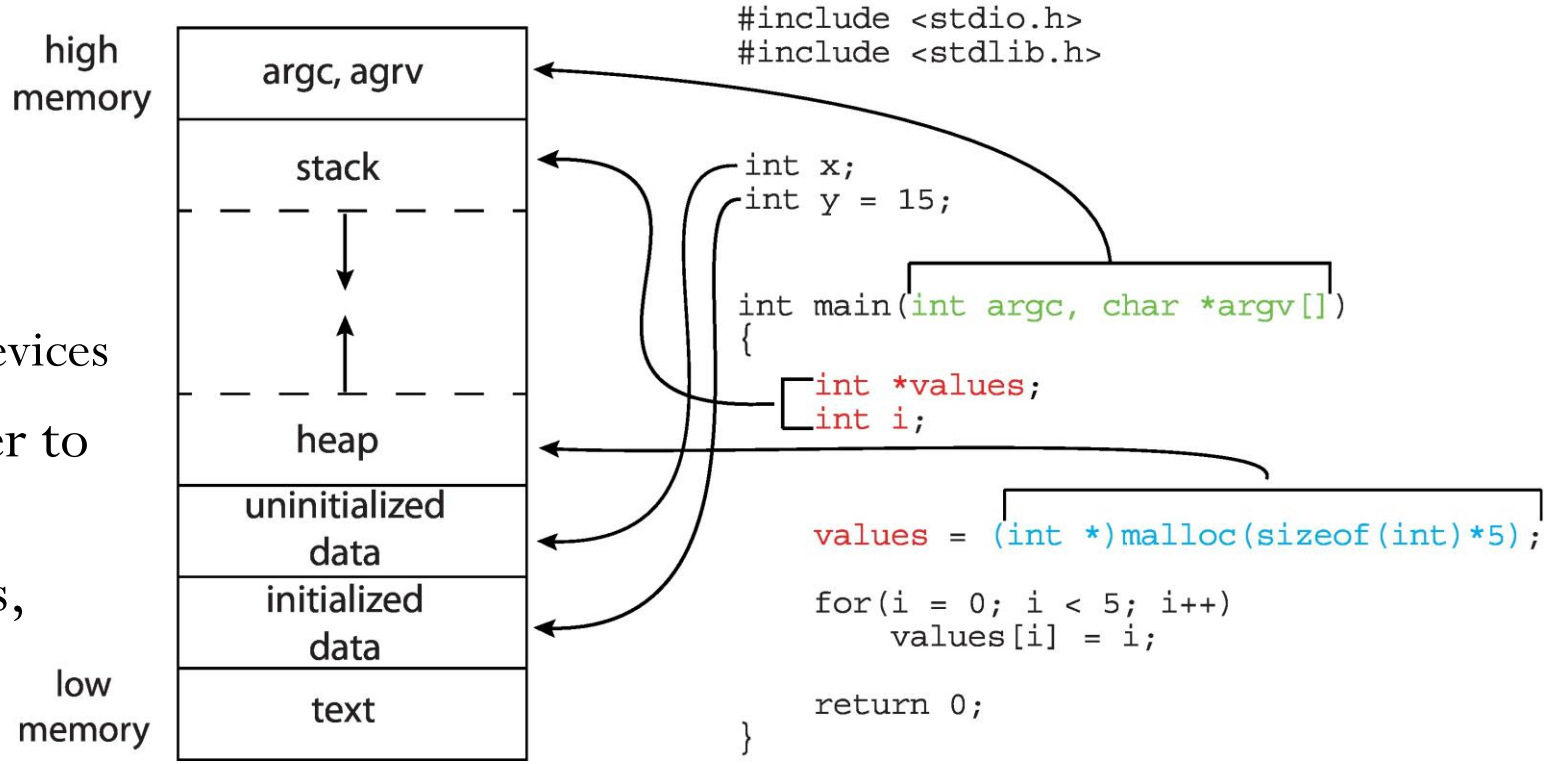
Program and Process

- 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)



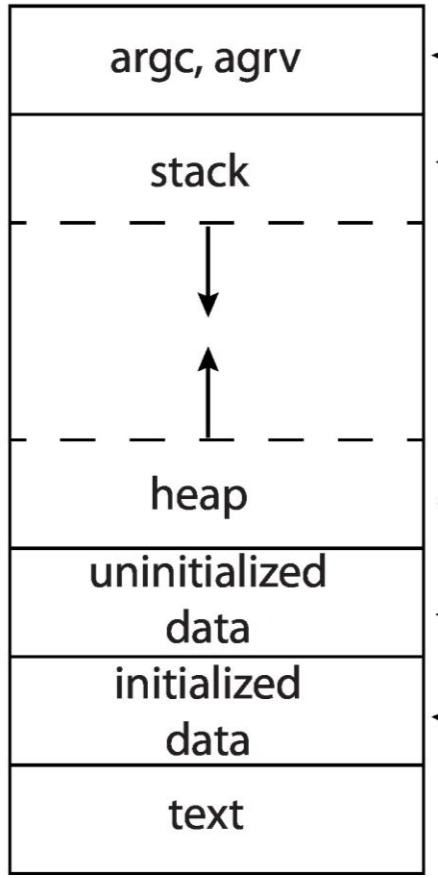
Program and Process

- 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.



Motivation: Program Code to Memory

- **Abstraction** of complex usage Program as a Memory (RAM or Cache).
- Conversion of **High level language to Low level language**
- Static/global variables are allocated in the executable
  - Local variables of a function on Stack
  - Dynamic allocation with malloc on the heap



Process as a Memory

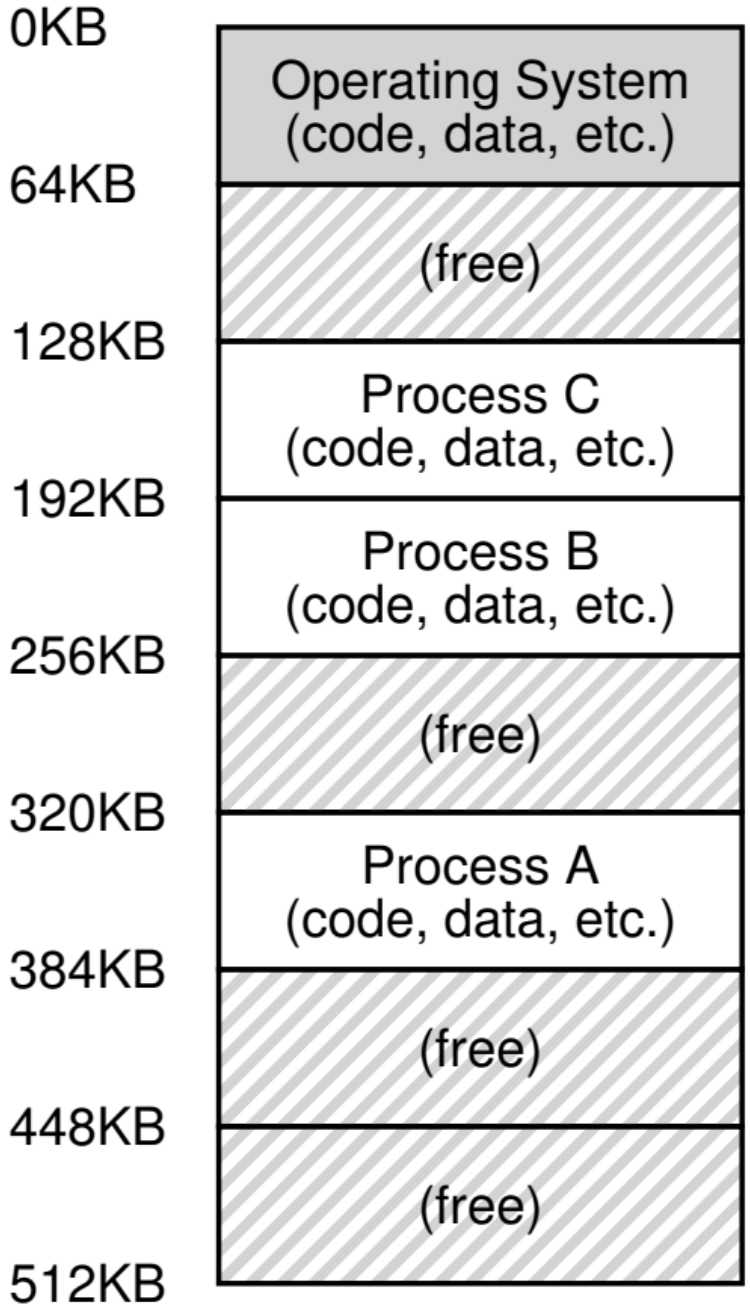
```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Program as a Code

Memory management and File systems

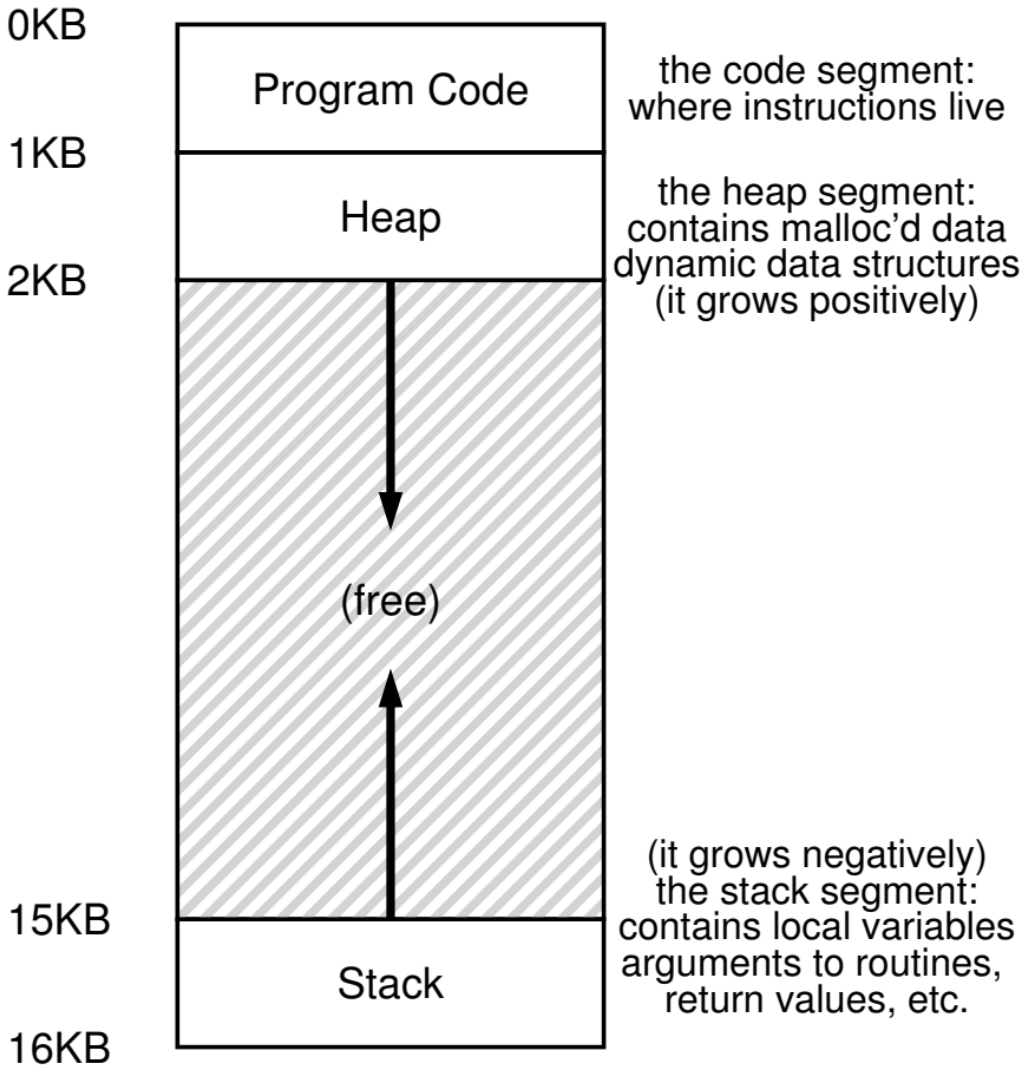
Why virtualize memory?

- Because real view of memory is messy!
- Earlier, memory had only code of one running process (and OS code)
- Now, multiple active processes timeshare CPU
  - Memory of many processes must be in memory
  - Non-contiguous too
- Need to hide this complexity from user
- Image of Three Processes in Sharing Memory



Abstraction: (Virtual) Address Space

- **Virtual address space**: every process assumes it has access to a large space of memory from address 0 to a MAX
- Contains program code (and static data), **heap (dynamic allocations)**, and **stack (used during function calls)**
- **Stack** and **Heap** grow during runtime
- CPU issues loads and stores to virtual addresses



Goals of memory virtualization

- **Transparency**: user programs should not be aware of the messy details
- **Efficiency**: minimize overhead and wastage in terms of memory space and access time
- **Isolation and protection**: a user process should not be able to access anything outside its address space
- Address translation from virtual addresses (VA) to physical addresses (PA)
  - CPU issues loads/stores to VA but memory hardware accesses PA
- OS allocates memory and tracks location of processes

Address Translation

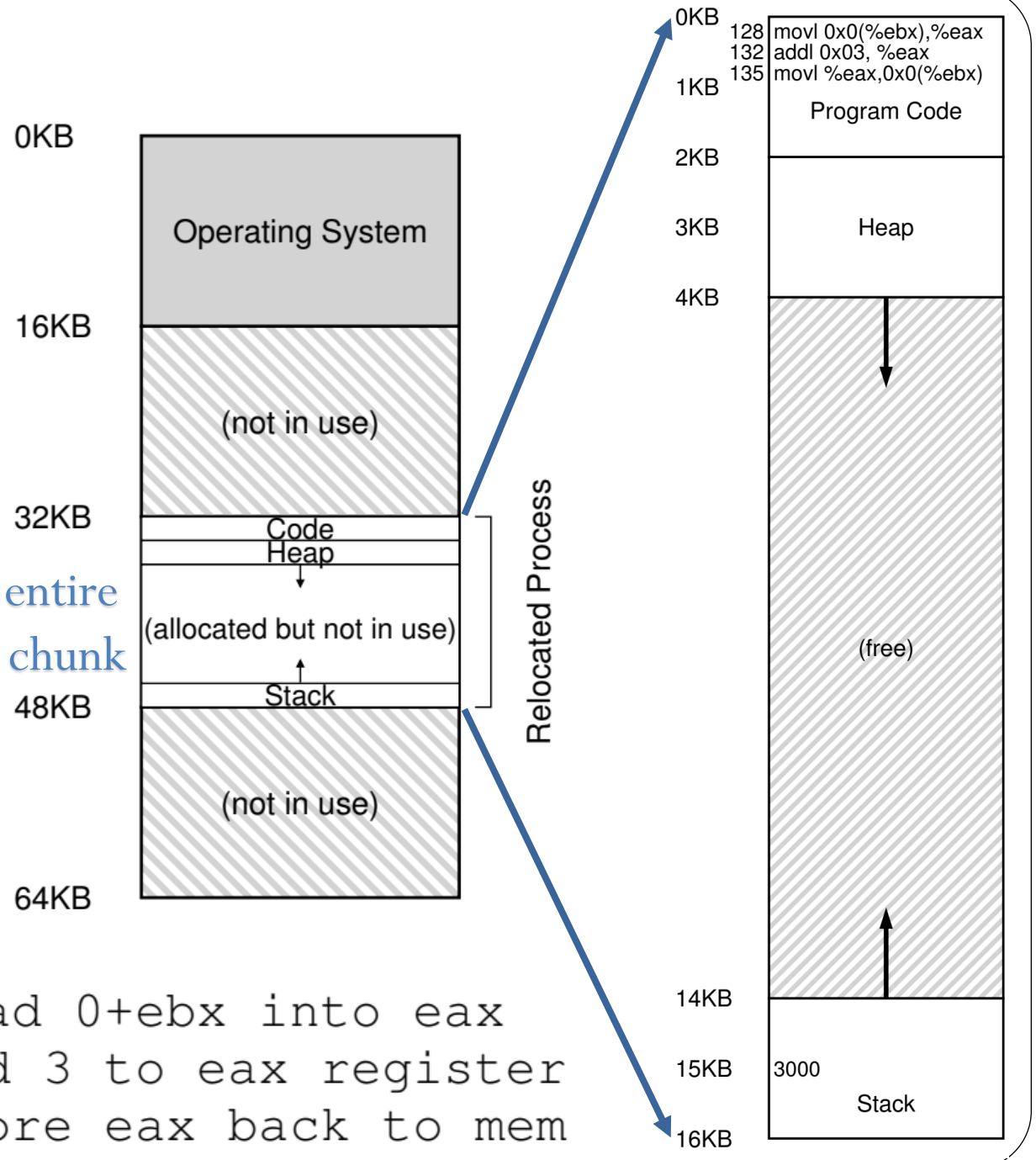
- Virtual address space is setup by OS during process creation
- Translation from **VA to PA**
  - 128 to 32896 (32KB + 128)
  - 1KB to 33 KB
  - 20KB? Error!

Simplified OS: places entire memory image in one chunk

```
void func() {
    int x = 3000;
    x = x + 3;
    ...
```

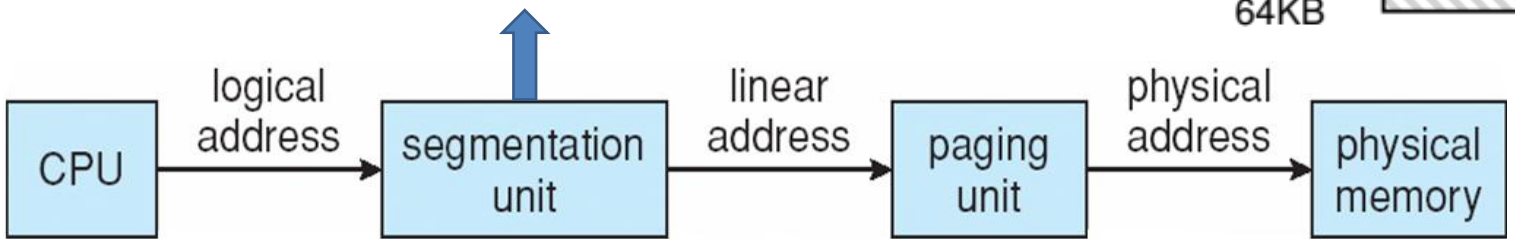
Compiler

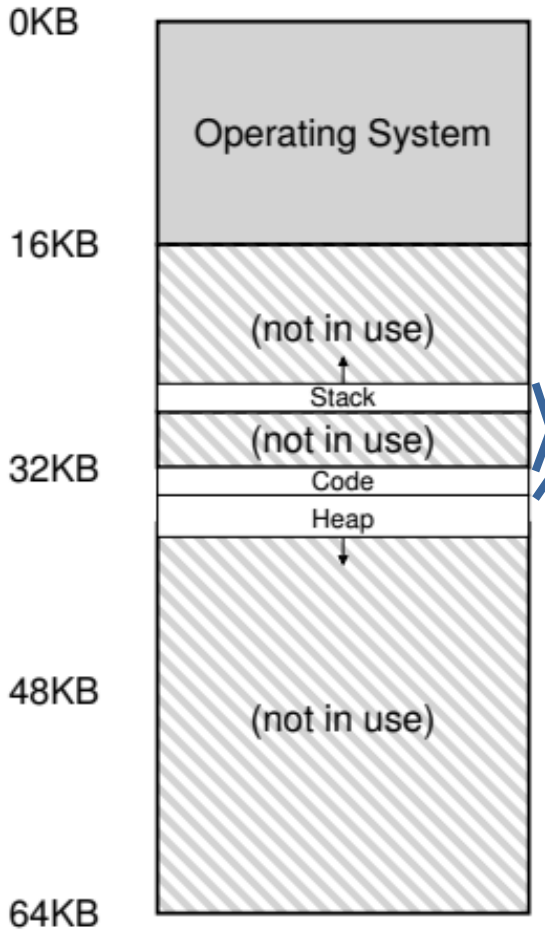
```
128: movl 0x0(%ebx), %eax     ;load 0+ebx into eax
132: addl $0x03, %eax         ;add 3 to eax register
135: movl %eax, 0x0(%ebx)     ;store eax back to mem
```

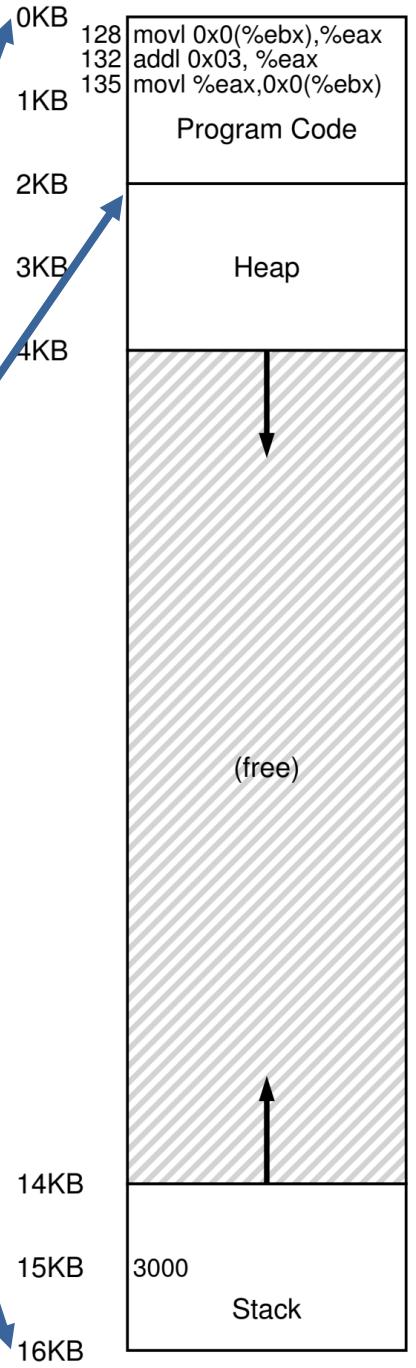


Segmentation

- Generalized base and bounds
- Each segment of memory image placed separately
- Multiple (base, bound) values stored in MMU
- Good for sparse address spaces
- But variable sized allocation leads to external fragmentation – Small holes in memory left between segments



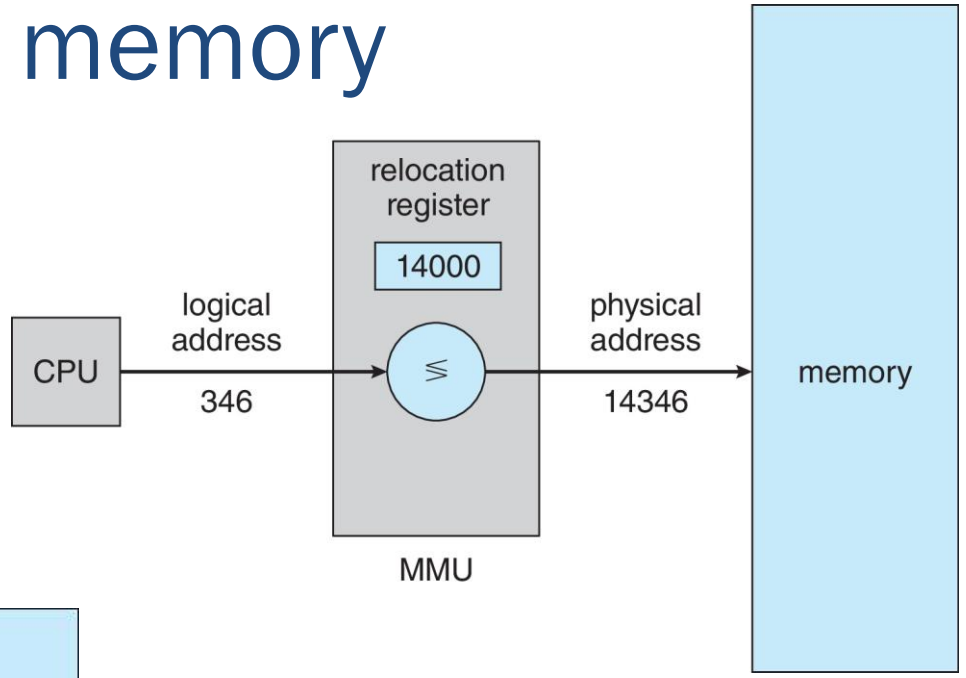


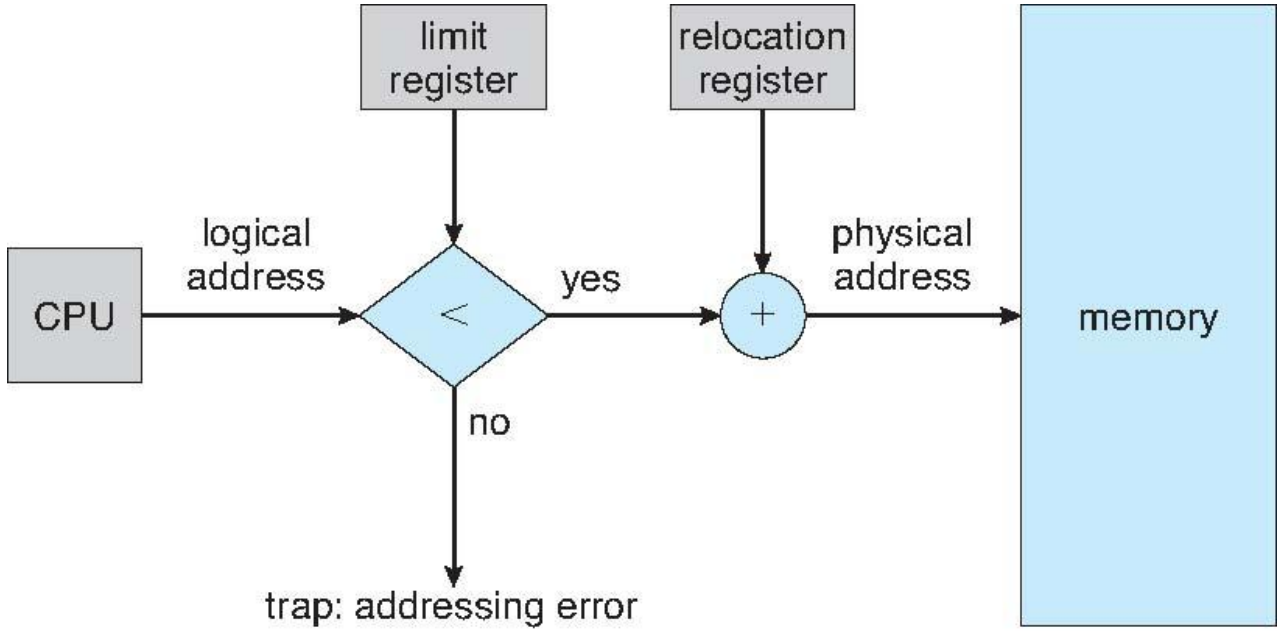


Physical memory and Virtual memory

- OS provides the **base** (starting address) and **bound** (total size of process) values to
  - **Memory Management Unit (MMU)** (a hardware)
- Memory hardware MMU calculates PA from VA

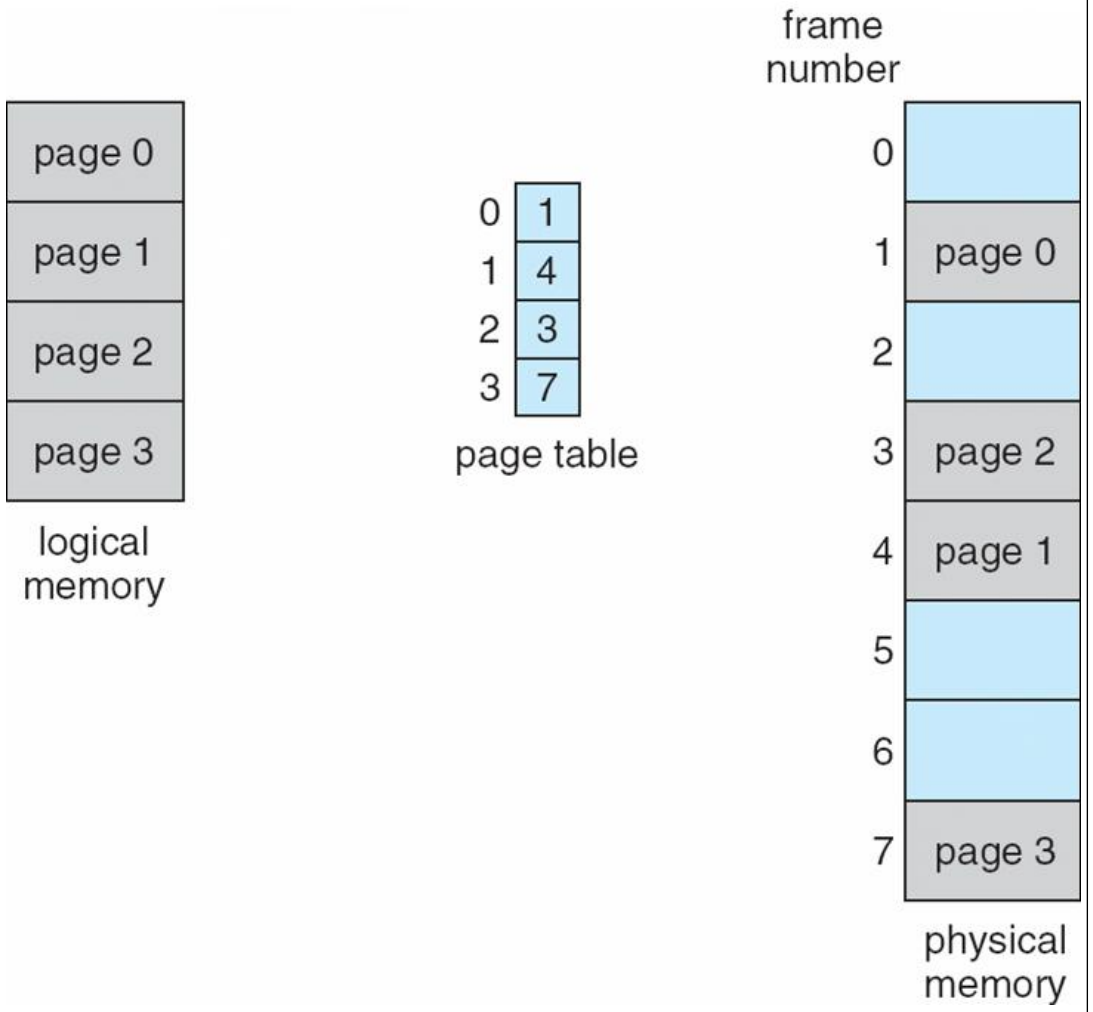
  Physical Address (PA) = Virtual Address (VA) + Base





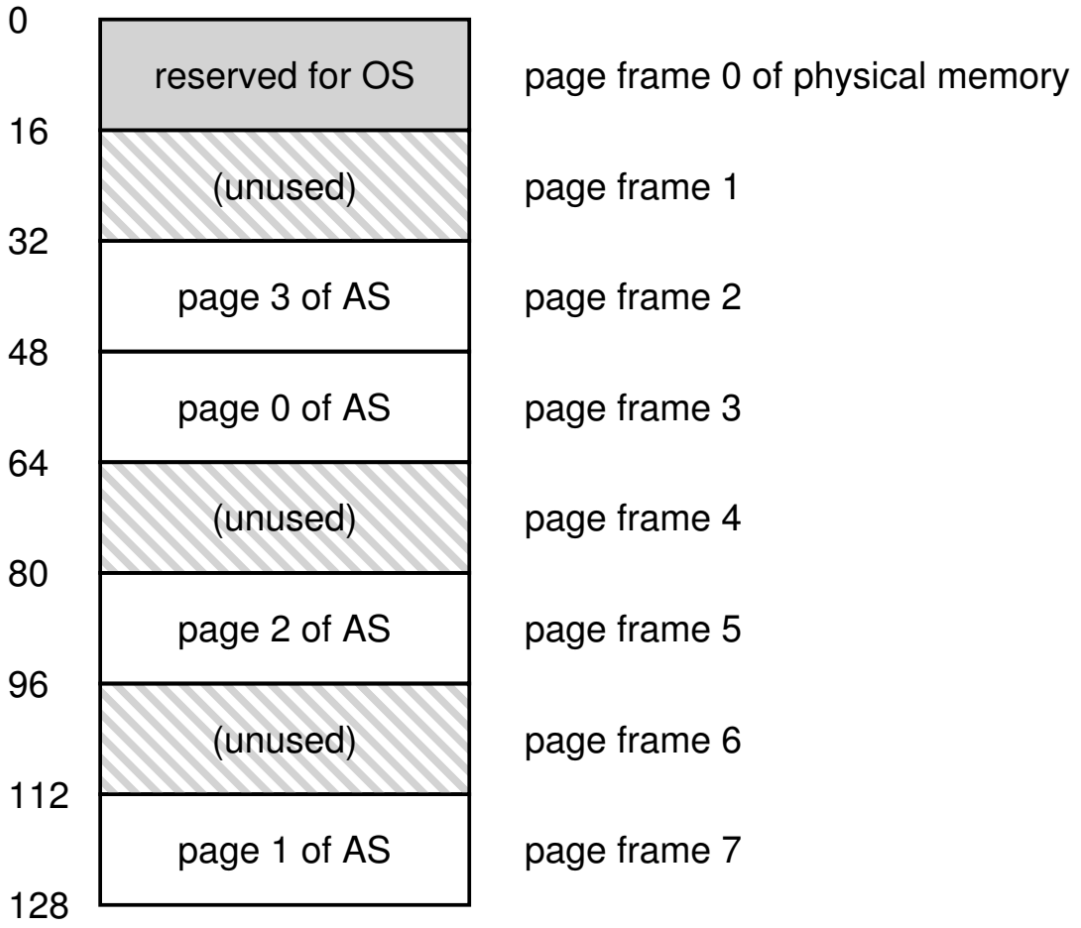
Paging, Logical memory to Frame number

- OS divides virtual address space into **fixed size pages**, physical memory into **frames**
- To allocate memory, a page is mapped to a free physical frame
- **Page table** stores mappings from virtual page number to physical frame number for a process (e.g., page 0 to frame 3)
- MMU has access to page tables, and uses it to translate VA to PA



Page Table

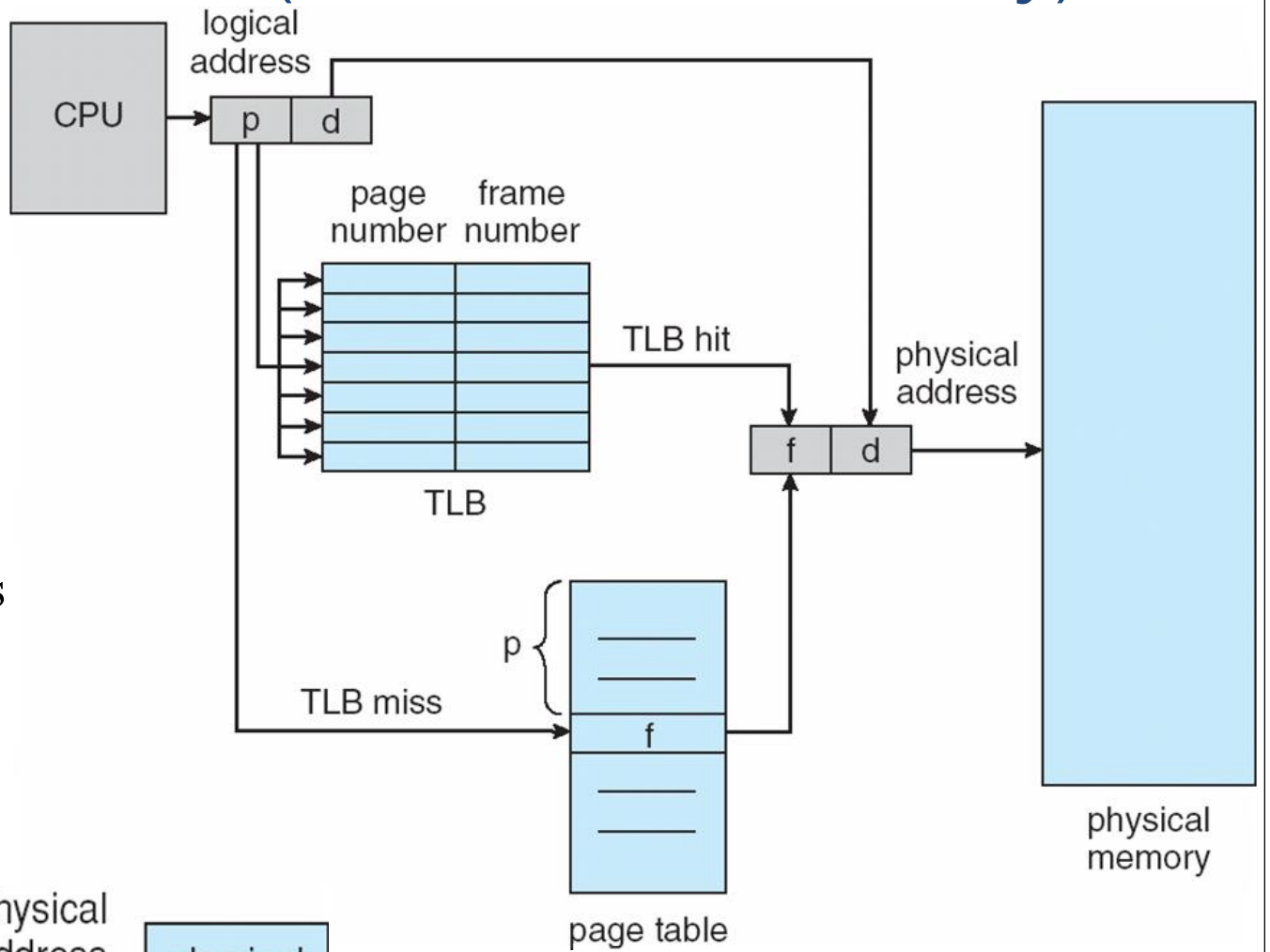
- Per process data structure to help VA-PA translation.
- Array stores mappings
  - from **Virtual Page Number (VPN)**
    to **Physical Frame Number (PFN)**
  - E.g., VPN 0 → PFN 3, VPN 1 → PFN 7
- Part of OS memory (in PCB)
- MMU has access to page table and uses it for address translation
- OS updates page table upon context switch

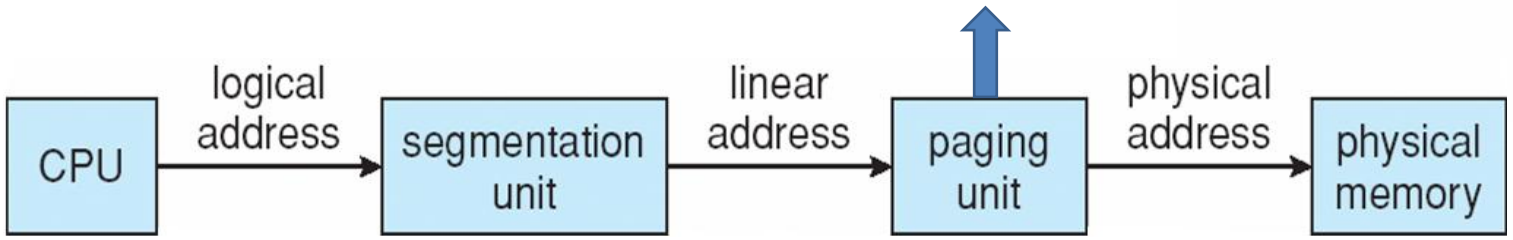


a 64-Byte address space in a 128-Byte physical memory

Translation Look-Aside Buffer (Associative memory)

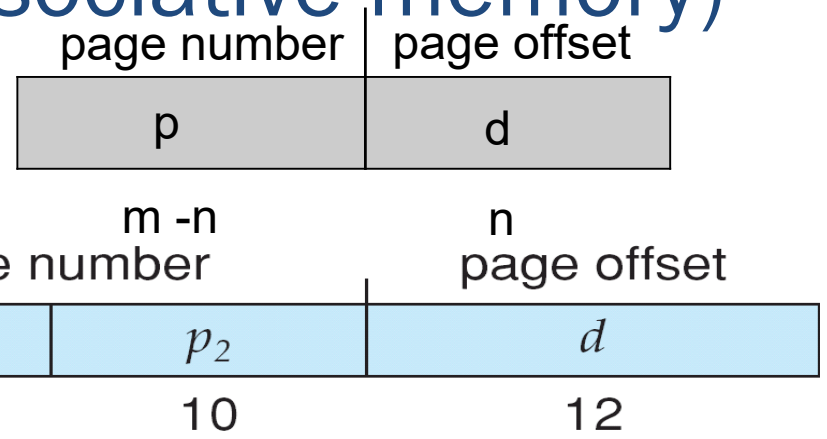
- A cache of recent VA-PA mappings
- To translate VA to PA, MMU first looks up TLB
- If **TLB hit**, obtaining PA, fetches memory location and returns to CPU (via CPU caches)
- If **TLB miss**, then MMU performs additional memory accesses to "walk" page table



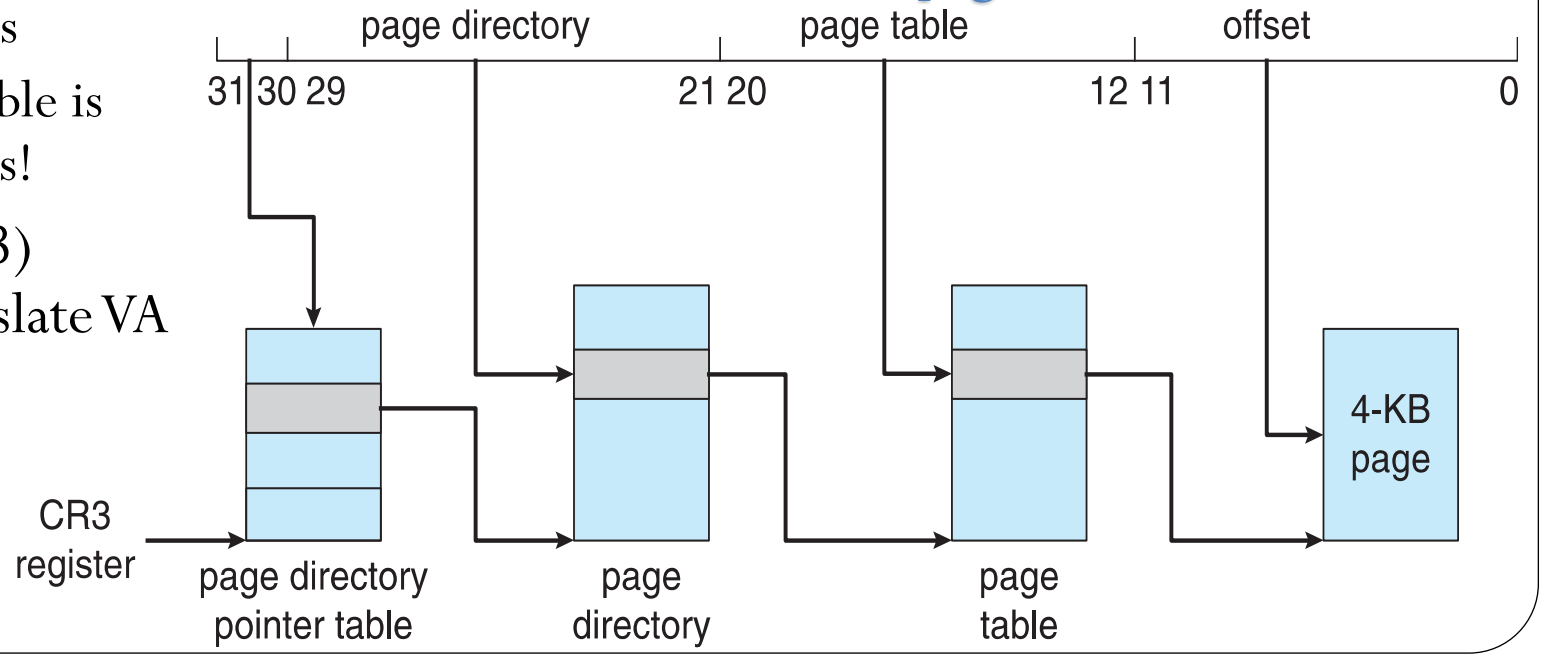


Translation Look-Aside Buffer (Associative memory)

- 32 bit VA, 4 KB pages,
  - 2^32 / 2^12 = 2^20 entries
  - If each PTE is 4 bytes, then page table is **4MB**
  - One such page table per process!
- To reduce the size of page tables
  - Larger pages, so fewer entries
  - For such large tables, Page table is itself split into smaller chunks!
- **Control register** (e. g. CR3) enables the processor to translate VA into PA by locating the page directory and page tables



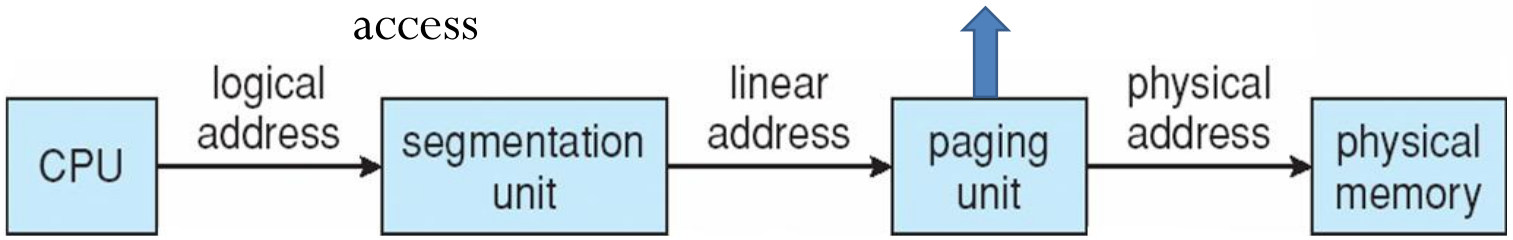
Multilevel page tables

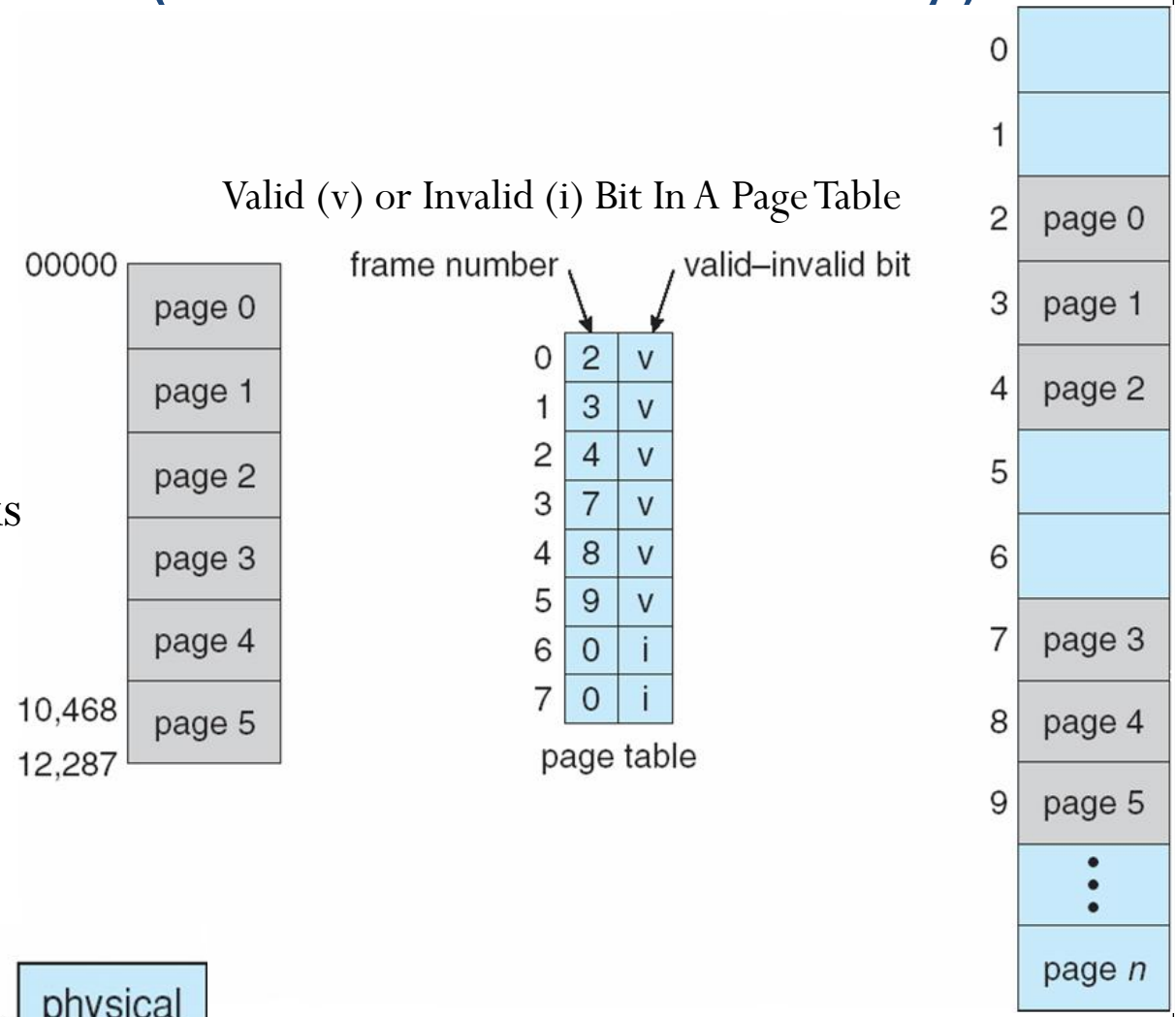


Intel Architecture IA-32 bit

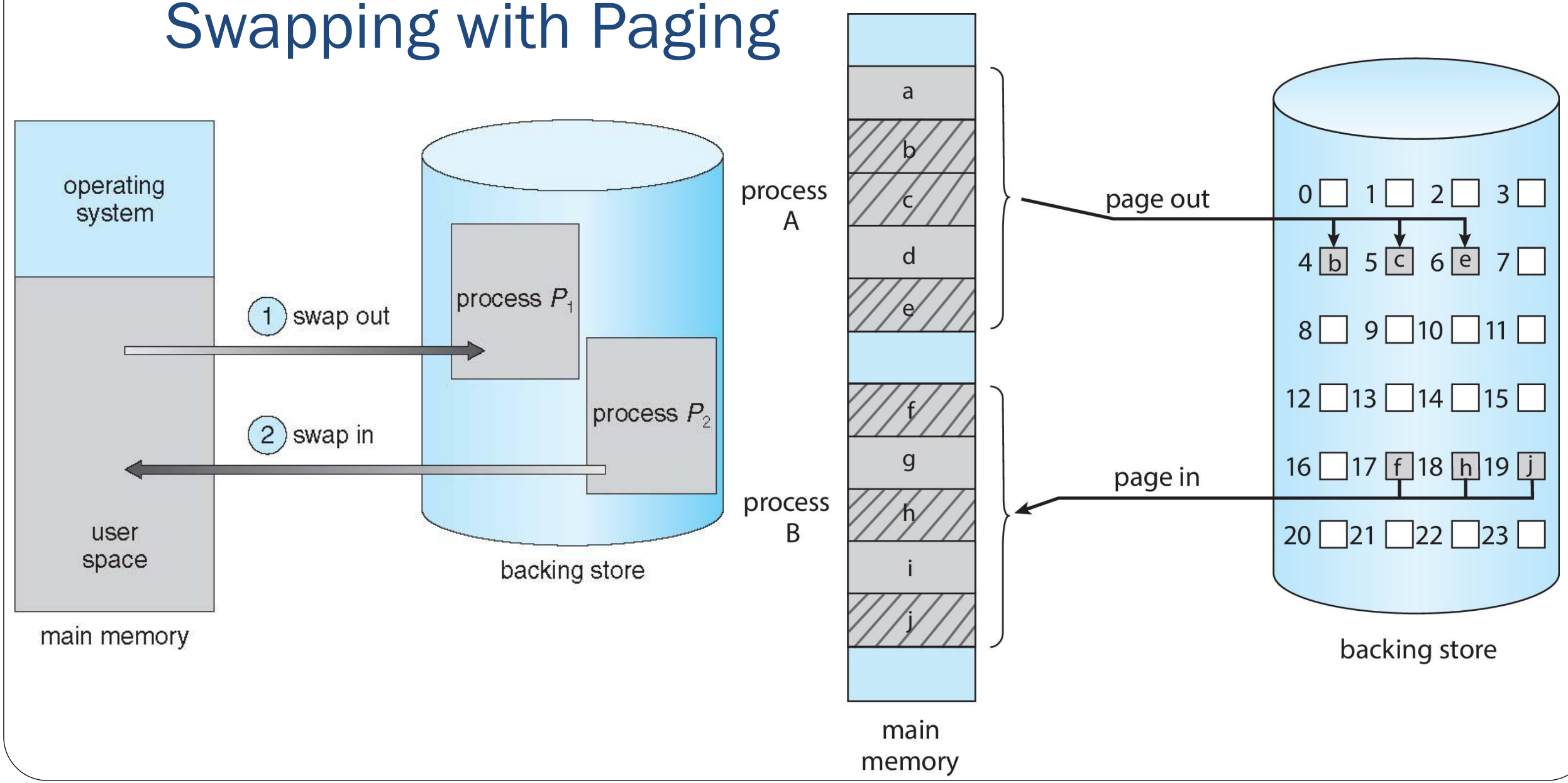
Translation Look-Aside Buffer (Associative memory)

- **TLB misses are expensive** (multiple memory accesses)
- **Locality of reference** helps to have high **hit rate**
- If **TLB hit**: PA can be directly used
- If **TLB miss**: MMU accesses memory, walks page table, and obtains page table entry
  - If present bit set in PTE, accesses memory
  - If not present but valid, raises **page fault**.
  - If invalid page access, trap to OS for illegal access





Swapping with Paging



CPU and Process Scheduling

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

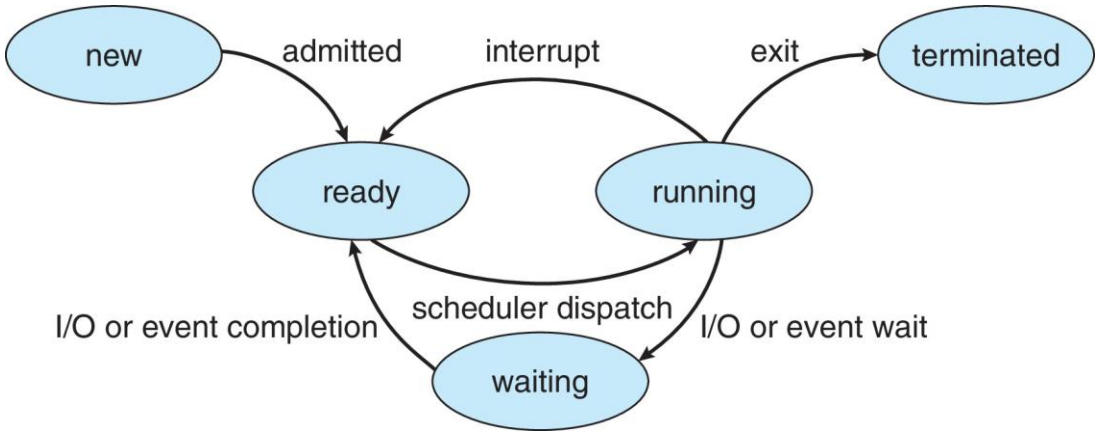
Scheduling policy

- 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)

Process states

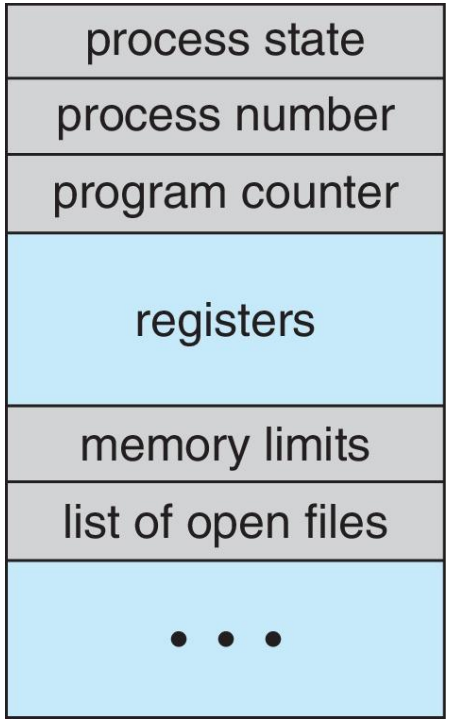
As a process executes, it changes **state**

- **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



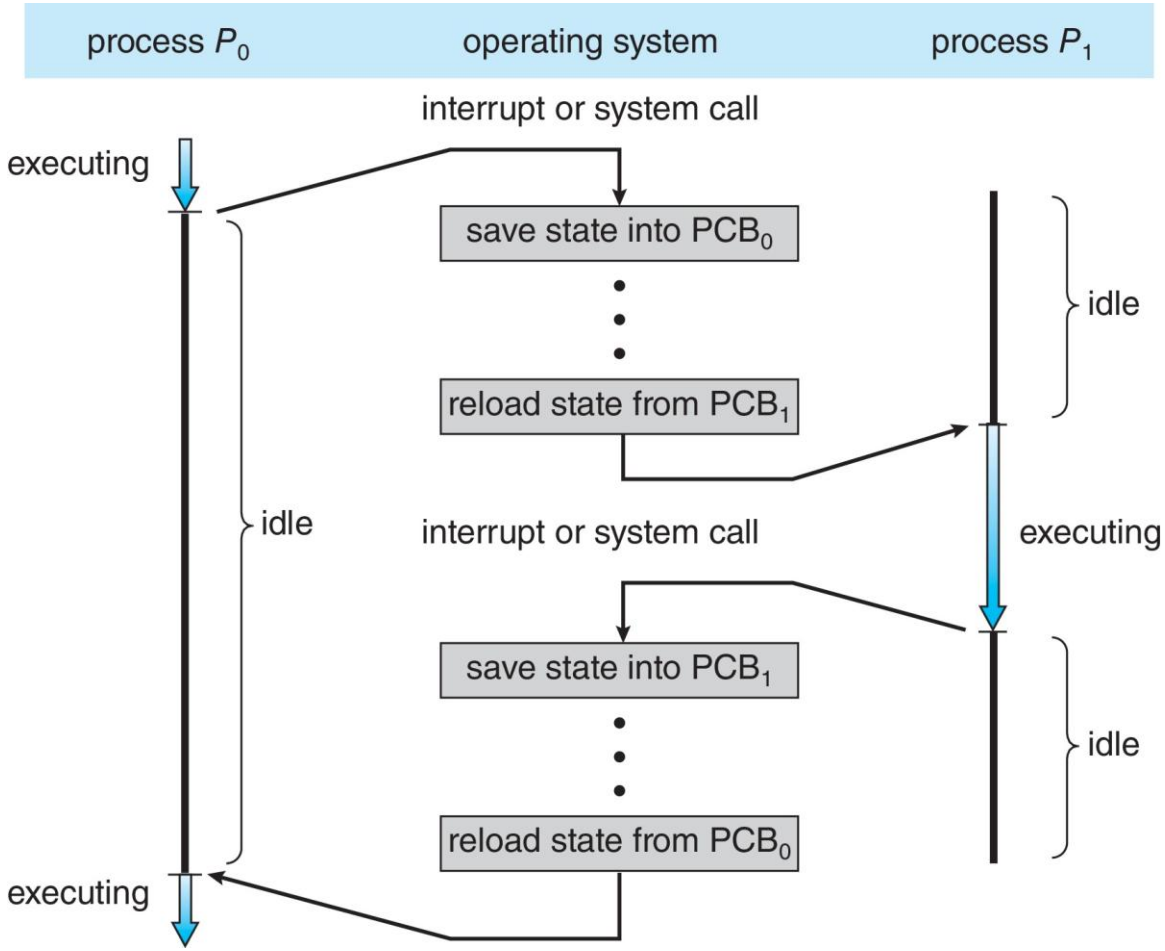
Process Control Block (PCB)

- 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



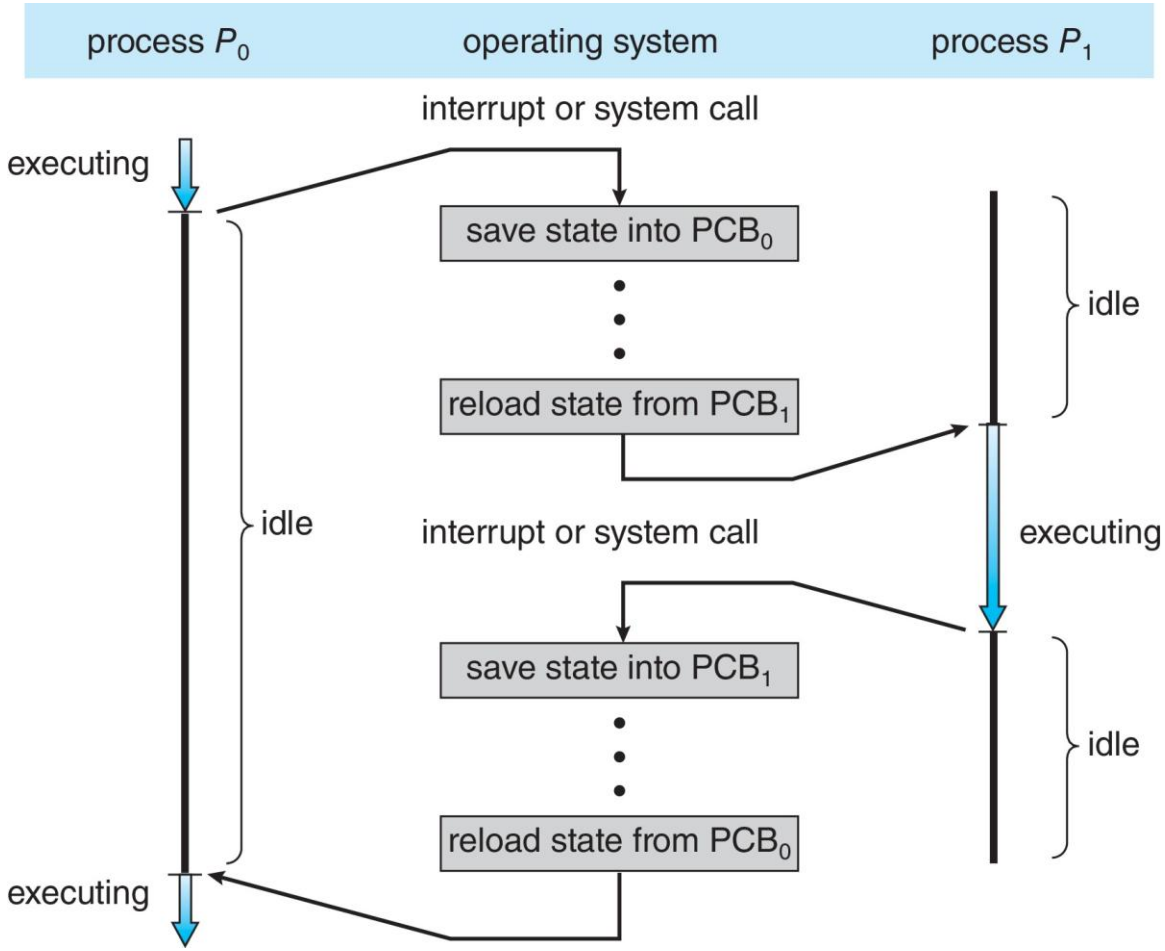
CPU Switch From Process to Process

- 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



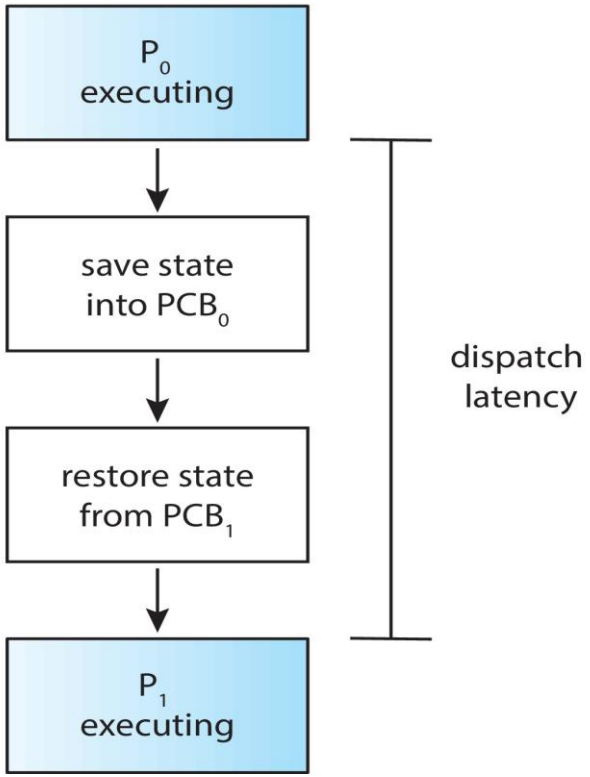
Mechanism of Context Switch

- 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



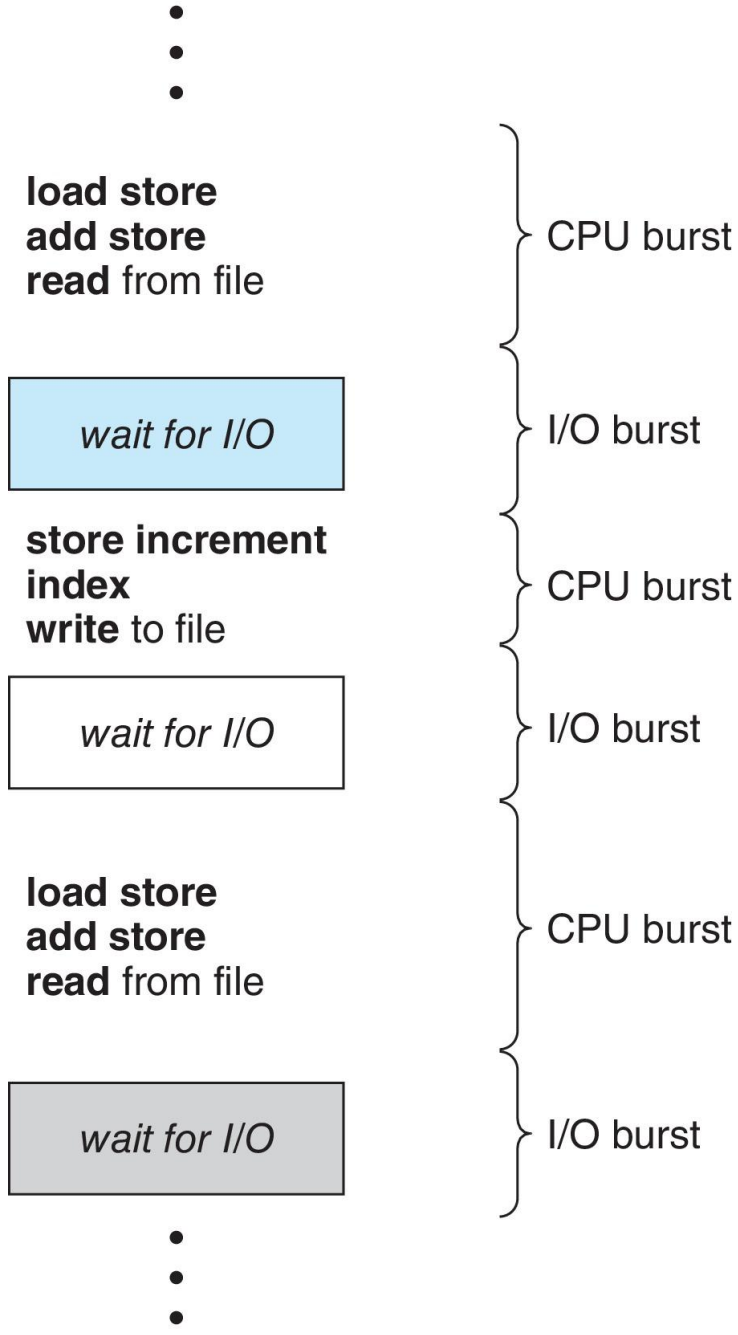
Mechanism of Context Switch (Dispatcher)

- 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



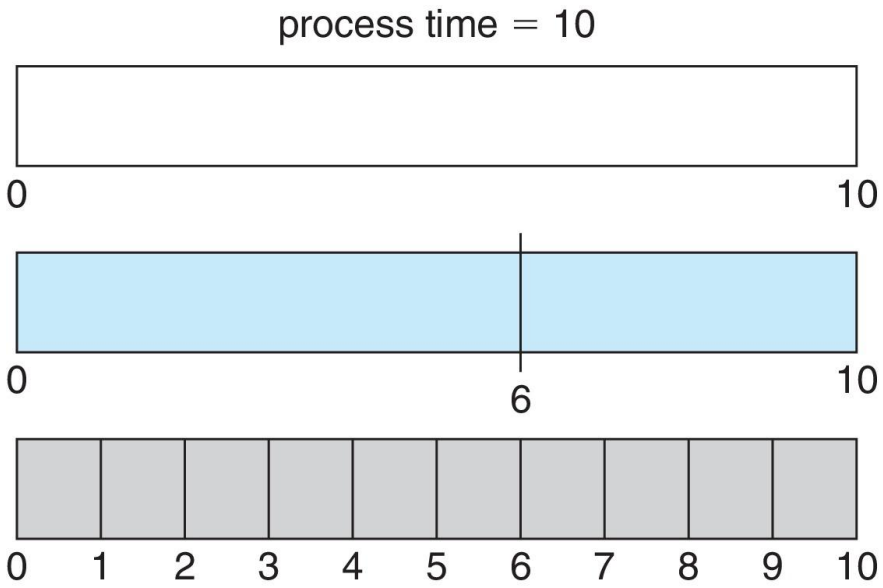
CPU Scheduler optimization

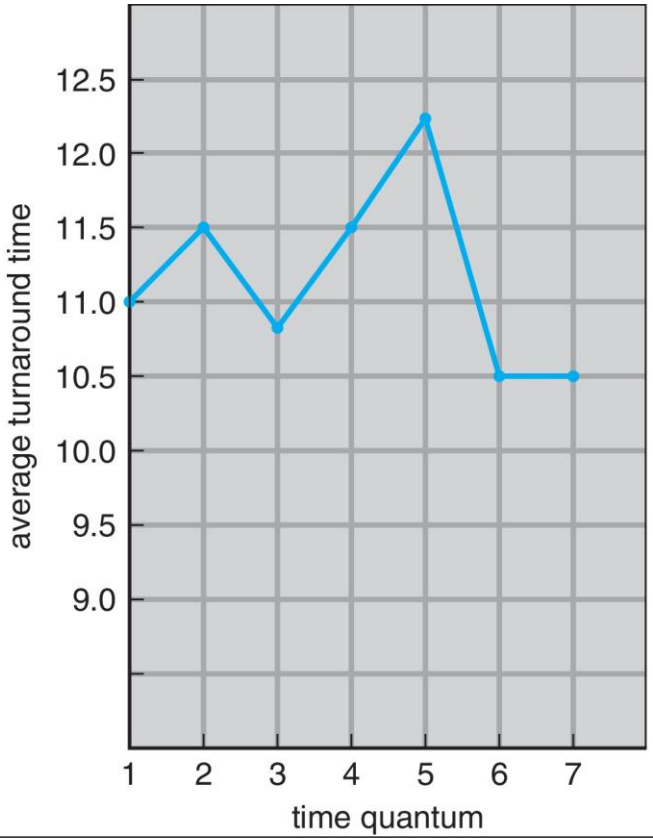
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern
- CPU scheduling decisions may take place when a process:
  1.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



Round Robin (RR) with varying Time Quantum

- Time Quantum and Context Switch Time



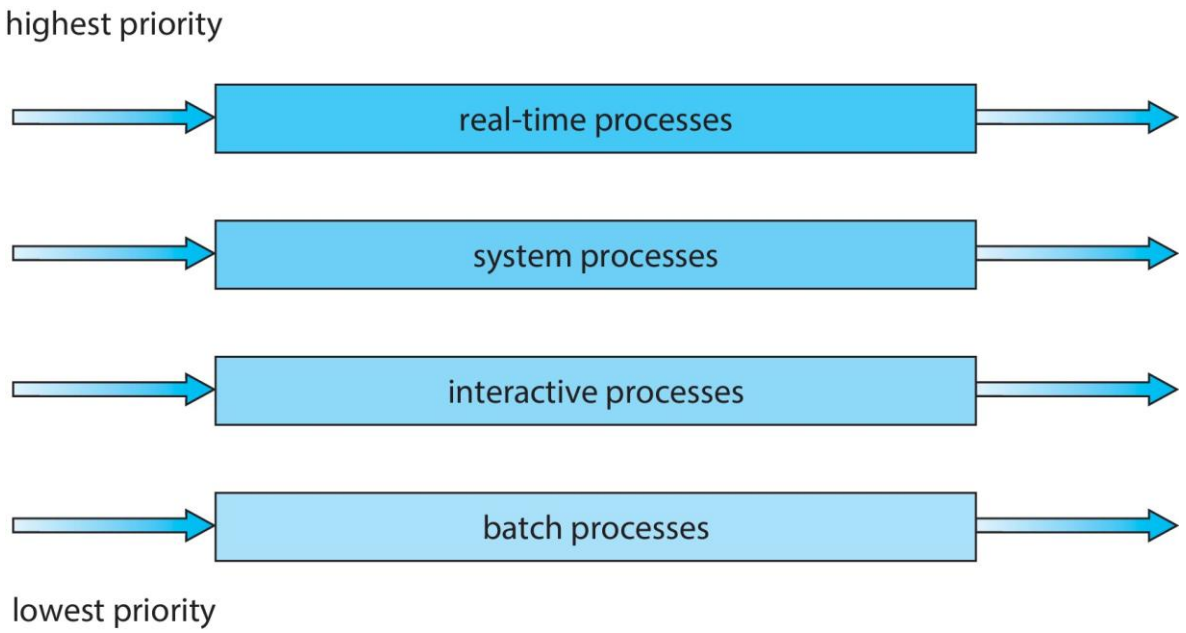


process	time
P_1	6
P_2	3
P_3	1
P_4	7

- Turnaround Time Varies With The Time Quantum

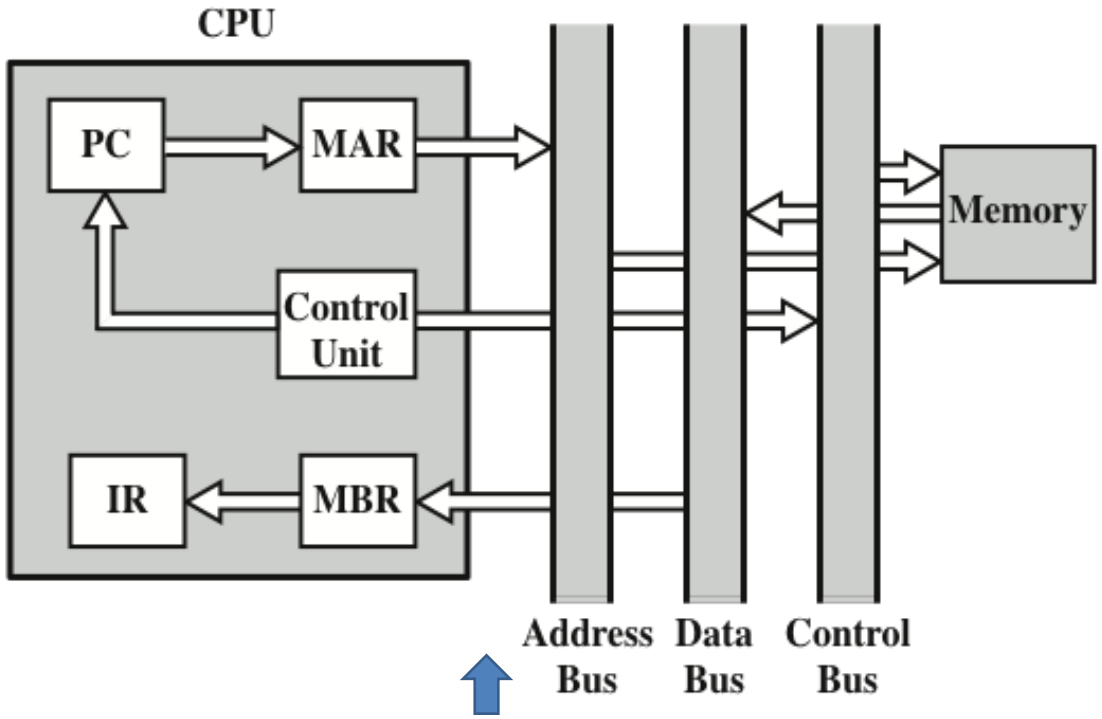
Priority Multilevel Queue

- Prioritization based upon process type

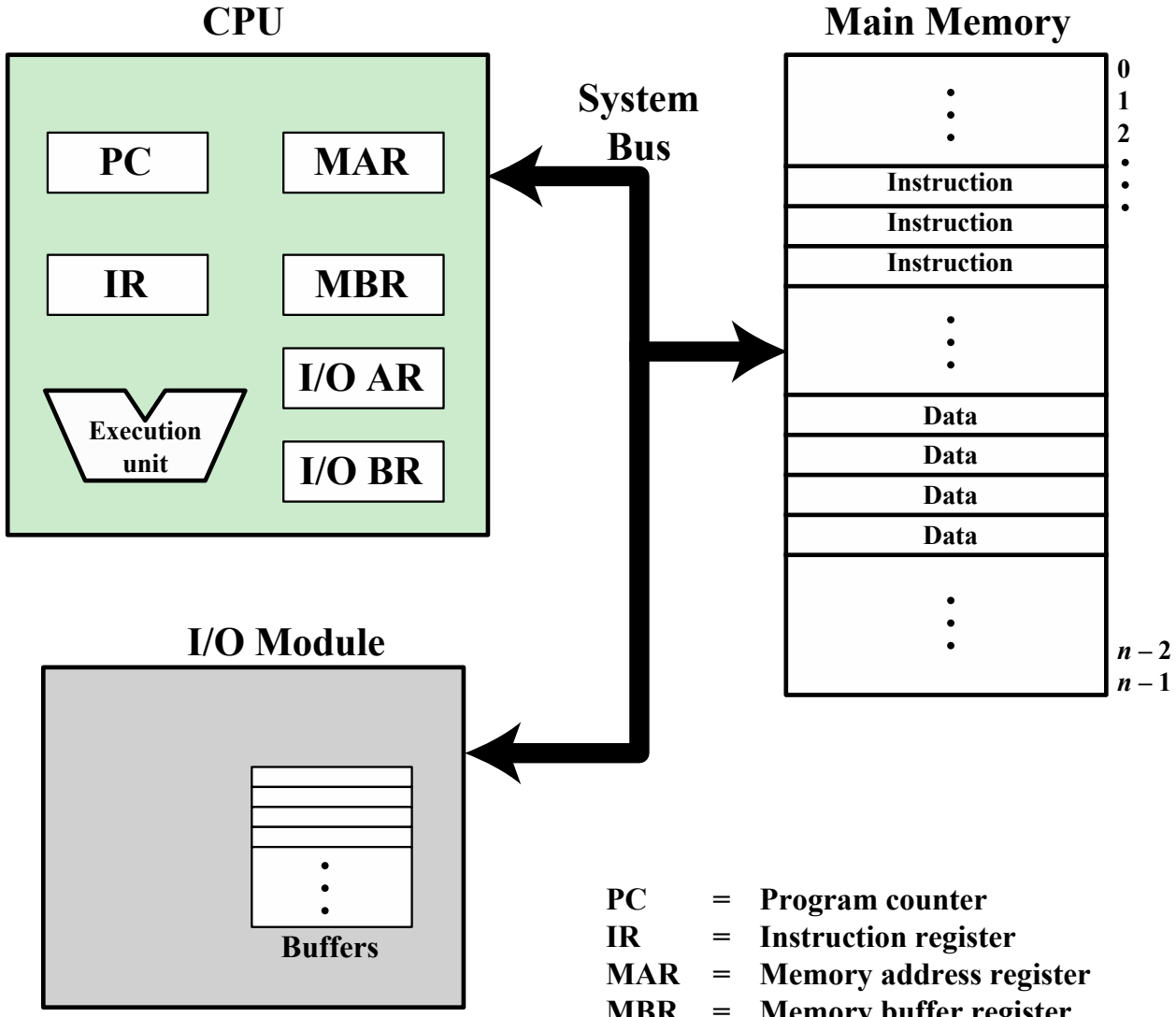


Process Concurrency on CPUs

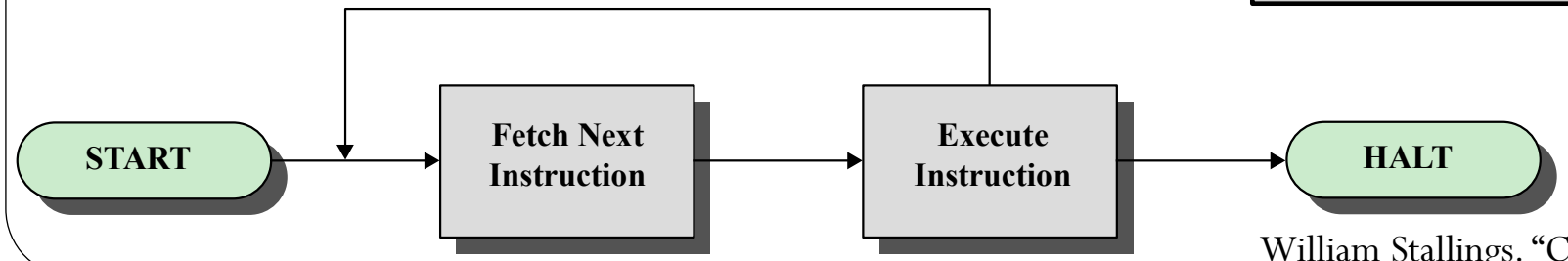
CPU and Memory



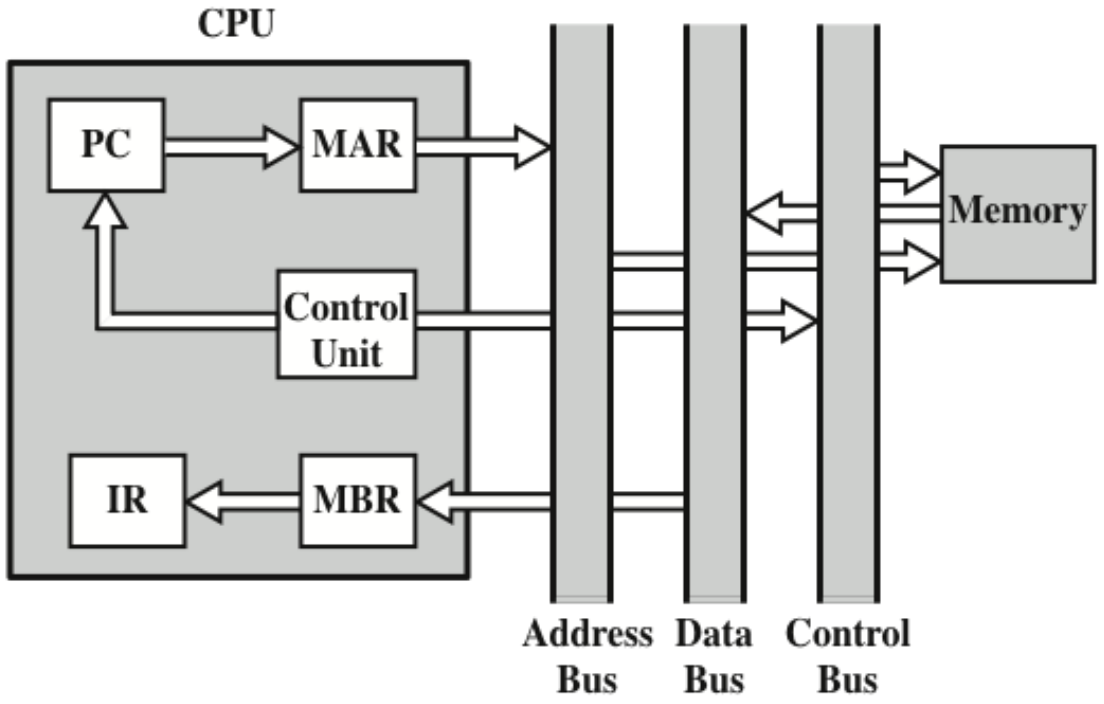
Fetch Cycle Execute Cycle



PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register



CPU and Memory

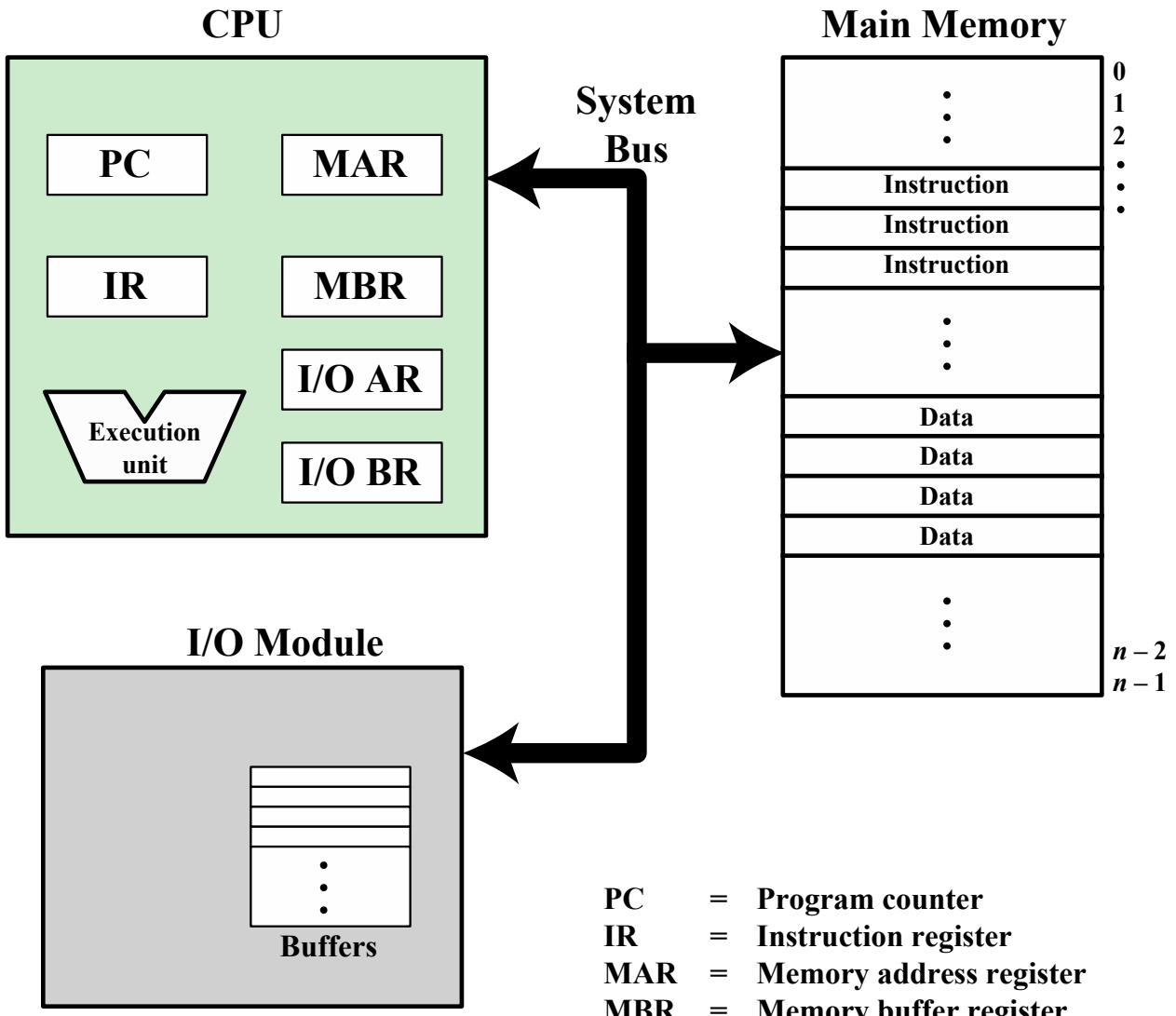


t1 : MAR ← PC

t2 : MBR ← MEMORY

PC ← (PC) + I

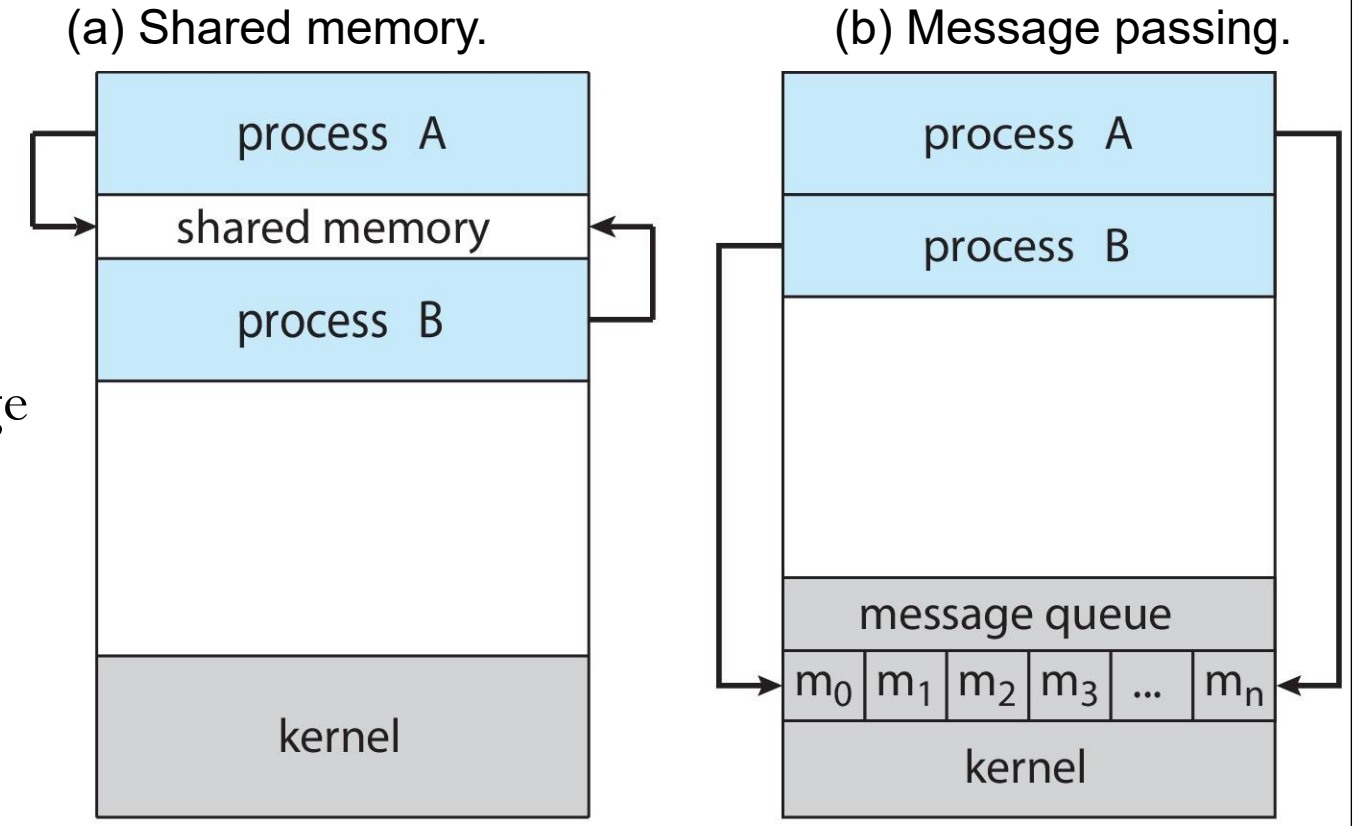
t3 : IR ← (MBR)



PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

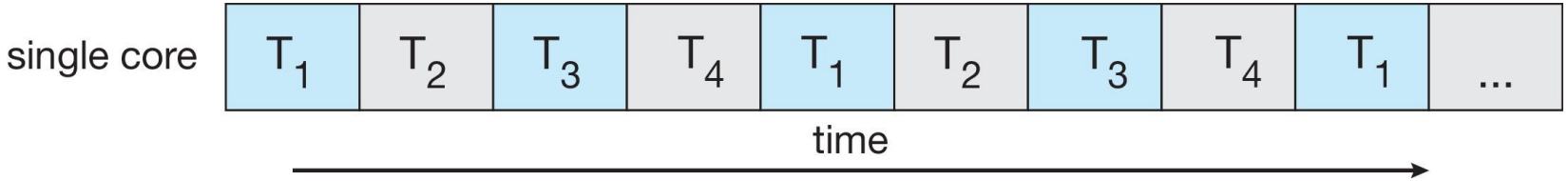
Inter Process Communication (IPC)

- Shared Memory: Processes can access same segment of memory via system call
- Message Queues:
  - Process can open a mailbox at a specified location.
  - Processes can send/receive message from mailbox
  - OS buffers messages between send and receive

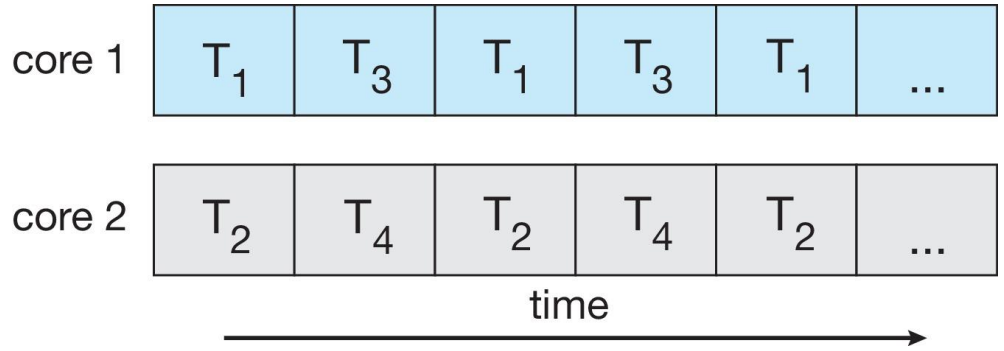


Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

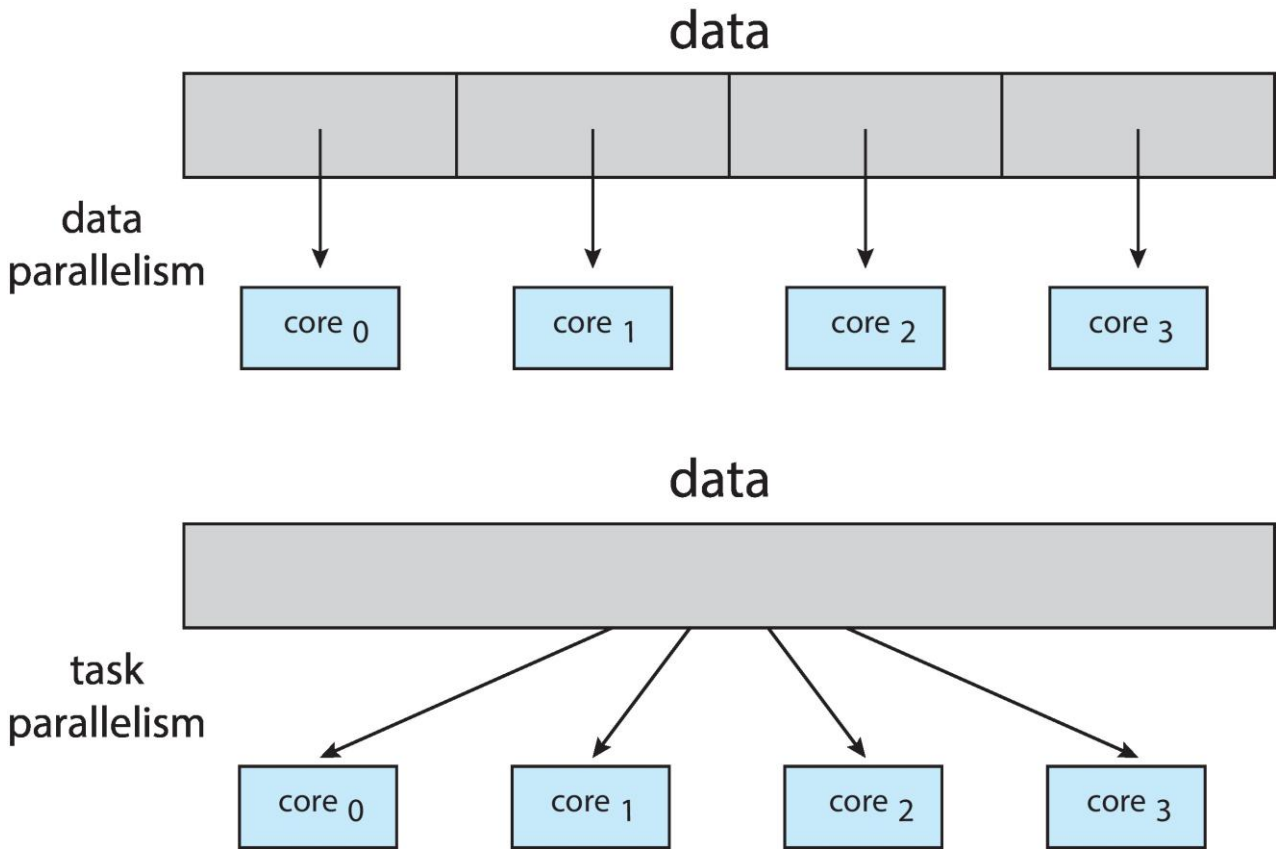


- **Parallelism on a multi-core system:**



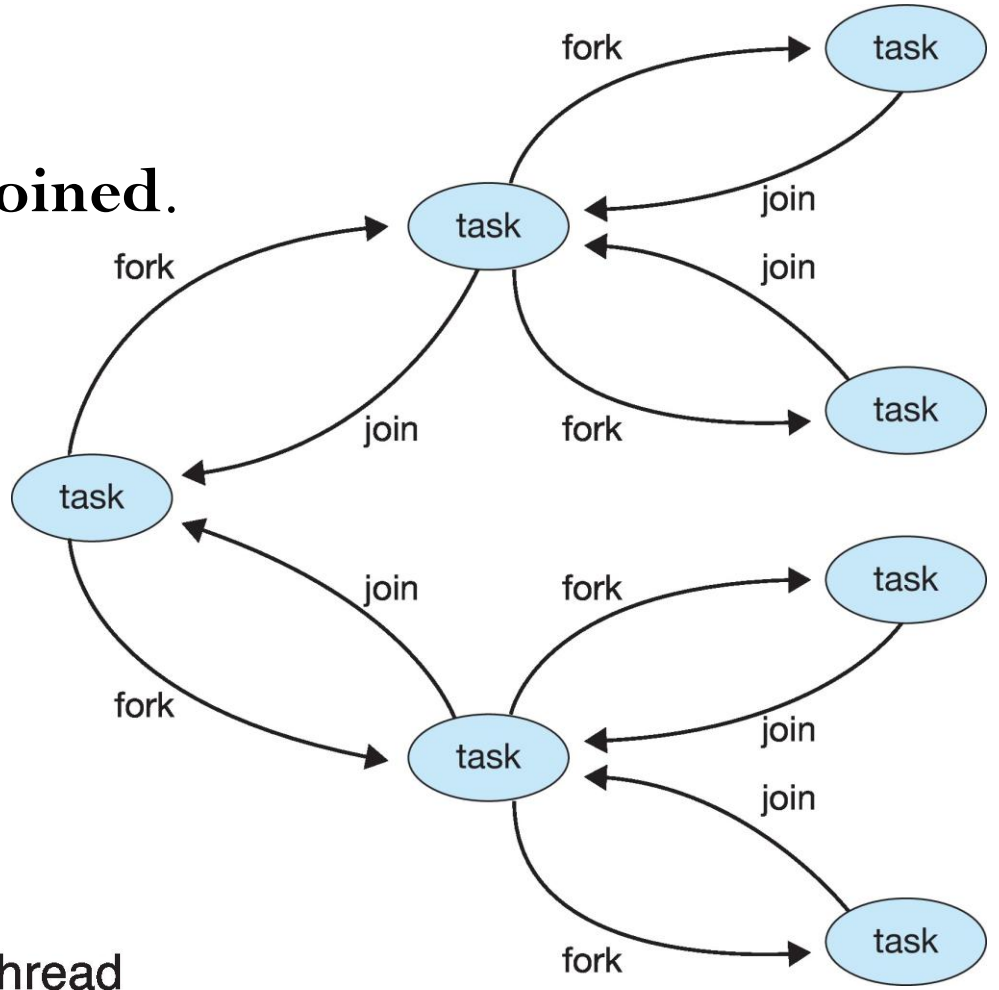
Parallelism

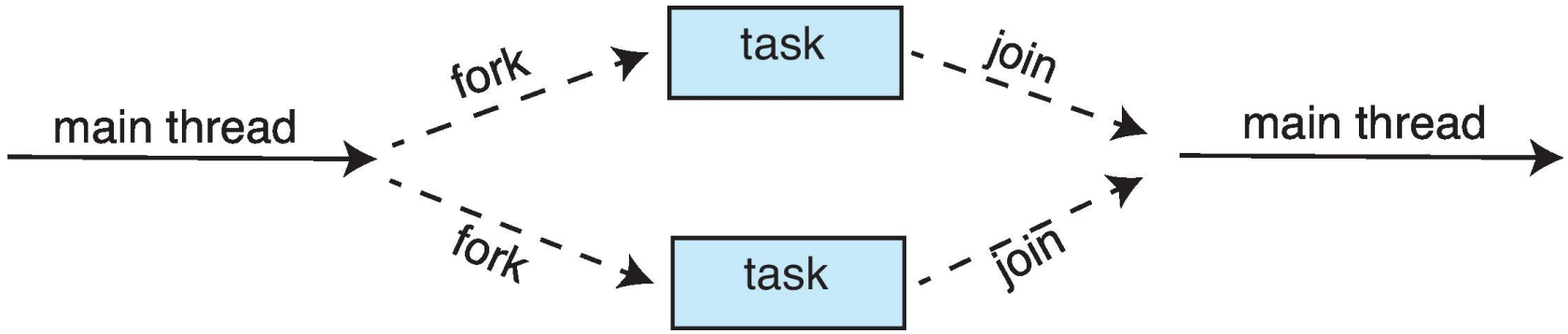
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation



Parallelism using Fork-Join

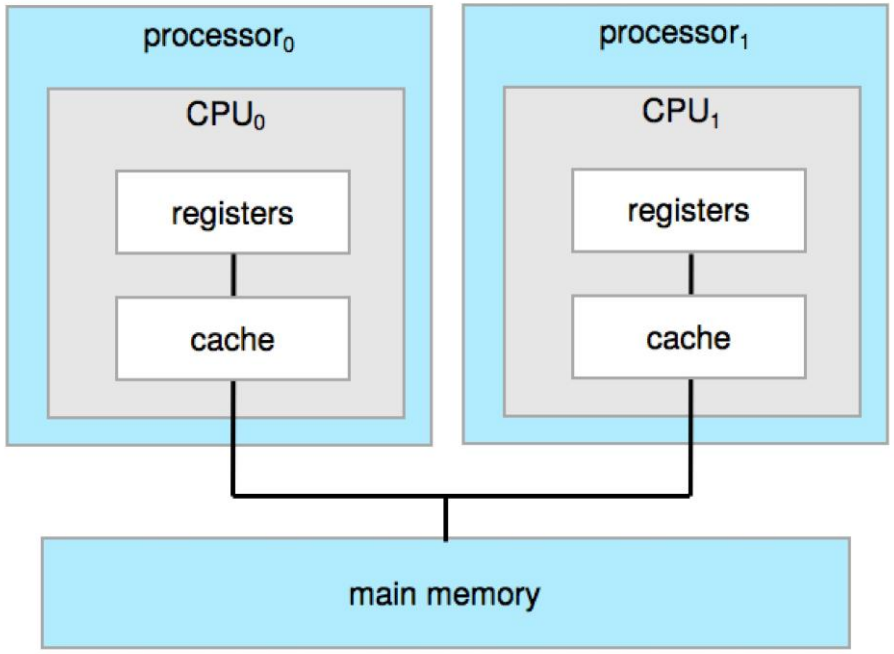
- Multiple threads of tasks are **forked**, and then **joined**.



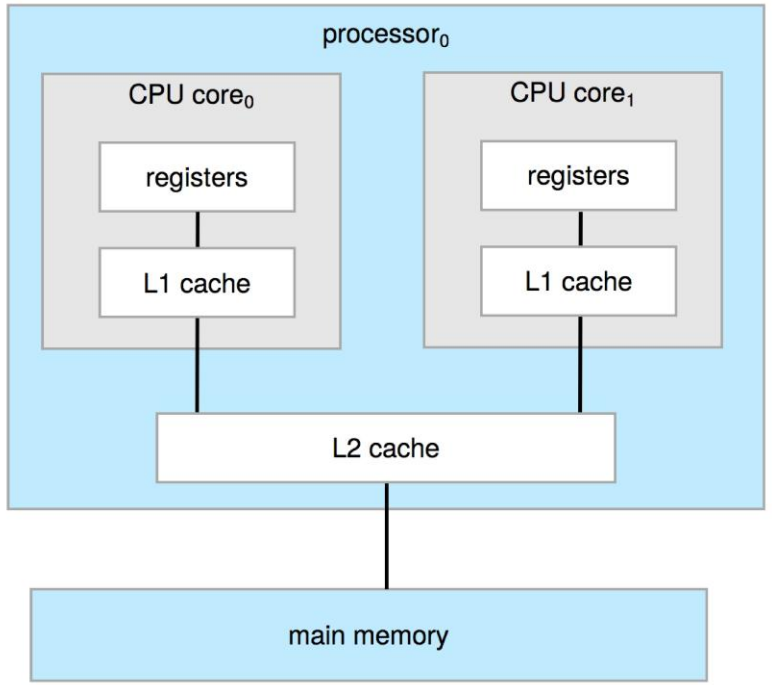


Multiprocessing

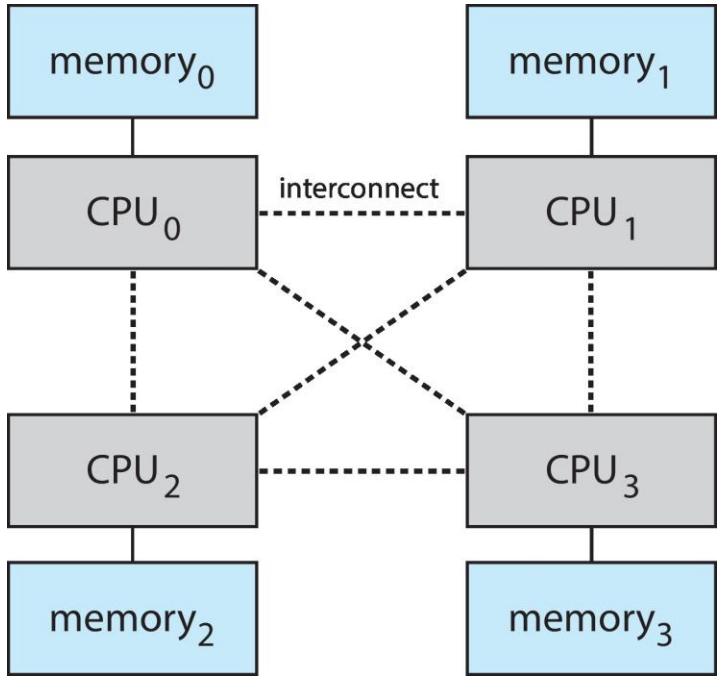
- Two types:
  1. **Asymmetric Multiprocessing** – each processor is assigned a specific task.
  2. **Symmetric Multiprocessing** – each processor performs all tasks



Symmetric Multiprocessing Architecture

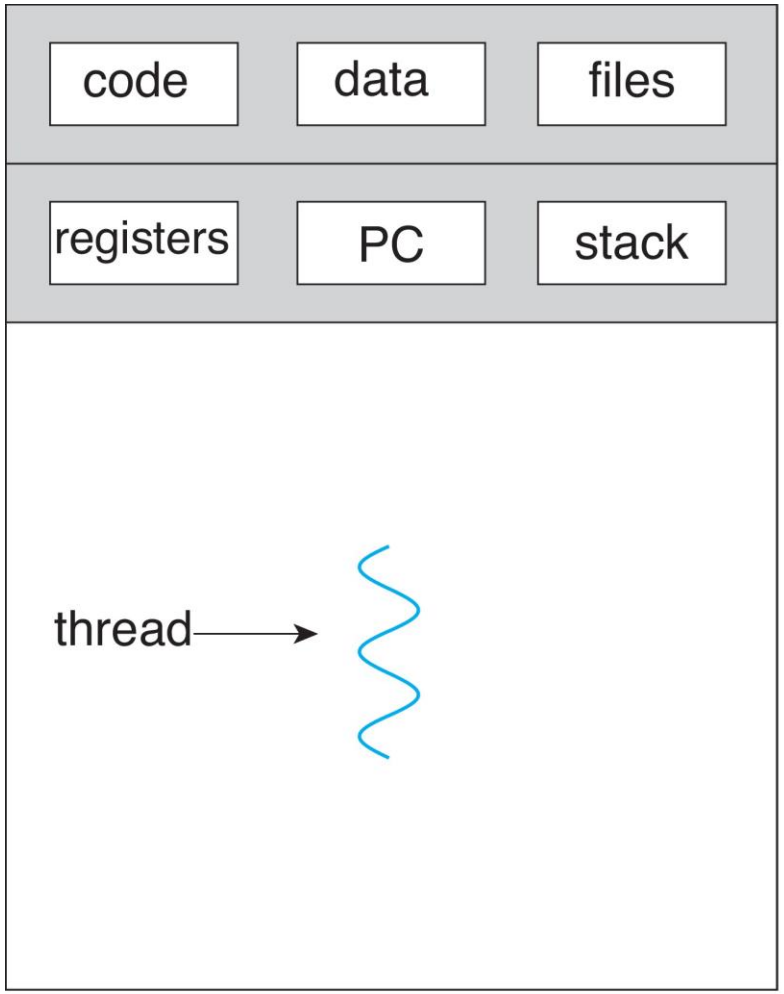


Dual-Core Design

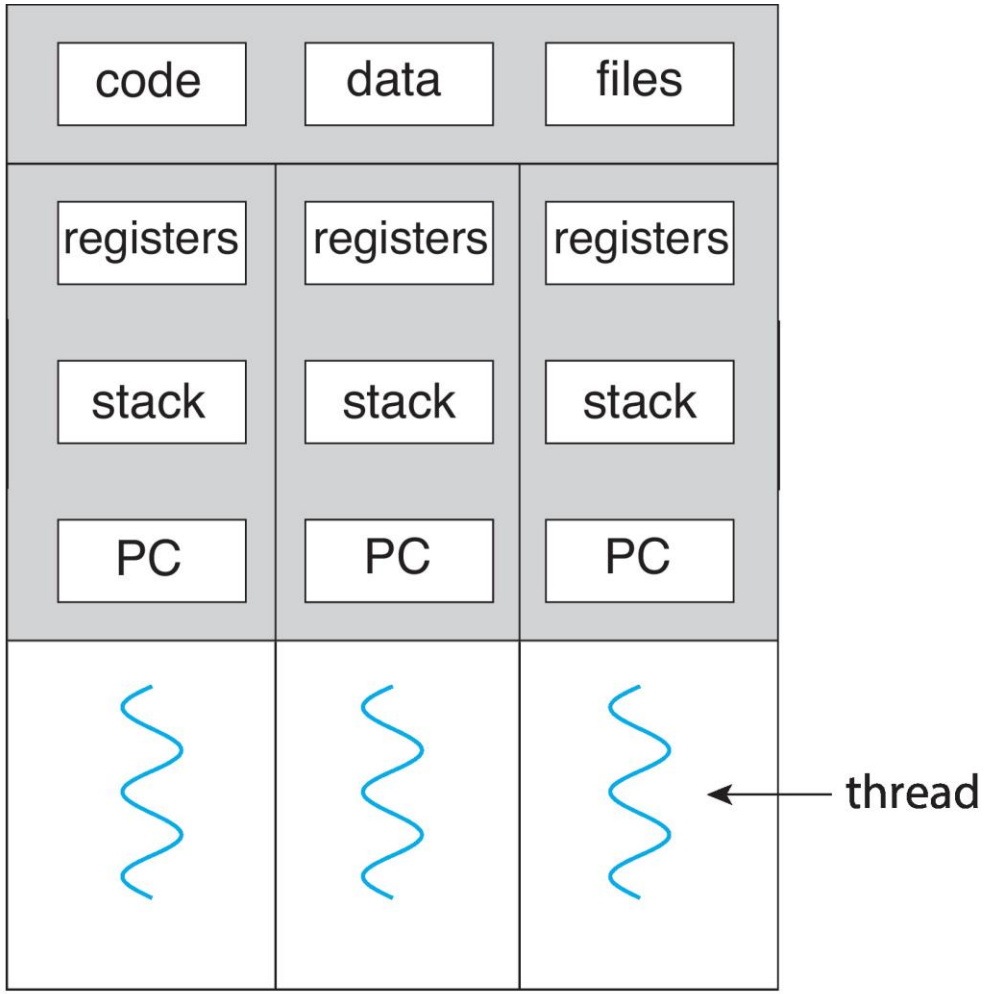


Non-Uniform Memory Access System

Multiprocessing



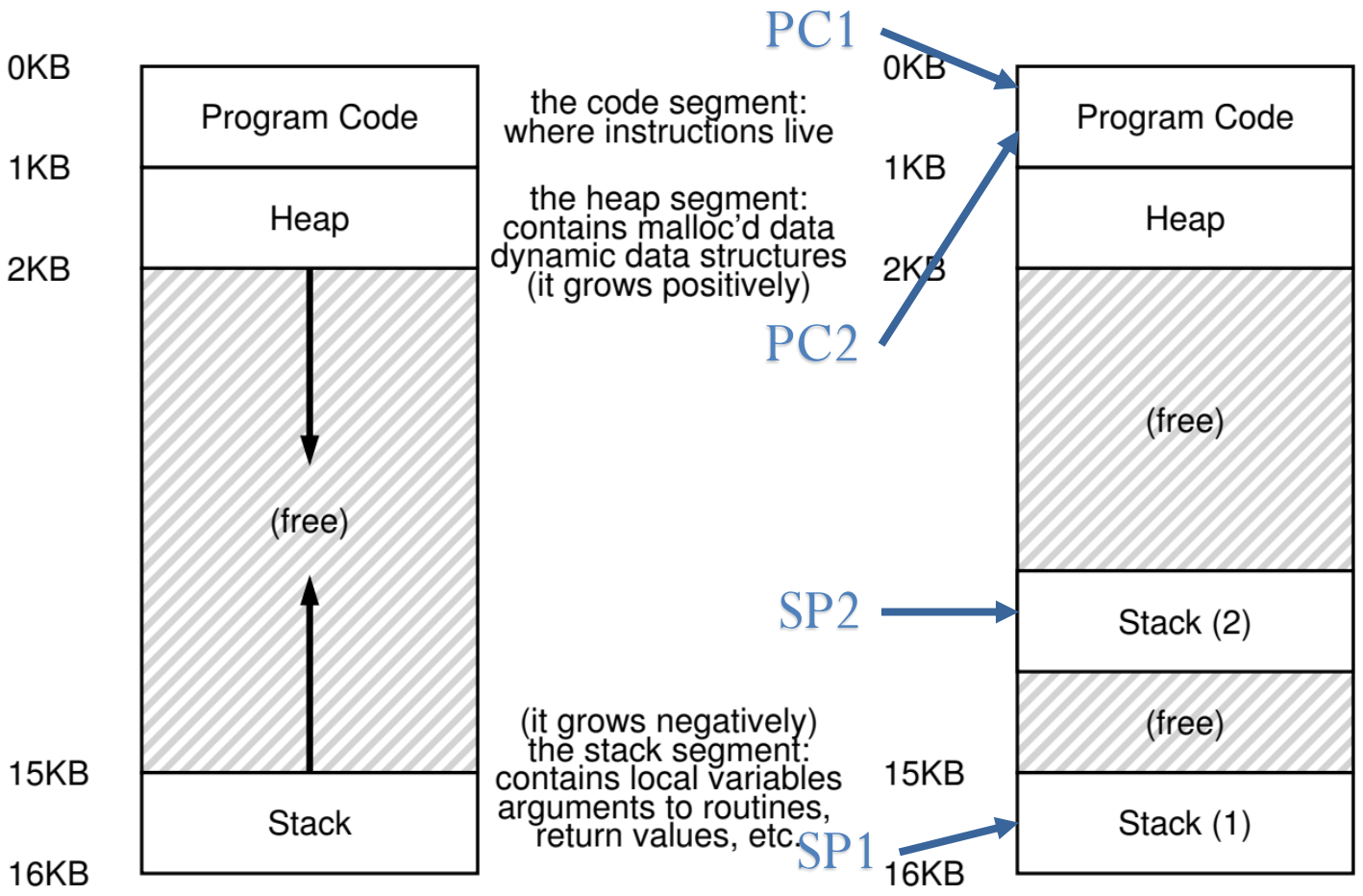
single-threaded process



multithreaded process

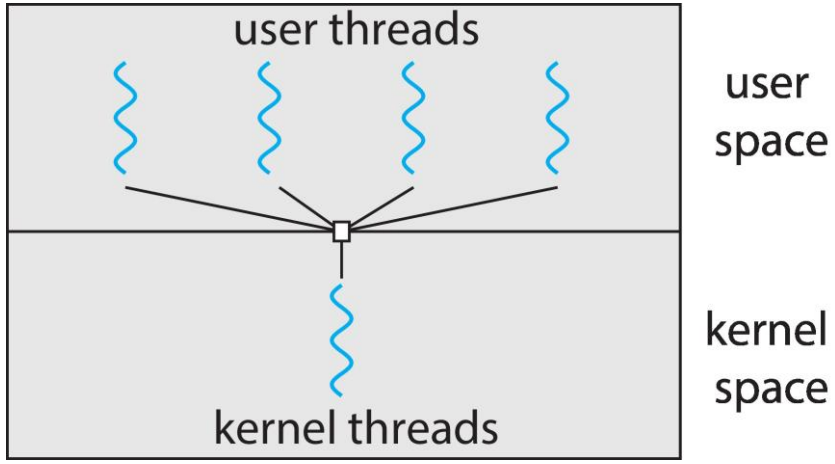
Multi threaded process

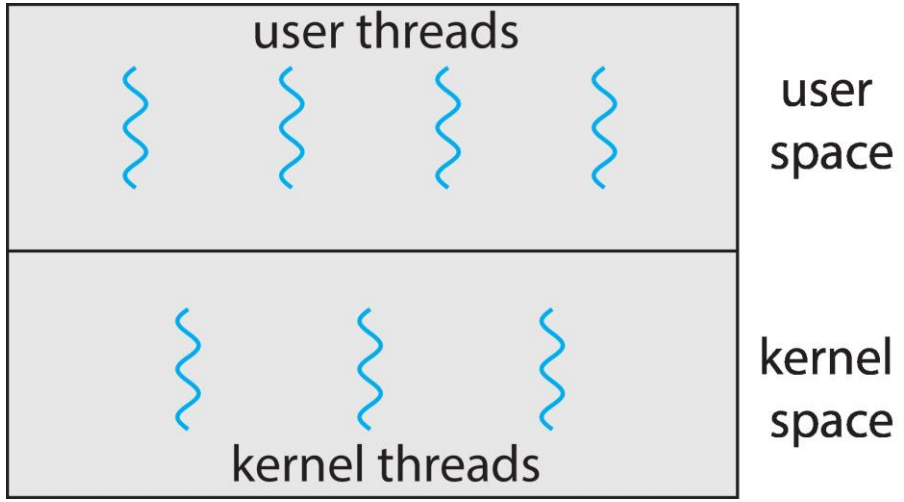
- A thread is like another copy of a process that executes independently
- Threads shares the same address space (code, heap)
- Each thread has separate PC
  - Each thread may run over different part of the program
- Each thread has separate stack for independent function calls



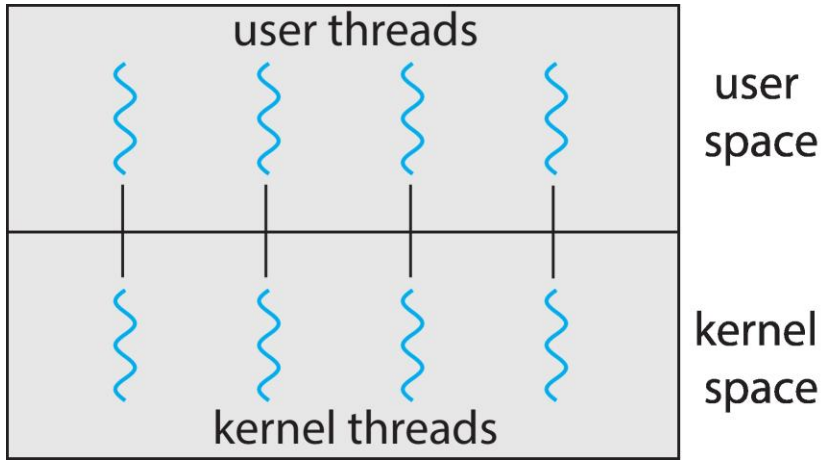
Threads: User and Kernel

- Many-to-One

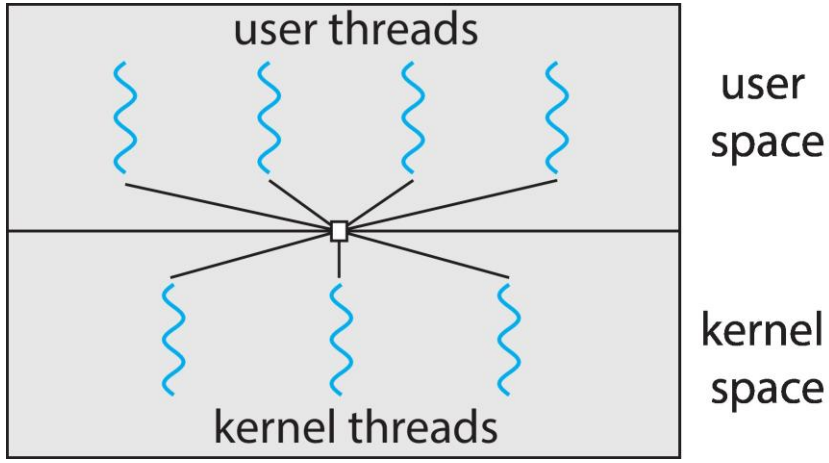




- One-to-One



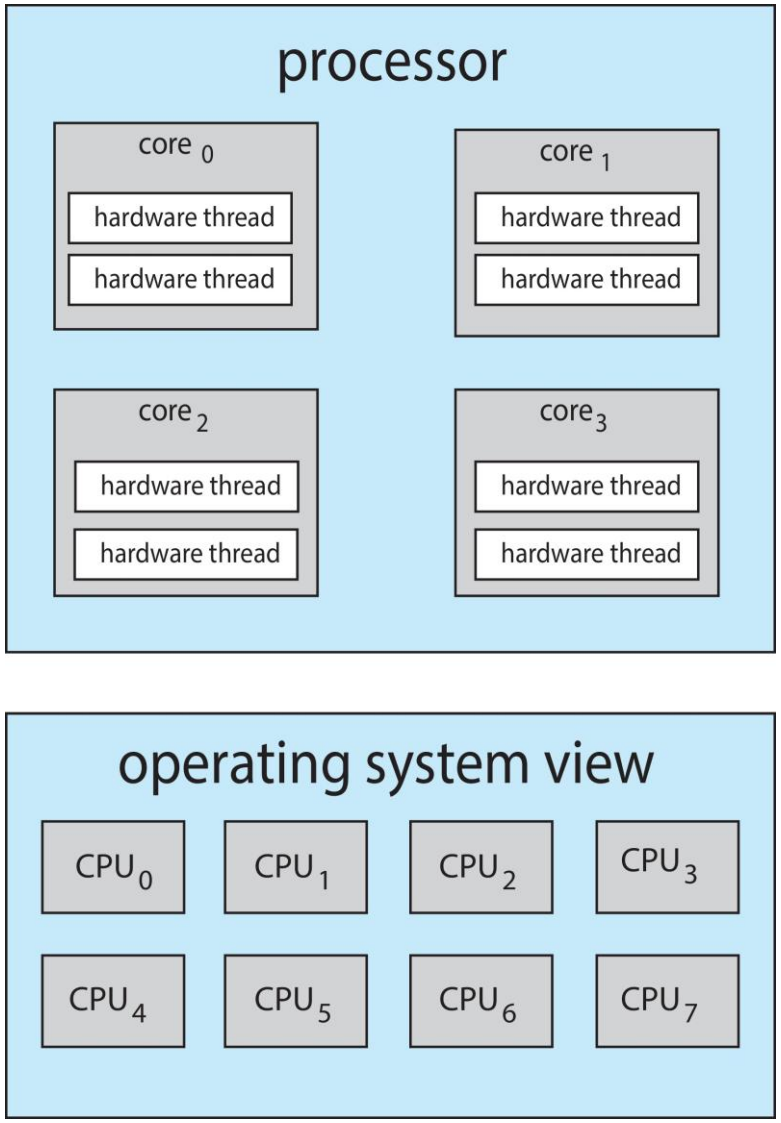
- Many-to-Many



Multithreaded Multicore System

- **Chip-multithreading** (CMT) assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)

- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.

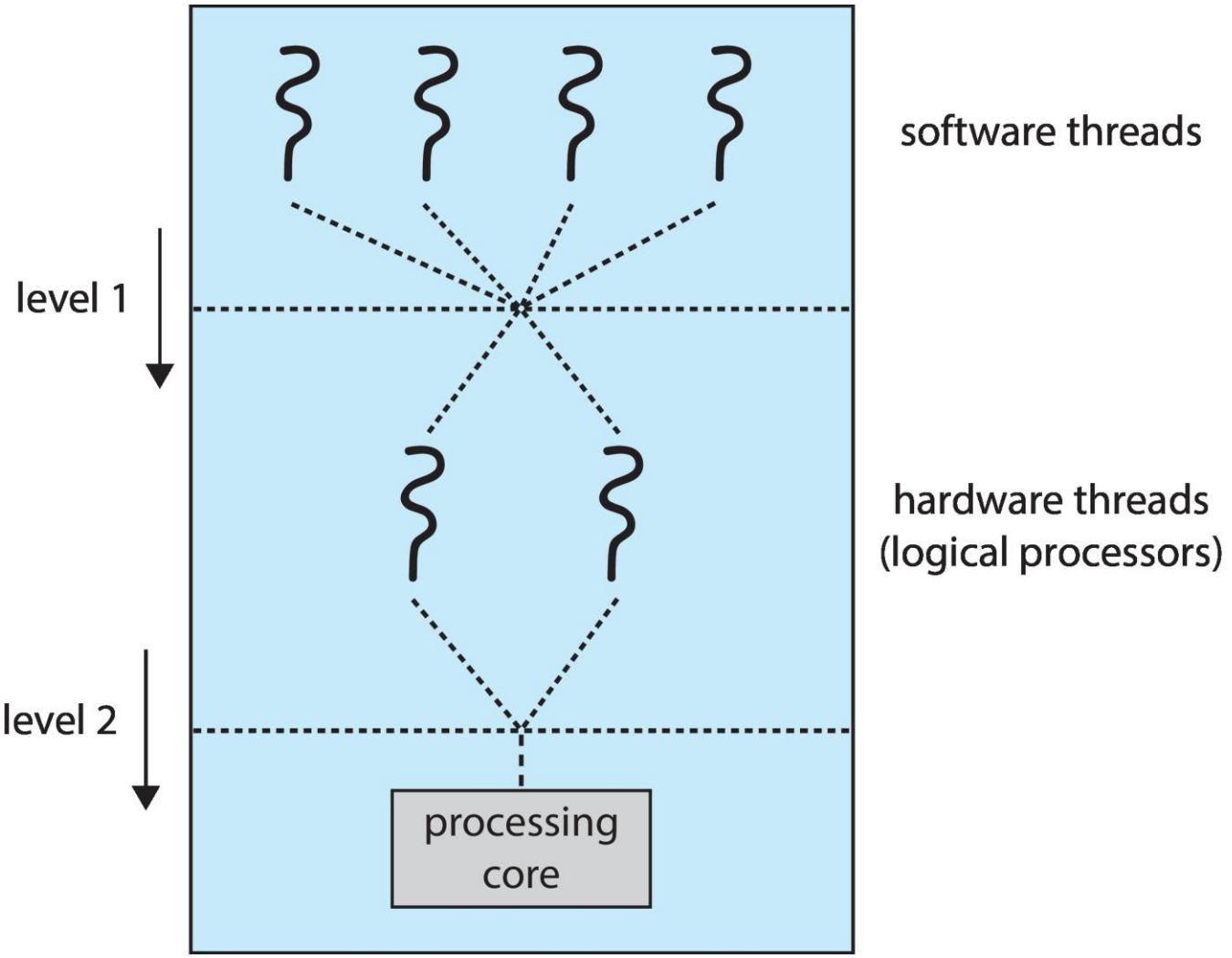


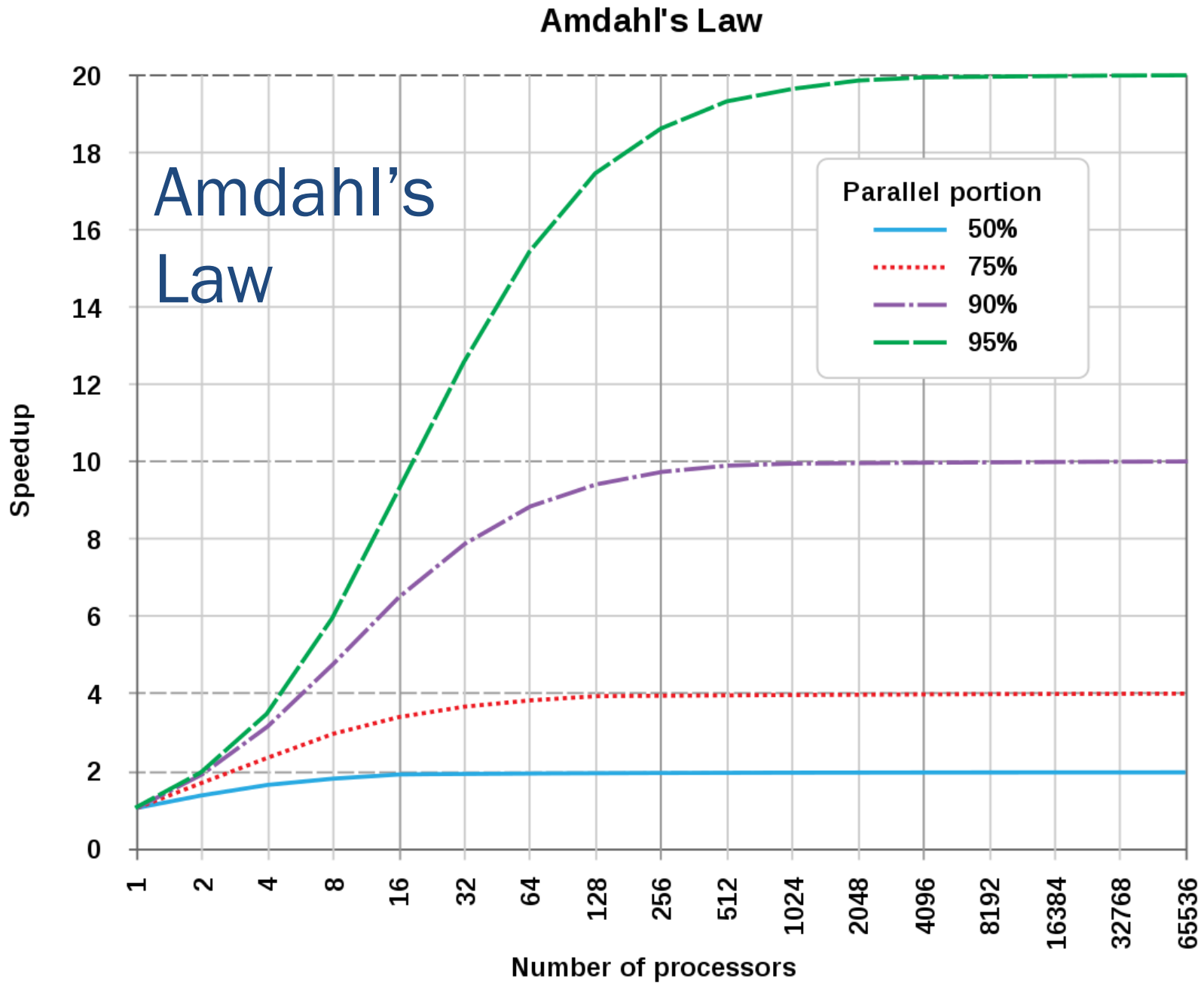
Multithreaded Multicore System

Two levels of scheduling:

1. The operating system deciding which software thread to run on a logical CPU

2. How each core decides which hardware thread to run on the physical core.




Amdahl's Law
Amdahl's Law
Speedup
Number of processors
Parallel portion
50%
75%
90%
95%
0
2
4
6
8
10
12
14
16
18
20
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536

Race conditions and synchronization

- **Race condition:** Concurrent execution can lead to different results
- **Critical section:** portion of code that can lead to race conditions
- What we need: mutual exclusion
  - Only one thread should be executing critical section at any time
- What we need: atomicity of the critical section
  - The critical section should execute like one uninterruptible instruction
- How is it achieved? Locks

Critical section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \cdots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

Critical section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- Assume that each process executes at a nonzero speed
- No assumption concerning **relative speed** of the ***n*** processes

Locks

- Consider update of shared variable `balance = balance + 1;`
  - We can use a special lock variable to protect it

```
1  lock_t mutex; // some globally-allocated lock 'mutex'
2  ...
3  lock(&mutex);
4  balance = balance + 1;
5  unlock(&mutex);
```

- All threads accessing a critical section share a lock
- One threads succeeds in locking – owner of lock
- Other threads that try to lock cannot proceed further until lock is released by the owner

Intuitive Solution

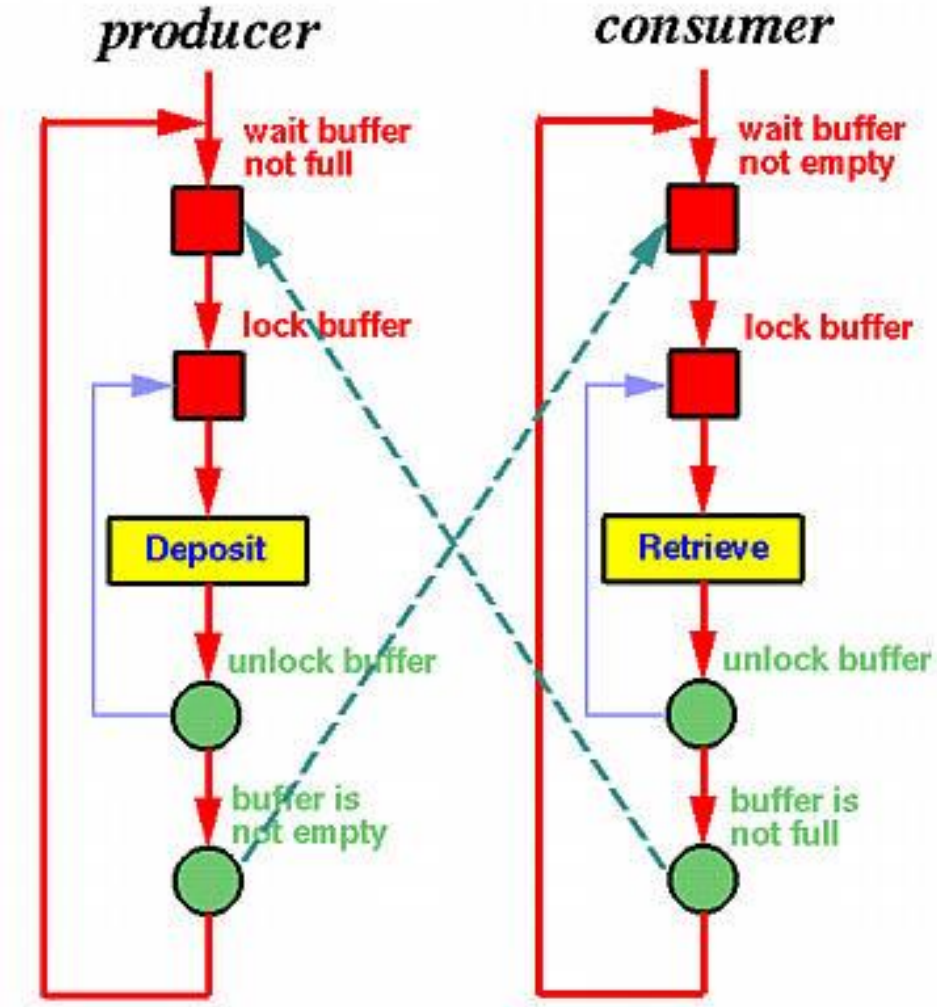
```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

```
while (true) {
        entry section
            critical section
        exit section
            remainder section
}
```

- Entry section: disable interrupts
- Exit section: enable interrupts
- Is this enough?
- No, not always!
- Many issues here:
  - Disabling interrupts is a privileged instruction and program can misuse it (e.g., run forever)
  - Will not work on multiprocessor systems, since another thread on another core can enter critical section
- Used to implement locks on single processor systems inside the OS
  - Need better solution for other situations

Semaphore for Producer/Consumer

- Need two semaphores for signaling
  - One to track empty slots, and make producer wait if no more empty slots
  - One to track full slots, and make consumer wait if no more full slots
- One semaphore to act as mutex for buffer



https://pages.mtu.edu/~shene/NSF-3/e-Book/SEMA/TM-example-buffer.html

Semaphore for Producer/Consumer

- Correct use of Mutex

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);         // Line P1
        sem_wait(&mutex);         // Line P1.5 (MUTEX HERE)
        put(i);                   // Line P2
        sem_post(&mutex);         // Line P2.5 (AND HERE)
        sem_post(&full);          // Line P3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);          // Line C1
        sem_wait(&mutex);         // Line C1.5 (MUTEX HERE)
        int tmp = get();          // Line C2
        sem_post(&mutex);         // Line C2.5 (AND HERE)
        sem_post(&empty);         // Line C3
        printf("%d\n", tmp);
    }
}
```

Semaphore for Producer/Consumer

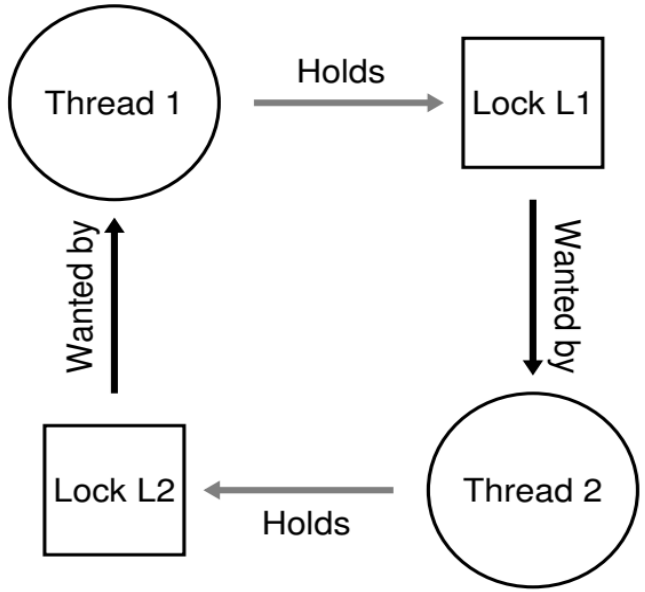
- What if lock is acquired before signaling?
- Waiting thread sleeps with mutex and the signaling thread can never wake it up

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);       // Line P0 (NEW LINE)
        sem_wait(&empty);       // Line P1
        put(i);                 // Line P2
        sem_post(&full);        // Line P3
        sem_post(&mutex);       // Line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);       // Line C0 (NEW LINE)
        sem_wait(&full);        // Line C1
        int tmp = get();        // Line C2
        sem_post(&empty);       // Line C3
        sem_post(&mutex);       // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

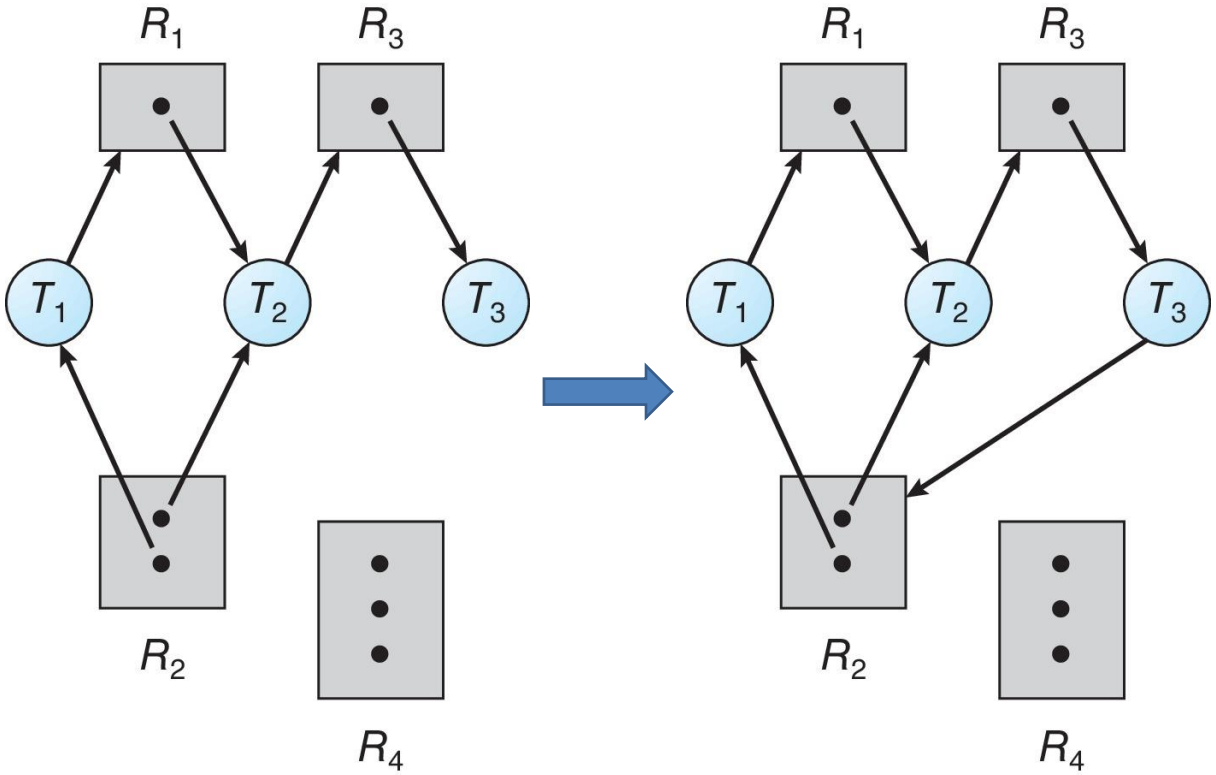
Liveness

- **Liveness:** a set of properties that a system must satisfy to ensure processes make progress. Indefinite waiting is an example of a liveness failure.
- Consider if P_0 executes wait(S) and P_1 wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q).
- Let ***S*** and ***Q*** be two semaphores initialized to 1. Then, P_1 is waiting until P_0 execute signal(S). Since these signal() operations will never be executed, P_0 and P_1 are **deadlocked**.

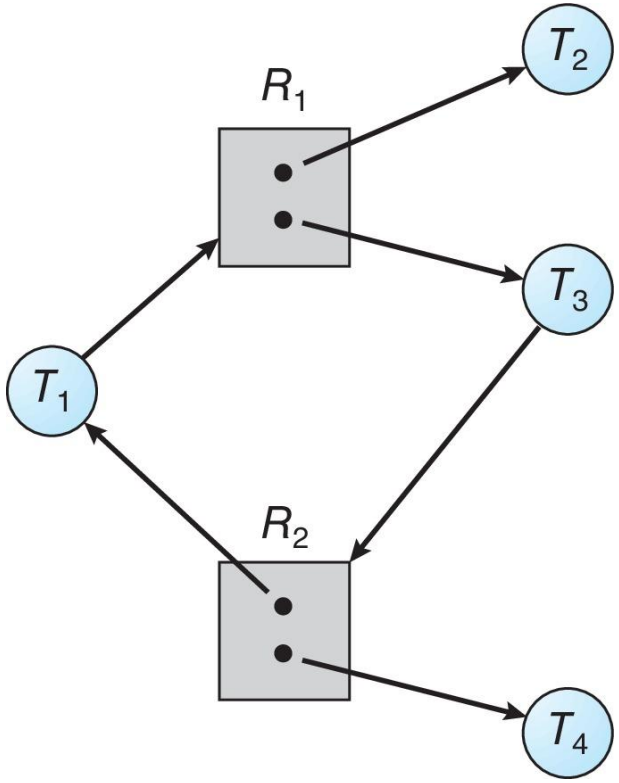
P_0	P_1
wait(S);	**wait(Q);**
wait(Q);	**wait(S);**
...	**...**
signal(S);	**signal(Q);**
signal(Q);	**signal(S);**



Deadlock (Resource Allocation Graph)

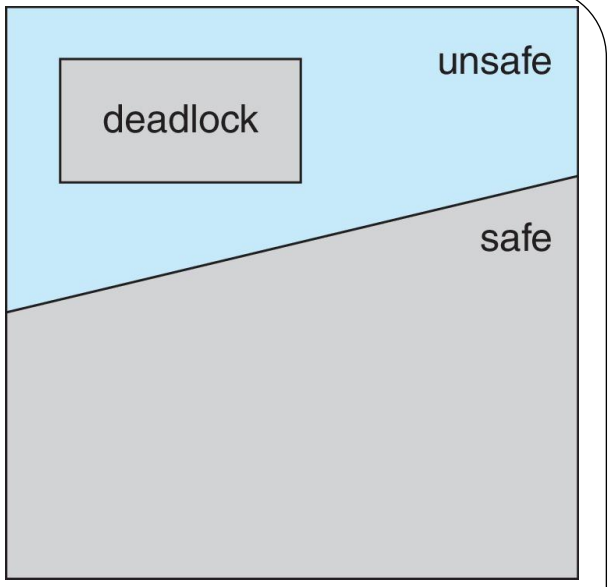


Deadlock



Graph with a Cycle But no Deadlock

Methods for Handling Deadlocks



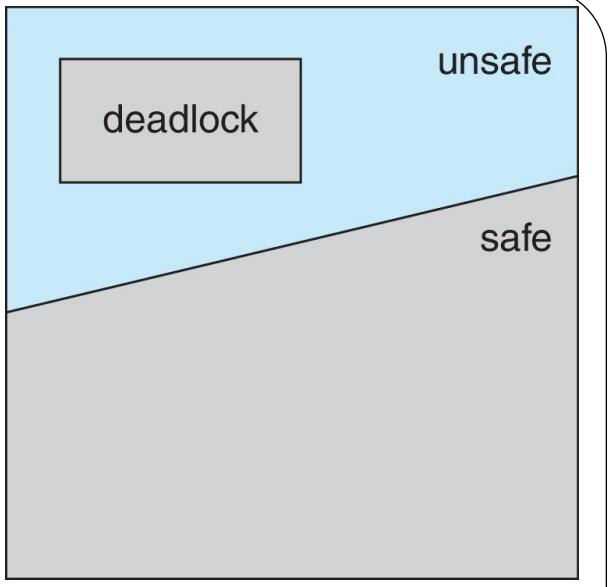
Ensure that the system will **never** enter a deadlock state:

1. Deadlock prevention
   - Invalidate one of the four necessary conditions for deadlock
2. Deadlock avoidance
   - Requires that the system has some additional ***a priori*** information available

Otherwise,

1. Allow the system to enter a deadlock state and then recover
2. Ignore the problem and pretend that deadlocks never occur in the system.

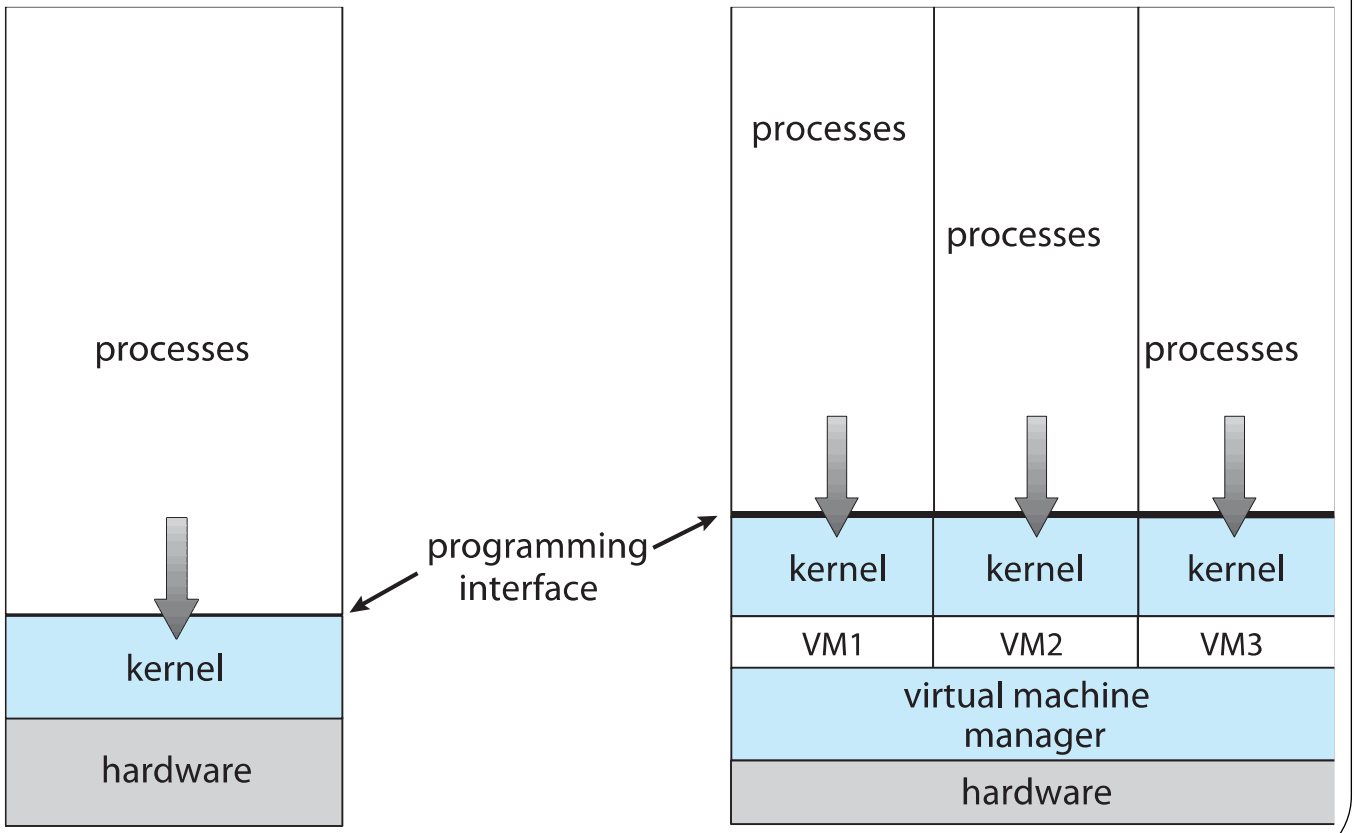
Deadlock avoidance



- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.
- Multiple instances of a resource type then use the Banker's Algorithm
- Banker's algorithm is very popular, but impractical in real life to assume this knowledge
- When a thread requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

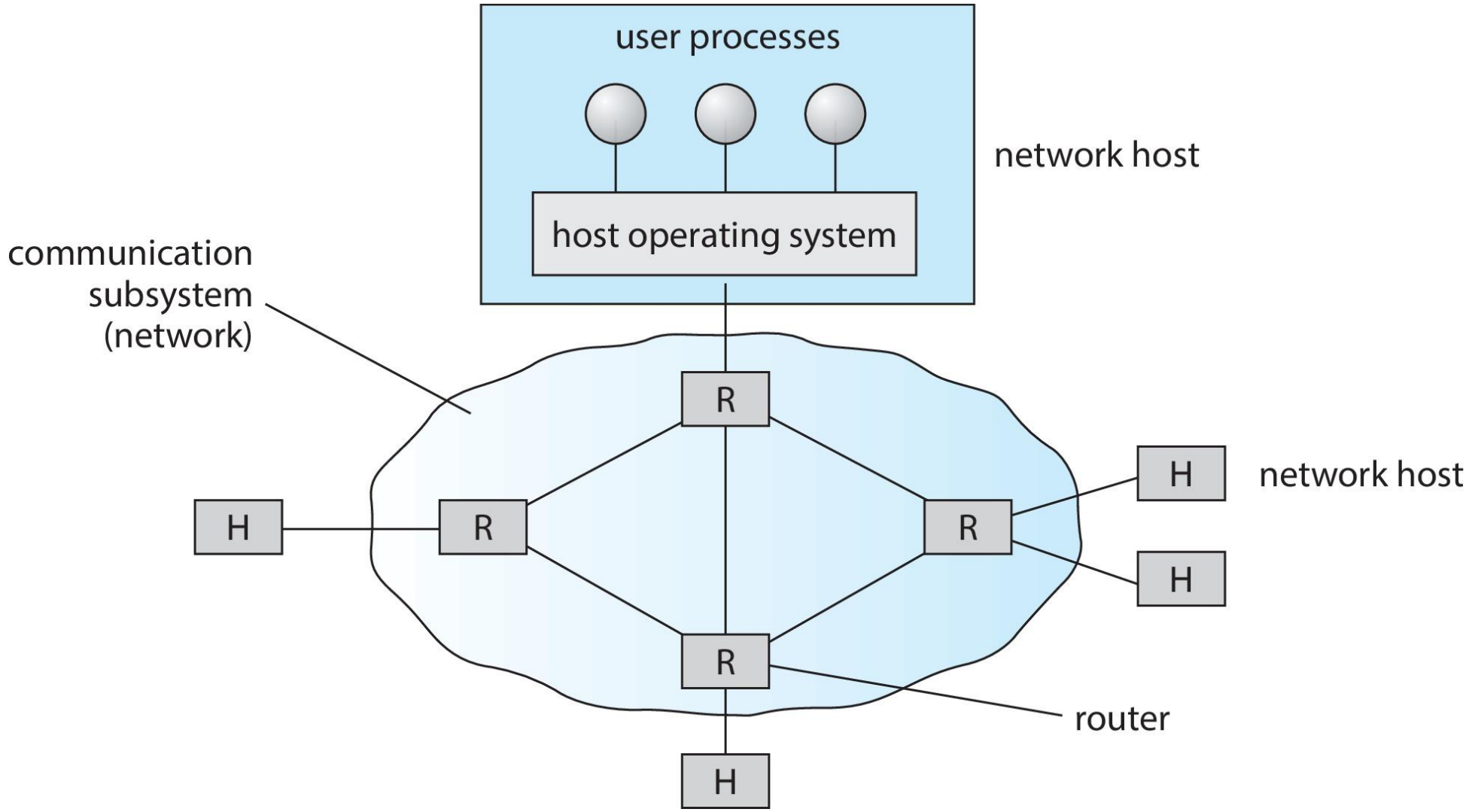
Virtualizations and Services for Cloud

- **Virtualization** – OS natively compiled for CPU, running **guest** OSes also natively compiled
  - Consider VMware running WinXP guests, each running applications, all on native WinXP **host** OS
  - **VMM** (virtual machine Manager)
    - provides virtualization services
- **Infrastructure as a Service** (IaaS)
- **Platform as a Service** (PaaS)
- **Container as a Service** (CaaS)
- **Software as a Service** (SaaS)



Computer Networks

Wide-Area Network (WAN)



Communication Protocol

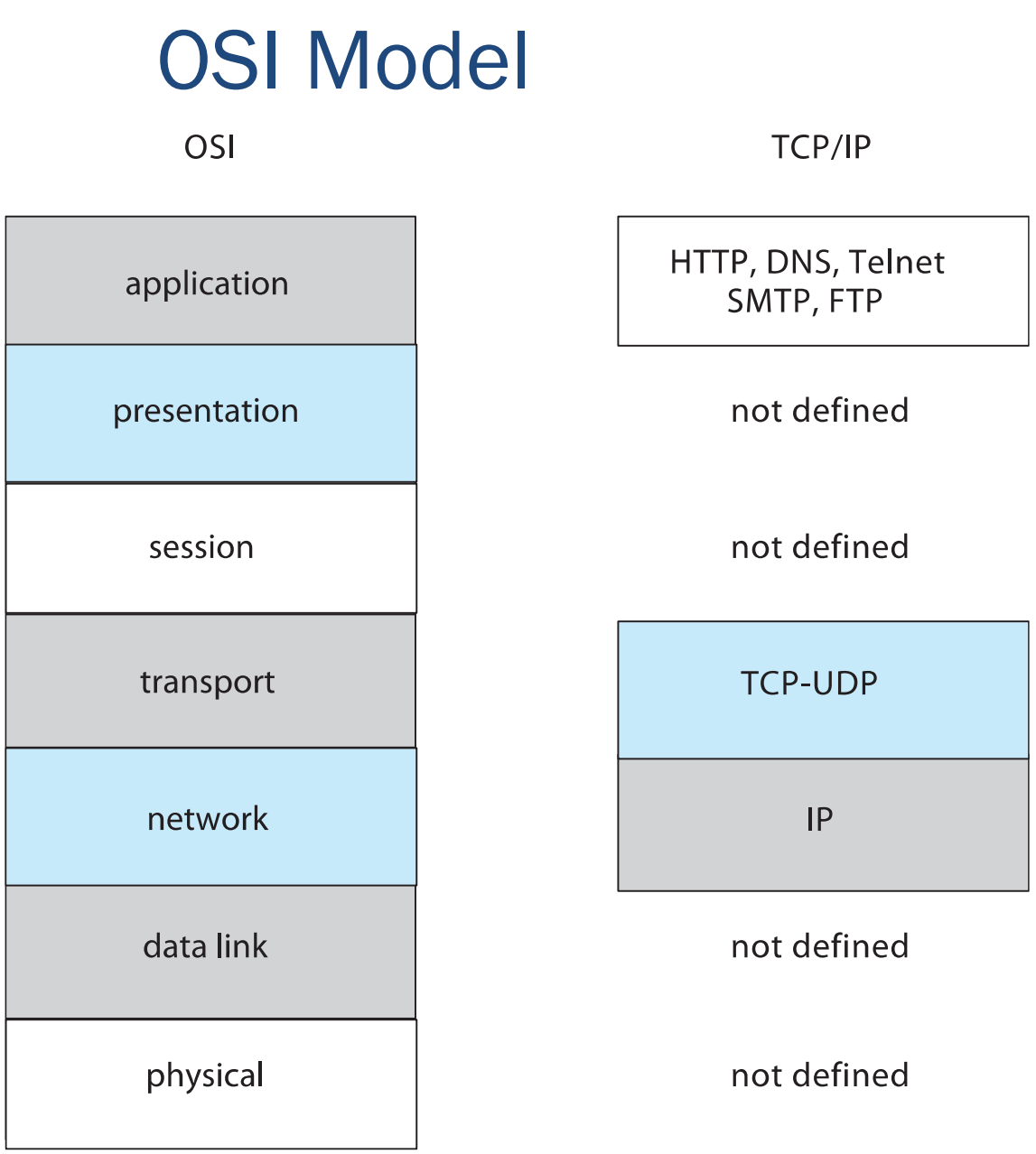
The communication network is partitioned into the following multiple layers:

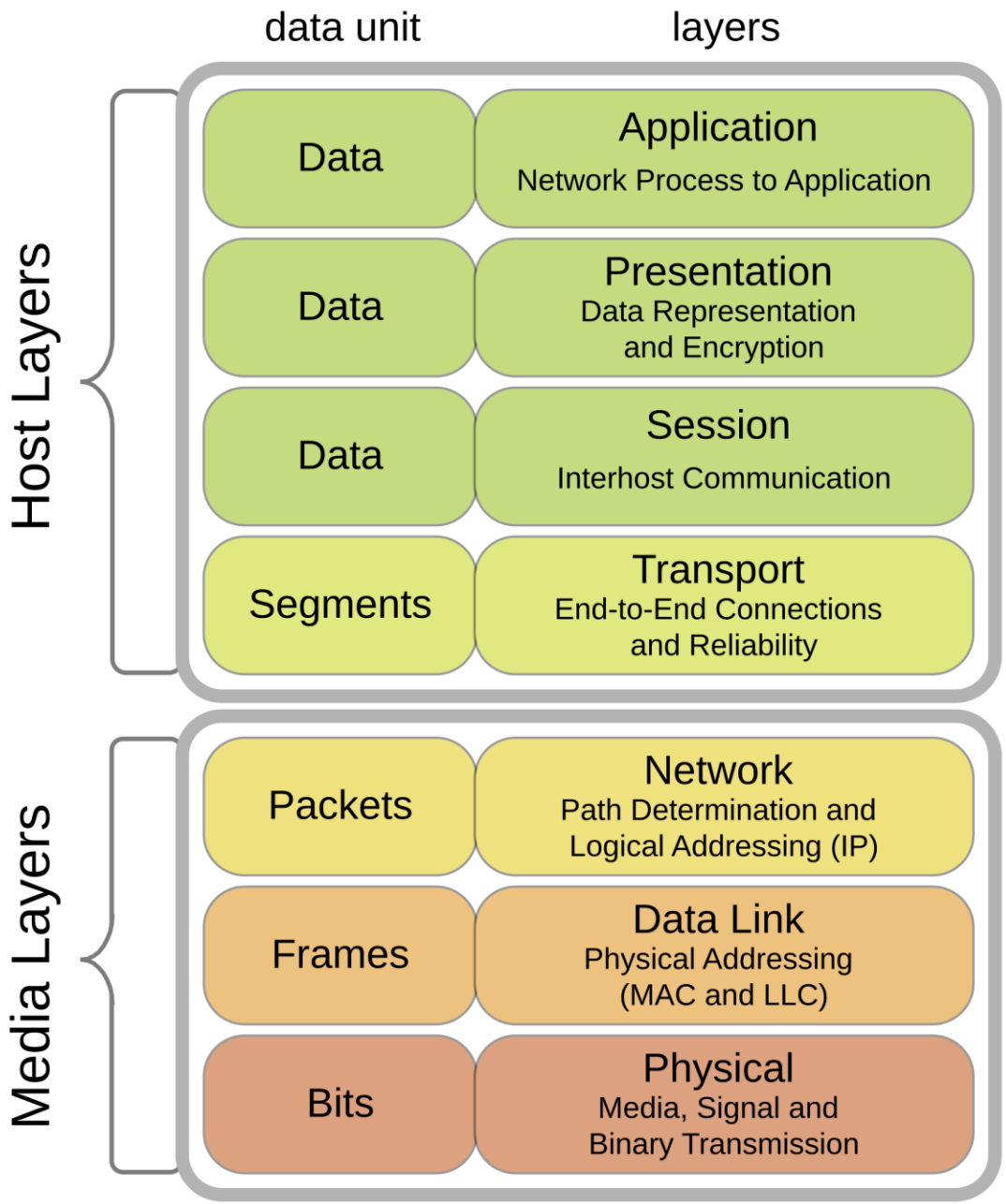
- **Layer 1: Physical layer** – handles the mechanical and electrical details of the physical transmission of a bit stream
- **Layer 2: Data-link layer** – handles the *frames*, or fixed-length parts of packets, including any error detection and recovery that occurred in the physical layer
- **Layer 3: Network layer** – provides connections and routes packets in the communication network, including handling the address of outgoing packets, decoding the address of incoming packets, and maintaining routing information for proper response to changing load levels

Communication Protocol (Cont.)

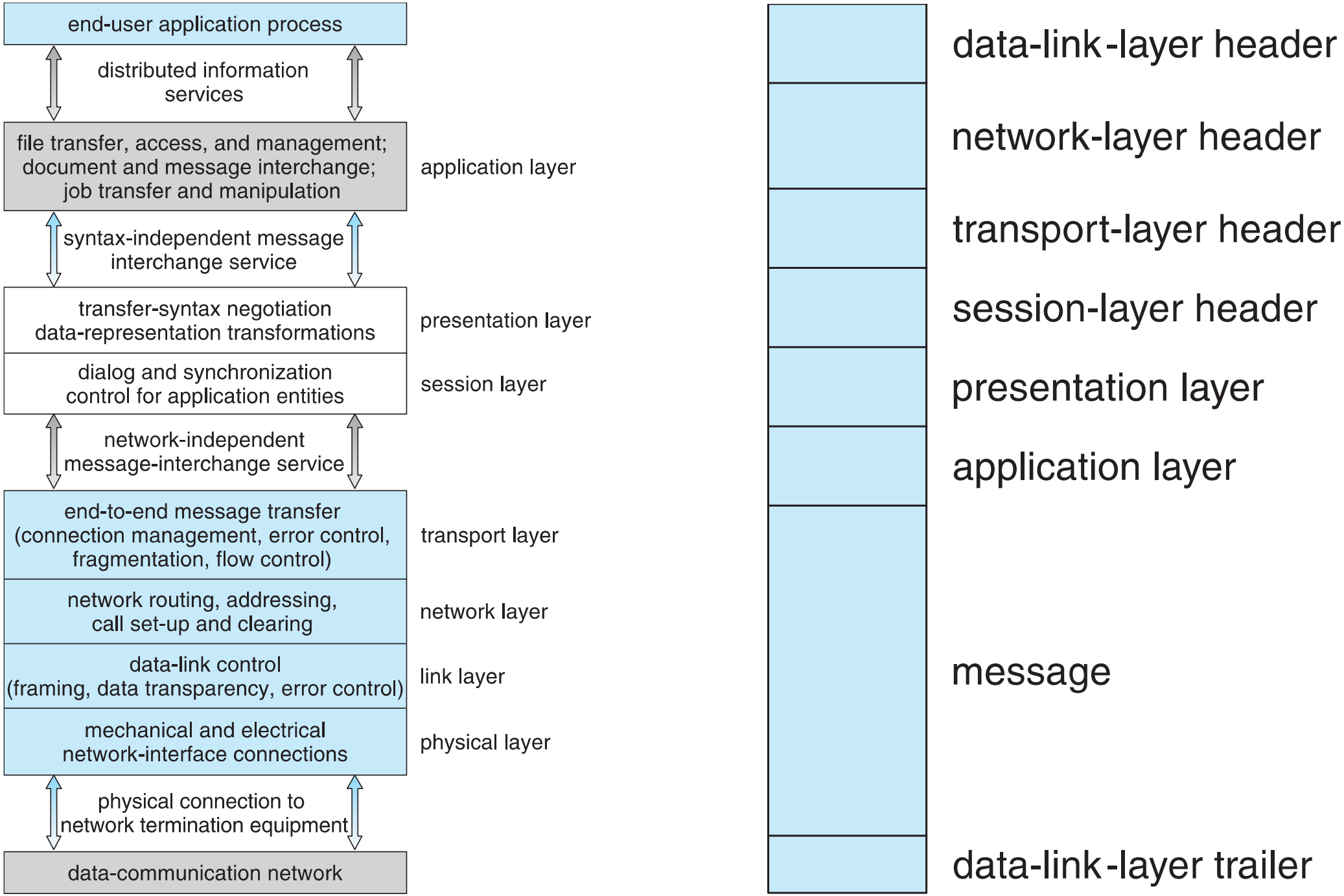
- **Layer 4: Transport layer** – responsible for low-level network access and for message transfer between clients, including partitioning messages into packets, maintaining packet order, controlling flow, and generating physical addresses
- **Layer 5: Session layer** – implements sessions, or process-to-process communications protocols
- **Layer 6: Presentation layer** – resolves the differences in formats among the various sites in the network, including character conversions, and half duplex/full duplex (echoing)
- **Layer 7: Application layer** – interacts directly with the users, deals with file transfer, remote-login protocols and electronic mail, as well as schemas for distributed databases

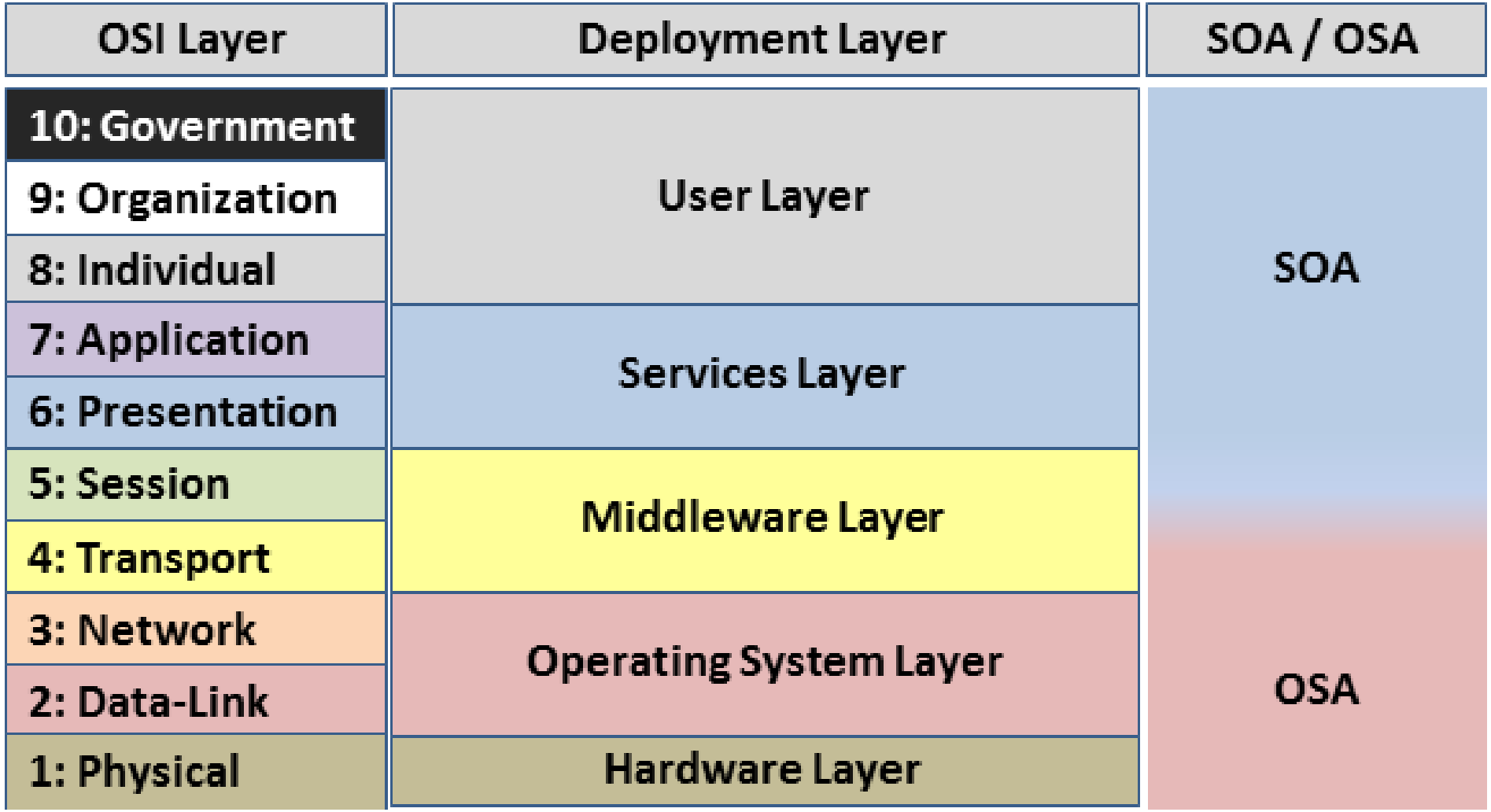
OSI Model





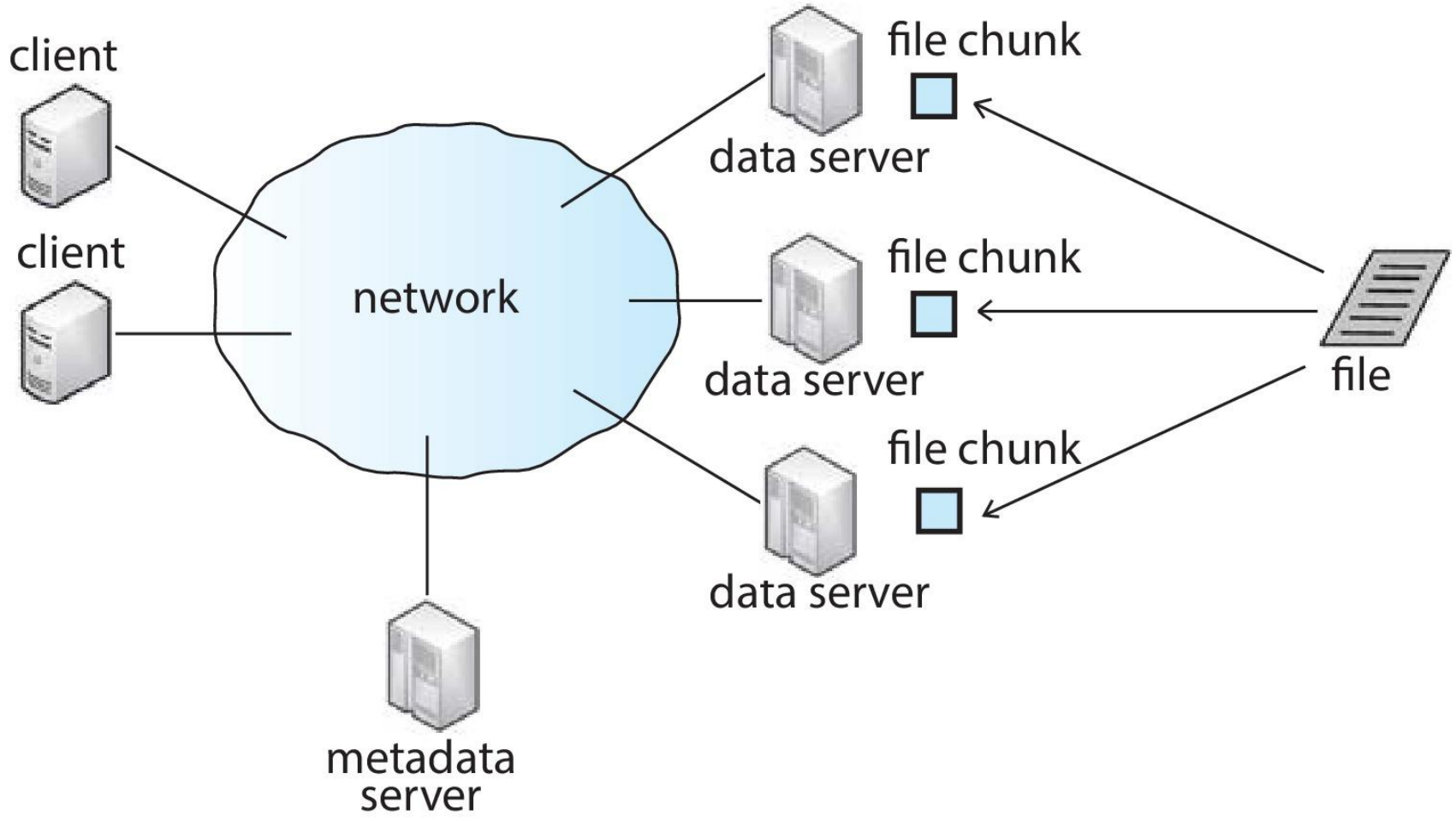
https://commons.wikimedia.org/wiki/File:OSI_Model_v1.svg


end-user application process
distributed information services
file transfer, access, and management; document and message interchange; job transfer and manipulation
application layer
syntax-independent message interchange service
transfer-syntax negotiation data-representation transformations
presentation layer
dialog and synchronization control for application entities
session layer
network-independent message-interchange service
end-to-end message transfer (connection management, error control, fragmentation, flow control)
transport layer
network routing, addressing, call set-up and clearing
network layer
data-link control (framing, data transparency, error control)
link layer
mechanical and electrical network-interface connections
physical layer
physical connection to network termination equipment
data-communication network
data-link-layer header
network-layer header
transport-layer header
session-layer header
presentation layer
application layer
message
data-link-layer trailer



https://commons.wikimedia.org/wiki/File:OSI_user_layers.png

Cluster-based DFS Model



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

- Online textbook Operating Systems: Three Easy Pieces (OSTEP)

- William Stallings. "Computer Organization and Architecture". 10th Edition

תודה רבה
Hebrew

Ευχαριστώ
Greek

Danke
German

Спасибо
Russian

धन्यवादः
Sanskrit

Merci
French

ধন্যবাদ
Bangla

நன்றி
Tamil

شكراً
Arabic

ಧನ್ಯವಾದಗಳು
Kannada

Thank You
English

നന്ദി
Malayalam

Grazie
Italian

多謝
Traditional Chinese

ధన్యవాదాలు
Telugu

આભાર
Gujarati

ਧੰਨਵਾਦ
Punjabi

धन्यवाद
Hindi & Marathi

Gracias
Spanish

多谢
Simplified Chinese

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

Obrigado
Portuguese

ありがとうございました
Japanese

ขอบคุณ
Thai

감사합니다
Korean