Topic 11

Concurrency and Parallelism

Objectives

- Understand the concepts of concurrency and parallelism.
- Understand the terms of critical section, mutual exclusion, race condition, deadlock, livelock, starvation
- Be aware of tools for enforcing mutual exclusions
- Be aware of the strategies for tackling deadlock: prevention, avoidance and detection
- Understand the concept of threads and its benefits.
- Understand different thread implementations: user-level and kernel-level threads.
- Be aware of Flynn's taxonomy and parallel computing
- Understand symmetrical multiprocessing (SMP)
- Understand Microkernel architecture

Readings

- Stallings: Ch 4.1 – 4.3
- Stallings: Ch 5.1 to 5.2
- Stallings: Ch 6.1 to 6.4
- Skim the rest of Ch 4, 5 and 6.

Concurrency

- Concurrency means more than one execution flow exists at the same time.
- These execution flows often share resources.
- In some systems, these execution flows share the same processor, thus they interleave with each other.
- In other systems they run on multiple processors, so they can run in parallel.

Benefit of Concurrency

- Concurrency allows more efficient use of resources - time, processor, memory, input and output devices etc.
- All modern operating systems are concurrent systems, as the operating system kernel and multiple processes exist at the same time.

Problems with Concurrency

- Race condition - two or more processes read and write shared data and the final results depend on the relative timing of the processes.
- Deadlock - two or more processes wait for each other and none can proceed any further.

Problems with Concurrency

- Starvation - a runnable process is indefinitely overlooked by the scheduler. Although it is able to proceed, but is never chosen.
- Livelock - two or more processes continuously change their states in response to changes in other processes without doing any useful work.

Difficulties of Concurrency

- Sharing of global resources
- Operating system managing the allocation of resources optimally
- Difficult to locate programming errors

Race Condition - A Simple Example

- Function **`echo`** and the two global variables **`chin`** and **`chout`** are shared by processes P1 and P2.


```
void echo()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

Race Condition - A Simple Example

```
   Process P1                    Process P2

.                                .
chin = getchar();                .
.                                chin = getchar();
chout = chin;                    .
.                                chout = chin;
putchar(chout);                  .
.                                putchar(chout);
.                                .
```

Mutual Exclusion

- To avoid race conditions, we must enforce mutual exclusion in the critical sections of the processes
- Critical Section - a section of code within a process that requires access to shared resources which may not be executed while another process is in a corresponding section of code.
- Mutual Exclusion - only one process at a time is allowed in its critical section, e.g., only one process at a time is allowed to send command to the printer

Requirements for Mutual Exclusion

- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its non-critical section must do so without interfering with other processes
- No deadlock or starvation

Requirements for Mutual Exclusion

- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

Mutual Exclusion: Hardware Support

- Interrupt Disabling
  - So the process is not interrupted by another process while it is in its critical section
  - This is dangerous if there is a bug in the code
- Special Machine Instructions
  - Test and Set instruction
  - Exchange instruction

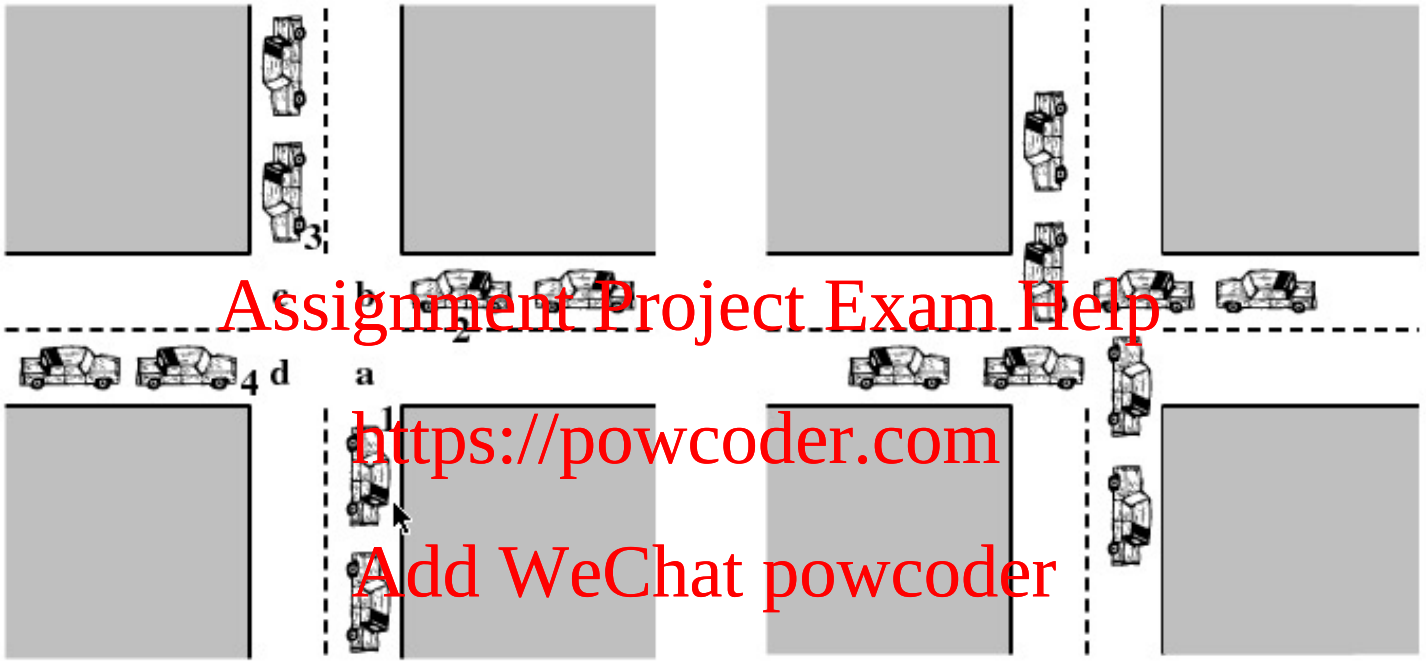
Mutual Exclusion: Semaphore

- Semaphores, such as binary semaphores, are often used in user processes to enforce mutual exclusions.
- Example: use a shared binary semaphore s:

P1:	P2:
P(s)	P(s)
use chin and cout	use chin and cout
V(s)	V(s)

Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No efficient solution
- Involve conflicting needs for resources by two or more processes
- Deadlocks are often the result of enforcing mutual exclusions when accessing shared resources.



(a) Deadlock possible (b) Deadlock

Figure 6.1 Illustration of Deadlock

Reusable Resources

- Used by only one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes
- Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other

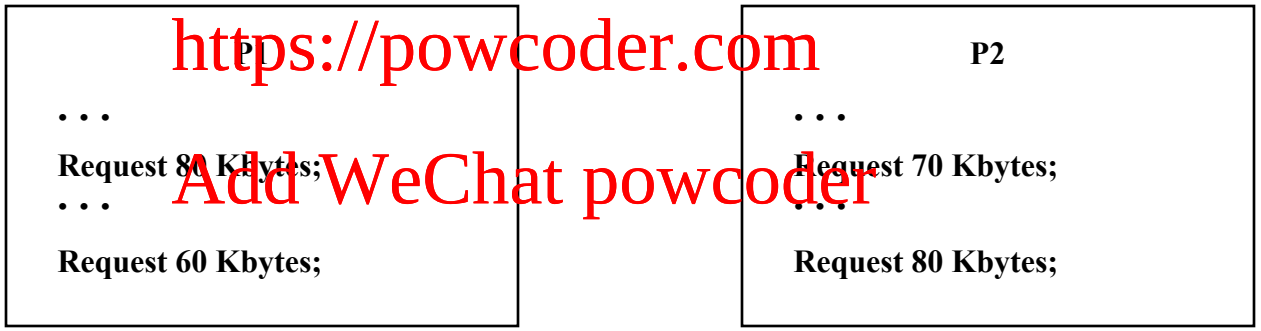
Example of Deadlock

Process P		Process Q	
Step	**Action**	**Step**	**Action**
p_0	Request (D)	q_0	Request (T)
p_1	Lock (D)	q_1	Lock (T)
p_2	Request (T)	q_2	Request (D)
p_3	Lock (T)	q_3	Lock (D)
p_4	Perform function	q_4	Perform function
p_5	Unlock (D)	q_5	Unlock (T)
p_6	Unlock (T)	q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

Another Example of Deadlock

- Space is available for allocation of 200Kbytes, and the following sequence of events occur



- Deadlock occurs if both processes progress to their second request

Consumable Resources

- Created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock

Example of Deadlock

- Deadlock occurs if receive is blocking

P1	P2
……	……
Receive(P2)	Receive(P1)
……	…….
Send(P2, M1)	Send(P1, M2)

Resource Allocation Graphs

- Directed graph that depicts a state of the system of resources and processes

(a) Resouce is requested (b) Resource is held

Resource Allocation Graphs

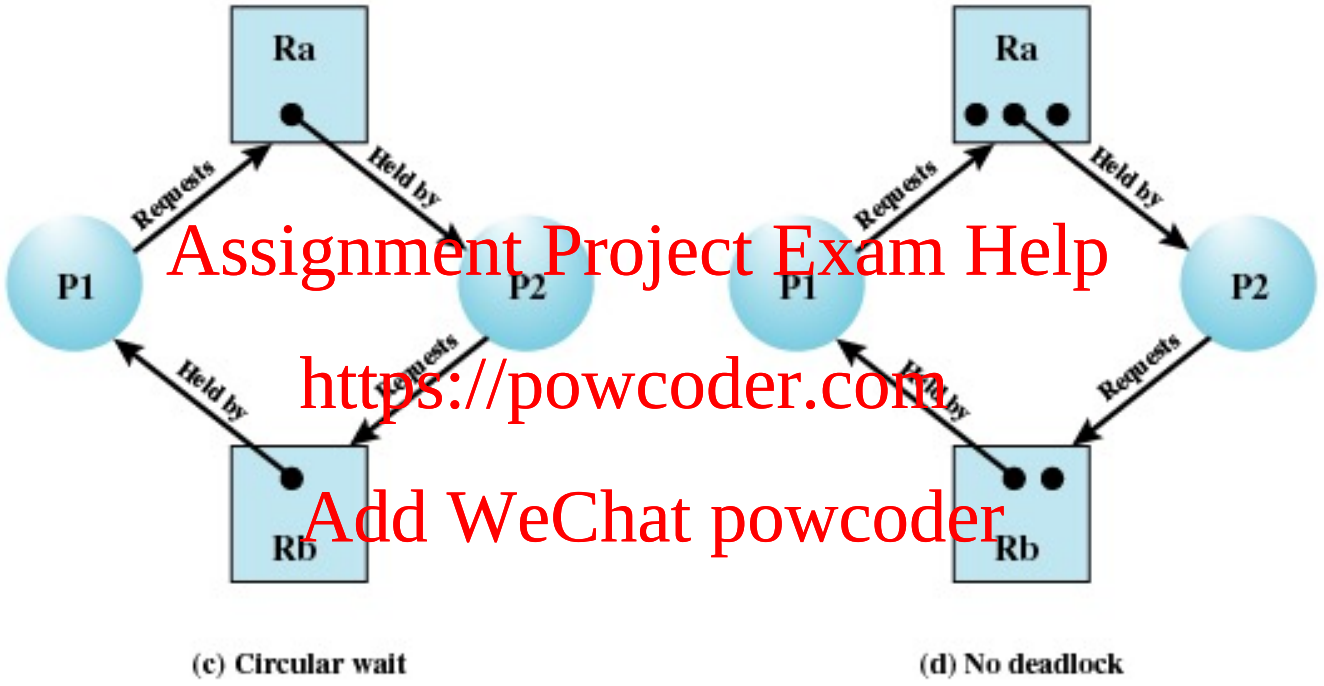


Figure 6.5 Examples of Resource Allocation Graphs

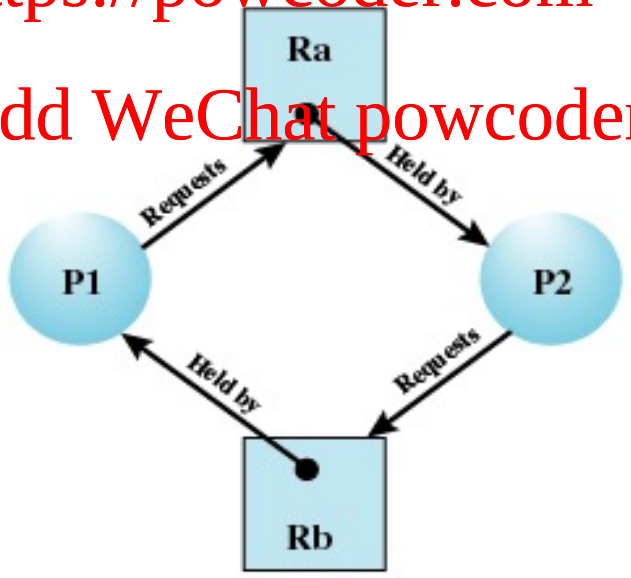
Necessary Conditions for Deadlock

- For deadlock to occur, the following three conditions must exist:

  - Mutual exclusion
    - Only one process may use a resource at a time
  - Hold-and-wait
    - A process may hold allocated resources while awaiting assignment of others
  - No preemption
    - No resource can be forcibly removed from a process holding it

Necessary and Sufficient Conditions for Deadlock

- Circular wait - a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain



(c) Circular wait

Resource Allocation Graph

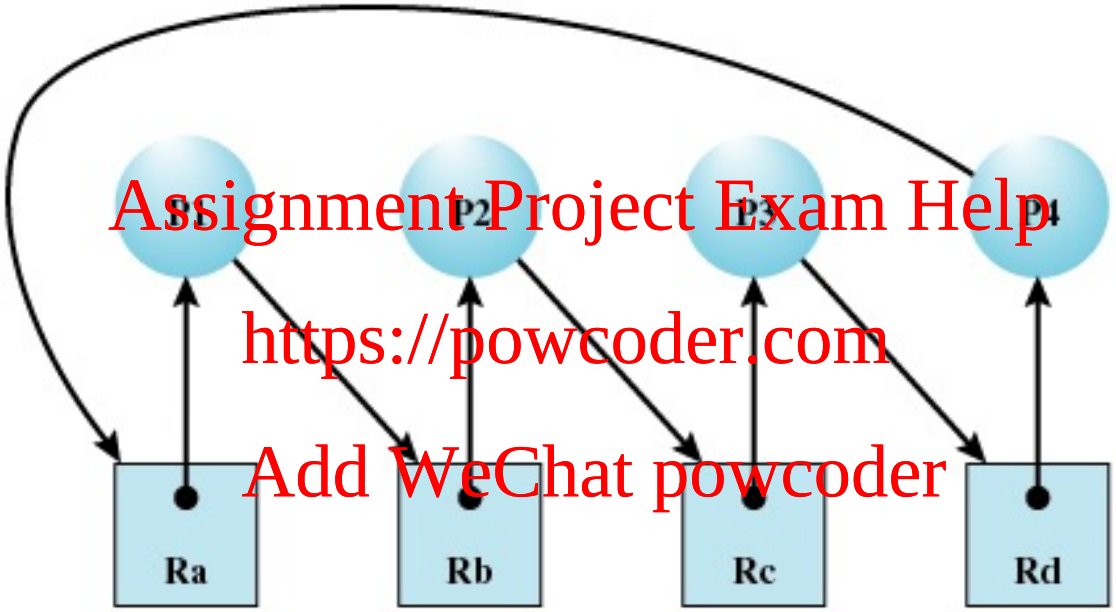


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Deadlock Prevention

- Mutual Exclusion
  - not to enforce mutual exclusion, e.g., reading a shared file
  - not always achievable
- Hold and Wait
  - Require a process request all of its required resources at one time

Deadlock Prevention

- No Preemption
  - Process must release resource and request again
  - Operating system may preempt a process to require it releases its resources
- Circular Wait
  - Define a linear ordering of resource types

Deadlock Avoidance

- A decision is made dynamically on whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future process request

Two Approaches to Deadlock Avoidance

- Do not start a process if its demands might lead to deadlock
- Do not grant an incremental resource request to a process if this allocation might lead to deadlock

The Banker's Algorithm

- State of the system is the current allocation of resources to processes
- Safe state is where there is at least one sequence that does not result in deadlock
- Unsafe state is a state that is not safe

Strategies once Deadlock Detected

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart all process
  - Original deadlock may occur
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Starvation - Dining Philosophers Problem

Figure 6.11 Dining Arrangement for Philosophers

UNIX Concurrency Mechanisms

- Pipes
- Messages
- Shared memory
- Semaphores
- Signals

Process vs Thread

- A traditional process has two functionalities:
  - Resource ownership - process includes a virtual address space to hold the process image
  - Scheduling/execution - follows an execution path that may be interleaved with other processes
- These two characteristics are treated independently by the operating system

Process vs Thread

- Scheduling and execution is referred to as a thread
- Resource ownership is referred to as a process or task

Multithreading

- Operating system supports multiple threads of execution within a single process
- MS-DOS supports a single thread
- Traditional UNIX supports multiple user processes but only supports one thread per process
- Windows, Solaris, Linux, Mach, and OS/2 and Darwin (macOS) support multiple threads

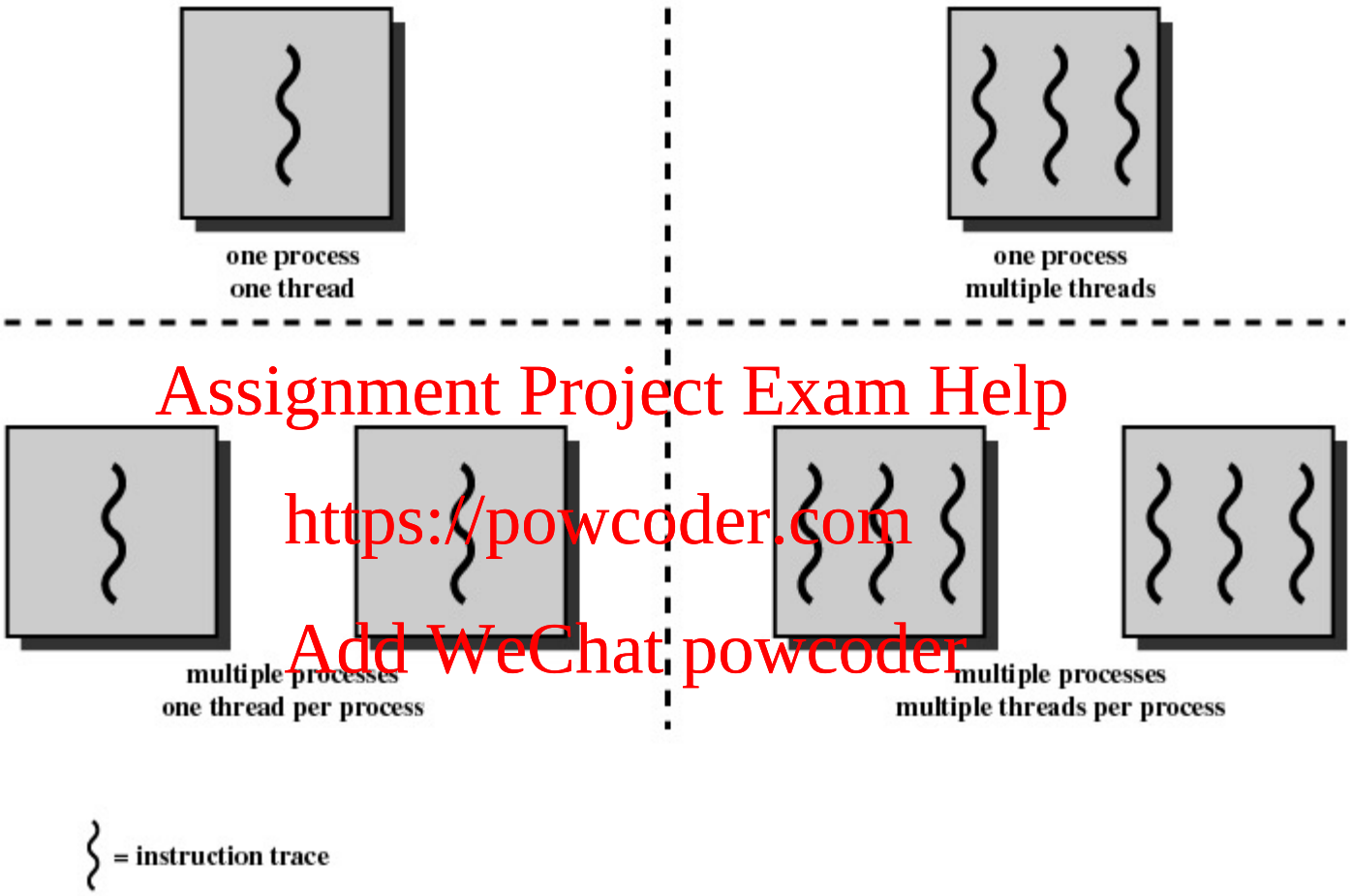


Figure 4.1 Threads and Processes [ANDE97]

Process

- Have a virtual address space which holds the process image
- Protected access to processors, other processes, files, and I/O resources

Thread

- An execution state (running, ready, etc.)
- Saved thread context when not running
- Has an execution stack
- Some per-thread static storage for local variables
- Access to the memory and resources of its process
  - all threads of a process share this

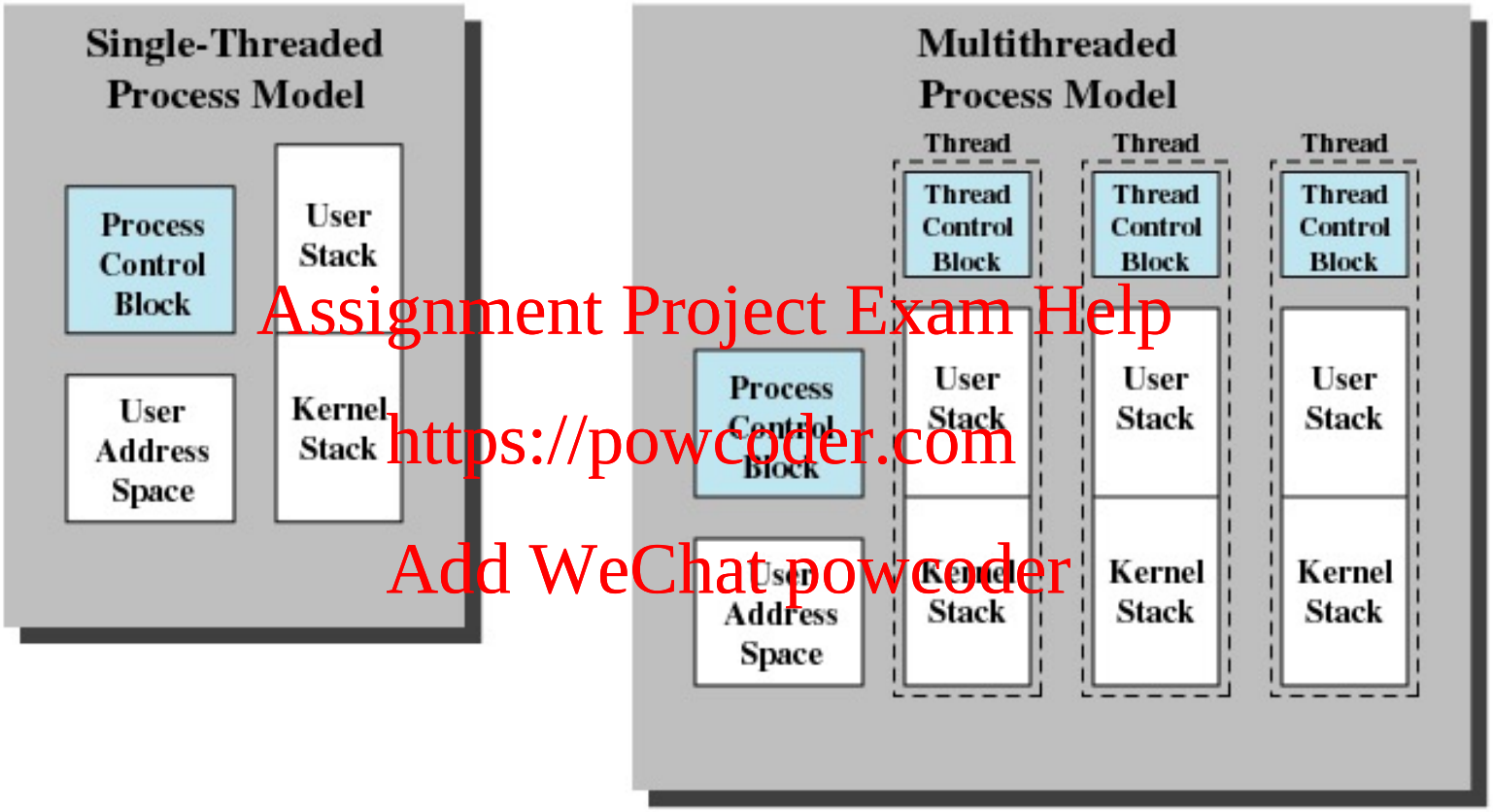


Figure 4.2 Single Threaded and Multithreaded Process Models

Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process
- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel

Threads

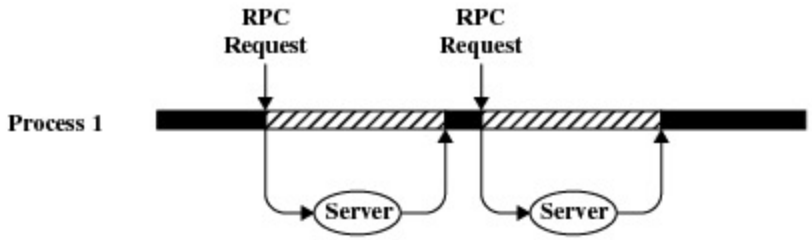
- Suspending a process involves suspending all threads of the process since all threads share the same address space
- Termination of a process, terminates all threads within the process

Thread States

- States associated with a change in thread state
  - Spawn
    - Spawn another thread
  - Block
  - Unblock
  - Finish
    - Deallocate register context and stacks

Example - Remote Procedure Call Using Single Thread

- The process has to wait two different servers sequentially, thus taking longer to complete.



(a) RPC Using Single Thread

Example: Remote Procedure Call Using Threads

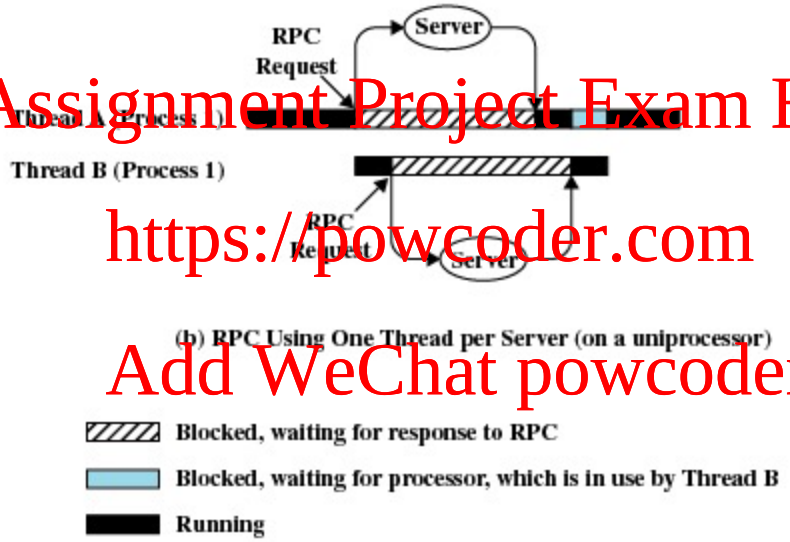


Figure 4.3 Remote Procedure Call (RPC) Using Threads

Multithreading

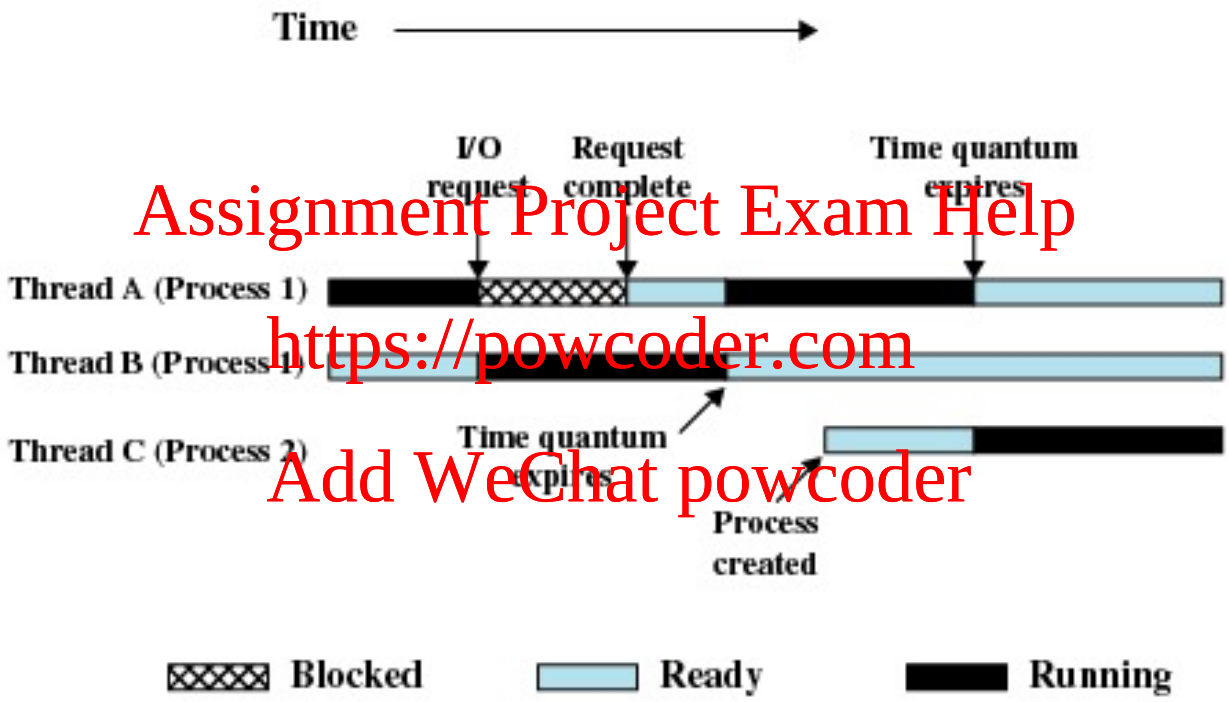
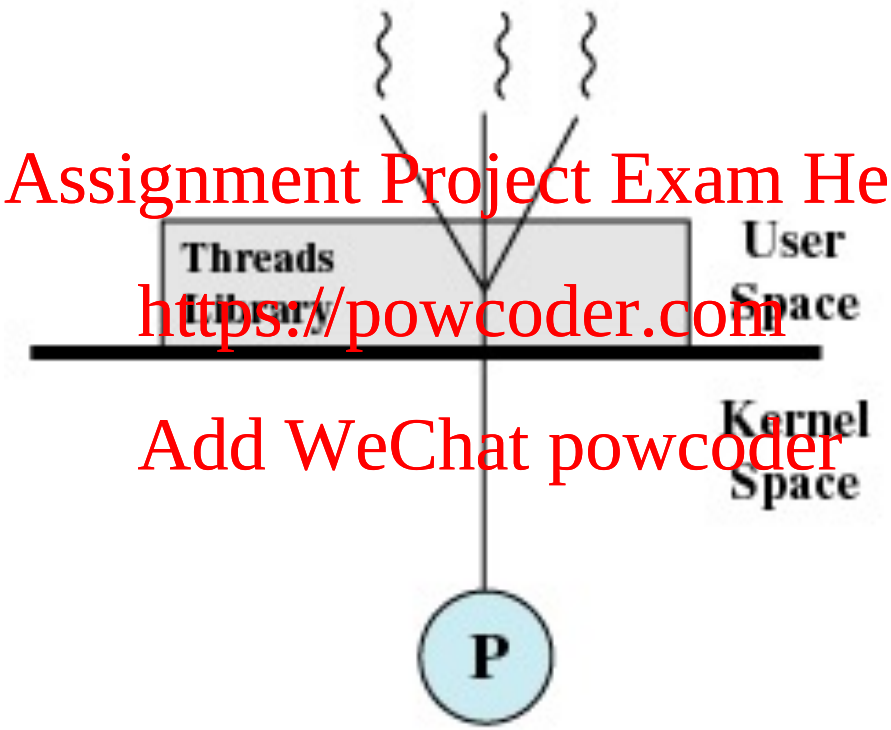


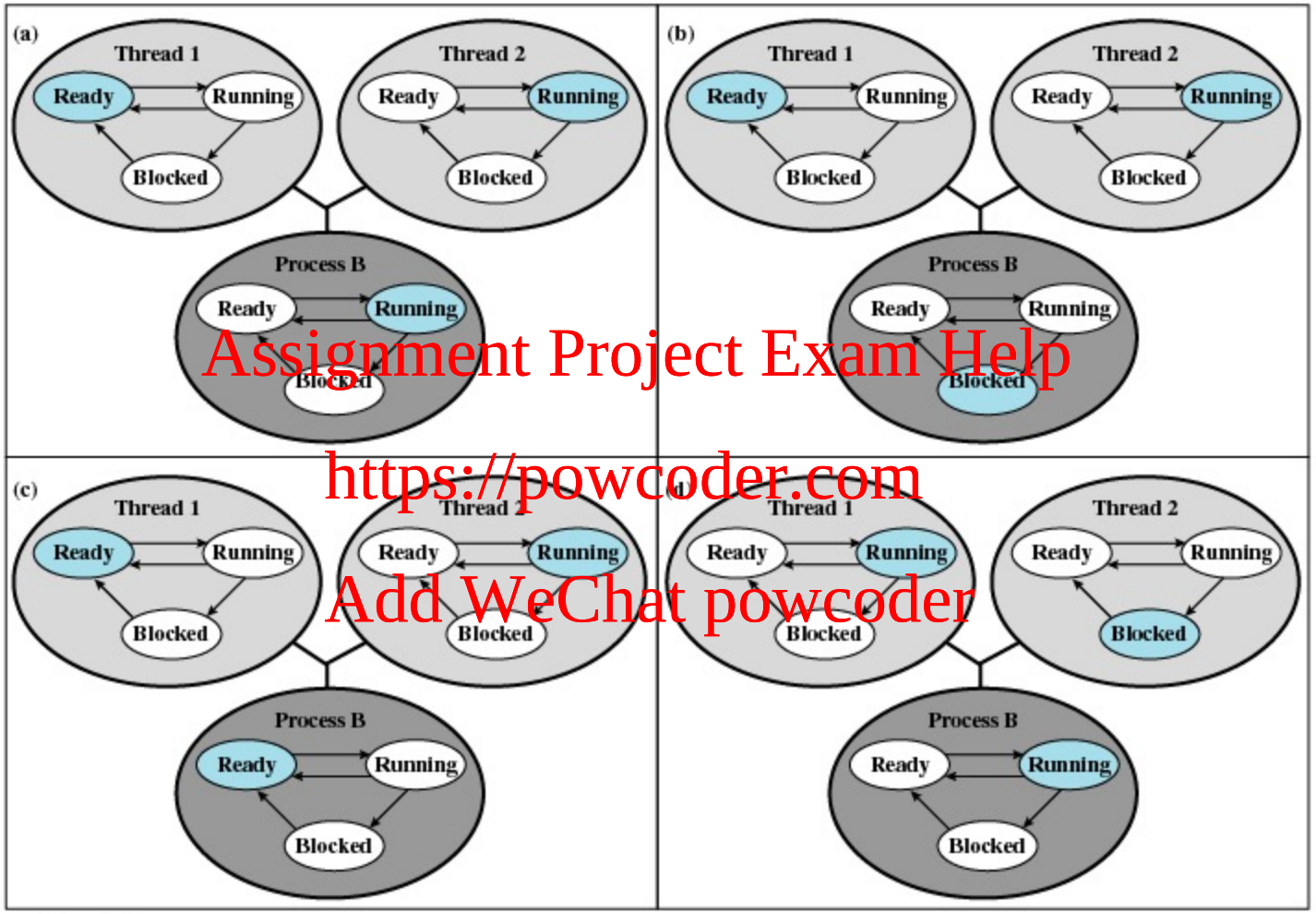
Figure 4.4 Multithreading Example on a Uniprocessor

User-Level Threads

- All thread management is done by the application
- The kernel is not aware of the existence of threads

User-Level Threads





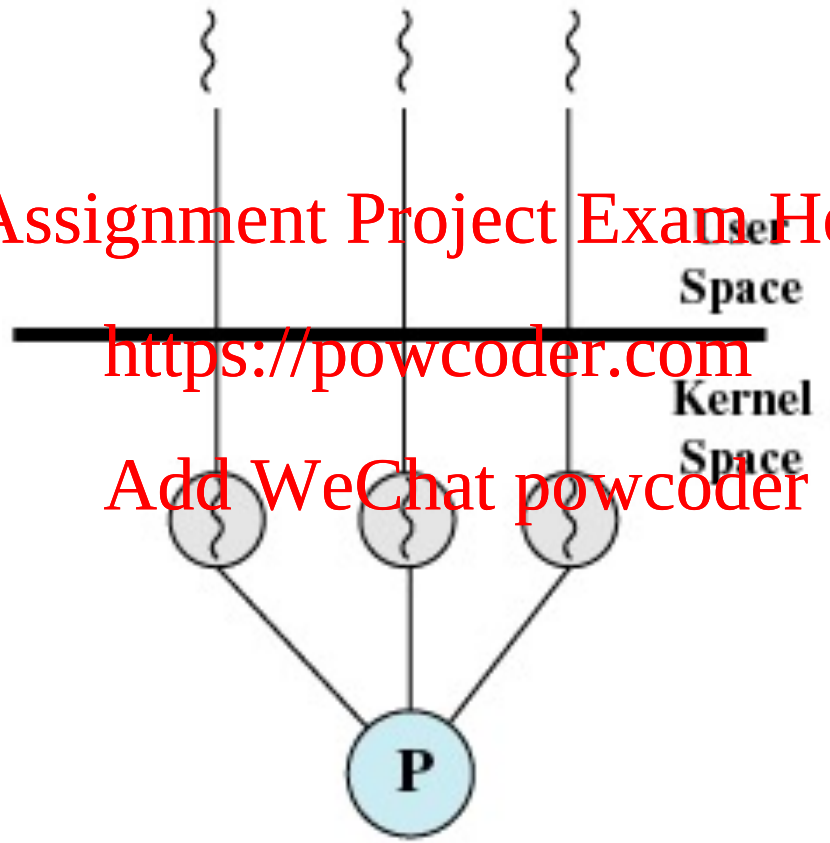
Colored state
is current state

Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

Kernel-Level Threads

- Windows is an example of this approach
- Kernel maintains context information for the process and the threads
- Scheduling is done on a thread basis

Kernel-Level Threads

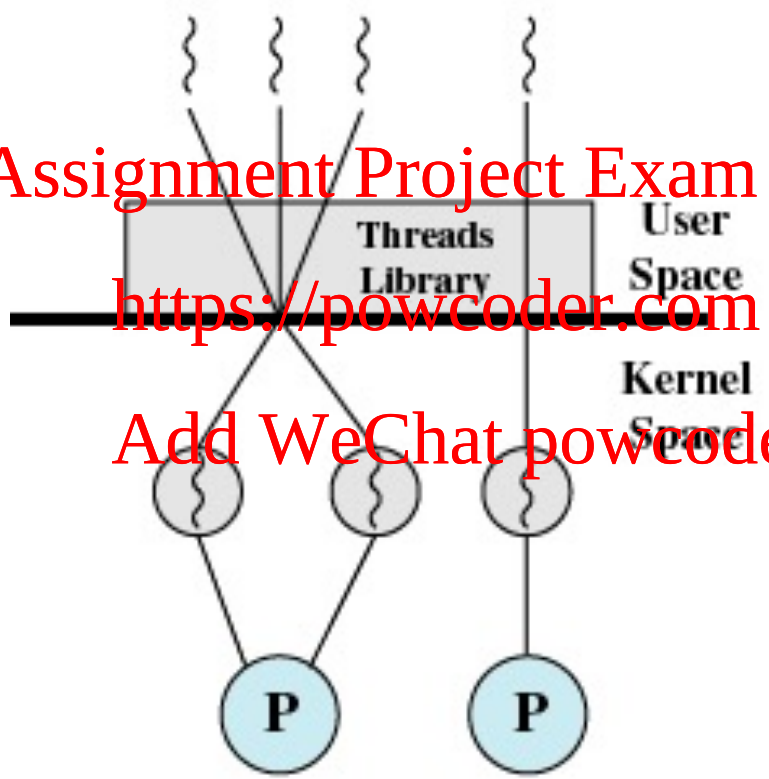


(b) Pure kernel-level

Combined Approaches

- Example is Solaris
- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads within application

Combined Approaches



(c) Combined

Relationship Between Threads and Processes

Table 4.2 Relationship Between Threads 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

Categories of Computer Systems - Flynn's taxonomy

- 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 the different processors

Categories of Computer Systems - Flynn's Taxonomy

- Multiple Instruction Single Data (MISD) stream
  - A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. Never implemented
- Multiple Instruction Multiple Data (MIMD)
  - A set of processors simultaneously execute different instruction sequences on different data sets

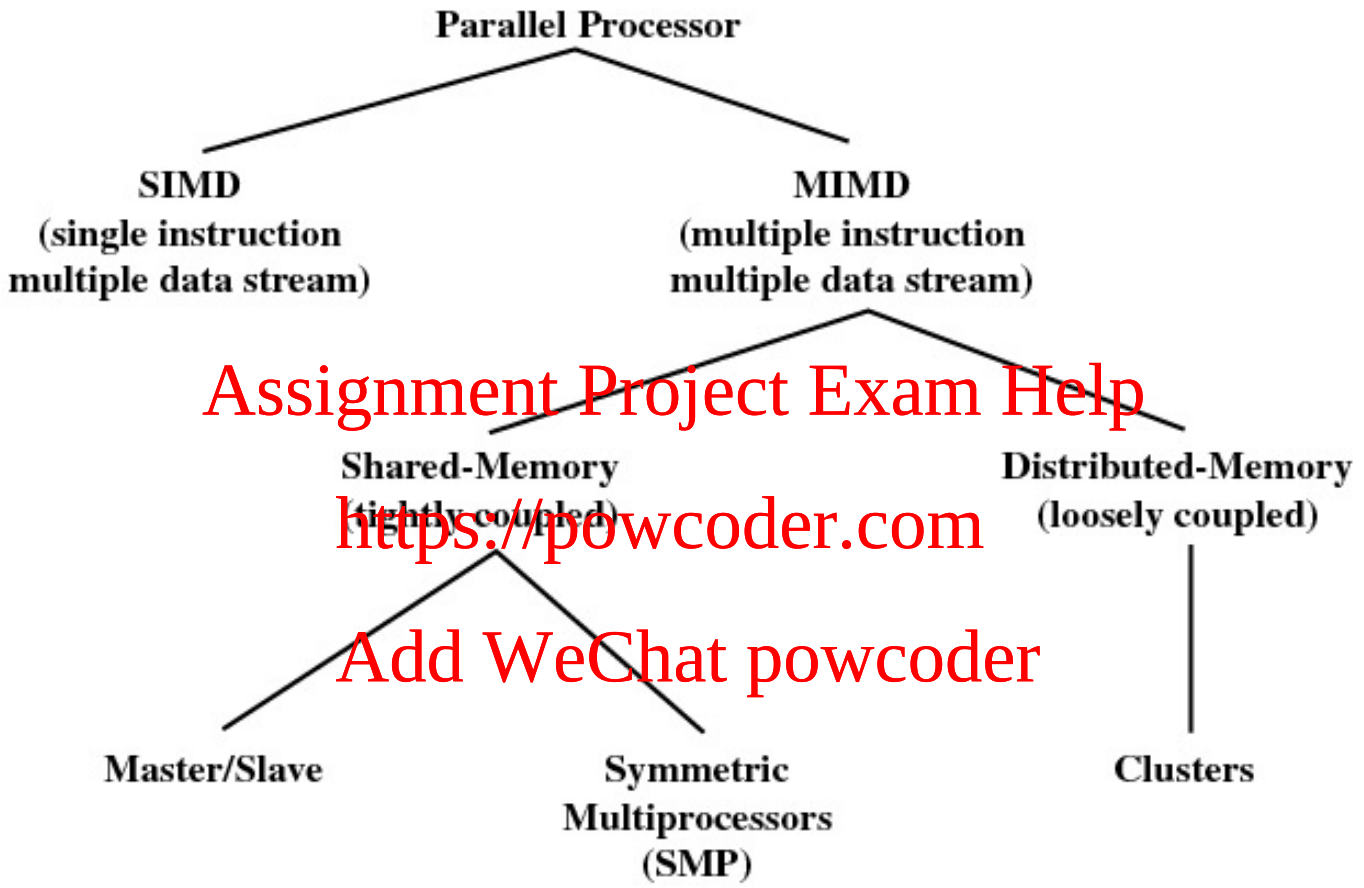


Figure 4.8 Parallel Processor Architectures

Symmetric Multiprocessing (SMP)

- Kernel can execute on any processor
- Typically, each processor does self-scheduling from the pool of available process or threads

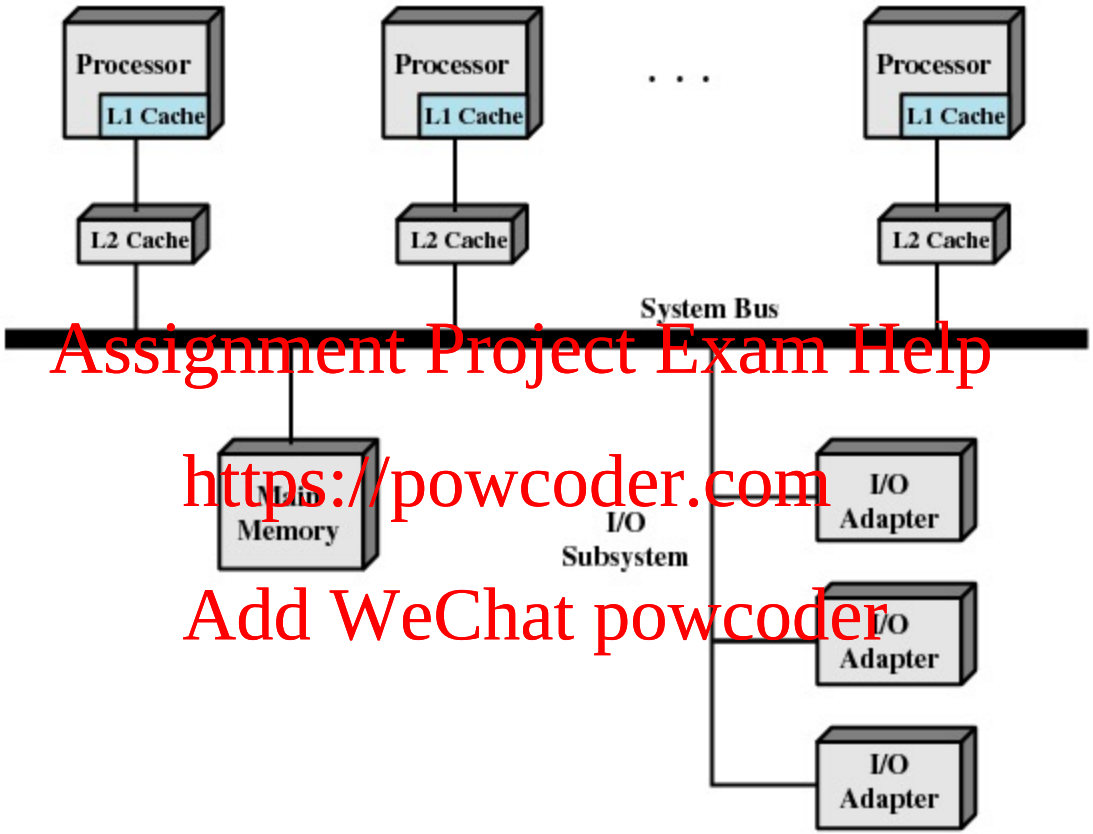


Figure 4.9 Symmetric Multiprocessor Organization

Parallel Computing

- Parallel Computing - Use multiple processors to solve *one* task.
- Need to split the program into multiple parts that can run on multiple processors *in parallel*.
- The main aim is to shorten the execution time.

Multiprocessor Operating System Design Considerations

- Simultaneous concurrent processes or threads
- Scheduling
- Synchronization
- Memory management
- Reliability and fault tolerance

Microkernels

- Small operating system core
- Contains only essential core operating systems functions
- Many services traditionally included in the operating system are now external subsystems
  - Device drivers
  - File systems
  - Virtual memory manager
  - Windowing system
  - Security services

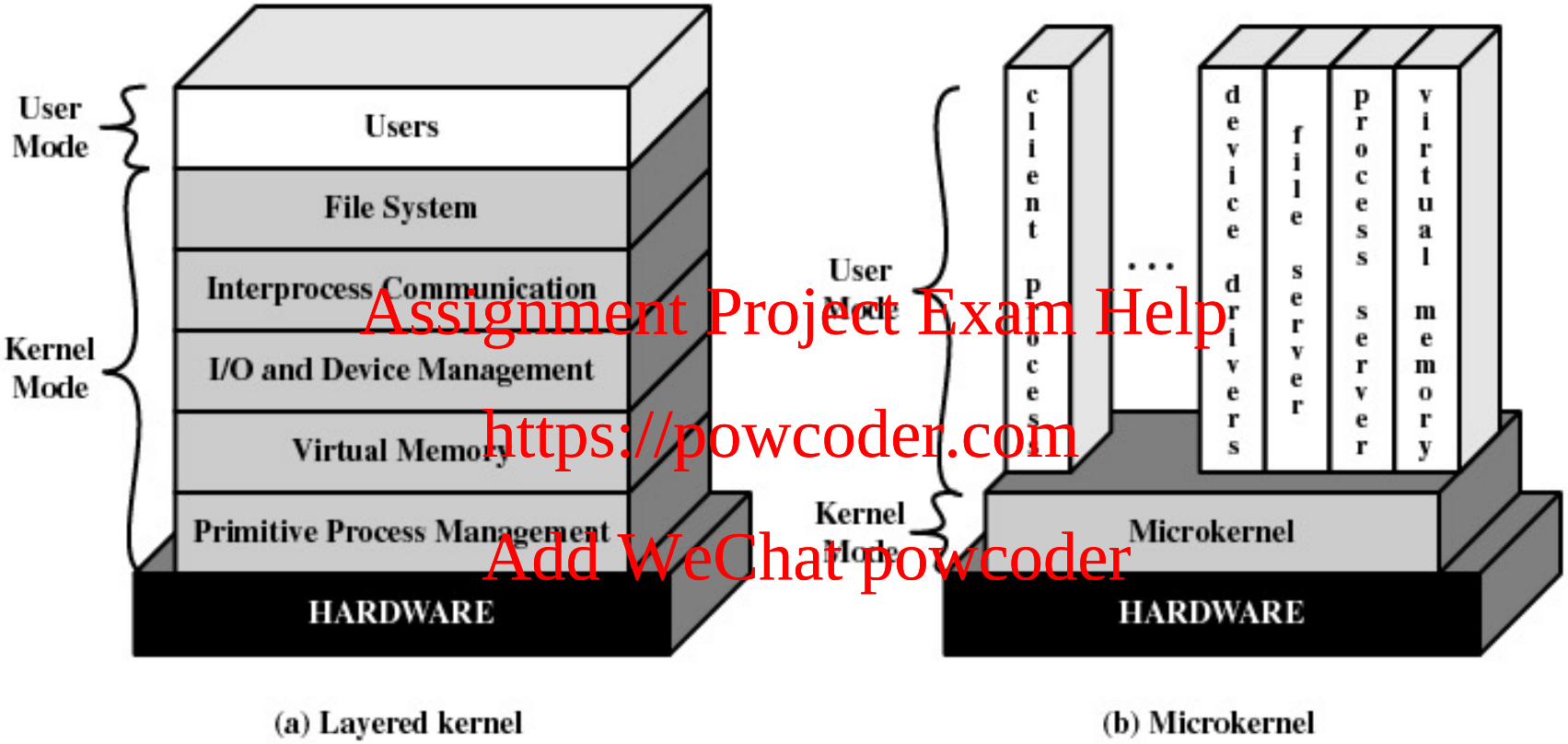


Figure 4.10 Kernel Architecture

Benefits of a Microkernel Organization

- Uniform interface on request made by a process
  - Don't distinguish between kernel-level and user-level services
  - All services are provided by means of message passing
- Extensibility
  - Allows the addition of new services
- Flexibility
  - New features added
  - Existing features can be subtracted

Benefits of a Microkernel Organization

- Portability
  - Changes needed to port the system to a new processor is changed in the microkernel - not in the other services
- Reliability
  - Modular design
  - Small microkernel can be rigorously tested

Benefits of Microkernel Organization

- Distributed system support
  - Messages are sent without knowing what the target machine is
- Object-oriented operating system
  - Components are objects with clearly defined interfaces that can be interconnected to form software

Microkernel: Examples

- Windows 2000, XP etc
- Mac OS X (darwin)
- Linux is a notable exception