

Amdahl's Law

Amdahl's Law explains how much performance improvement can be achieved when more processing cores are added to run an application. It assumes that the application has two parts: a serial part that cannot be parallelized and a parallel part that can run simultaneously on multiple cores.

The formula for Amdahl's Law is:

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

Where:

- S is the fraction of the program that is serial (cannot be parallelized)
- N is the number of processing cores

For example, if 25% of the application is serial and 75% is parallel, increasing cores from 1 to 2 will result in a speedup of only 1.6 times. As the number of cores approaches infinity, the maximum speedup is limited to $1/S$ meaning the serial part limits the total performance gain.

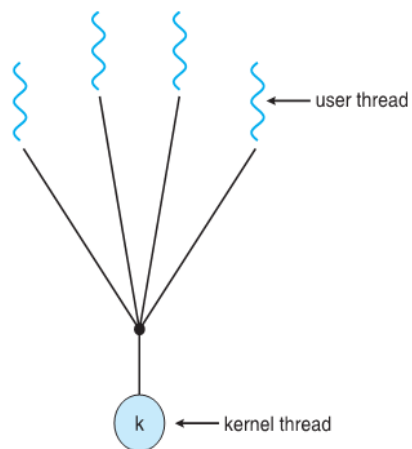
This law shows that even a small serial portion of a program greatly limits how much the performance can improve by adding more cores.

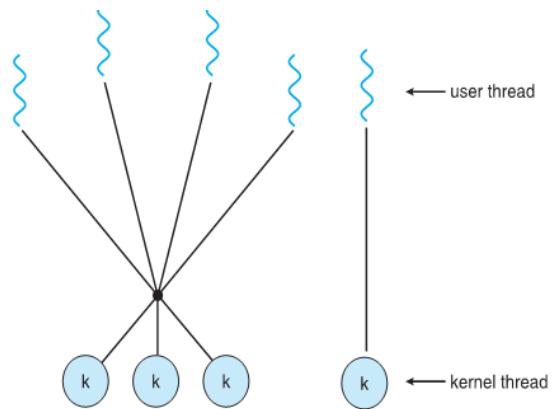
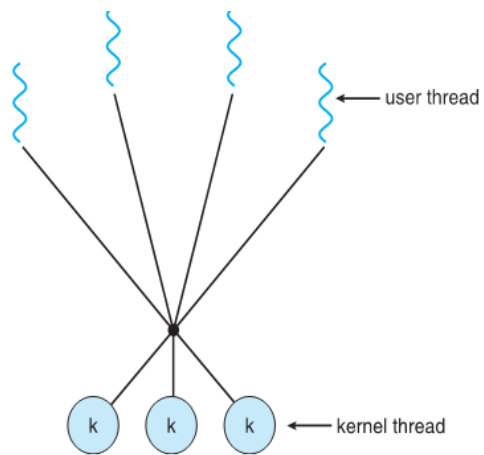
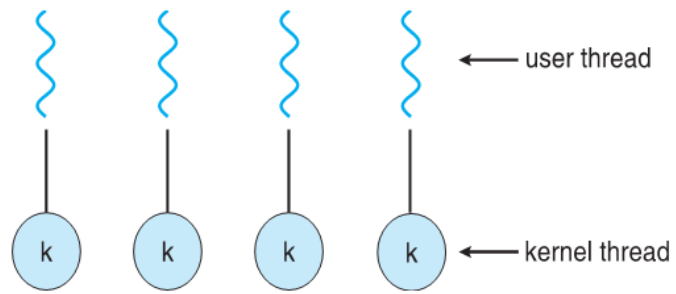
However, Amdahl's Law does not fully consider modern multicore systems, where parallelism is more complex and dynamic. New hardware designs and software optimizations can sometimes overcome these limits to some extent.

Compare concurrency and parallelism:

Point	Concurrency	Parallelism
1.	Doing many tasks at once , but not exactly at the same time	Doing many tasks exactly at the same time
2.	Tasks take turns using the same CPU	Tasks run on different CPUs/cores at the same time
3.	Looks like things are happening together	Things actually happen together
4.	Used when you have one processor	Used when you have multiple processors/cores
5.	Example: Switching quickly between chatting and browsing	Example: Watching a movie and downloading a file at the same time using different cores
6.	Focuses on managing multiple tasks	Focuses on speeding up tasks
7.	Like a single chef cooking many dishes by switching between them	Like multiple chefs each cooking one dish at the same time

COMPARISON OF MULTITHREADED MODELS:





Model	User Threads	Kernel Threads	Concurrency	Blocking Impact	Complexity	Performance	Parallelism	Examples
Many-to-One	Many	One	Low	One thread blocks all	Low	Poor due to blocking	No	Solaris Green Threads, GNU PT
One-to-One	One-to-One	One-to-One	High	One thread blocks itself	Medium	Good	Yes	Windows, Linux, Solaris 9+
Many-to-Many	Many	Many	High	Efficient thread blocking	High	Very Good	Yes	Solaris (before v9), Windows
Two-Level	Many	Many	High	Mix of bound/unbound thread blocking	High	Excellent	Yes	IRIX, HP-UX, Tru64 UNIX, Solaris 8

Lightweight Processes (LWP):

Definition:

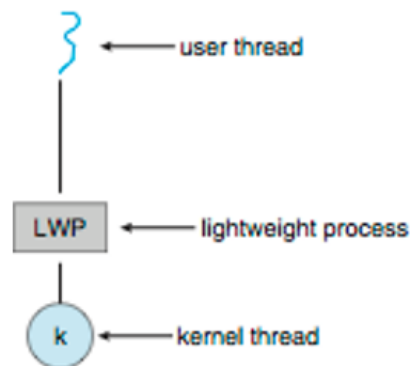
Lightweight Processes (LWPs) are special entities that act as a bridge between user-level threads and kernel-level threads. Each LWP is connected to exactly one kernel thread and allows user threads to be scheduled on the CPU.

Scheduler Activation:

- **Scheduler Activation** is a technique used in **many-to-many** or **two-level threading models**.
- It helps the **user-level thread library** and the **kernel** to **cooperate** in managing threads efficiently.
- It allows the system to support **concurrency** and **parallelism** while keeping thread management **efficient**.

How are LWPs linked to Scheduler Activations?

Link	Explanation	
1. Bridge Between User and Kernel Threads	LWPs act as a bridge between user threads and the kernel. Scheduler activations use LWPs to allow user threads to run on actual CPU cores.	
2. Kernel Creates LWPs	The kernel creates LWPs based on the number of CPUs and system load.	
3. User Threads Use LWPs	When a user-level thread wants to run, it uses an available LWP.	
4. Scheduler Activations Notify the User Library	When an LWP is blocked (e.g., due to I/O), the kernel notifies the user thread library via a scheduler activation , so it can schedule another user thread.	
5. Efficient Context Switching	This method ensures efficient use of CPU time , as the user library always knows the state of LWPs and can manage user threads better.	
6. Scalability	It scales well in multi-core systems since multiple LWPs can run in parallel, giving true parallelism for user threads.	



Thread Cancellation

- Thread cancellation means stopping a thread before it finishes its work.
- The thread you want to stop is called the **target thread**.
- There are two main ways to cancel a thread:
 1. **Asynchronous cancellation:** This stops the target thread immediately.

2. **Deferred cancellation:** This lets the target thread check regularly if it should stop or not, so it cancels itself safely.

- Example code to create and cancel a thread using pthreads:

```
pthread_t tid;
```

```
pthread_create(&tid, 0, worker, NULL);
```

```
pthread_cancel(tid);
```

- When you request to cancel a thread, whether it actually stops depends on the thread's current state.

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

If the thread's cancellation is **disabled**, the cancel request waits until the thread allows cancellation.

- By default, cancellation is **deferred**, meaning the thread cancels only at certain points called **cancellation points**.

ADVANTAGES OF MULTI-THREADED SYSTEMS:

Better Performance

Multiple threads run at the same time, so programs can do more work faster.

Resource Sharing

Threads in the same process share memory and resources, making communication and data sharing easier.

Responsiveness

Even if one thread is busy or waiting, other threads can keep the program responsive (like a UI staying smooth).

Efficient Use of CPU

Threads can run on multiple CPU cores simultaneously, using the processor power better.

Simpler Program Structure

You can split complex tasks into smaller threads, making the program easier to design and maintain.

Cost-Effective

Creating and switching between threads is faster and requires fewer resources compared to full processes.

Improved Scalability

Multi-threading helps applications scale well on modern multi-core systems.

Semaphore Implementation with No Busy Waiting:

In semaphore implementation without busy waiting, each semaphore maintains an associated waiting queue. This queue holds the processes that are waiting to access the critical section.

- Each entry in the waiting queue consists of:
 - An integer value (usually representing the semaphore count or status).
 - A pointer to the next process in the queue.

The implementation uses two key operations to manage processes in this queue:

1. **block:** This operation places the process that calls it into the semaphore's waiting queue,

2. effectively blocking the process until it can proceed.
3. **wakeup:** This operation removes a process from the waiting queue and moves it to the ready queue, allowing it to continue execution.

By using this queue mechanism, processes do not spin or busy wait for the semaphore to become available. Instead, they are put to sleep (blocked), freeing the CPU to execute other tasks. This method avoids the inefficiency and CPU wastage of busy waiting, especially important when critical sections can take longer time.

In code, the semaphore structure can be represented as:

```
typedef struct {  
    int value;  
  
    struct process *list; // pointer to waiting queue  
} semaphore;
```

This implementation is efficient and widely used in operating systems to handle process synchronization without wasting CPU resources.

Deadlock

Deadlock is a situation in a multiprogramming environment where two or more processes are waiting indefinitely for resources held by each other, and none of them can proceed. It occurs when the following four conditions hold simultaneously:

1. **Mutual Exclusion** – At least one resource is held in a non-shareable mode.
2. **Hold and Wait** – A process is holding at least one resource and waiting for more.
3. **No Preemption** – Resources cannot be forcibly taken away from a process.

4. **Circular Wait** – A circular chain of processes exists, each waiting for a resource held by the next.

Example:

Process A holds Resource R1 and waits for R2, while Process B holds R2 and waits for R1 → deadlock occurs.

Starvation

Starvation (or indefinite blocking) happens when a process waits for a resource for an indefinite time because other higher-priority processes keep getting scheduled before it. The low-priority process never gets a chance to execute.

Example:

In priority scheduling, if new high-priority processes keep arriving, a low-priority process may never get CPU time, leading to starvation.

Solution: Aging – gradually increasing the priority of waiting processes over time.

Priority Inversion

Priority inversion is a condition where a lower-priority process holds a resource needed by a higher-priority process, but the higher-priority process is blocked because the lower-priority one is not allowed to complete.

Example:

- Low-priority process holds a lock.
- High-priority process needs that lock but gets blocked.
- A medium-priority process (not needing the lock) preempts the low-priority process, delaying it further.

Solution: Priority Inheritance Protocol – temporarily raise the priority of the low-priority process holding the resource.

Dining Philosophers Problem

The Dining Philosophers Problem is a classic synchronization problem introduced by **Edsger Dijkstra** to illustrate the issues of **deadlock**, **concurrency**, and **resource sharing**.

Problem Statement:

- There are **five philosophers** sitting around a circular table.
- Each philosopher has **one plate of food** and needs **two chopsticks** (one on the left and one on the right) to eat.
- Between each pair of philosophers lies **one chopstick**, so there are **five chopsticks** in total.
- Philosophers alternate between **thinking** and **eating**.
- A philosopher can only eat if they pick up both the **left and right chopsticks**.
- After eating, they put the chopsticks back.

Problems Demonstrated:

1. **Deadlock:**
If all philosophers pick up their left chopstick at the same time and wait for the right one, none can eat. They all wait forever → **deadlock**.
2. **Starvation:**
If a philosopher is always the last to get chopsticks, they may never eat → **starvation**.
3. **Concurrency Issues:**
Improper access to chopsticks (shared resources) can lead to incorrect or undefined behavior.

Solutions (Approaches):

1. **Resource Hierarchy Solution:**
Number the chopsticks and require philosophers to pick them up in a specific

order (e.g., lower-numbered first). This breaks the circular wait condition.

2. **Allow at most 4 philosophers to sit at the table:**

Ensures at least one philosopher can always eat, preventing deadlock.

3. **Use of Semaphores or Mutexes:**

Use a semaphore for each chopstick and ensure atomic pick-up and put-down actions.

4. **Arbitrator (Waiter) Solution:**

Introduce a waiter who controls access to chopsticks and ensures only a safe number of philosophers eat simultaneously.

Conclusion:

The Dining Philosophers Problem is used to model and understand synchronization challenges in operating systems and multithreaded environments. It highlights the importance of handling **mutual exclusion**, **deadlock avoidance**, and **resource allocation**.



Definition of RPC (Remote Procedure Call):

Remote Procedure Call (RPC) is a protocol that allows a program to request a service or execute a function on another computer in a network, as if it were calling a local function.

Explanation in Simple Words:

- Normally, when you write a function in your program, it runs **on your own computer**.
- But sometimes, the function or data you need is on **another computer** (like a server).
- **RPC lets you call that remote function just like a regular function** — and it handles all the network communication for you.
- It **sends your request over the network**, the remote system **runs the function**, and then **sends the result back** to your program.

Example:

Suppose you have this in your code:

```
weather = getWeather("London")
```

This looks like a normal function call.

But with RPC:

- This call is actually **sent to a remote server**.
- The server runs the real `getWeather("London")` function.
- It returns the result (e.g., "Cloudy, 18°C") back to your program.

You didn't need to worry about how it happened — it just works like a local function call!

Execution of RPC

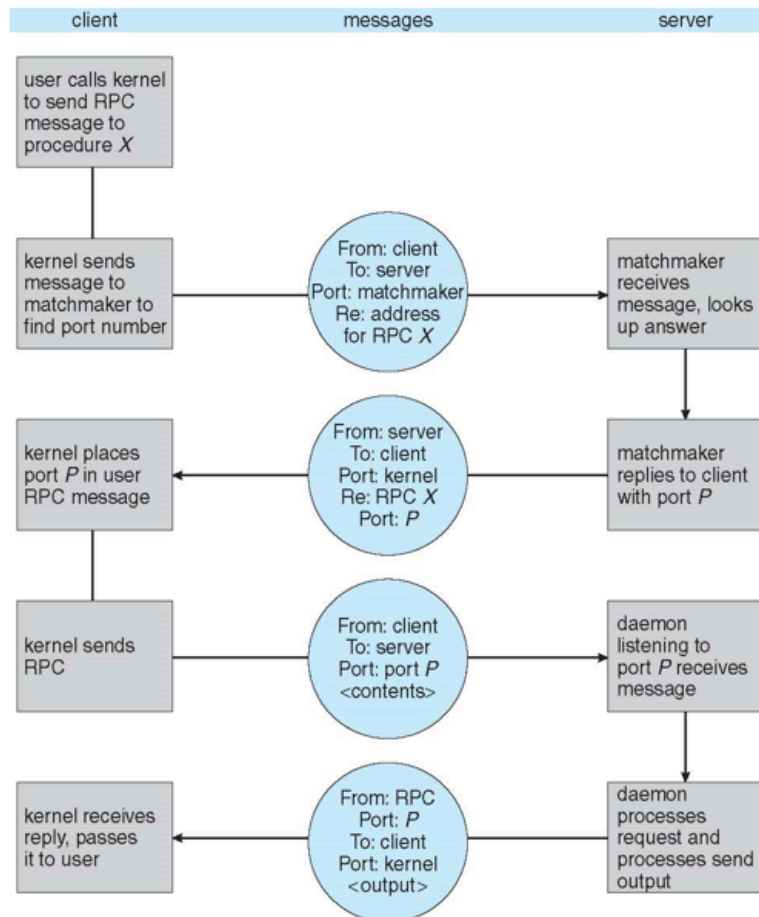
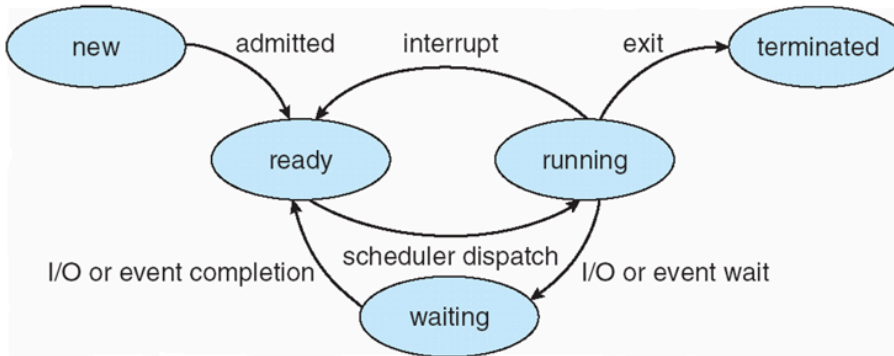


Diagram of Process State



Process State

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

- **new**: The process is being created.
- **running**: Instructions are being executed.
- **waiting**: The process is waiting for some event to occur.
- **ready**: The process is waiting to be assigned to a processor.
- **terminated**: The process has finished execution.

Process Control Block (PCB)

(also called **Task Control Block**)

The PCB is a **data structure** used by the operating system to store **all important information about a process**.

COMPONENTS OF PCB:

1. Process State

- Tells whether the process is running, ready, waiting, etc.

2. **Program Counter**

- Holds the address of the **next instruction** the process will execute.

3. **CPU Registers**

- Stores the contents of all the **CPU registers** that the process was using (used to resume the process correctly).

4. **CPU Scheduling Information**

- Includes things like process **priority** and pointers to scheduling **queues**.

5. **Memory Management Information**

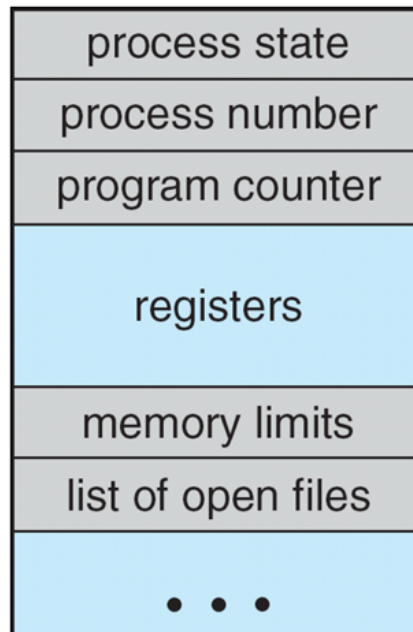
- Details about the **memory** allocated to the process, like base and limit registers.

6. **Accounting Information**

- Tracks how much **CPU time** the process has used, **start time**, and **time limits**.

7. **I/O Status Information**

- Information about **input/output devices** the process is using and the **list of open files**.



Give reason why monitors are not powerful enough to model some synchronization schemes.

MONITORS:

Monitors are **not powerful enough** to model some synchronization schemes because they **only allow one process or thread inside at a time**. This makes it hard to handle situations where **more flexible control is needed**, like allowing **multiple readers but only one writer** at a time.

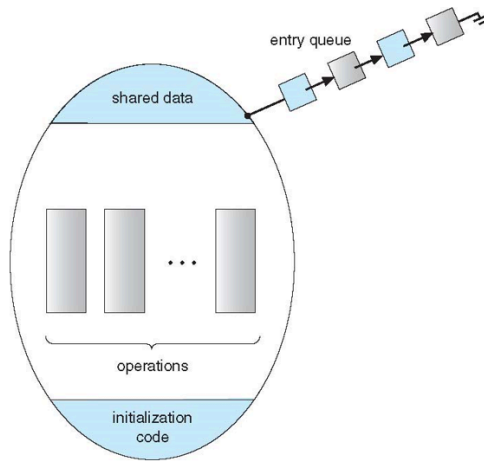
A **monitor** works like a room with a **single lock** — only **one person can enter** at a time. While inside, they can do their work and leave. If others want to come in, they have to **wait outside**.

This is good for **simple synchronization**, but for **complex cases**, like when:

- **Multiple readers** should be allowed to read at the same time,
- But **only one writer** can write (and no one else can read or write during that time),


Monitors **don't support this well**, because they **block everyone** except one thread.

Example: Readers-Writers Problem



Differentiate between synchronous and asynchronous inter process communication of message passing

DIFFERENCE OF SYNCHRONOUS AND ASYNCHRONOUS IPC:

Point	Synchronous IPC	Asynchronous IPC
1.	Sender waits for receiver to receive the message.	Sender does not wait , just sends the message and continues.
2.	Both sender and receiver must be ready at the same time .	Sender and receiver can work at different times .
3.	Slower , due to waiting for response.	Faster , as sender doesn't wait.
4.	Example: Phone call (both talk live).	Example: Email or WhatsApp message .
5.	Used when timing is important (e.g., live updates).	Used when timing is flexible .
6.	Easier to manage order of communication .	Needs more logic to manage message order and delivery .
7.	Message is received immediately .	Message is stored in a queue or buffer until received.
8.	Tightly connected processes.	Loosely connected processes.
9.	Simpler logic , easier to implement.	More complex , needs buffer/queue management.
10.	No need for message storage .	Requires buffer or queue to hold messages.
11.	Good for real-time systems.	Good for background or non-urgent tasks.
12.	If receiver is busy, sender must wait .	Sender keeps working even if receiver is busy .
13.	Harder to scale in large systems. 	Easier to scale , better for distributed systems.

Illustrate the windows thread data structure diagrammatically

Windows Threads:

Each thread in Windows has some important structures that store its information:

1. ETHREAD (Executive Thread Block)

- It points to the process that the thread belongs to.

- It also points to KTHREAD.
- This is stored in **kernel space** (part of the OS memory).

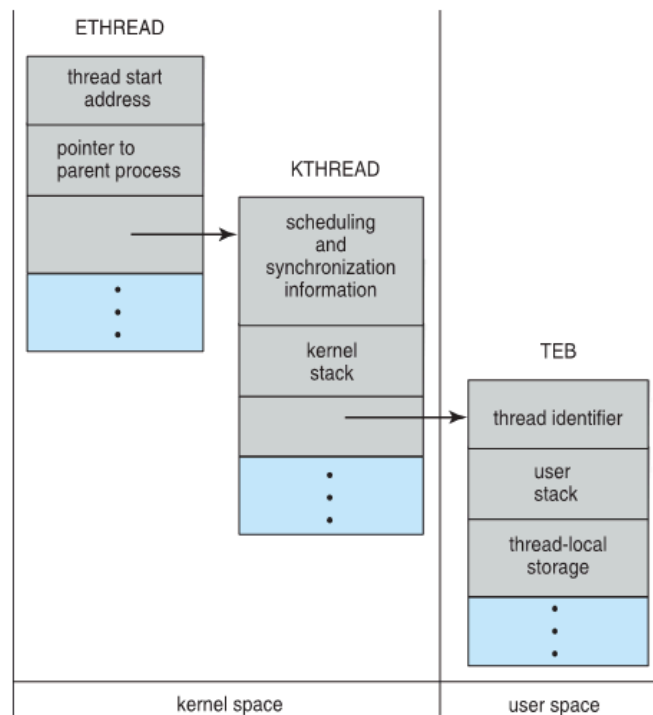
2. KTHREAD (Kernel Thread Block)

- It stores scheduling info (when and how the thread runs).
- It has synchronization info, kernel-mode stack, and a pointer to TEB.
- This is also in **kernel space**.

3. TEB (Thread Environment Block)

- It keeps the thread ID, user-mode stack, and thread-local data.
- This part is in **user space** (where user programs run).

Windows Threads Data Structures



PREEMPTIVE VS NON-PREEMPTIVE :

Feature	Preemptive Approach	Non-Preemptive Approach
Interruption	Process can be interrupted anytime	Process cannot be interrupted once it starts
Control	OS controls when to switch processes	Process keeps control until it finishes
Critical Section Safety	Needs extra care to avoid issues like conflicts	Safer as no one else enters until it's done
CPU Switching	Can switch tasks anytime	No switching during the critical section
Waiting Time	May be less , but risky	May be more , but safe
Complexity	More complex to manage	Easier to manage
Example	Someone takes your turn midway	Others wait for your turn to finish