Process Concurrency on CPUs

Dr. Animesh Chaturvedi

Assistant Professor: IIIT Dharwad

Post Doctorate: King's College London & The Alan Turing Institute

PhD: IIT Indore MTech: IIITDM Jabalpur

CPU and Memory

CPU

PC   MAR

IR   MBR

Control
Unit

Memory

Address Bus **Data Bus** **Control Bus**

Fetch Cycle   **Execute Cycle**

CPU

PC   MAR

IR   MBR

Execution unit   I/O AR

          I/O BR

System Bus

Main Memory

```
0
1
2
```

Instruction
Instruction
Instruction

Data
Data
Data
Data

n − 2
n − 1

I/O Module

Buffers

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

START → Fetch Next Instruction → Execute Instruction → HALT

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

CPU and Memory

CPU

PC	MAR
IR	MBR
Execution unit	I/O AR
	I/O BR

Main Memory

```
0
1
2
   Instruction
   Instruction
   Instruction

   Data
   Data
   Data
   Data

n − 2
n − 1
```

System Bus

I/O Module

Buffers

Address Bus Data Bus Control Bus

t1 : MAR ⟵ PC
t2 : MBR ⟵ MEMORY
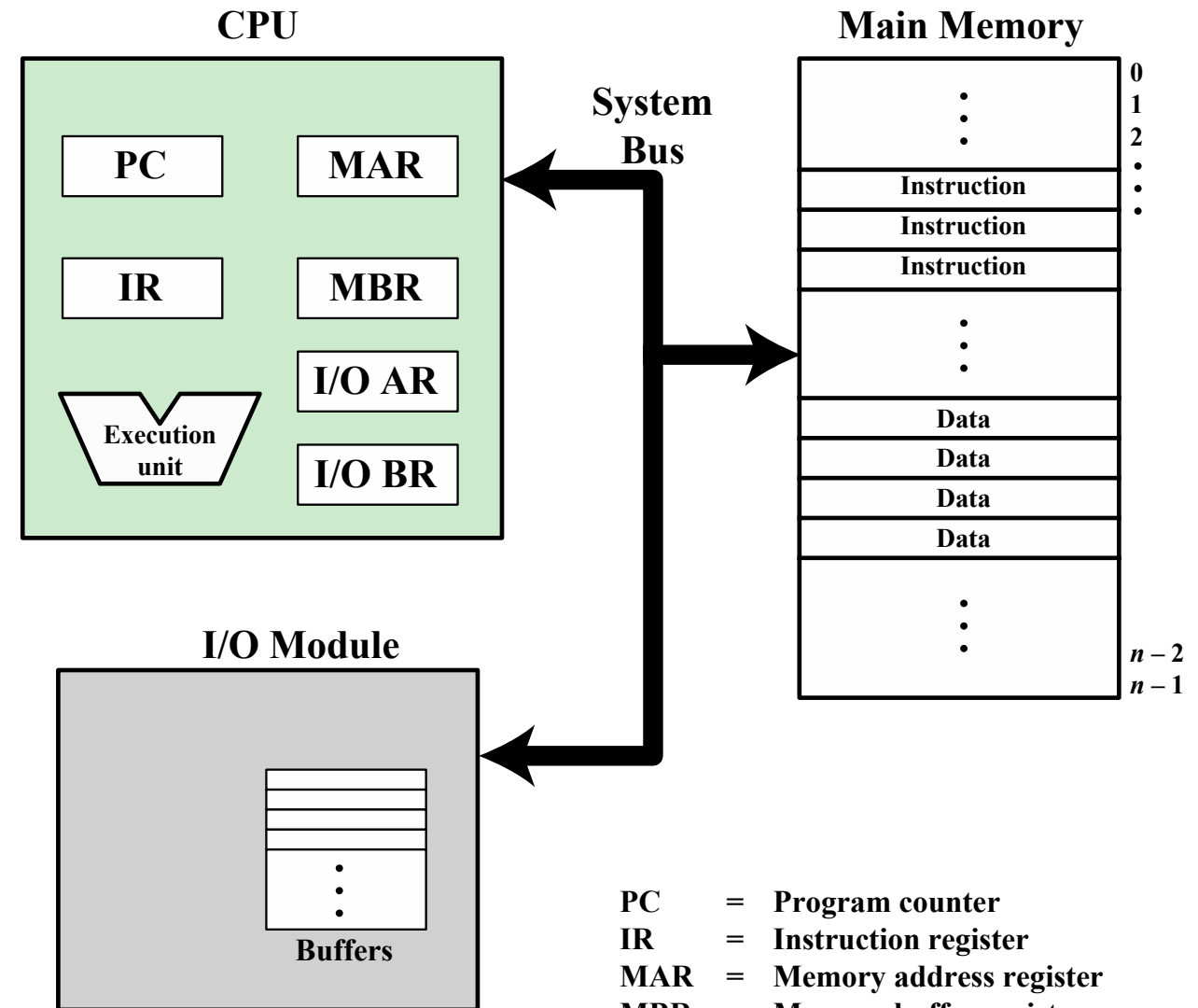   PC ⟵ (PC) + I
t3 : IR ⟵ (MBR)

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

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

Inter-process communication

Inter Process Communication (IPC)

Why IPC? Because
- Processes do not share any memory with each other
- Some processes might want to work together for a task, so need to communicate information

- IPC mechanisms to share information between processes.
- Sockets: Used to communicate between processes on same or different machines
  - Processes open sockets and connect them to each other
  - Messages written into one socket can be read from another
  - OS transfers data across socket buffers
- Pipe system call returns two file descriptors: write handle and read handle
 - Pipe data buffered in OS buffers between write and read handle

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

(a) Shared memory.
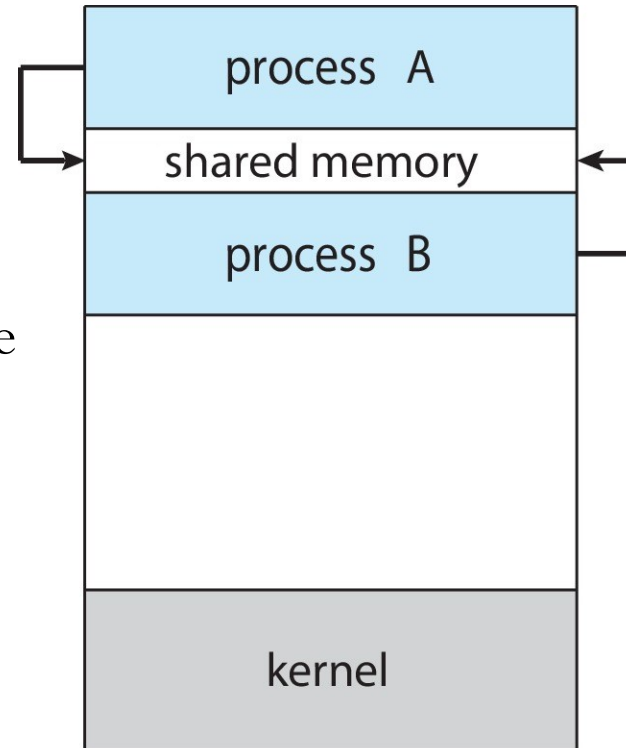
(b) Message passing.
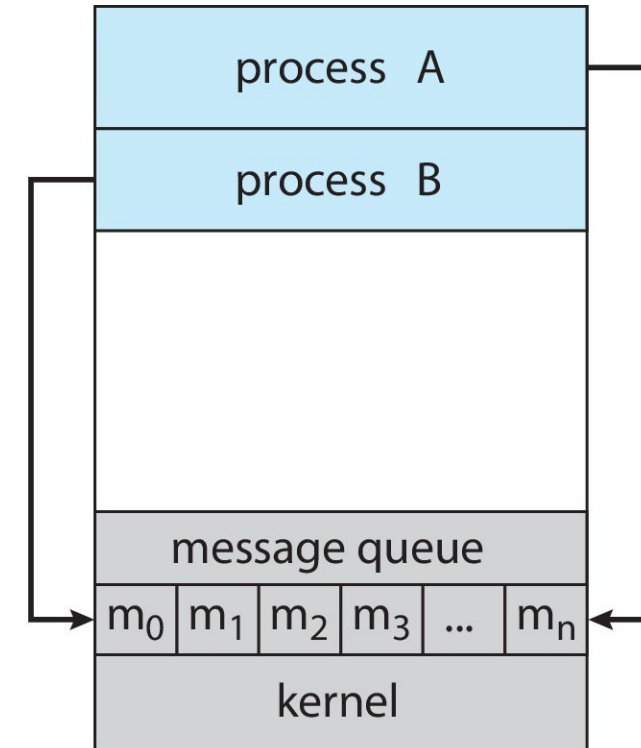
Concurrency vs. Parallelism

Concurrency vs. Parallelism

- **Multicore** or **multiprocessor** systems puts pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- *Parallelism* implies a system can perform more than one task simultaneously
- *Concurrency* supports more than one task making progress
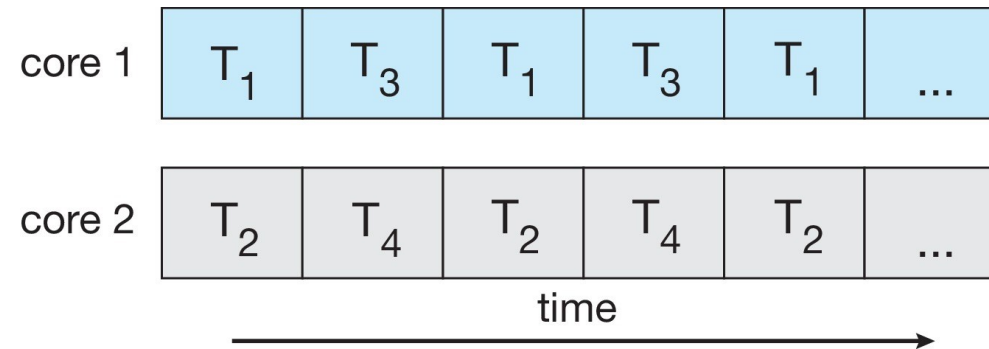 - Single processor / core, scheduler providing concurrency

Concurrency vs. Parallelism

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

single core	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4	T_1	...

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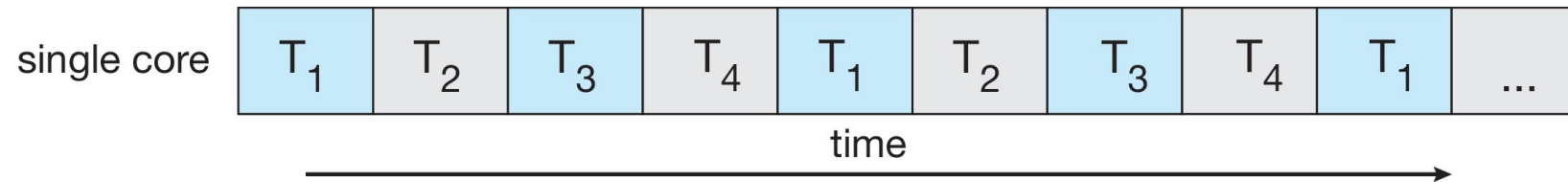
- **Parallelism on a multi-core system:**

core 1	T_1	T_3	T_1	T_3	T_1	...

| core 2	T_2	T_4	T_2	T_4	T_2	...

time →

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

data

data parallelism

| core 0 | core 1 | core 2 | core 3 |

data

task parallelism

| core 0 | core 1 | core 2 | core 3 |
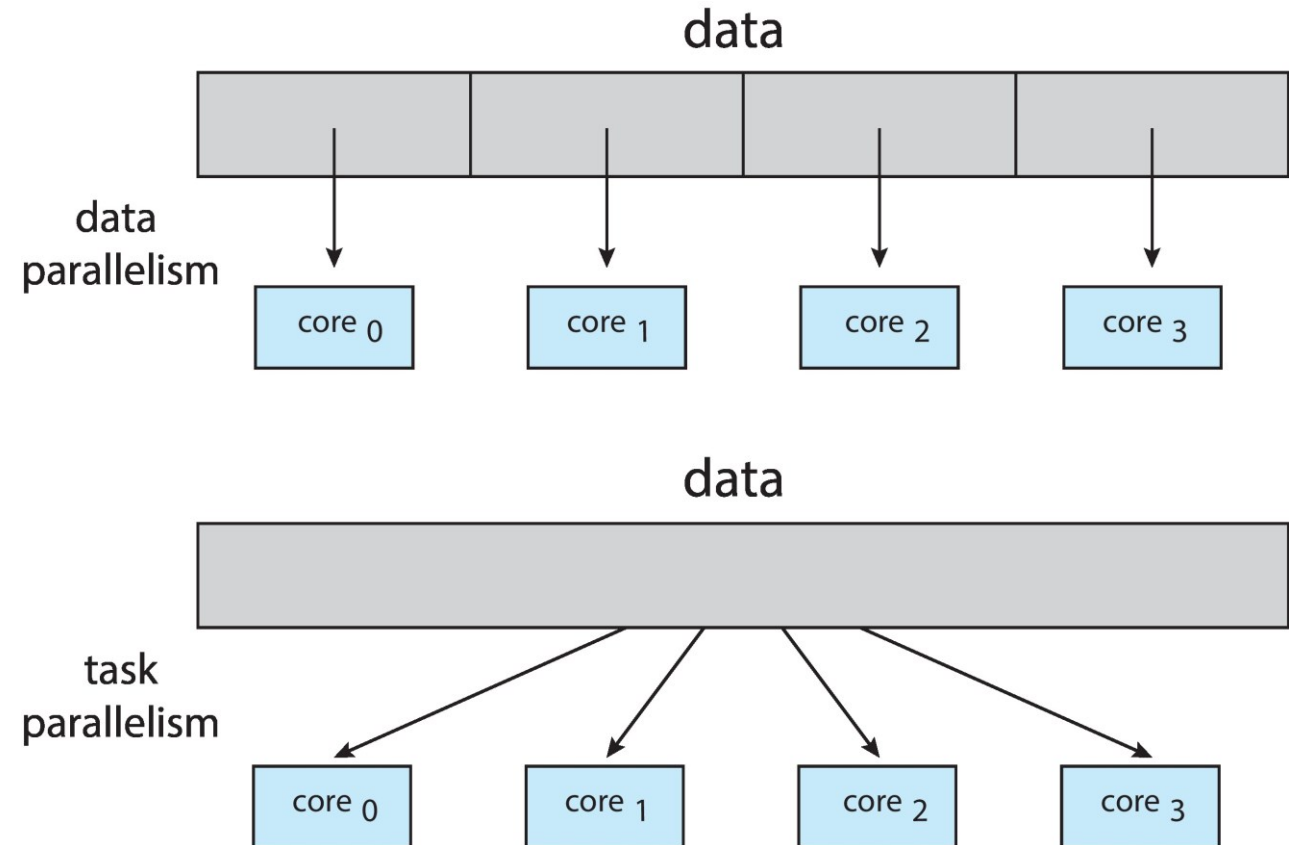
Parallelism using Fork-Join

- Multiple threads of tasks are **forked**, and then **joined**.
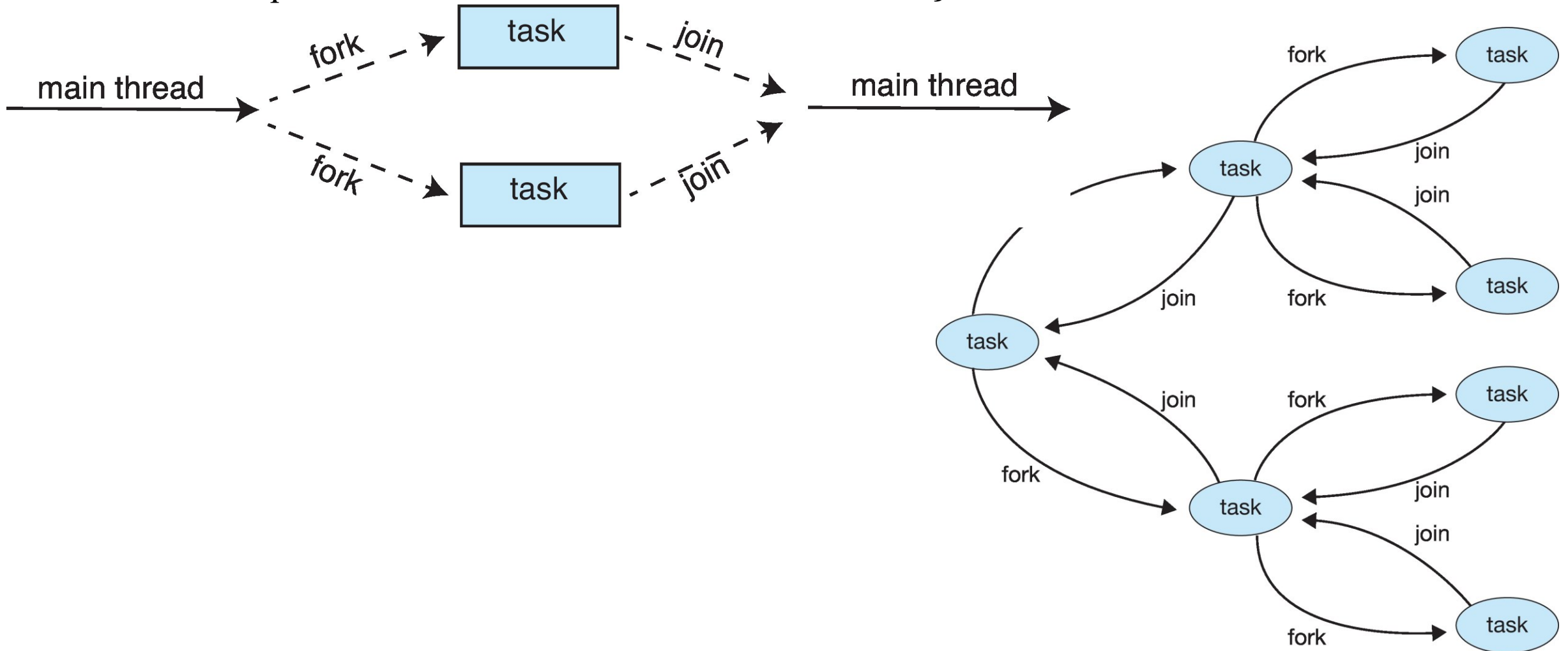
# Parallelism in Client-Server

Multithreaded Server Architecture

- **Responsiveness –** may allow continued execution if part of process is blocked
- **Resource Sharing –** threads share resources of process,
- **Economy –** thread creation is cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multicore architectures
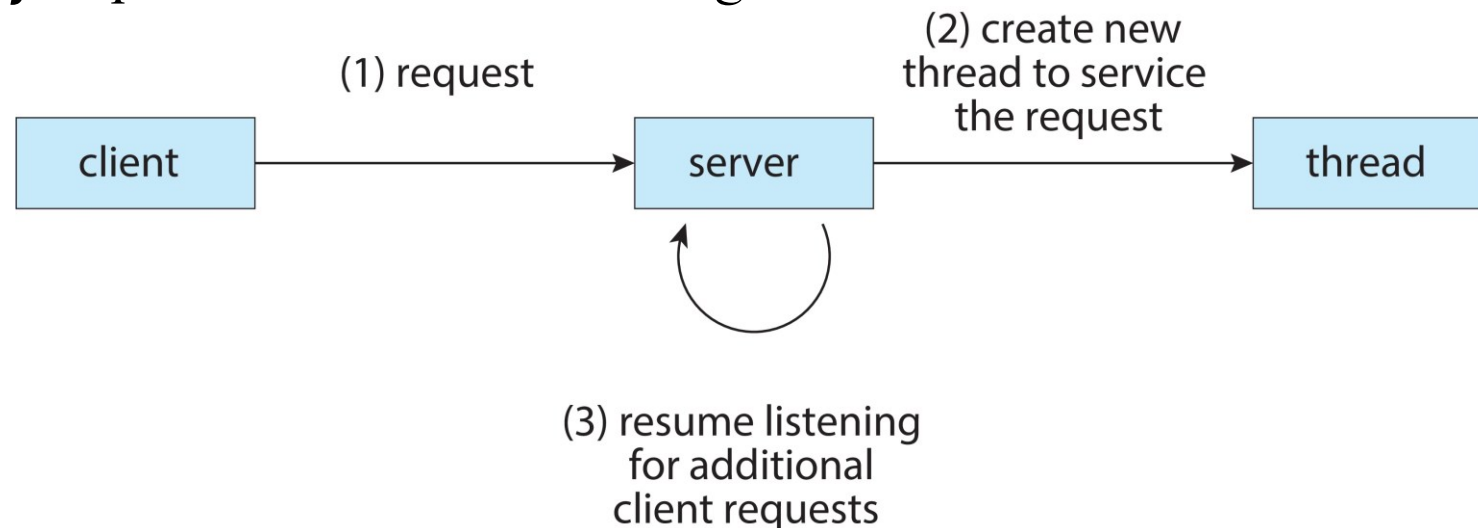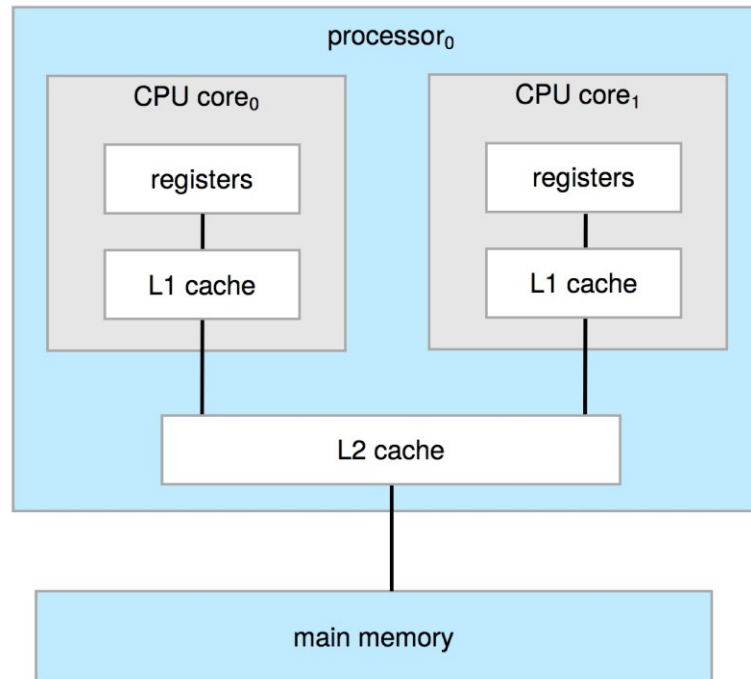
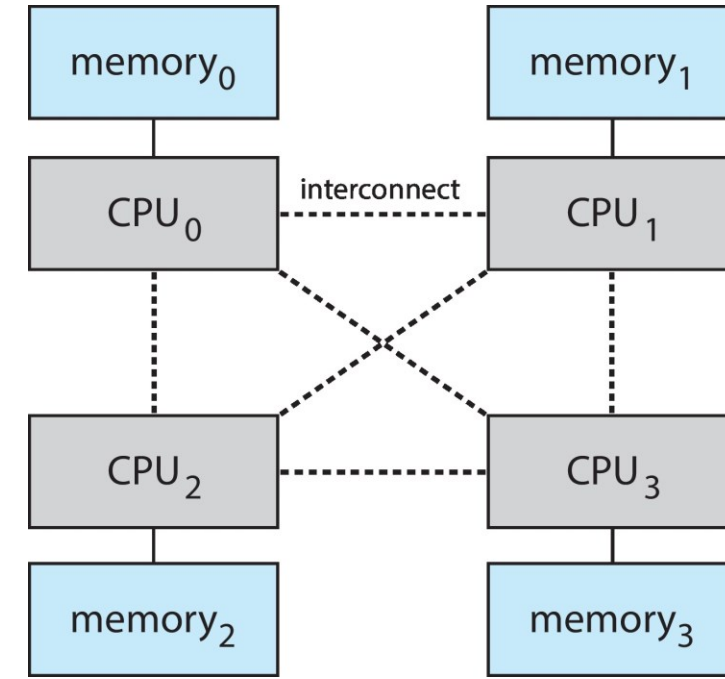# Multiprocessing, Threads, Multithreading

Multiprocessing

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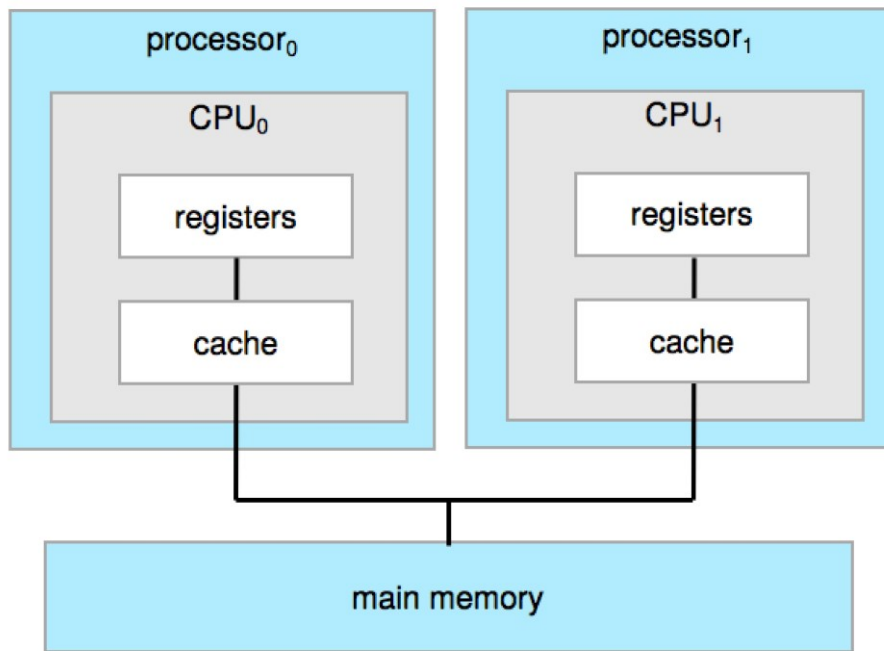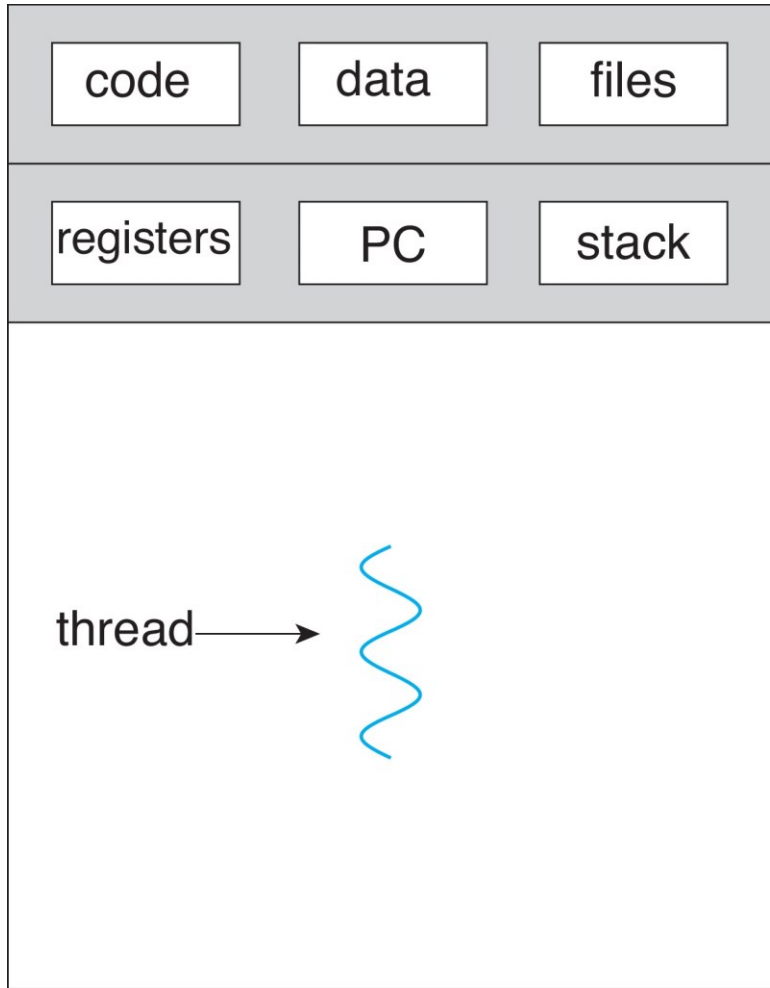

Symmetric Multiprocessing Architecture

Dual-Core Design

Non-Uniform Memory Access System

Multiprocessing

single-threaded process   multithreaded process
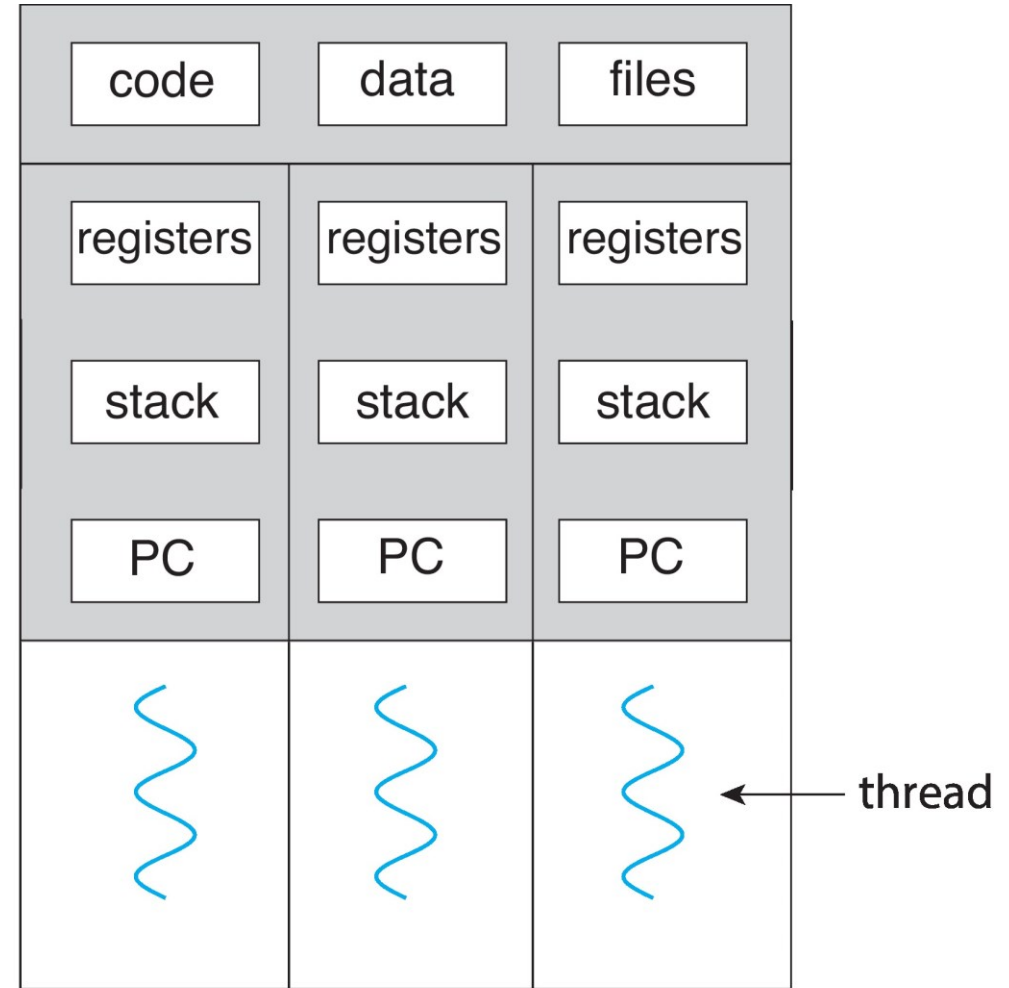
Why Multi threaded process?

- Parallelism: a single process can effectively utilize multiple CPU cores
  - Understand the difference between concurrency and parallelism
  - Concurrency: running multiple threads/processes at the same time, even on single CPU core, by interleaving their executions
  - Parallelism: running multiple threads/processes in parallel over different CPU cores
- Even if no parallelism, concurrency of threads ensures effective use of CPU when one of the threads blocks (e.g., for I/O)

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
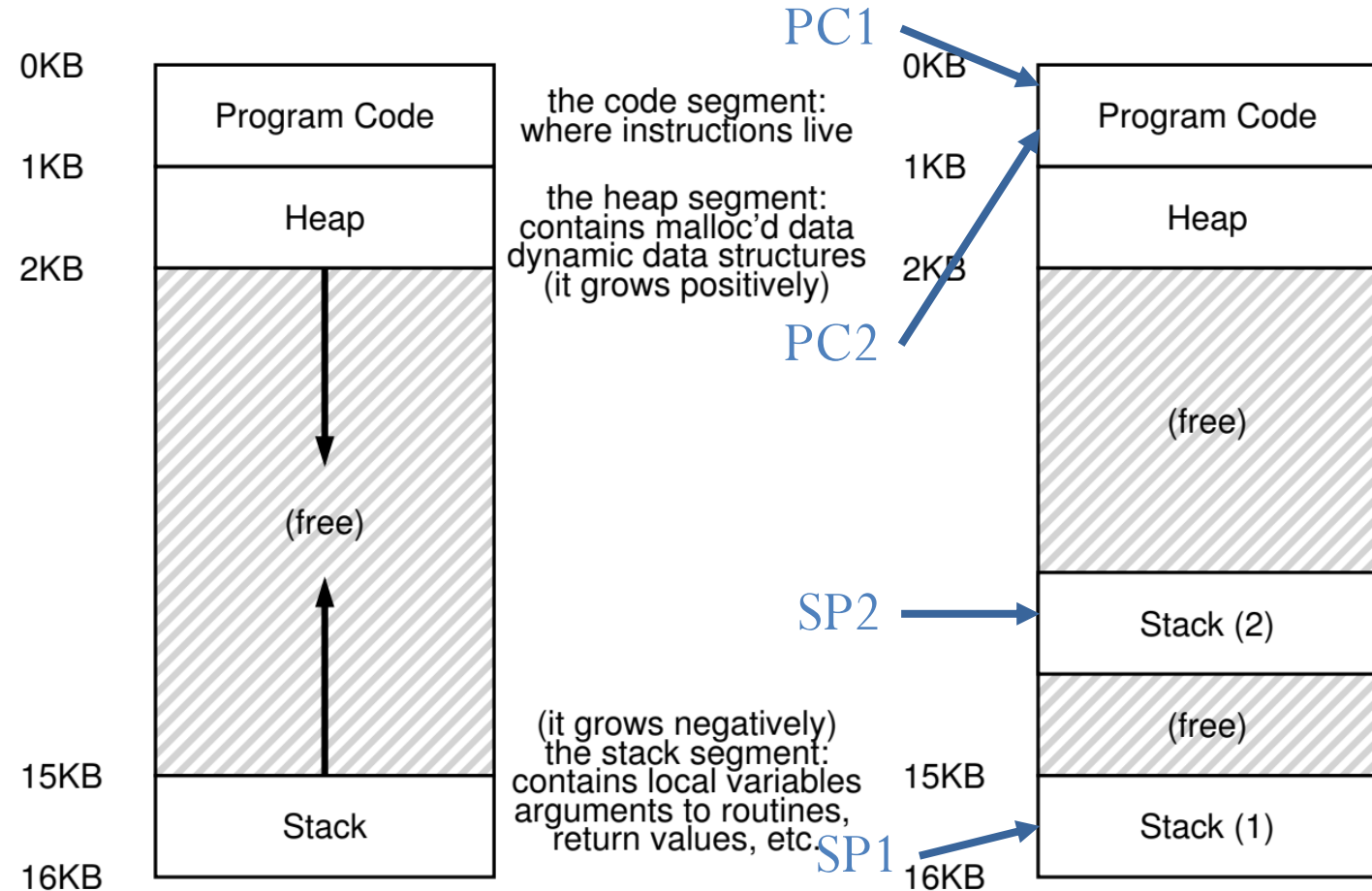
Process vs. threads

- Parent P forks a child C
  - P and C do not share any memory
  - Extra copies of code, data in memory
  - Need complicated IPC mechanisms to communicate
- Parent P executes two threads T1 and T2
  - T1 and T2 share parts of the address space
  - Global variables can be used for communication
  - Smaller memory footprint (amount of main memory that a program uses while running)
- Threads are like separate processes,
  - except they share the same address space

Scheduling threads

- OS schedules threads that are ready to run independently, much like processes
- Thread Control Block (TCB)
  - Context of a thread (PC, registers) is saved into/restored from TCB
  - Every PCB has one or more linked TCBs
- Threads that are scheduled independently by kernel are called kernel threads
  - E.g., Linux pthreads are kernel threads
- Libraries provide user-level threads
  - User program sees multiple threads
  - Library multiplexes larger number of user threads over a smaller number of kernel threads
  - Low overhead of switching between user threads (no expensive context switch)
  - Multiple user threads cannot run in parallel

Threads: User and Kernel

- Many-to-One
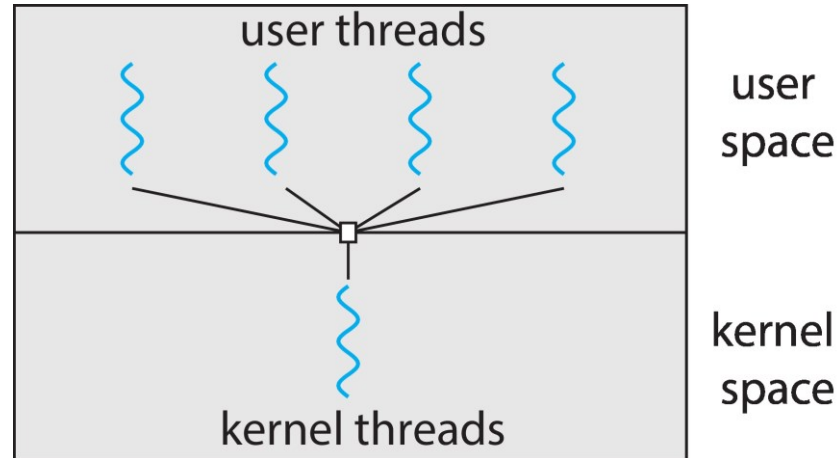
- One-to-One
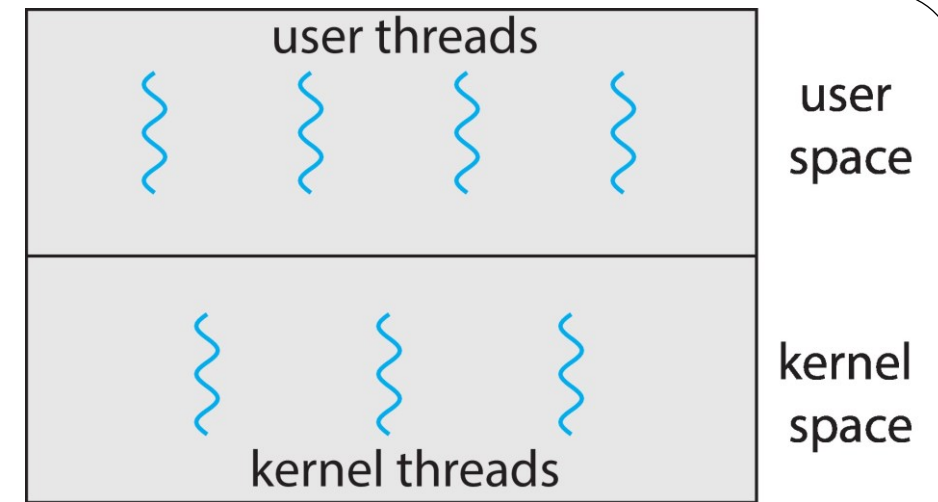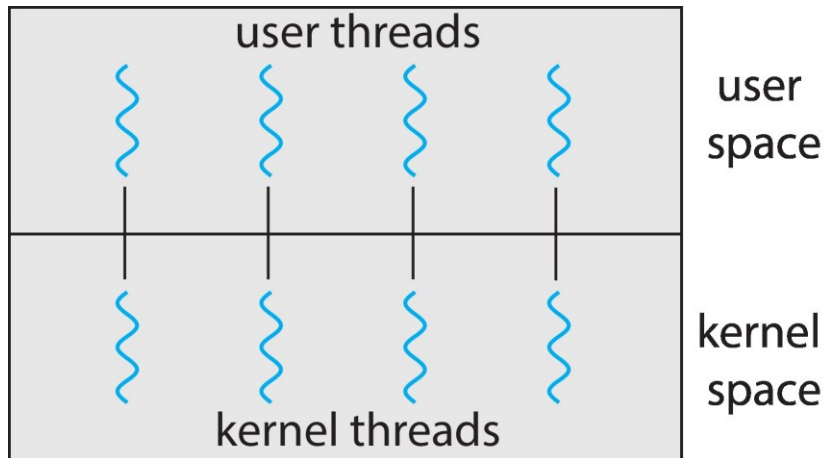
- Many-to-Many
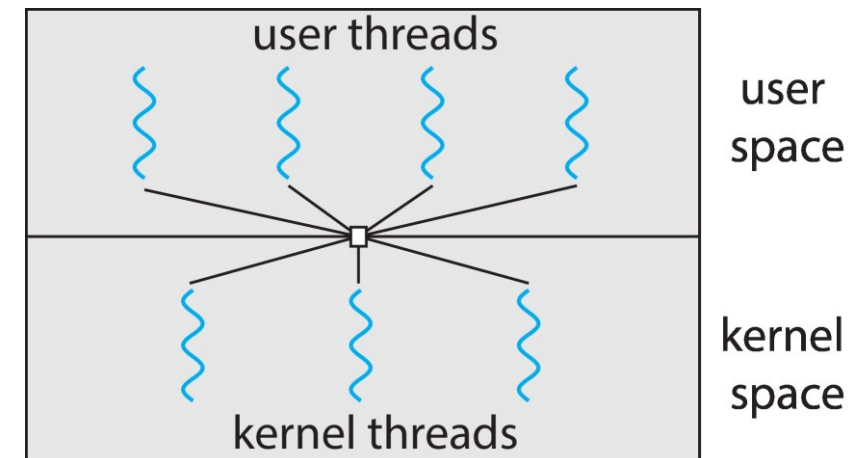
Multithreaded Multicore System

- **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.

processor

core 0	core 1
hardware thread	hardware thread
hardware thread	hardware thread

core 2	core 3
hardware thread	hardware thread
hardware thread	hardware thread

operating system view

CPU_0	CPU_1	CPU_2	CPU_3
CPU_4	CPU_5	CPU_6	CPU_7
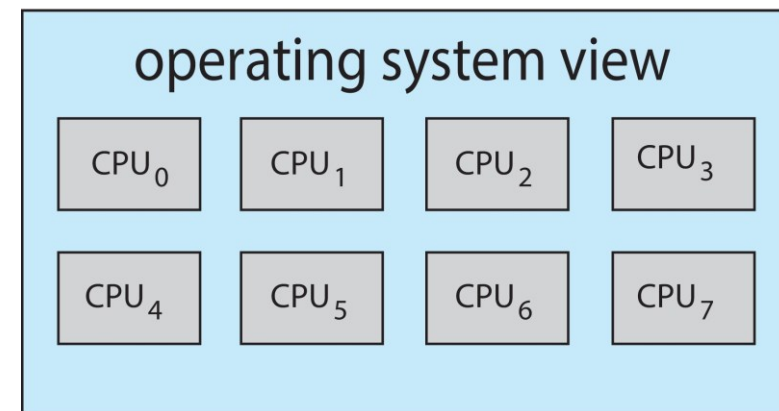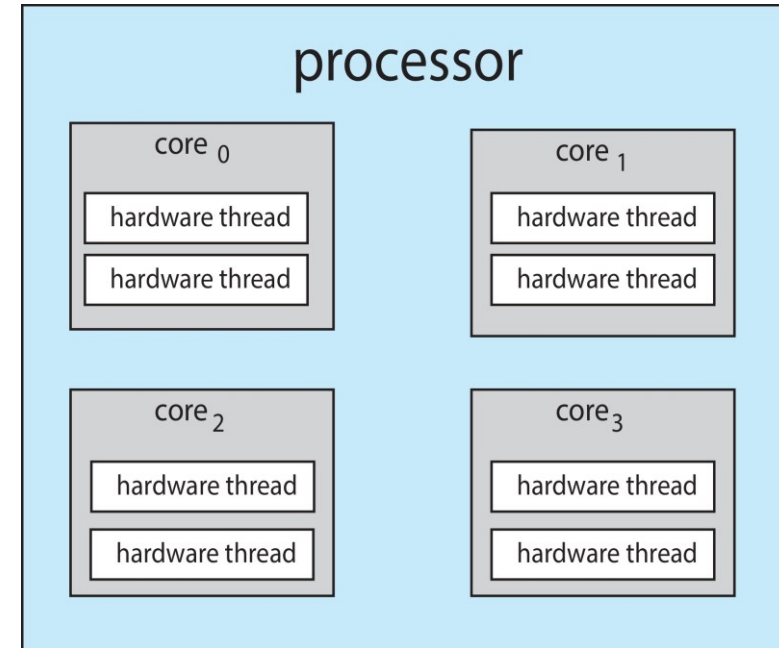
Multithreaded Multicore System

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.

level 1

level 2

software threads

hardware threads (logical processors)

processing core
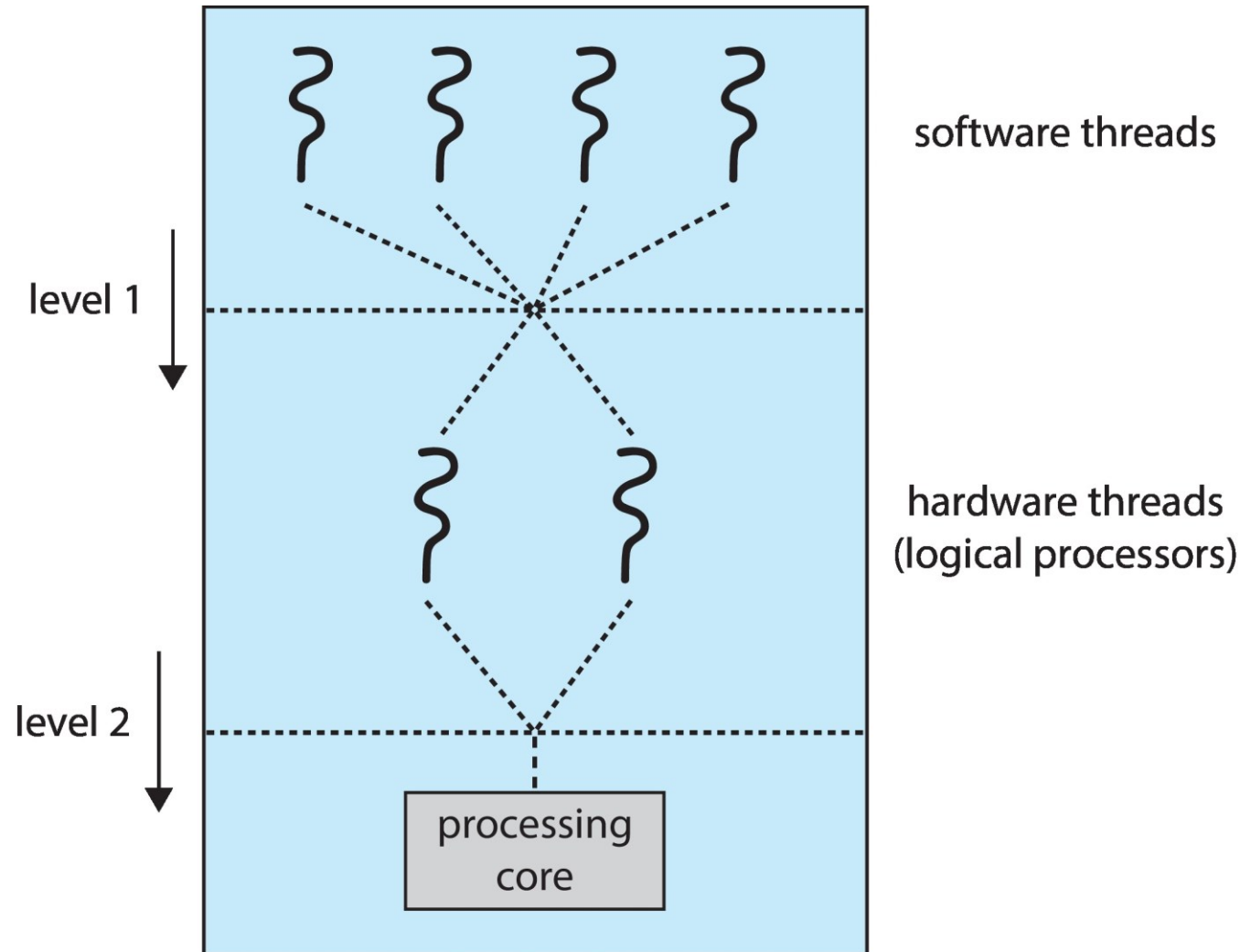
Amdahl's Law
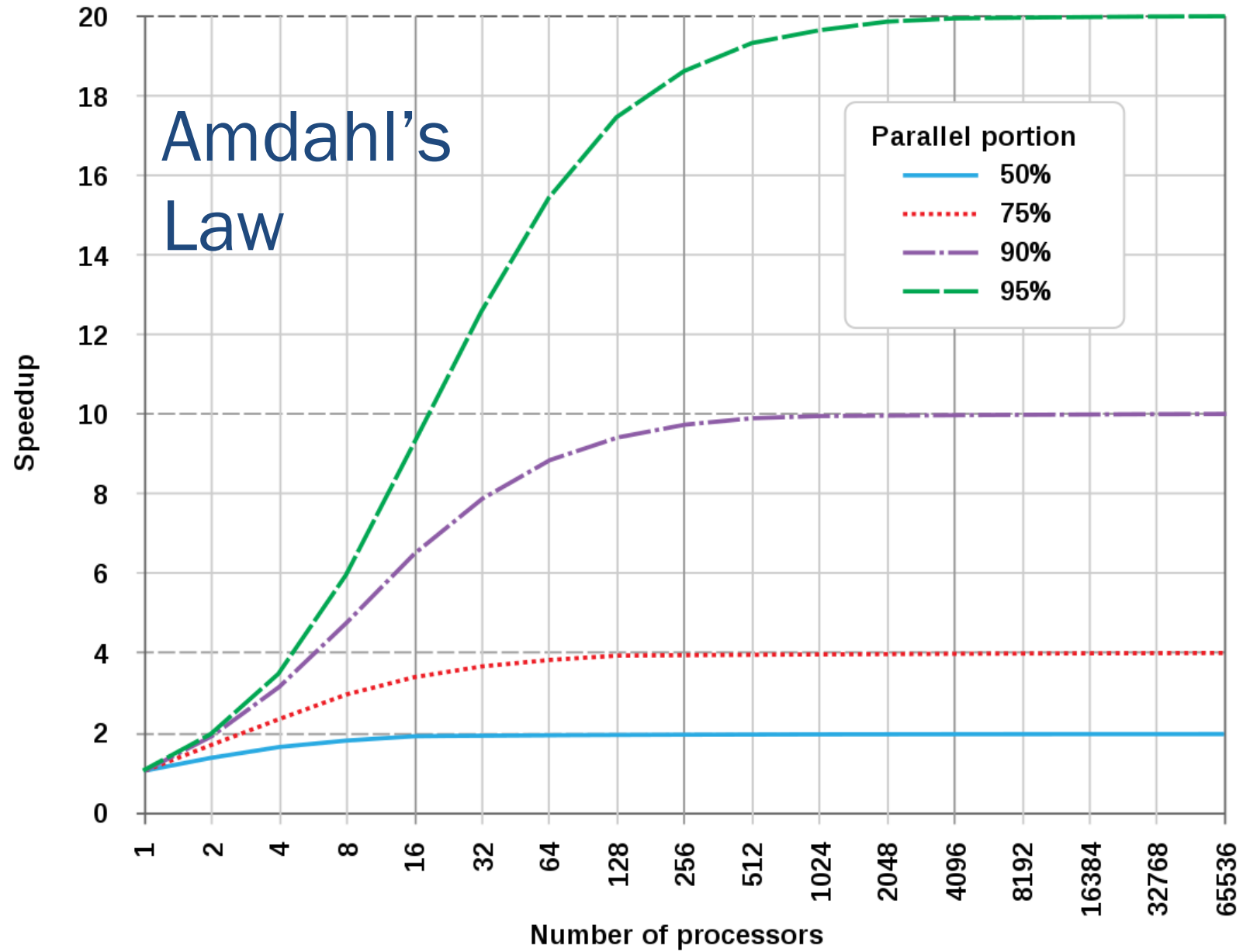
Race conditions, Synchronization, and Critical section

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

```
while (true) {

    entry section

    critical section

    exit section

    remainder section

}
```

- 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**

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

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

Building a lock

- Goals of a lock implementation
  - Mutual exclusion (obviously!)
  - Fairness: all threads should eventually get the lock, and no thread should starve
  - Low overhead: acquiring, releasing, and waiting for lock should not consume too many resources
- Implementation of locks are needed for both userspace programs (e.g., pthreads library) and kernel code
- Implementing locks needs support from hardware and OS

How should locks be used?

- A lock should be acquired before accessing any variable or data structure that is shared between multiple threads of a process – "Thread-safe" data structures
- All shared kernel data structures must also be accessed only after locking
- Coarse-grained vs. fine-grained locking:
  - one big lock for all shared data vs. separate locks
  - Fine-grained allows more parallelism
  - Multiple fine-grained locks may be harder to manage
- OS only provides locks, correct locking discipline is left to the user

Failed lock

- Because "No Mutual Exclusion"
- Lock: spin on a flag variable until it is unset, then set it to acquire lock
- Unlock: unset flag variable
- Thread 1 spins, lock is released, ends spin
- Thread 1 interrupted just before setting flag
- Race condition has moved to the lock acquisition code!

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)   // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;           // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Thread 1	Thread 2
call lock()	
while (flag == 1)	
interrupt: switch to Thread 2	
	call lock()
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Solution is Atomic hardware code

- Very hard to ensure atomicity of flag in software

- Solution of Mutual Exclusion: Hardware atomic instructions

- Modern architectures provide hardware atomic instructions

- Example of an atomic instruction functions/procedures

  These lock is called a spinlock – spins until lock is acquired

  1. **test-and-set()**

  2. **compare-and-swap()**

  3. Update a **Condition variable** and return old value, all in one hardware instruction

Simple lock using test-and-set

- If TestAndSet(flag, 1) returns 1, it means the lock is held by someone else, so wait busily

```
1   int TestAndSet(int *old_ptr, int new) {
2       int old = *old_ptr; // fetch old value at old_ptr
3       *old_ptr = new;     // store 'new' into old_ptr
4       return old;         // return the old value
5   }
```

spinlock

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Spinlock using compare-and-swap

- Atomic instruction: compare-and-swap

```
1    int CompareAndSwap(int *ptr, int expected, int new) {
2        int original = *ptr;
3        if (original == expected)
4            *ptr = new;
5        return original;
6    }
```

spinlock

- Spinlock using compare-and-swap

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

Alternative to spinning

- Alternative to spinlock: a (sleeping) mutex
- Instead of spinning for a lock, a contending thread could simply give up the CPU and check back later
  - **yield()** moves thread from running to ready state

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Spinlock vs. sleeping mutex

- Most userspace lock implementations are of the sleeping mutex kind
  - CPU wasted by spinning contending threads
  - More so if a thread holds spinlock and blocks for long
- Locks inside the OS are always spinlocks
  - Why? Who will the OS yield to?
- When OS acquires a spinlock:
  - It must disable interrupts (on that processor core) while the lock is held. Why? An interrupt handler could request the same lock, and spin for it forever.
  - It must not perform any blocking operation – never go to sleep with a locked spinlock!
- In general, use spinlocks with care, and release as soon as possible

Condition variables

Condition variables

- Locks allow one type of synchronization between threads – mutual exclusion
- Another common requirement in multi-threaded applications
  - waiting and signaling
  - E.g., Thread T1 wants to continue only after T2 has finished some task
- Can accomplish this by busy-waiting on some variable, but inefficient
- Need a new synchronization primitive: **condition variables**

Condition Variable

- A condition variable (CV) is a queue that a thread can put itself into when waiting on some condition

- Another thread that makes the condition true can signal the CV to wake up a waiting thread

- Pthreads provides CV for user programs
  - OS has a similar functionality of wait/signal for kernel threads

- Signal wakes up one thread, signal broadcast wakes up all waiting threads

Producer/Consumer problem

- A common pattern in multi-threaded programs

- Example: in a multi-threaded web server, one thread accepts requests from the network and puts them in a queue.

- Worker threads get requests from this queue and process them.

- Setup: one or more producer threads, one or more consumer threads, a shared buffer of bounded size

Semaphores

Semaphores

- Synchronization primitive like condition variables
- Semaphore is a variable with an underlying counter
- Two functions on a semaphore variable
  - Up/post increments the counter
  - Down/wait decrements the counter and blocks the calling thread if the resulting value is negative
- A semaphore with init value 1 acts as a simple lock (binary semaphore = mutex)

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
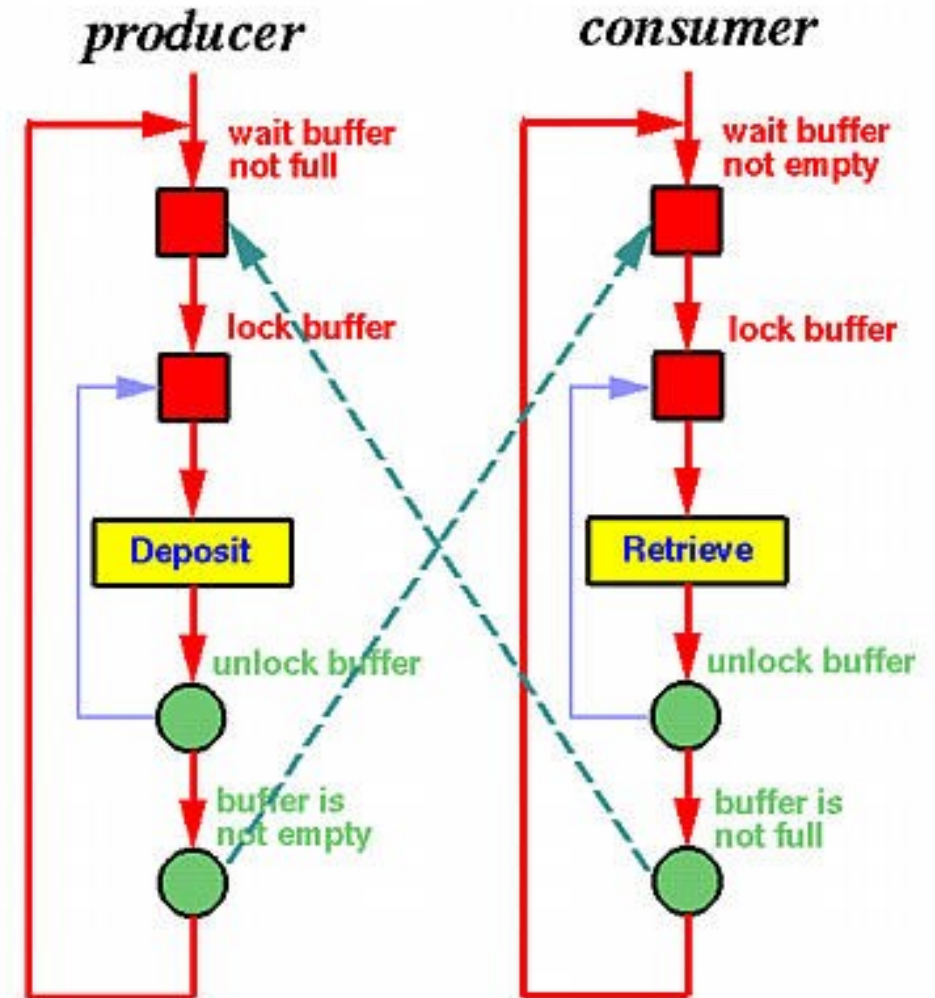
Semaphore for Producer/Consumer

- 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 and Deadlock

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);
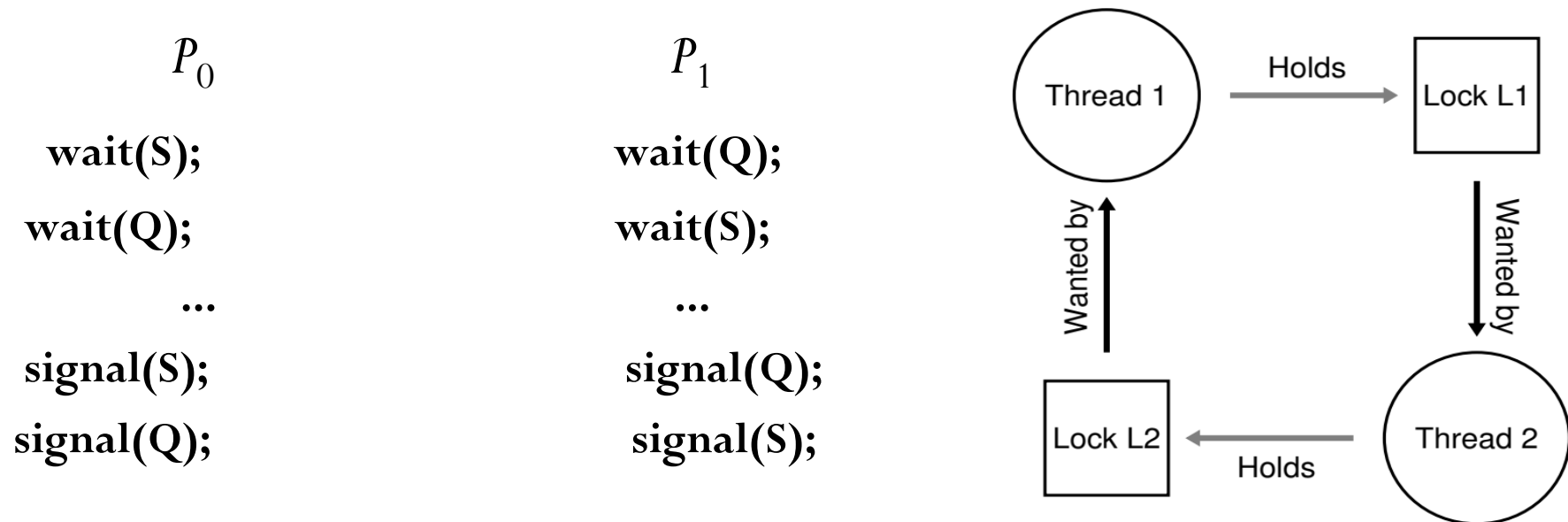
Conditions for Deadlock

- Mutual exclusion: a thread claims exclusive control of a resource (e.g., lock)
- Hold-and-wait: thread holds a resource and is waiting for another
- No preemption: thread cannot be made to give up its resource (e.g., cannot take back a lock)
- Circular wait: there exists a cycle in the resource dependency graph
- ALL four of the above conditions must hold for a deadlock to occur
- Build systems to prevent, avoid, or at least detect and recover from deadlock

Deadlock solution (Resource Allocation Graph)

- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $T = \{T_1, T_2, \ldots, T_n\}$, the set consisting of all the threads in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
  - **request edge** – directed edge $T_i \rightarrow R_j$
  - **assignment edge** – directed edge $R_j \rightarrow T_i$
- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

# Deadlock (Resource Allocation Graph)
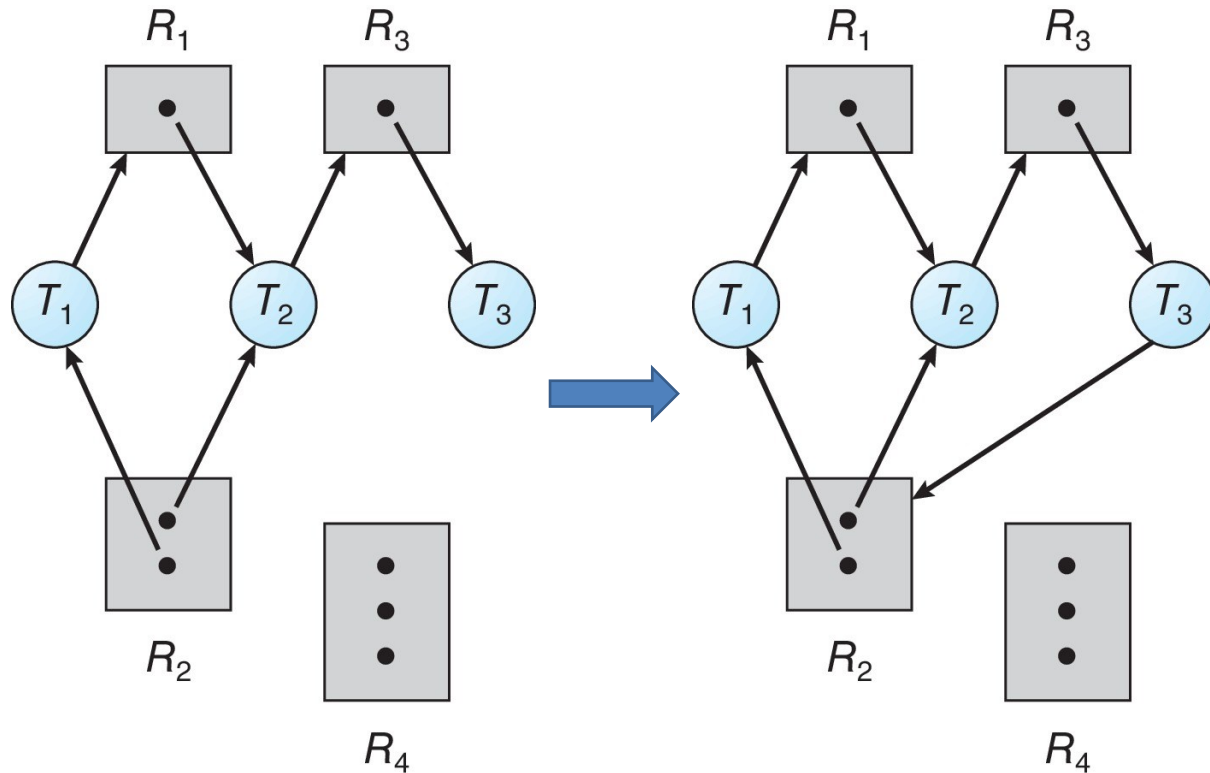
Deadlock
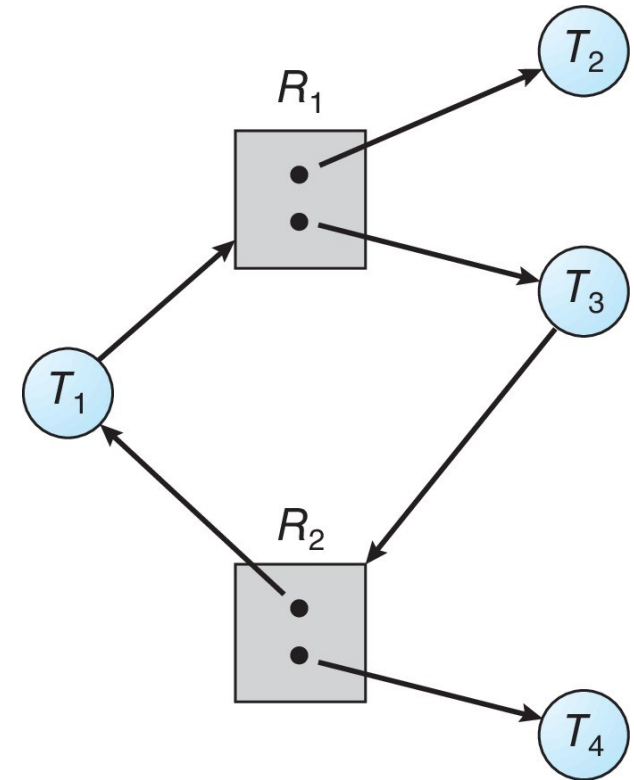
Graph with a Cycle But no Deadlock

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

3. Allow the system to enter a deadlock state and then recover

4. Ignore the problem and pretend that deadlocks never occur in the system.
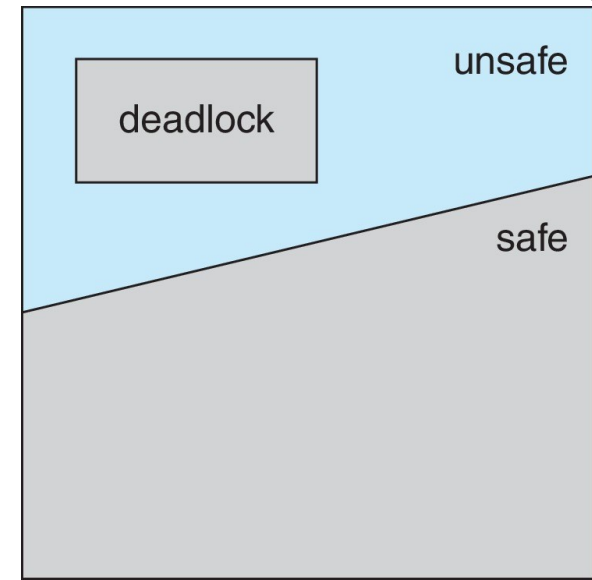
Deadlock avoidance

- If a system is in safe state $\Rightarrow$ no deadlocks

- If a system is in unsafe state $\Rightarrow$ possibility of deadlock

- Avoidance $\Rightarrow$ 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.
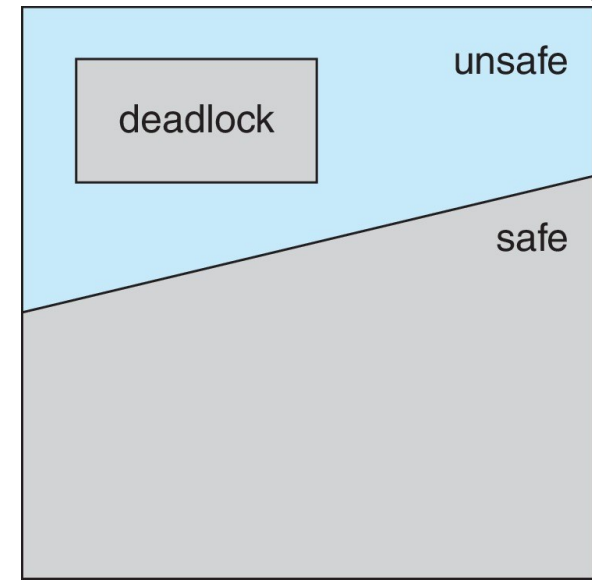
Concurrency bugs

Concurrency bugs

- Bugs are non-deterministic and occur based on execution order of threads
  - very hard to debug
- Two types of bugs
  - Deadlocks: threads cannot execute any further and wait for each other
  - Non-deadlock bugs: non deadlock but incorrect results when threads execute

Concurrency bugs

- Atomicity bugs – atomicity assumptions made by programmer are violated during execution of concurrent threads
  - Fix: locks for mutual exclusion
- Order-violation bugs – desired order of memory accesses is flipped during concurrent execution
  - Fix: condition variables
- Deadlock bugs: Mutual exclusion, Hold-and-wait, No preemption of the above conditions must hold for a deadlock to occur
  - Fix: Resource allocation graph

Advance Applications
Virtualizations, Services, and Distributed systems

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)
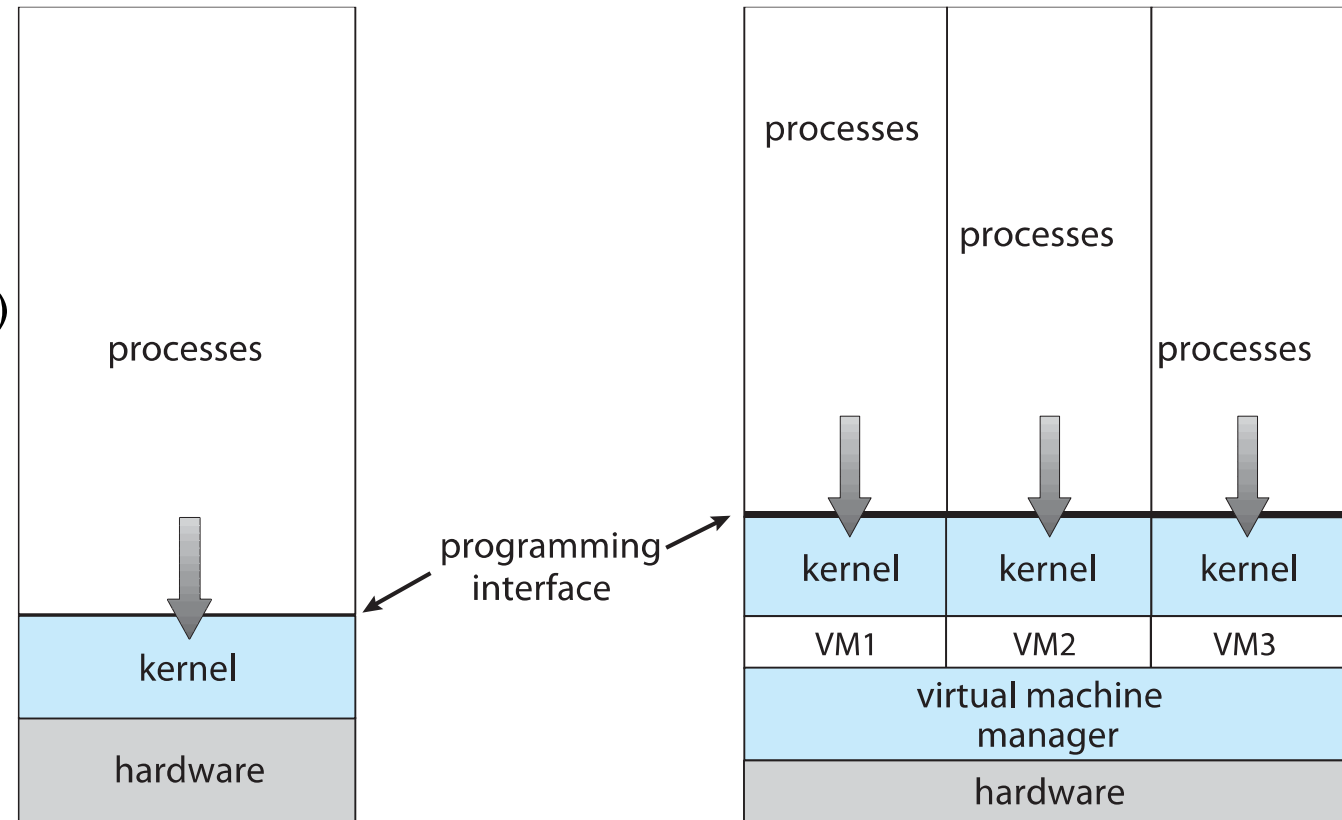
Distributed Systems for Cloud

- Collection of separate, possibly heterogeneous, systems networked together
  - **Network** is a communications path, **TCP/IP** most common
    - **Local Area Network** (**LAN**)
    - **Wide Area Network** (**WAN**)
    - **Metropolitan Area Network** (**MAN**)
    - **Personal Area Network** (**PAN**)
- **Distributed Operating System** provides features between systems across network
  - Communication scheme allows systems to exchange messages
  - Illusion of a single system

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

ขอบคุณ

Thai

Grazie

Italian

תודה רבה

Hebrew

धन्यवादः

Sanskrit

ধন্যবাদ

Bangla

ಧನ್ಯವಾದಗಳು

Kannada

Ευχαριστώ

Greek

Thank You

English

Спасибо

Russian

Gracias

Spanish

Obrigado

Portuguese

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

شكراً

Arabic

Merci

French

多謝

Traditional Chinese

धन्यवाद

Hindi

Danke

German

多谢

Simplified Chinese

நன்றி

Tamil

Tamil

ありがとうございました

Japanese

감사합니다

Korean