K. J. Somaiya College of Engineering, Mumbai

(A constituent College of Somaiya Vidyavihar University)

Operating System

Module 2. Process Concept and scheduling

Dr. Prasanna Shete
Dept. of Computer Engineering
prasannashete@somaiya.edu
Mobile/WhatsApp 9960452937

Processes and Threads

- Processes have two characteristics:
- Resource ownership– ownership of resources like main memory, I/O channels/devices and files
  - process includes a virtual address space to hold the process image
- Scheduling/execution- Execution of process follows an execution path (trace) through one or more programs;
  - may be interleaved with other processes
- Process has execution state and dispatching priority

- These two characteristics are treated independently by the operating system

Somaiya TRUST

Processes and Threads

- The unit of dispatching is referred to as a ***thread*** (or lightweight process)

- The unit of resource ownership is referred to as a ***process*** (or ***task***)

Multithreading

- The ability of an OS to support multiple, concurrent paths of execution within a single process.

- 4 approaches (relationship between threads and processes)
  - 1 Process, 1 Thread (e.g MS-DOS)
  - 1 Process, Multiple Threads (JRE)
  - Multiple Processes, 1 Thread / Process (Unix)
  - Multiple Processes, Multiple Threads / Process (Linux, Windows, Solaris)

Multithreading

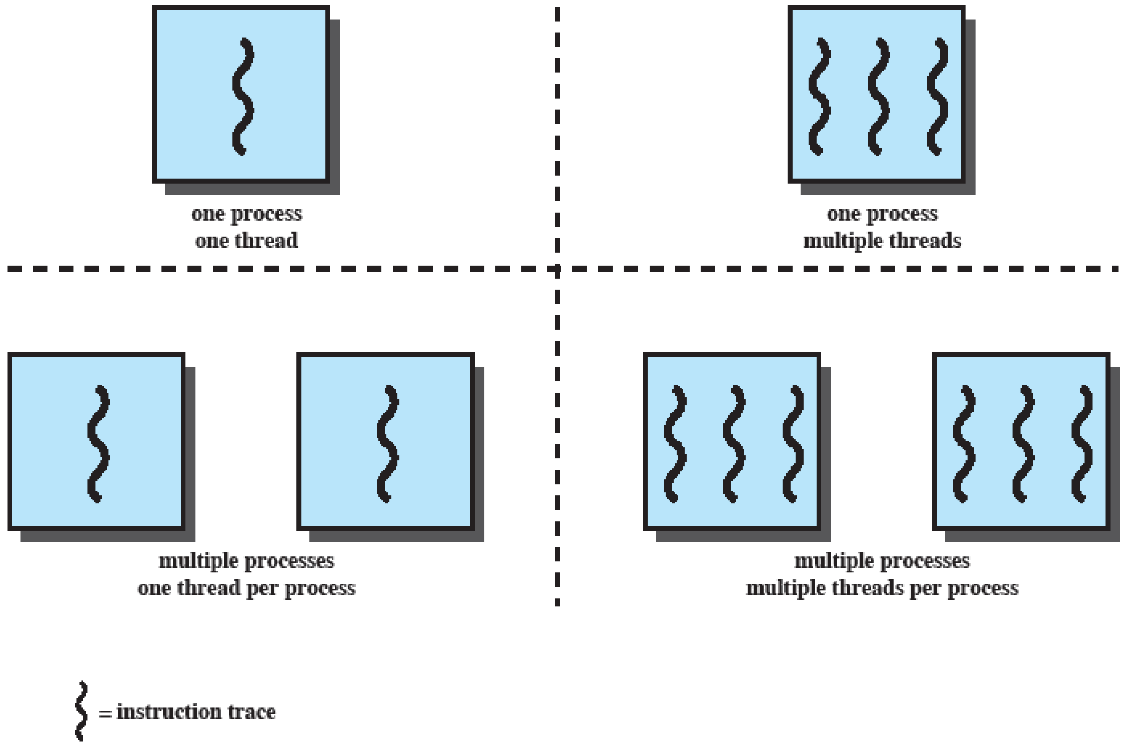


Figure 4.1 Threads and Processes [ANDE97]

Multithreading

- In multithreaded environment a Process is defined as the unit of resource allocation and a unit of protection;
  - following things are associated with it:
- A virtual address space that holds the process image
- Protected access to
  - Processors
  - Other processes
  - Files
  - I/O resources

Threads within a process

- Each thread has
  - An execution state (running, ready, etc.)
  - Saved thread context when not running

    (can be viewed as independent PC operating within a process)
  - An execution stack
  - Some per-thread static storage for local variables
  - Access to the memory and resources of its process (shared with all threads in that process)

Threads V/s processes

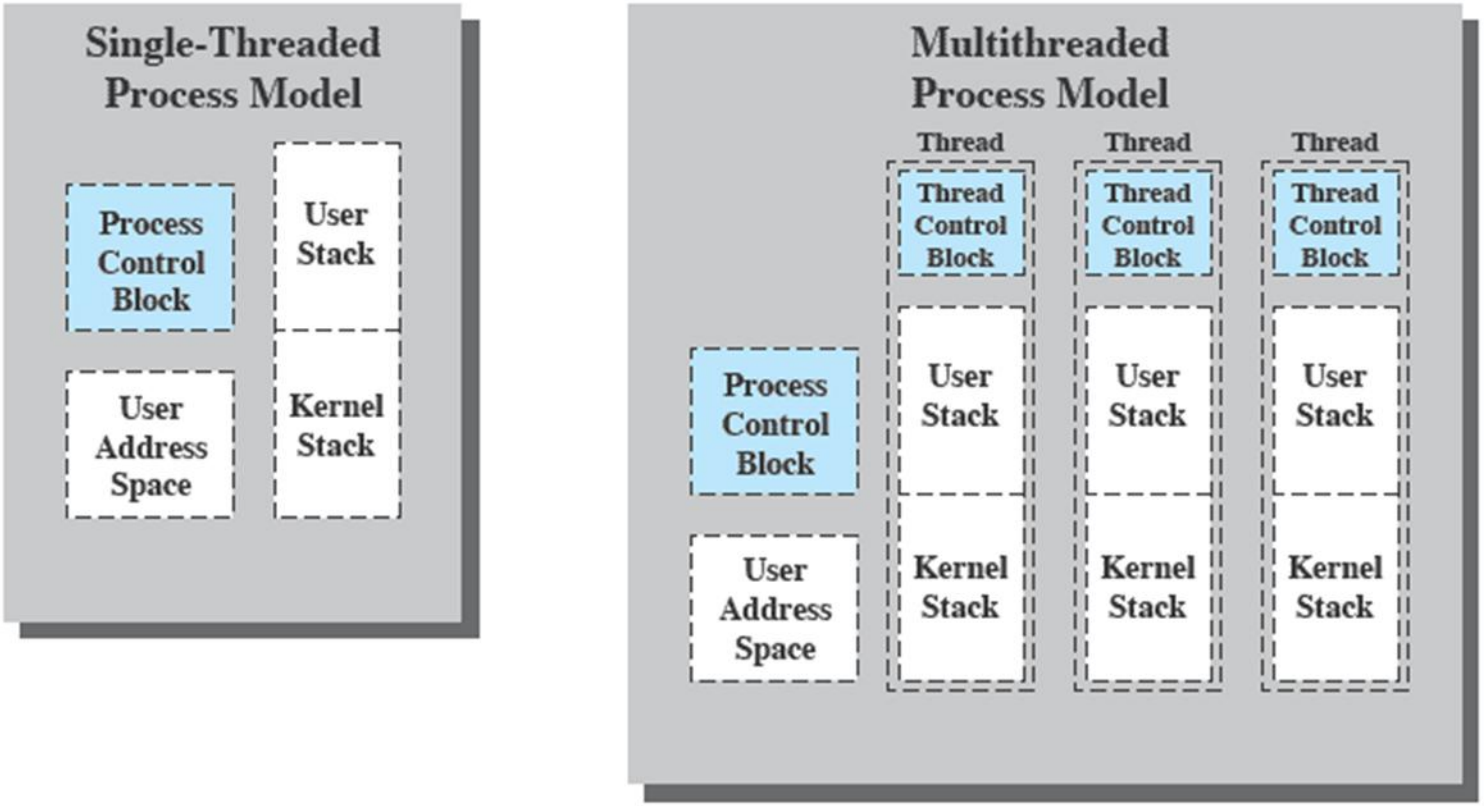


Figure 4.2 Single Threaded and Multithreaded Process Models

Distinction between threads and processes from the point of view of process management

- **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

Distinction between threads and processes from the point of view of process management

- **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

Distinction between threads and processes from the point of view of process management

- **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

Benefits to performance

- **Less creation time**- Takes less time to create a new thread than a process (10 times faster)
- **Less termination time**- time required to terminate a thread is less than that for process
- **Less Switching time**- Switching between two threads within the same process takes less time than switching processes
- **Efficiency enhancement in communication between different executing programs**
  – Threads within a process share memory and files, thus can communicate with each other without invoking the kernel

Example scenarios of Threads in Single-User System

- Foreground and background work
  - Spreadsheet: One thread displays menu, other executes user commands
- Asynchronous processing
  - Power failure protection; write contents of RAM to disk every 100ms
- Speed of execution
  - Compute one batch of data while reading next batch; multi-processor scenario
- Modular program structure

Thread Functionality

- Thread States- Running, Ready, Blocked; Suspended state is not associated with threads
  - If a process is suspended, all its threads are suspended
- Thread operations:
  – Spawn (another thread)
  – Block
    - Issue: will blocking a thread block other, or *all,* threads
  – Unblock
  – Finish (thread)
    - Deallocate register context and stacks

Multithreading on a Uniprocessor

- Multithreading enables interleaving of multiple threads within multiple processes

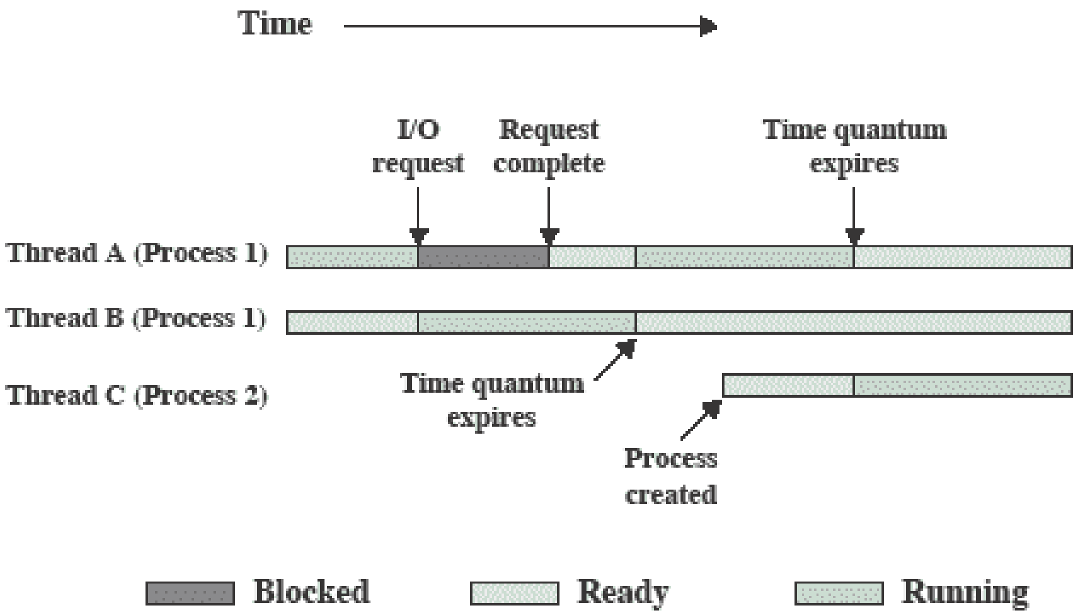


Figure 4.4 Multithreading Example on a Uniprocessor

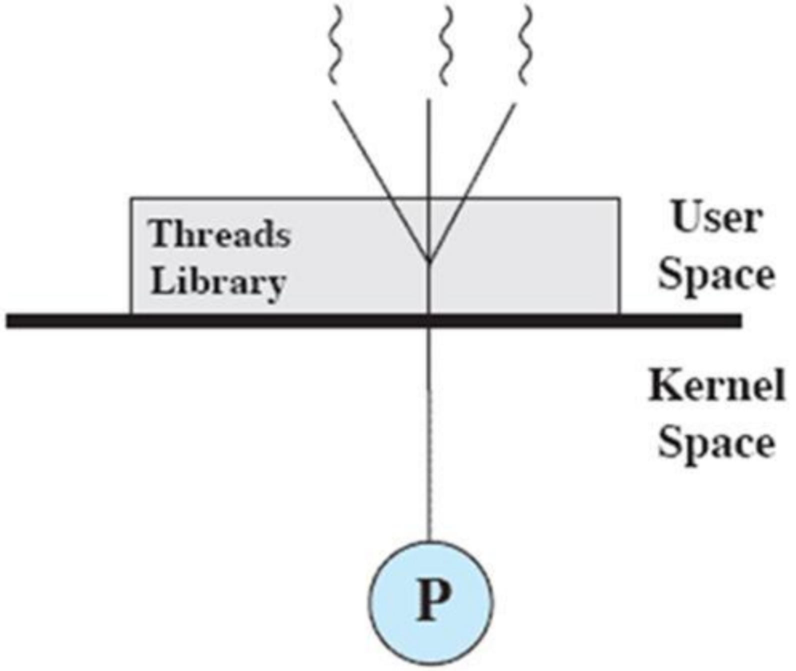
Thread Implementation: Categories

- User Level Thread (ULT)
- Kernel level Thread (KLT) also called:
  - kernel-supported threads
  - lightweight processes

Thread Implementation: ULTs

- All thread management done by the application
  - The kernel is not aware of the existence of threads
  - Threads Library: package of application level routines for thread management

- Kernel schedules process as unit



(a) Pure user-level

ULTs

- Threads Library: package of application level routines for thread mngmentsin
  - contains the code for creating and destroying threads, for passing messages and databetween threads, for scheduling threads and for saving and restoring thread contexts
- Application begins with single thread and starts running in that thread
- The application and all its threads are allocated to a single process managed by the kernel
- When the application is running it may spawn a new thread to run within the same process

ULTs

- Threads Spawing- is carried out by invoking the *spawn utility* in the threads library; Control passed to that utility by a procedure call
- Threads library creates the data structure for the new thread and then passes control to one of the threads that is in the Ready state using some scheduling algorithm

- When control is passed to the library, the context of current thread is saved; and restored when control is passed back to the thread
- These activities take place in the user space; kernel is unaware
- Kernel continues to schedule the process as a unit

Relationships between ULT Thread and Process States

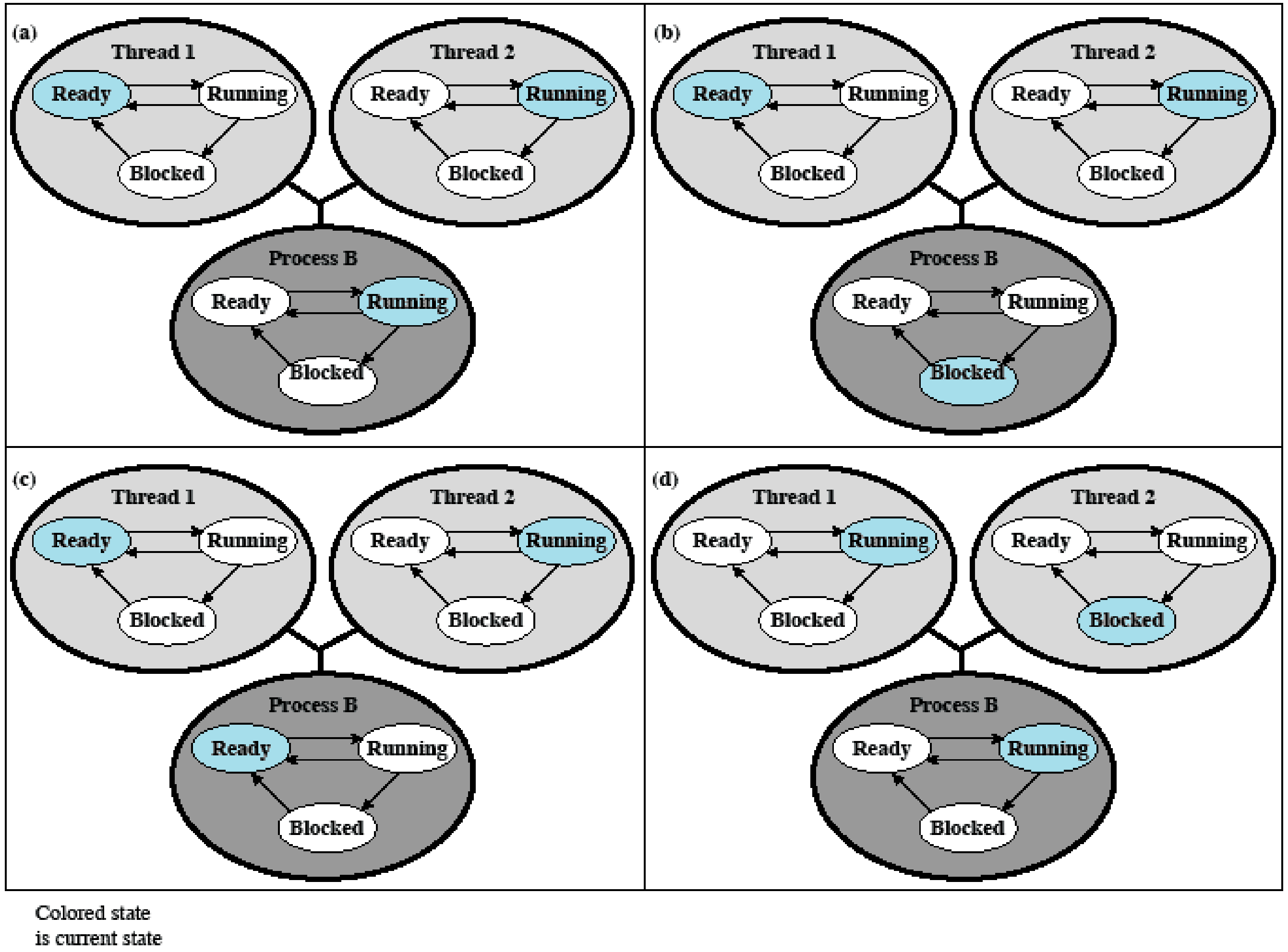


Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

Relationships between ULT Thread and Process States

- Suppose Process B has 2 threads and is executing in Thread 2
  1. Process B blocked: During execution the application makes a system call that blocks process B →kernel switches control to another process
     - Thread 2 will still be in the running sate- not actually executing on the processor- but perceived as being in the running state by the threads library
  2. Clock interrupt→ process B moved to ready state by kernel; switches control to another process
     - Data structures maintained by threads library indicate that thread 2 is in running state
  3. Thread blocked: Thread 2 needs some action to be performed by thread 1,→ Thread 2 is blocked; thread 1 transitions from Ready to running state
     - Data structures maintained by threads library are updated

ULT Advantages

- Thread switching does not need Kernel level privileges
  - Thread management data structures are within user space of process
- Scheduling can be application specific
  - Scheduling algorithms can be tailored to the application without disturbing underlying OS scheduler
- ULTs can run on any OS
  - No changes in the kernel are required to support ULT

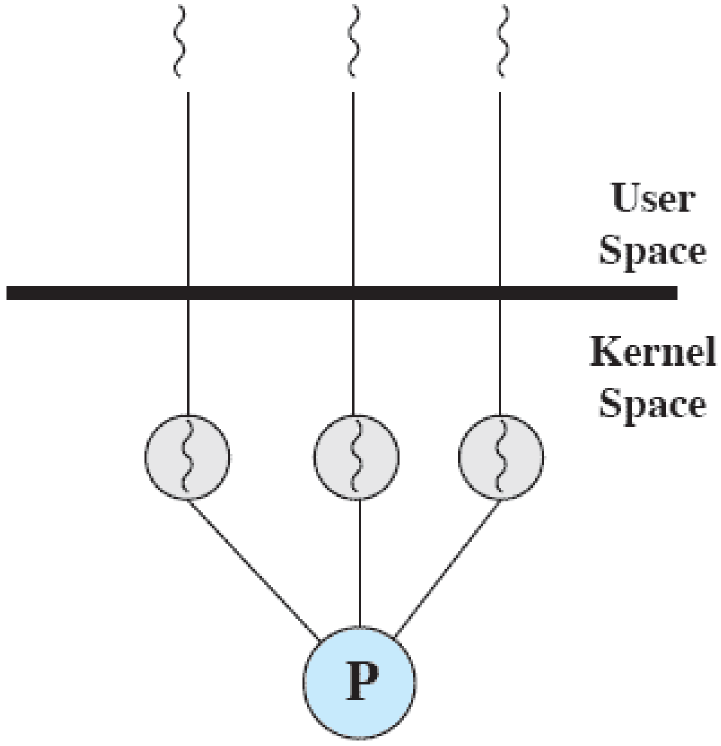
ULT Disadvantages

- Multi threaded applications cannot take advantage of multiprocessing
- When one thread blocks on a system call, all threads within that process are blocked
- Remedy:
- Write application with multiple processes rather than threads
  - → eliminates the advantage of threads
- Jacketing- Converts a blocking system call into non-blocking system call
  - Rather than directly calling the system I/O routine, thread calls an application level I/O jacket routine; checks the availability of I/O

Thread Implementation: KLTs

- Kernel maintains context information for the process and the threads
  - No thread management done by application

- Scheduling is done on a thread basis
  - Windows is an example of this approach



(b) Pure kernel-level

Advantages of KLT

- Kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines themselves can be multithreaded

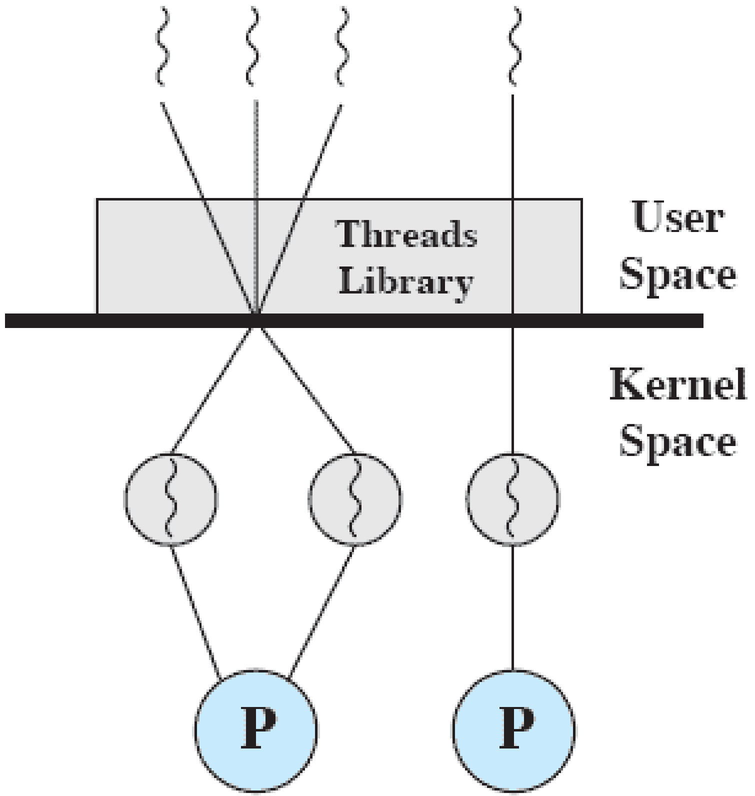
- Disadvantages
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Somaiya TRUST

Thread Implementation: Combined Approach

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads by the application
- Multiple ULTs from single application are mapped onto smaller (or equal) number of KLTs
- Multiple threads within same application can run in parallel on multiple processors
  - Blocking system call need not block the entire process

- Example: Solaris



(c) Combined

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

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

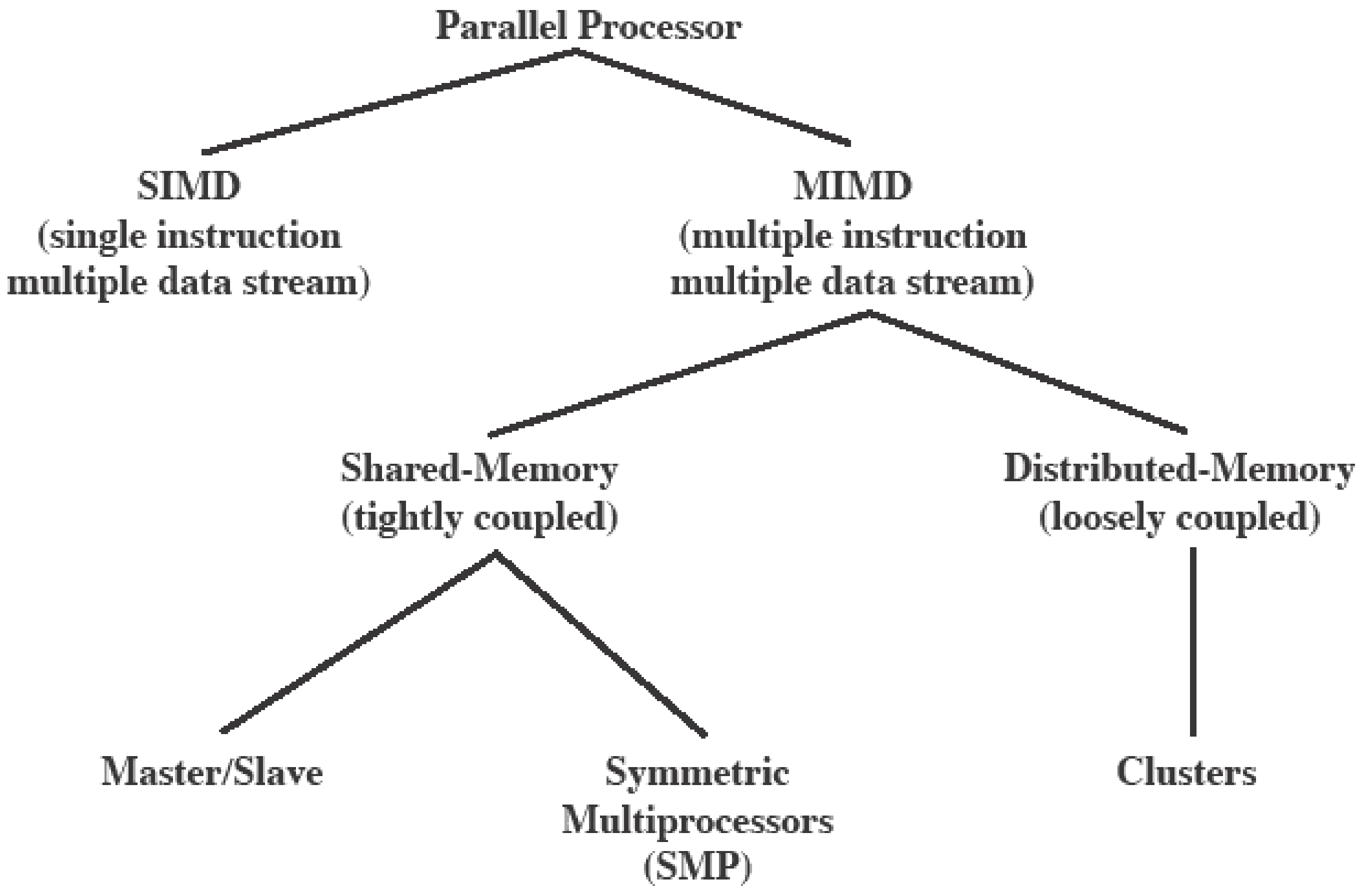


Figure 4.8 Parallel Processor Architectures

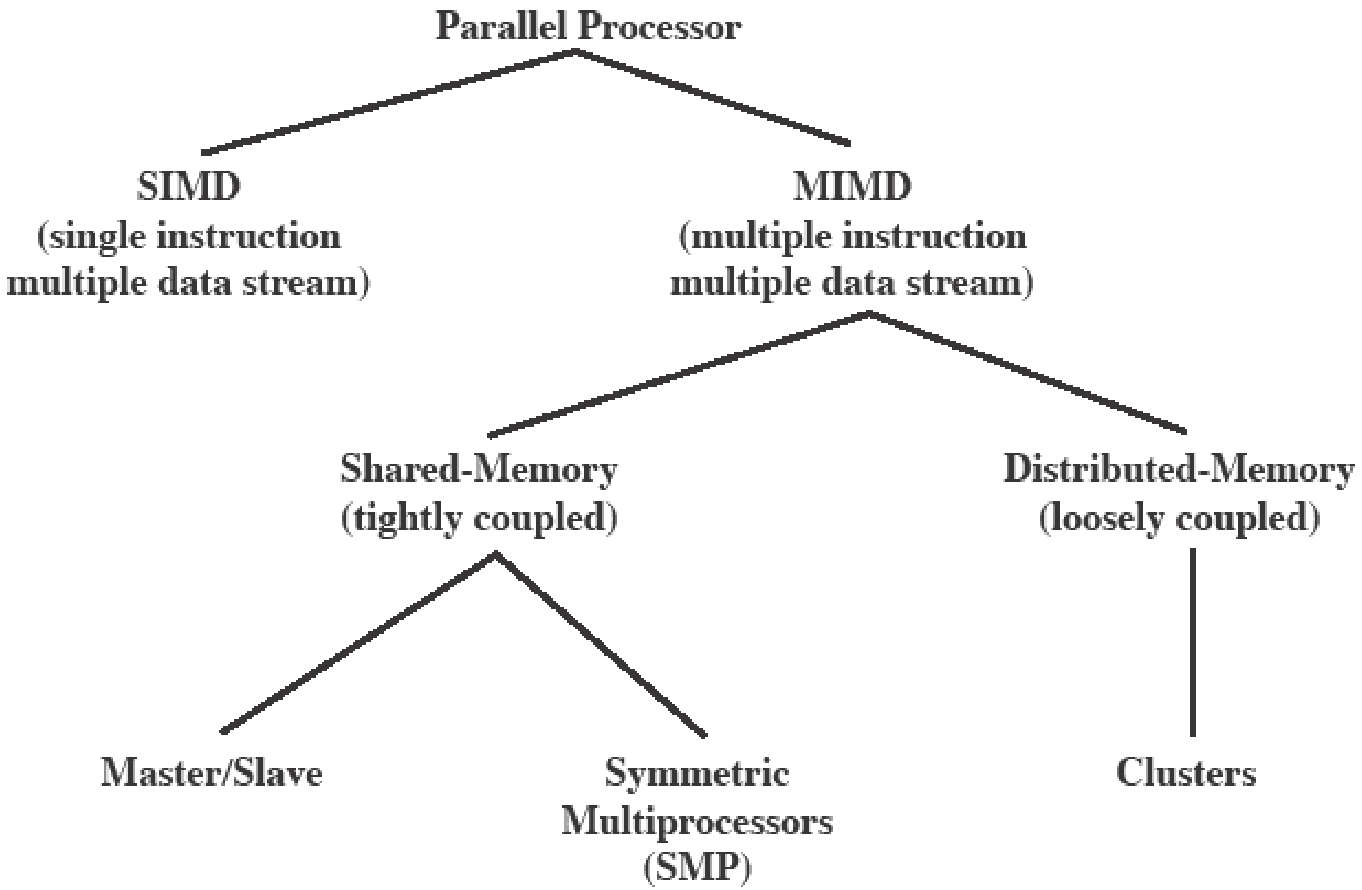


Figure 4.8 Parallel Processor Architectures

Symmetric Multiprocessing

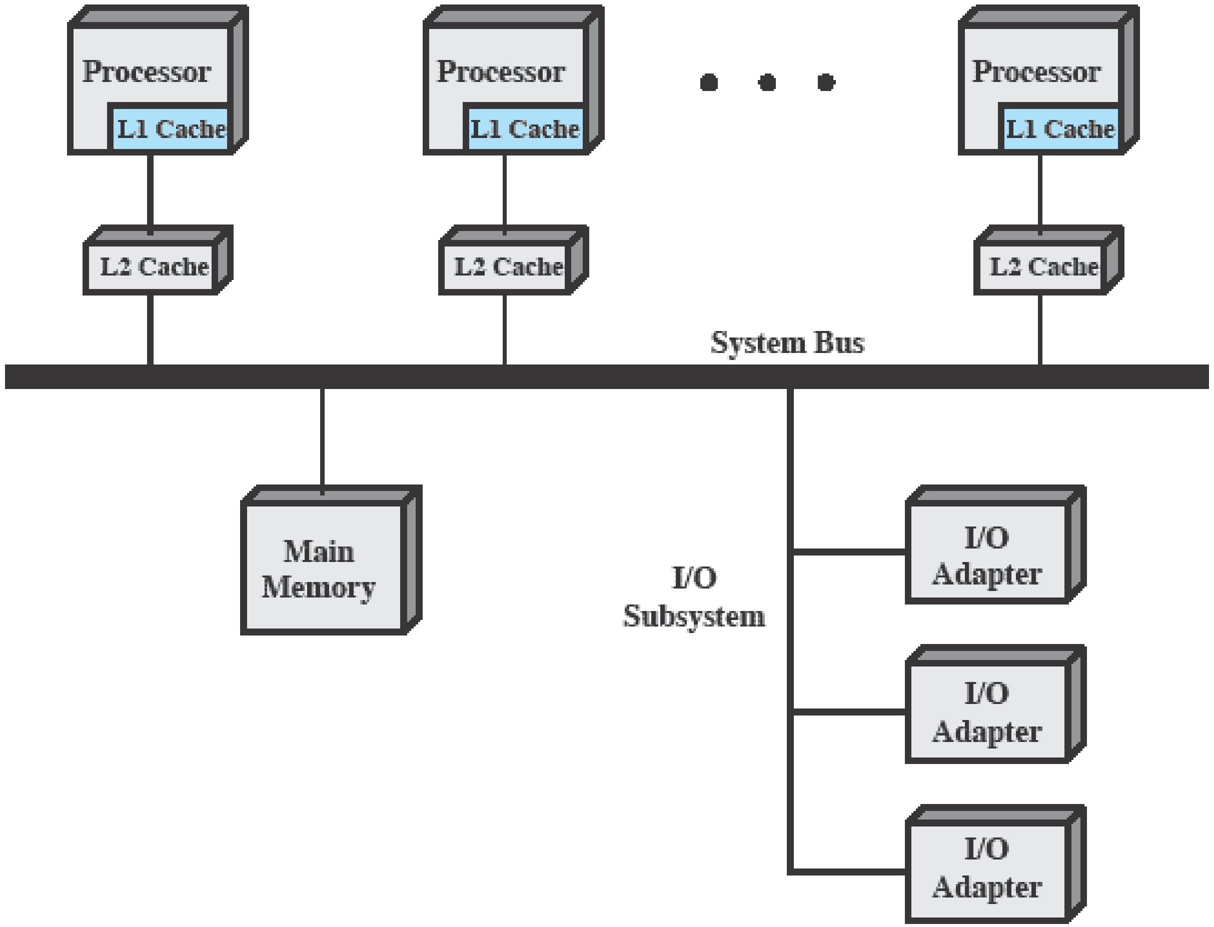


Figure 4.9 Symmetric Multiprocessor Organization

Symmetric Multiprocessing

- In a symmetric multiprocessor, Kernel can execute on any processor
  - Each processor does self-scheduling from the pool of available process or threads (typically)

- The kernel can be constructed as multiple processes or multiple threads, allowing portions of the kernel to execute in parallel

- Complicates the OS
  - must ensure that two processors do not choose the same process and that processes are not somehow lost from the queue.
  - Techniques must be employed to resolve and synchronize claims to resources

Multiprocessor OS Design Considerations

- The key design issues include
  - Simultaneous concurrent processes or threads
  - Scheduling
  - Synchronization
  - Memory Management
  - Reliability and Fault Tolerance

Multiprocessor OS Design Considerations

1. **Simultaneous concurrent processes or threads:**
   - Kernel routines need to be re-entrant- to allow several processors to execute the same kernel code simultaneously
   - With multiple processors executing the same or different parts of the kernel, kernel tables and management structures must be managed properly to avoid deadlock or invalid operations

2. **Scheduling:**
   - Scheduling may be performed by any processor, so conflicts must be avoided
   - If kernel-level multithreading is used, then there is opportunity to schedule multiple threads from the same process simultaneously on multiple processors

Multiprocessor OS Design Considerations

3. Synchronization:

- With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective synchronization
- Synchronization is a facility that enforces mutual exclusion and event ordering
    - A common synchronization mechanism used in multiprocessor operating systems is locks

Somaiya TRUST

Multiprocessor OS Design Considerations

4. **Memory management:**
   - Must deal with all of the issues found on uniprocessor computers
   - OS needs to exploit the available hardware parallelism, such as multiported memories, to achieve the best performance
   - The paging mechanisms on different processors must be coordinated to enforce consistency when several processors share a page or segment and to decide on page replacement
5. **Reliability and fault tolerance:**
   - OS should provide graceful degradation in the case of processor failure
   - The scheduler and other portions of the OS must recognize the loss of a processor and restructure management tables accordingly

Multiprocessor Scheduling Design Issues

- Scheduling on a multiprocessor involves 3 interrelated issues:
  - Assignment of processes to processors
  - Use of multiprogramming on individual processors
  - Actual dispatching of a process

- Approach taken depends on the degree of granularity of applications and the number of processors available

Assignment of Processes to Processors

- Assuming all processors are equal, treat processors as a pooled resource and
- assign process to processors on demand
  - Should the assignment be static or dynamic?

- Dynamic Assignment:
  - threads are moved from a queue for one processor to a queue for another processor

Assignment of Processes to Processors

- Static Assignment:
- Permanently assign of process/thread to a processor
  - Dedicated short-term queue for each processor
  - Less overhead
  - Allows the use of ‘group’ or ‘gang’ scheduling (see later)
- May result in processor remaining idle; while others have a backlog/ pending work
  - Solution: use a common queue
  - All processes go into one global queue and are scheduled on to any available processor

Assignment of Processes to Processors

- Both dynamic and static methods require some way of assigning a process to a processor
- Two methods:
  - Master/Slave
  - Peer-to-peer

- There are a spectrum of approaches between these two extremes

Assignment of Processes to Processors:

- Master / Slave Architecture
  - Key kernel functions always run on a particular processor- Master
  - Master is responsible for scheduling
  - Slave sends service request to the master

  Disadvantages
  - Failure of master brings down whole system
  - Master can become a performance bottleneck
- Peer Architecture:
  - Kernel can execute on any processor
  - Each processor does self-scheduling
  - Complicates the operating system
    - Need to make sure two processors do not choose the same process

Process Scheduling

- Usually processes are not dedicated to processors
- A single queue is used for all processes
- Or multiple queues are used for differing priorities
  - All queues feed to the common pool of processors

Thread Scheduling

- Threads execute separate from the rest of the process
- An application can be a set of threads that cooperate and execute concurrently in the same address space
  - Dramatic gains in performance in multi-processor systems, compared to running in uniprocessor system
- Four general approaches:
  – Load Sharing
  – Gang Scheduling
  – Dedicated Processor assignment
  – Dynamic scheduling

Thread Scheduling: Load Sharing

- Threads are not assigned to a particular Processor
- Global queue of ready threads is maintained
- Each processor, when idle, selects a thread from the queue
- Advantages:
  - Simple Approach
  - Load is distributed evenly across the processors
  - No centralized scheduler required
    - whenever a processor is available, scheduling routine of OS runs on that processor to select the next thread
  - The global queue (of tasks) can be organized and accessed using any scheduling algorithm

Thread Scheduling: Load Sharing

- 3 different versions of load sharing approach
  - – FCFS
    - Each thread of a job is placed consecutively at the end of shared queue
    - When processor becomes idle, it picks up next ready thread and executes it till completion (or blocking)
  - – Smallest Number of Threads First
    - Priority based shared queue; highest priority to threads of that job which has least no. of threads
  - – Pre-emptive Smallest Number of Threads First
    - Job with smallest no. of threads than the job which is already in execution, will pre-empt the threads of scheduled job

Thread Scheduling: Load Sharing

- Disadvantages:
- Central queue resides in the region of memory that needs to be accessed in a mutually exclusive manner
  - Can lead to bottlenecks
- Preemptive threads are unlikely to resume execution on the same processor
  - Caching becomes less efficient
- If all threads are in the global queue, all threads of a given program will not gain access to the processors at the same time

Thread Scheduling: Gang Scheduling

- Scheduling set of threads simultaneously on a set of processors
  - A set of related threads/processes is scheduled to run on a set of processors at the same time
- Parallel execution of closely related processes may reduce overhead such as: process switching and synchronization blocking
  - Improves Performance
- Gang scheduling is usually applied to simultaneous scheduling of threads that make up a single process
  - Used for medium to fine grained parallel applications

Somaiya TRUST

Thread Scheduling: Gang Scheduling

- Involves Following activities:
  1. Group of related threads are scheduled as a unit (or gang)
  2. All members of gang run simultaneously on different processors
  3. All members start and end their time slices together

- Gang scheduling works because all CPUs are scheduled synchronously, by dividing time into discrete quanta

Thread Scheduling: Dedicated Processor Assignment

- Opposite to load sharing
- Implicit scheduling defined by Assignment of threads to processors
- Each program is allocated with
- Number of Processors = No. of threads in the program for the duration of program execution
  – When program terminates processors return to general pool; these processors can be allocated to other programs

- Some processors may remain idle
- No multiprogramming of processors

Thread Scheduling: Dynamic Scheduling

- Number of threads in a process may be/are altered dynamically by the application during the course of execution
  - By using some language/ system tools; e.g a thread giving birth to many child threads
- Dynamic Scheduling allows the OS to adjust the load to improve utilization/performance
- OS as well as application are involved in scheduling decisions