1. Processes: Resource Ownership & Scheduling

- **Processes have two main characteristics:**
    1. **Resource Ownership**:
        - Each process owns resources like **memory, I/O devices, and files**.
        - The OS allocates a **virtual address space** for each process.
    2. **Scheduling & Execution**:
        - A process follows an **execution path (trace)** through programs.
        - The OS **dispatches** and manages execution states (Ready, Running, Blocked).
        - Each process has a **dispatching priority**.

2. Threads

- A **thread (lightweight process)** is the **unit of execution** within a process.
- A **process (task)** is the **unit of resource ownership**.
- Threads **share process resources** (memory, files, I/O).

---

3. Multithreading

- **Definition**: An OS's ability to support multiple, concurrent paths of execution within a single process.
- **Types of Multithreading Models**:
    1. **Single process, single thread** (e.g., MS-DOS).
    2. **Single process, multiple threads** (e.g., Java Runtime Environment).
    3. **Multiple processes, single thread per process** (e.g., traditional UNIX).
    4. **Multiple processes, multiple threads per process** (e.g., Windows, Linux).

Advantages of Multithreading

✅ **Faster thread creation & termination** than process creation.
✅ **Efficient communication** (threads share memory and files).
✅ **Lower context switch time** than processes.
✅ **Better resource utilization** in multiprocessor environments.

---

4. Thread Components

Each thread has:

- **Execution state** (Running, Ready, Blocked).
- **Thread context (registers, PC, stack pointer)**.
- **Execution stack** (local function variables).
- **Shared process resources** (memory, files).

5. Single-threaded vs Multithreaded Model

Feature	Single-threaded	Multithreaded
Context Switching	Slow (requires saving/restoring process state)	Fast (only thread state needs to change)
Resource Utilization	Low	High (shared resources)
Execution Speed	Slower	Faster
Concurrency	No	Yes

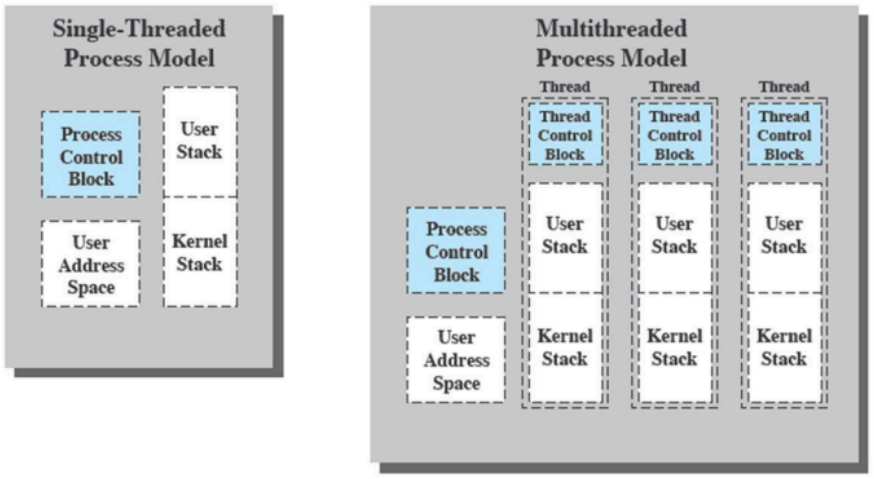


Figure 4.2 Single Threaded and Multithreaded Process Models

(1)

- **Single-threaded process model-**
- No distinct concept of thread
- the representation of a process includes
  - its PCB (process control block)
  - user address space,
  - user and kernel stacks to manage the call/return behaviour of the execution of the process.

While the process is running, it controls the processor registers

When the process is not running, Contents of these registers are saved

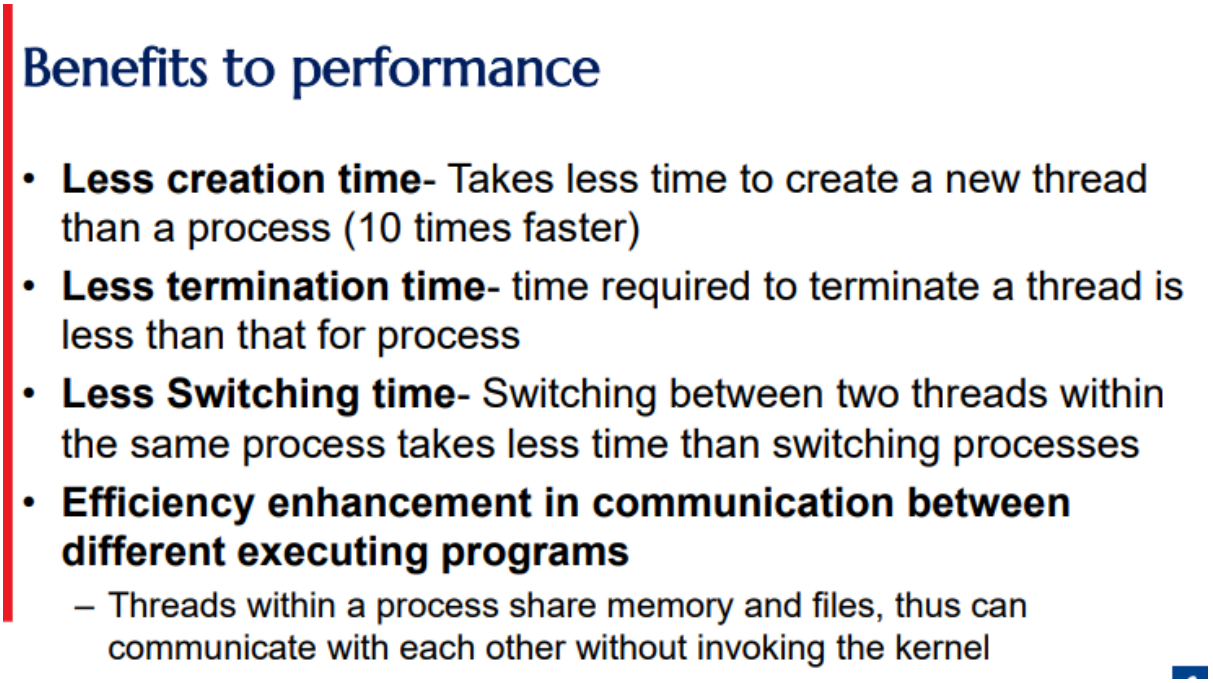
(2)

- **Multithreaded environment…**
  - All threads of a process share the state and resources of that process
  - They reside in the same address space and have access to the same data.
  - When one thread alters data in the memory, other threads see the results, as and when they access that location
  - Threads in the same process can read the contents of the same memory location concurrently
    - If one thread opens a file with read privileges, other threads in the same process can also read from that file

- **In a multithreaded environment** there is single PCB and user address space associated with the process, **but** separate stacks for each thread

- Separate control block called **Thread Control Block**, is maintained for each thread containing register values, priority, and other thread-related state information

BENEFITS OF MULTITHREADING



Q.Thread States -

- Running, Ready, Blocked;
- Suspended state is not associated with threads • If a process is suspended, all its threads are suspended

Threads can perform the following operations:

- **Spawn** (create a new thread).
- **Block** (suspend execution until an event occurs).
- **Unblock** (resume execution).
- **Finish** (terminate and free resources).

7. Multithreading on a Uniprocessor

- Threads share CPU **time slices**.
- Context switching allows **interleaved execution** of multiple threads.
- Improves responsiveness of applications.

8. Thread Implementation Models

Threads can be implemented in three ways:

1. **User-Level Threads (ULTs)**

2. **Kernel-Level Threads (KLTs)**
3. **Hybrid Model (Combined Approach)**

---

1. User-Level Threads (ULTs)

- **All thread management is done in user space** (i.e., by the application).
- The **kernel is unaware** of the existence of threads.
- Uses a **Thread Library** that provides routines to create, destroy, and manage threads.
- The **OS only schedules processes, not threads**.

How ULT Works?

1. The application starts **with a single thread**.
2. It **creates additional threads** using the thread library.
3. The **threads library manages scheduling, synchronization, and context switching** between threads within the same process.
4. The **kernel schedules the process as a whole**, not individual threads.

Advantages of ULTs

✅ **Faster Context Switching**:

- Since **the kernel is not involved**, switching between threads is **much faster**.

✅ **Efficient Scheduling**:

- Application can implement **custom thread scheduling** without interfering with the OS scheduler.

✅ **Portable**:

- Works on any OS because the **kernel does not need to support threading**.

✅ **No Kernel Mode Switching**:

- **Thread operations happen in user space**, avoiding expensive mode switches between user and kernel modes.

Disadvantages of ULTs

❌ **One thread blocking the system call blocks the entire process**:

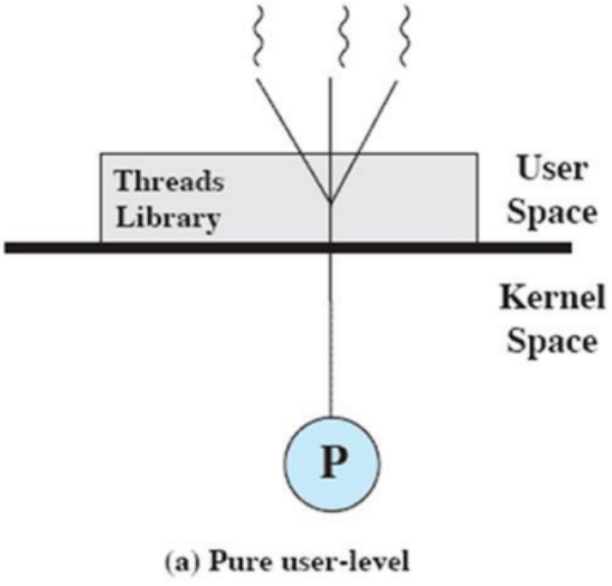
- If a thread makes a **blocking system call**, the entire process gets blocked because the **kernel only recognizes processes, not threads**.

❌ **Cannot Utilize Multiprocessing**:

- The kernel schedules **only one process at a time**. Even if a process has multiple threads, they **cannot run in parallel on multiple processors**.

Solution to ULT Problems

- **Jacketing**:
    - A technique where **blocking system calls are converted into non-blocking calls**.
    - Instead of directly calling the I/O system call, the thread calls a wrapper function (jacketing) that **checks availability before making the system call**.



(a) Pure user-level

---

2. Kernel-Level Threads (KLTs)

- The **OS manages and schedules threads** directly.
- The **kernel is aware** of threads and handles their execution.
- Each thread **has its own Thread Control Block (TCB)** in the kernel.

How KLT Works?

1. The **kernel provides system calls** for thread management.
2. The OS **schedules threads independently**, just like processes.
3. **If one thread blocks, others in the same process can continue execution**.
4. **Threads can be scheduled on multiple processors**, supporting true parallelism.

Advantages of KLTs

✅ **True Parallel Execution**:

- Multiple threads can **run simultaneously on different CPUs** in a multiprocessor system.

✅ **No Blocking Issue**:

- If one thread **blocks on I/O**, the **kernel can schedule another thread** from the same process.

✅ **Kernel-Level Multithreading**:

- The **kernel itself can be multithreaded**, improving OS efficiency.

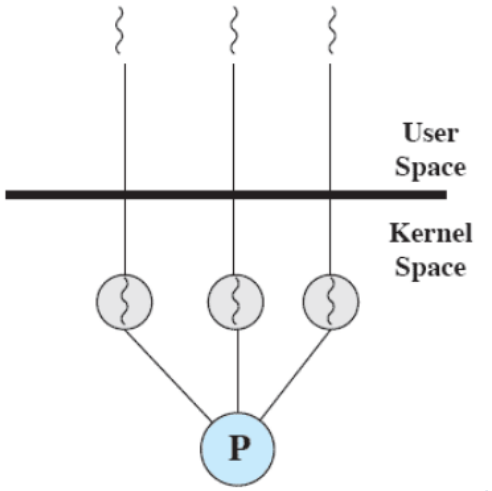
Disadvantages of KLTs

❌ **High Overhead**:

- **Thread creation, termination, and synchronization involve kernel intervention**, leading to **higher overhead**.

❌ **Mode Switching Overhead**:

- **Switching between threads requires mode switching** between **user mode and kernel mode**, making context switching **slower** than ULTs.



(b) Pure kernel-level

---

3. Hybrid Model (Combined Approach)

- **Combines the benefits of ULTs and KLTs**.

- Multiple **user-level threads (ULTs) are mapped onto kernel-level threads (KLTs)**.
- The **kernel is aware of some threads but not all**.

How Hybrid Model Works?

1. **Thread creation happens in user space** using thread libraries.
2. A **smaller number of kernel threads (KLTs) handle multiple ULTs**.
3. **If one ULT blocks**, the **kernel schedules another KLT** to keep the process running.

Advantages of Hybrid Model

✅ **Parallel Execution**:

- Unlike ULTs, threads **can run on multiple processors**.

✅ **Fast Context Switching**:

- Like ULTs, **most thread operations happen in user space**, avoiding frequent kernel intervention.

✅ **No Blocking Issue**:

- If a ULT **blocks**, another ULT can be **assigned to a different KLT** and continue execution.

✅ **Efficient Multiprocessing**:

- Multiple kernel threads allow **parallel execution**, but user threads reduce overhead.

Disadvantages of Hybrid Model

❌ **Complex Implementation**:

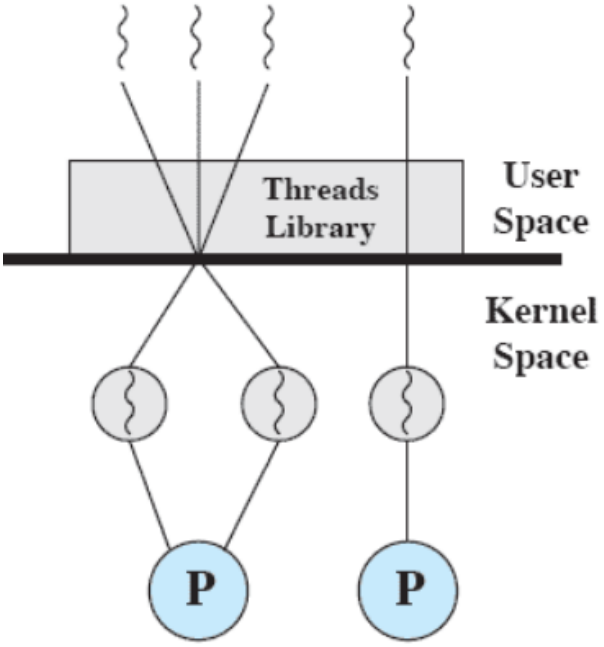
- Requires **both thread libraries** and **kernel-level support**, making implementation complex.

❌ **Extra Overhead**:

- Managing the mapping between **ULTs and KLTs** introduces additional **synchronization overhead**.

Example of Hybrid Model OS

- **Solaris OS** uses a hybrid model:
    - Maps **many user threads** to **a few kernel threads** for efficient execution.



(c) Combined

Comparison of ULT, KLT, and Hybrid Model

Feature	User-Level Threads (ULT)	Kernel-Level Threads (KLT)	Hybrid Model
Thread Creation & Management	Done in **user space**	Done in **kernel space**	Managed by **both user & kernel**
Context Switching Speed	**Fast** (no kernel involvement)	**Slow** (requires kernel mode switch)	**Moderate** (depends on ULT-KLT mapping)
Multiprocessing Support	❌ No	✅ Yes	✅ Yes
Blocking Issue	❌ Yes (one thread blocks all)	✅ No (another thread runs)	✅ No (maps ULTs to KLTs)
Portability	✅ Works on any OS	❌ OS-dependent	❌ OS-dependent
Complexity	✅ Simple	❌ Complex	❌ Most Complex

Final Thoughts

- **ULTs are fast but limited to single-core execution**.
- **KLTs enable true parallelism but introduce kernel overhead**.

- **Hybrid Model balances performance and flexibility**, making it a good choice for modern operating systems like **Solaris**.

Relationship Between User-Level Threads (ULT) and Process States

Since the **kernel is unaware** of **user-level threads (ULTs)**, the **scheduling and execution of ULTs are managed entirely by the application (thread library)**. This creates a unique relationship between **ULTs and process states**, which differs from kernel-managed threads.

Process & Thread State Relationship

- The **kernel schedules processes**, not individual user threads.
- The **state of a process determines the state of all its threads**.
- Since ULTs are managed in user space, a process with multiple threads still appears as a **single entity to the OS**.

Why Does This Relationship Cause Issues?

- **Problem**: Since the **kernel does not recognize ULTs**, it **treats the whole process as a single entity**.
- **Example Issue**: If **one ULT makes a blocking system call, all ULTs in the process are blocked**, even if others could continue executing.
- **Workaround**: **Jacketing (Wrapping System Calls)**
    - Instead of directly making a system call, a **wrapper function checks if the I/O resource is available**.
    - If unavailable, the ULT **switches to another thread instead of blocking the whole process**.

Q.

Relationship Between Thread and Processes

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

Q. Categories of computer system - SISD, SIMD, MISD, MIMD

Categories of Computer Systems

- Single Instruction Single Data (SISD) stream
  - Single processor executes a single instruction stream to operate on data stored in a single memory
- Single Instruction Multiple Data (SIMD) stream
  - Each instruction is executed on a different set of data by different processors
- Multiple Instruction Single Data (MISD) stream
  - A sequence of data is transmitted to a set of processors, each of them execute a different instruction sequence
- Multiple Instruction Multiple Data (MIMD)
  - A set of processors simultaneously execute different instruction sequences on different data sets

Categories of Computer Systems

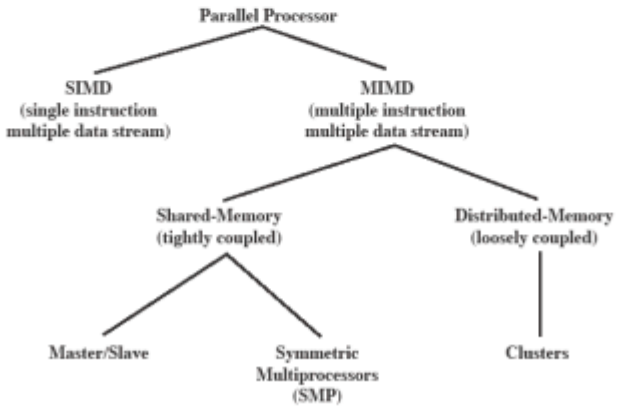


Figure 4.8 Parallel Processor Architectures

12. Symmetric Multiprocessing (SMP)

- **All processors are equal**.
- **OS executes on any processor**.
- **Advantages**:
  - Better resource utilization.
  - More parallelism.
- **Disadvantages**:
  - Synchronization complexity.

Multiprocessor Scheduling Design Issues

1. Key Challenges in Multiprocessor Scheduling

When scheduling processes on **multiple processors**, the OS must address several issues:

🔹 **Simultaneous execution of multiple processes/threads**

- The OS must manage multiple processes running in parallel on different processors.
- It needs **proper load balancing** to avoid idle processors.

🔹 **Synchronization of threads and processes**

- If multiple processes share resources (e.g., memory, I/O devices), the OS must ensure **proper synchronization**.
- **Race conditions** and **deadlocks** must be avoided.

🔹 **Memory management**

- When multiple processors access **shared memory**, the OS must maintain **consistency**.
- Example: If **Processor 1 updates a shared variable**, **Processor 2 should see the latest value**.

🔹 **Reliability & Fault Tolerance**

- If one processor **fails**, the system should continue functioning without crashing.
- The OS must detect failures and reassign processes.

---

13. Assignment of Processes to Processors

A. Static Assignment (Processor Affinity)

🔹 **Definition**:

- Each process is **permanently assigned** to a specific processor.
- The OS does **not move processes between processors**.

🔹 **Advantages**:
✅ **Better cache performance** → The process stays on the same CPU, reducing cache misses.
✅ **Less scheduling overhead** → No need to move processes frequently.

🔹 **Disadvantages**:
❌ **Uneven workload distribution** → Some processors may be overloaded while others stay idle.
❌ **Processor failures cause problems** → If a processor fails, all assigned processes are lost.

🔹 **When is it used?**

- Useful in **real-time systems** where predictability is critical.
- Example: **Air traffic control systems** use static assignment for reliability.

B. Dynamic Assignment (Load Balancing)

🔹 **Definition**:

- Processes are **dynamically assigned** to available processors.
- If a processor becomes idle, it can **steal work from another processor**.

🔹 **Advantages**:
✅ **Better load distribution** → The system adjusts workloads dynamically.
✅ **Processor failures are handled efficiently** → If a processor fails, its processes can be moved.

🔹 **Disadvantages**:
❌ **Cache inefficiency** → Processes may move between processors, leading to more cache misses.
❌ **Higher scheduling overhead** → The OS needs to constantly track and move processes.

🔹 **When is it used?**

- Used in **modern multi-core systems** (e.g., Linux, Windows).
- Example: **Cloud computing** dynamically assigns workloads to virtual processors.

Master-Slave vs. Peer-to-Peer Models

A. Master-Slave Model

🔹 **Definition**:

- A **single processor (Master)** is responsible for **all scheduling decisions**.
- Other processors (Slaves) execute tasks assigned by the Master.

🔹 **How it works?**

1. The Master **chooses which process should run**.
2. It assigns the process to an available Slave processor.
3. The Slave executes the task and reports back to the Master.

🔹 **Advantages**:
✅ **Simple to implement** → The scheduling logic is centralized.
✅ **Consistent decision-making** → No conflicts between processors.

🔹 **Disadvantages**:
❌ **Single Point of Failure** → If the Master processor fails, the entire system stops.
❌ **Bottleneck Issues** → The Master may become overloaded while Slaves remain idle.

🔹 **When is it used?**

- Used in **older multiprocessor systems** and **some embedded systems**.

B. Peer-to-Peer Model

🔹 **Definition**:

- **All processors are equal** and share scheduling responsibilities.
- Each processor **self-schedules** its own processes.

🔹 **How it works?**

1. Each processor has its **own queue of processes**.
2. When a processor **becomes idle**, it selects a process from its queue.
3. If its queue is empty, it may **steal a process from another queue**.

🔹 **Advantages**:
✅ **No single point of failure** → If one processor fails, others continue working.
✅ **Scalability** → More processors can be added without overloading a Master processor.

🔹 **Disadvantages**:
❌ **Complex OS design** → The OS must handle **synchronization between processors**.
❌ **Load imbalance risk** → Some processors may be overloaded while others are idle.

🔹 **When is it used?**

- Used in **modern multiprocessor OS** (e.g., Linux, Windows).
- Example: **Google's data centers** use peer-to-peer scheduling to distribute workloads.

4. Summary of Process Assignment Strategies

Assignment Strategy	How it Works	Advantages	Disadvantages
Static Assignment	Each process is permanently assigned to a processor	✅ Low scheduling overhead ✅ Good cache performance	❌ Load imbalance ❌ Hard to handle failures
Dynamic Assignment	Processes are assigned based on availability	✅ Better load balancing ✅ Handles failures well	❌ High scheduling overhead ❌ Poor cache performance
Master-Slave	One processor (Master) schedules all processes	✅ Simple implementation ✅ Predictable behavior	❌ Single point of failure ❌ Bottleneck risk
Peer-to-Peer	Each processor schedules its own processes	✅ No single point of failure ✅ Scalable	❌ Complex OS design ❌ Load imbalance risk

Final Takeaways

✅ **Multiprocessor scheduling** is **more complex than single-processor scheduling** because multiple CPUs must coordinate tasks efficiently.
✅ **Static assignment** is **simple but inefficient** for load balancing.
✅ **Dynamic assignment** ensures **better resource utilization** but requires **more overhead**.
✅ **Master-Slave model** is **easy to implement** but **not scalable**.
✅ **Peer-to-Peer model** provides **better fault tolerance** but is **complex** to implement.

14. Process Scheduling (Point 14 from PDF)

- Usually, processes are **not dedicated to specific processors**.
- A **single queue is used** for all processes.
- Alternatively, **multiple queues** may be used **for different priorities**.
- **All queues feed into the common pool of processors** for scheduling.

15. Thread Scheduling (Point 15)

What is Thread Scheduling?

- **Threads execute separately** from the main process.
- An application can consist of **multiple threads running concurrently**.
- **Significant performance improvements** can be achieved on **multiprocessor systems**.

Four General Approaches to Thread Scheduling:

1. Load Sharing

- **Threads are not assigned to specific processors**.
- A **global queue** of ready threads is maintained.
- **When a processor is idle, it selects a thread from the queue**.

Advantages of Load Sharing:

✅ **Simple approach**.
✅ **Load is distributed evenly** across processors.
✅ **No centralized scheduler is required**.
✅ **Whenever a processor is free, it selects the next thread from the queue**.

Disadvantages of Load Sharing:

❌ **Central queue bottleneck → If many processors access it simultaneously**, contention may occur.
❌ **Caching inefficiency → A preempted thread may not resume execution on the same processor**.
❌ **Thread locality issue → Threads of a process may not execute on the same processor at the same time**.

Different Versions of Load Sharing

🔹 First-Come-First-Served (FCFS)

- **Each thread is placed at the end of the queue in arrival order**.
- **Processors pick threads from the front of the queue**.

🔹 Smallest Number of Threads First

- **Priority is given to jobs that have the fewest threads**.

- Helps **reduce waiting time** for smaller jobs.

🔹 Preemptive Smallest Number of Threads First

- **If a new job arrives with fewer threads than the currently executing job, it preempts execution**.
- Helps **favor smaller jobs** over larger ones.

---

2. Gang Scheduling

- **A set of related threads is scheduled to run together on multiple processors at the same time**.
- **Parallel execution reduces synchronization overhead**.
- **Prevents context switching issues for related threads**.

How Gang Scheduling Works:

1. **A group (gang) of threads is scheduled as a unit**.
2. **All gang members execute simultaneously** on different processors.
3. **All members start and finish their time slice together**.

---

3. Dedicated Processor Assignment

- **Each program is allocated processors equal to its number of threads**.
- **Processors are reserved for the program's entire execution duration**.
- **When the program finishes, processors are released for other programs**.

Disadvantages:

❌ Some processors **may remain idle** if not fully utilized.
❌ **No multiprogramming on reserved processors**.

---

4. Dynamic Scheduling

- The **number of threads in a process can change dynamically**.
- New threads **can be created or terminated** during execution.
- The **OS dynamically adjusts execution** to improve efficiency.

---

3. Summary of Thread Scheduling Approaches

Scheduling Approach	How It Works	Advantages	Disadvantages
Load Sharing	Global queue, idle processors take tasks	✅ Simple ✅ Balances load	❌ Cache inefficiency ❌ Thread switching overhead
Gang Scheduling	Group of threads run simultaneously	✅ Improves synchronization	❌ Slowest thread delays all others
Dedicated Processor Assignment	Fixed processor for each thread	✅ No scheduling overhead ✅ Efficient for real-time	❌ Processors may remain idle
Dynamic Scheduling	Adjusts thread execution dynamically	✅ Maximizes CPU use ✅ Adapts to workload	❌ High scheduling complexity

---

Final Takeaways

✅ **Process scheduling** is critical for **efficient CPU utilization**.
✅ **Short-term scheduling** handles **real-time process execution**.
✅ **Thread scheduling optimizes CPU execution in multithreaded systems**.
✅ **Different thread scheduling approaches** (e.g., **Load Sharing, Gang Scheduling**) balance CPU load.