Threads

Threads

- A thread is a flow of execution through the process code with its own program counter that keeps track of which instruction to execute next, system registers which holds its current working variables and a stack which contains the execution history.
- Thread is also called lightweight process.
- Threads provide a way to improve application performance through parallelism.

Threads

- **Each thread belongs to exactly one process and no thread can exist outside a process.**

- Each Thread represents separate flow of control.

Process Characteristics

- **Unit of resource ownership - process is allocated:**
  - a virtual address space to hold the process image
  - control of some resources (files, I/O devices...)
- **Unit of dispatching - process is an execution path through one or more programs**
  - execution may be interleaved with other process
  - the process has an execution state and a dispatching priority

Process Characteristics

- These two characteristics are treated independently by some recent OS
- The unit of dispatching is usually referred to a thread or a lightweight process
- The unit of resource ownership is usually referred to as a process or task

Threads vs. Processes

- A thread has no data segment or heap
- Thread is lightweight, taking lesser resources.
- A thread cannot live on its own, it must live within a process
- Inexpensive creation
- Inexpensive context switching, does not need to interact with operating system.
- If a thread dies, its stack is reclaimed.
- Multiple threaded processes use fewer resources.

- A process has code/data/heap & other segments.
- Process is heavyweight or resource intensive.
- There must be at least one thread in a process
- Expensive creation
- Expensive context switching, needs interaction with operating system.
- If a process dies, its resources are reclaimed & all threads die.
- Multiple process without using threads use more resources.

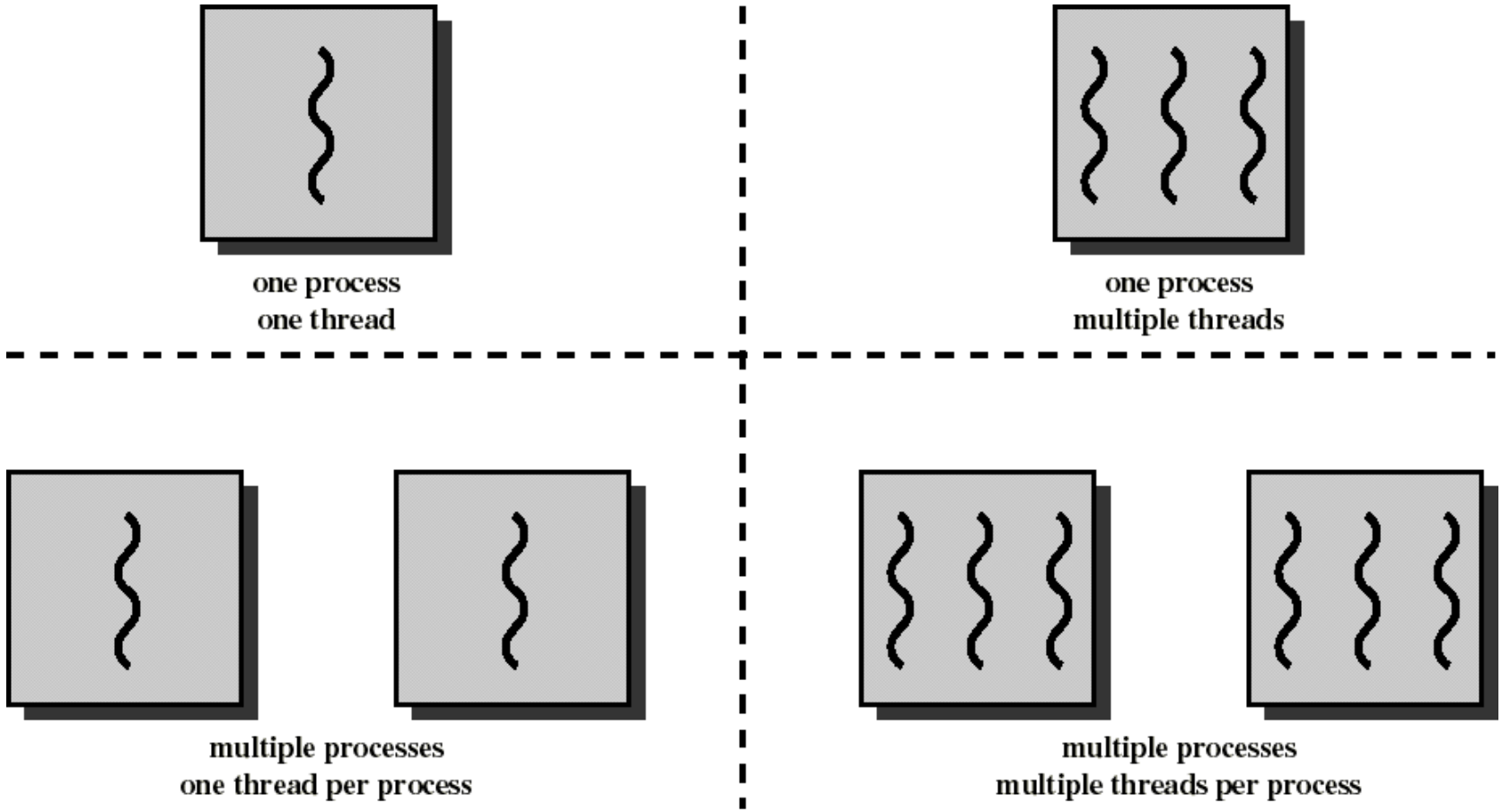
Threads

- Thread minimize the context switching time.
- **Use of threads provides concurrency within a process.**
- **Efficient communication.**
- **It is more economical to create and context switch threads.**
- **Threads allow utilization of multiprocessor architecture to a greater scale and efficiency.**

Multithreading vs. Single threading

- **Multithreading: when the OS supports multiple threads of execution within a single process**
- **Single threading: when the OS does not recognize the concept of thread**
- **MS-DOS support a single user process and a single thread**
- **UNIX supports multiple user processes but only supports one thread per process**
- **Solaris supports multiple threads**

Threads and Processes



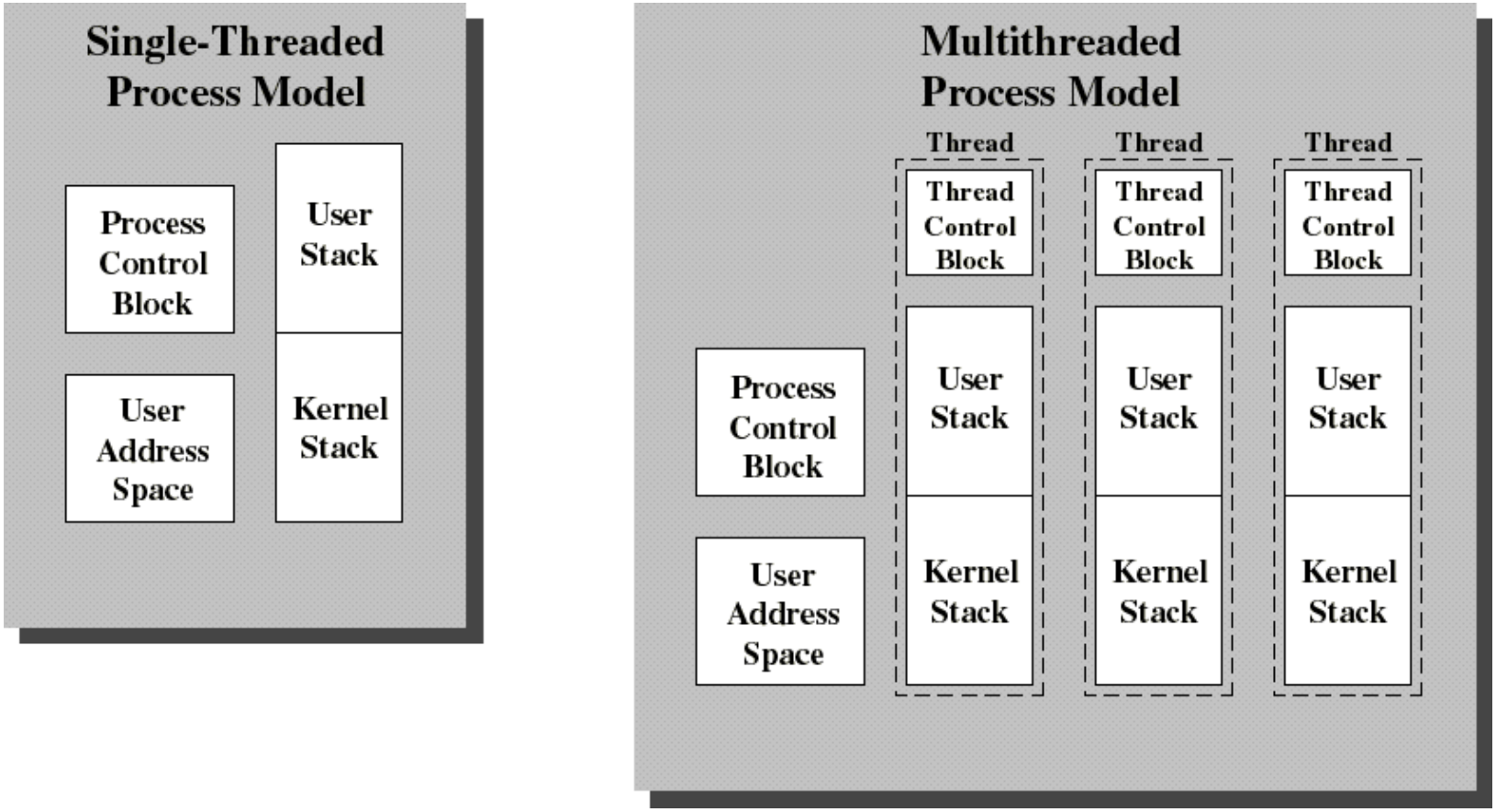
Processes

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

Threads

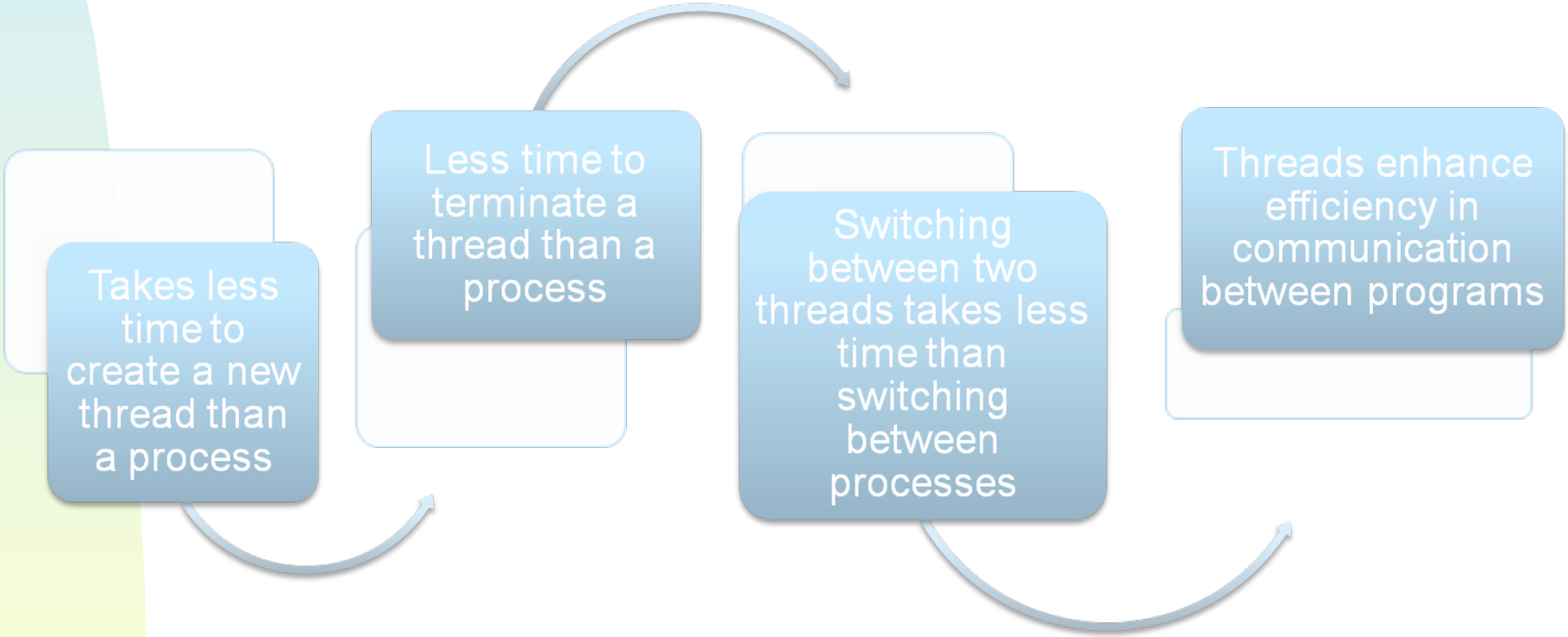
- **Has an execution state (running, ready, etc.)**
- **Saves thread context when not running**
- **Has an execution stack and some per-thread static storage for local variables**
- **Has access to the memory address space and resources of its process**
  - all threads of a process share this
  - when one thread alters a (non-private) memory item, all other threads (of the process) sees that
  - a file open with one thread, is available to others

Single Threaded and Multithreaded Process Models



Thread Control Block contains a register image, thread priority and thread state information

Benefits of Threads



Benefits of Threads

- Example: a file server on a LAN
- It needs to handle several file requests over a short period
- Hence more efficient to create (and destroy) a single thread for each request
- On a SMP machine: multiple threads can possibly be executing simultaneously on different processors
- Example2: one thread display menu and read user input while the other thread execute user commands

Application benefits of threads

- Consider an application that consists of several independent parts that do not need to run in sequence
- Each part can be implemented as a thread
- Whenever one thread is blocked waiting for an I/O, execution could possibly switch to another thread of the same application (instead of switching to another process)

Benefits of Threads

- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel
- Therefore necessary to synchronize the activities of various threads so that they do not obtain inconsistent views of the data

Thread Synchronization

- **It is necessary to synchronize the activities of the various threads**
  - all threads of a process share the same address space and other resources
  - any alteration of a resource by one thread affects the other threads in the same process

Example of inconsistent view

- **3 variables: A, B, C which are shared by thread T1 and thread T2**
- **T1 computes C = A+B**
- **T2 transfers amount X from A to B**
  - T2 must do: A = A -X and B = B+X (so that A+B is unchanged)
- **But if T1 computes A+B after T2 has done A = A-X but before B = B+X**
- **then T1 will not obtain the correct result for C = A + B**

Threads States

- **Three key states: running, ready, blocked**
- **They have no suspend state because all threads within the same process share the same address space**
  - Indeed: suspending (ie: swapping) a single thread involves suspending all threads of the same process
- **Termination of a process, terminates all threads within the process**

One or More Threads in a Process

Each thread has:

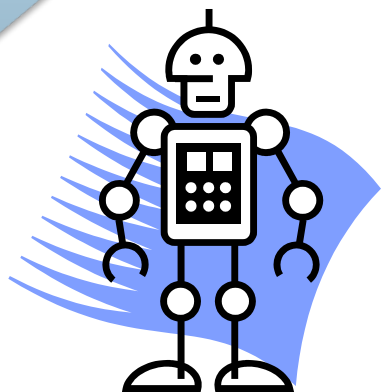
- an execution state (Running, Ready, etc.)
- saved thread context when not running (TCB)
- an execution stack
- some per-thread static storage for local variables
- access to the shared memory and resources of its process (all threads of a process share this)

Types of Threads

User Level Thread (ULT)

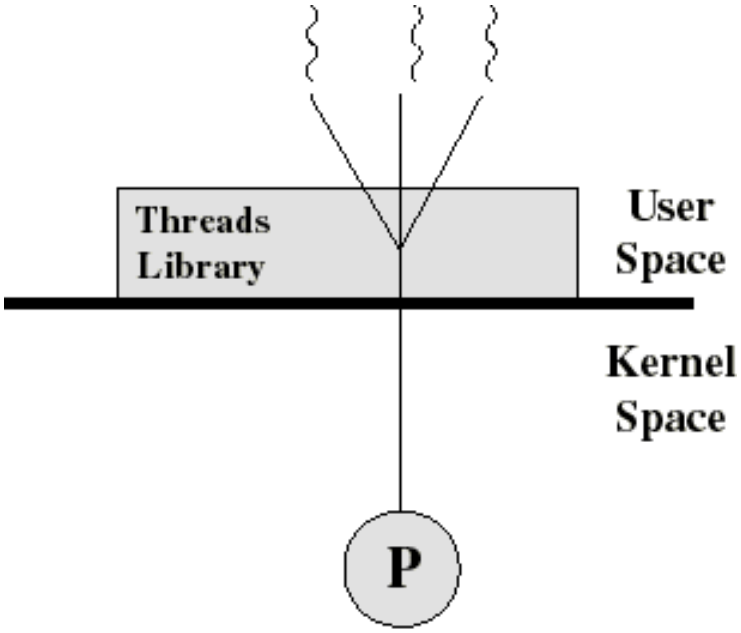
Kernel level Thread (KLT)

NOTE: we are talking about threads for *user* processes. Both ULT & KLT execute in user mode. An OS may also have threads but that is not what we are discussing here.



User-Level Threads (ULT)

- The kernel is not aware of the existence of threads
- All thread management is done by the application by using a thread library
- Thread switching does not require kernel mode privileges (no mode switch)
- Scheduling is application specific



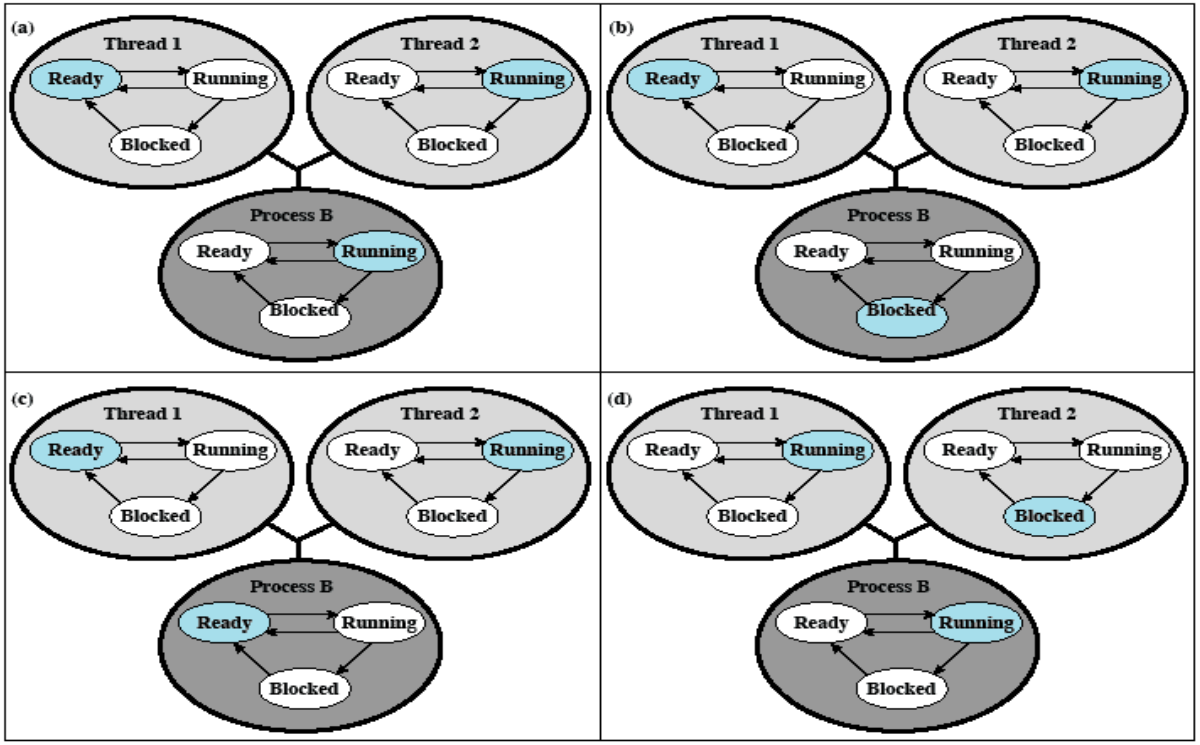
Threads library

- **Contains code for:**
  - creating and destroying threads
  - passing messages and data between threads
  - scheduling thread execution
  - saving and restoring thread contexts

Relationships Between ULT States and Process States

Possible transitions from a:

a → b
a → c
a → d



Colored state is current state

Examples of the Relationships between User-Level Thread States and Process States

Kernel activity for ULTs

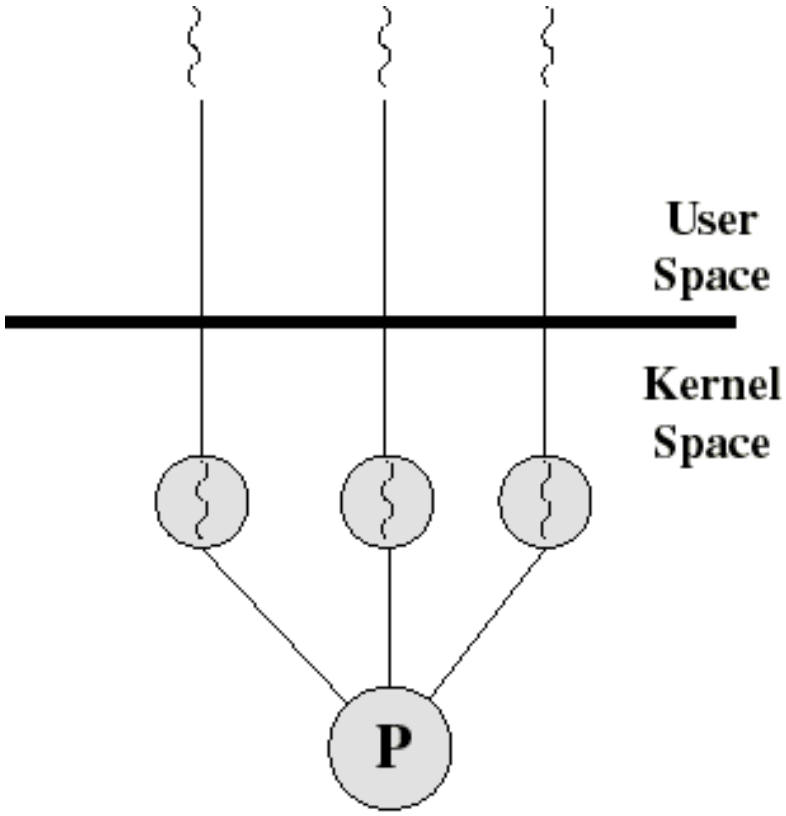
- The kernel is not aware of thread activity but it is still managing process activity
- When a thread makes a system call, the whole process will be blocked
- but for the thread library that thread is still in the running state
- So thread states are independent of process states

Advantages and inconveniences of ULT

- **Advantages**
  - Thread switching does not involve the kernel: no mode switching
  - Scheduling can be application specific: choose the best algorithm.
  - ULTs can run on any OS. Only needs a thread library
- **Inconveniences**
  - Most system calls are blocking and the kernel blocks processes. So all threads within the process will be blocked
  - The kernel can only assign processes to processors. Two threads within the same process cannot run simultaneously on two processors

Kernel-Level Threads (KLT)

- All thread management is done by kernel
- No thread library but an API to the kernel thread facility
- Kernel maintains context information for the process and the threads
- Switching between threads requires the kernel
- Scheduling on a thread basis
- Ex: Windows NT and OS/2



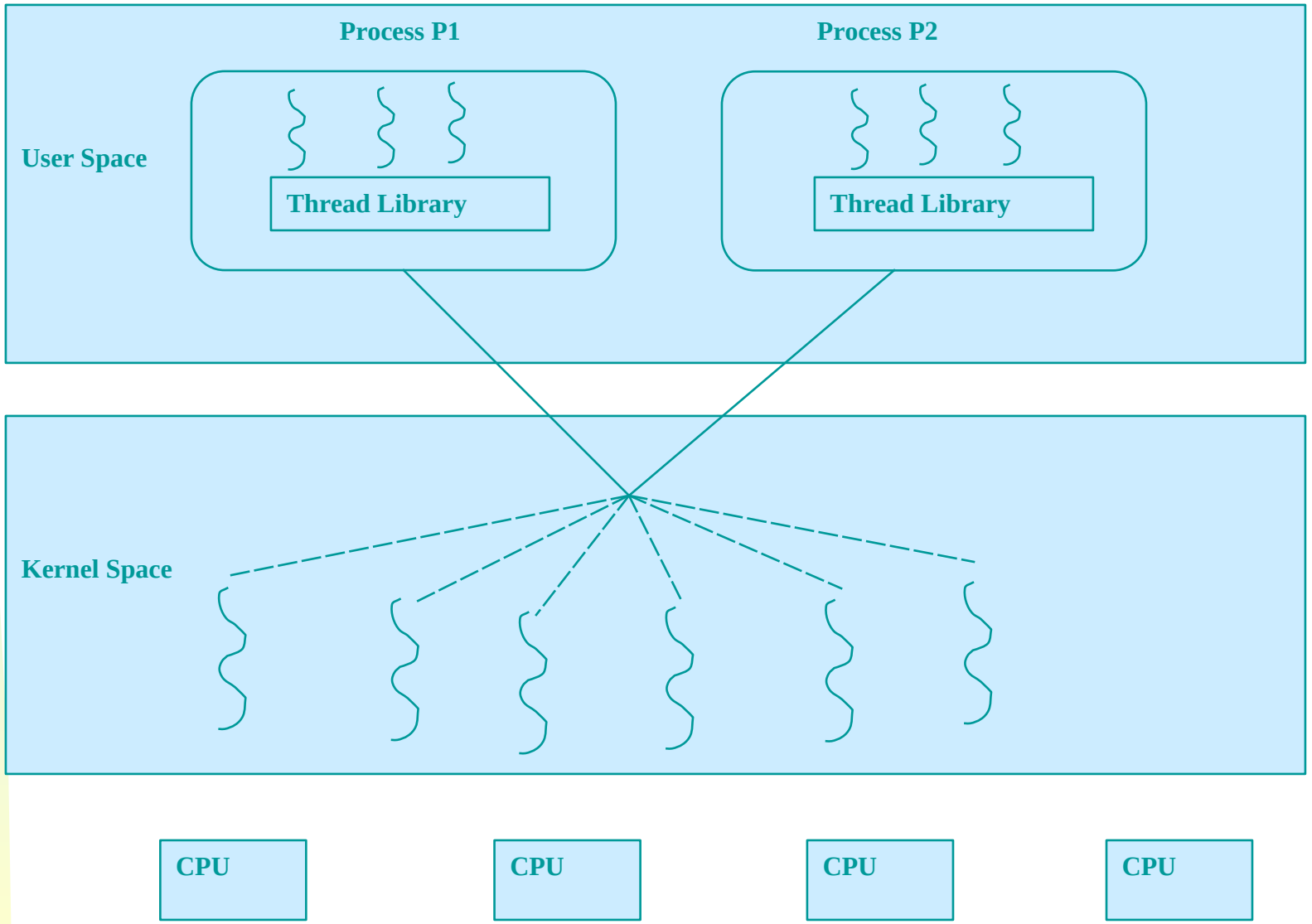
Advantages and inconveniences of KLT

- **Advantages**
  - the kernel can simultaneously schedule many threads of the same process on many processors
  - blocking is done on a thread level. **If one thread in a process is blocked, the kernel can schedule another thread of the same process**
  - kernel routines can be multithreaded
- **Inconveniences**
  - thread switching within the same process involves the kernel. We have 2 mode switches per thread switch
  - this results in a significant slow down

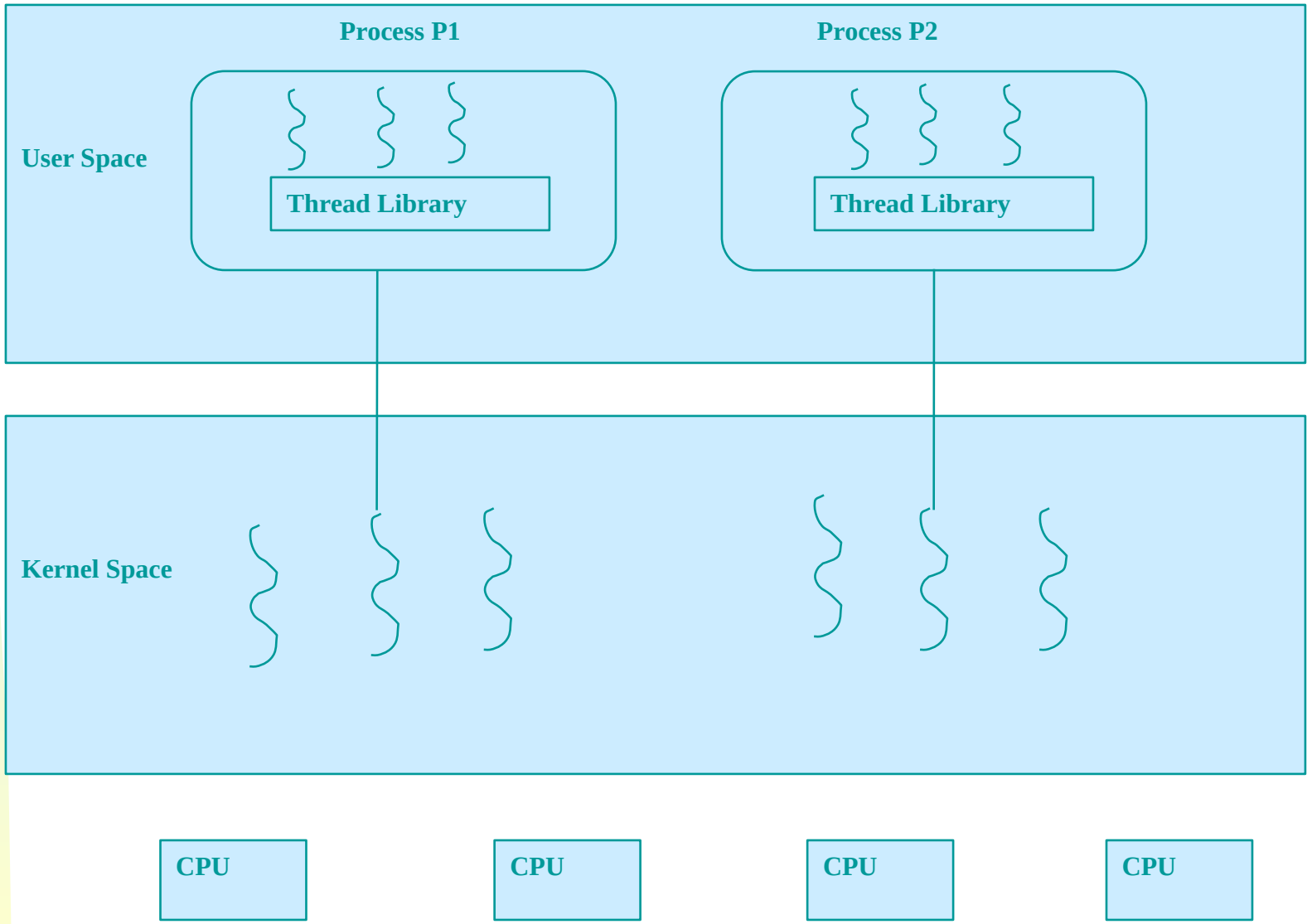
Multithreading Models

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

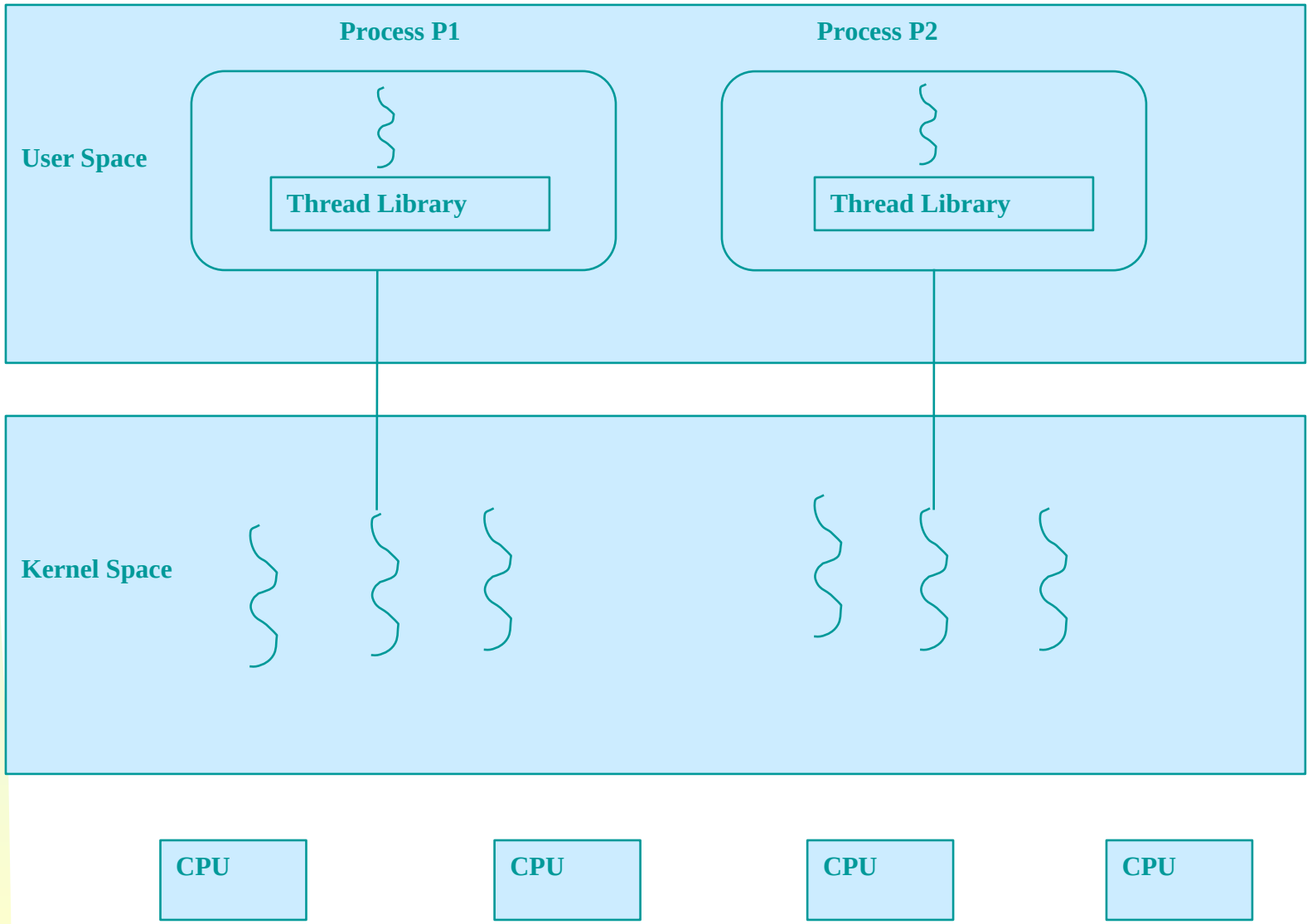
Multithreading Models(Many to Many)



Multithreading Models(Many to One)

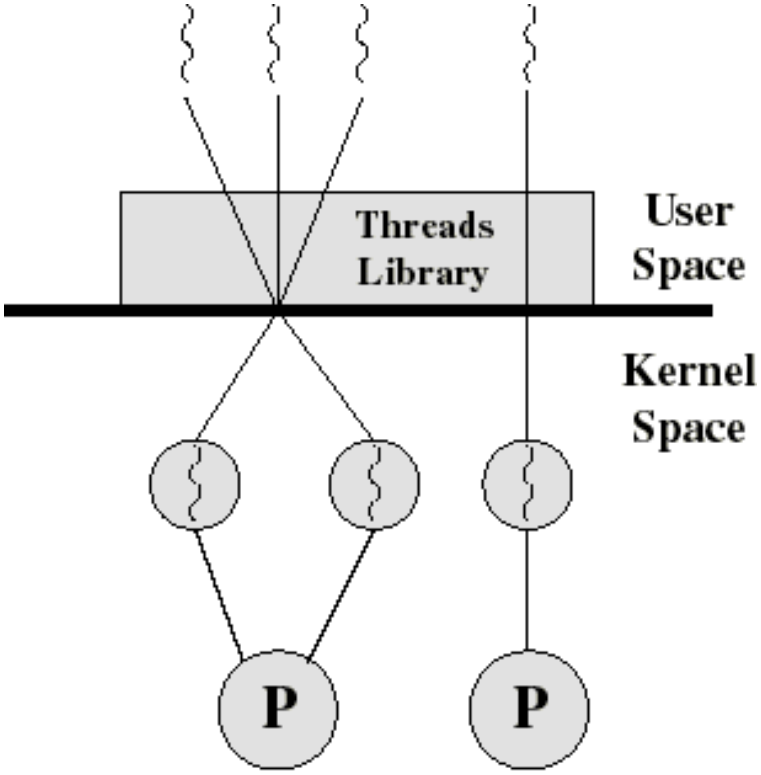


Multithreading Models(one to one)



Combined ULT/KLT Approaches

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space
- The programmer may adjust the number of KLTs
- May combine the best of both approaches
- Example is Solaris



Difference between ULT & KLT

- **ULT**
  - ULTs are faster to create and manage.
  - Implementation is by a thread library at the user level.
  - User level thread is generic and can run on any operating system.
  - Multithreaded applications can not take advantage of multiprocessing.
- **KLT**
  - **KLTs are slower to create and manage.**
  - **OS supports creation of kernel threads.**
  - **Kernel level thread is specific to the operating system.**
  - **Kernel routines themselves can be multithreaded.**

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

Relationship between Threads and Processes

THREADS

Various Implementations

PThreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Windows Threads

- Implements the one-to-one mapping
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads

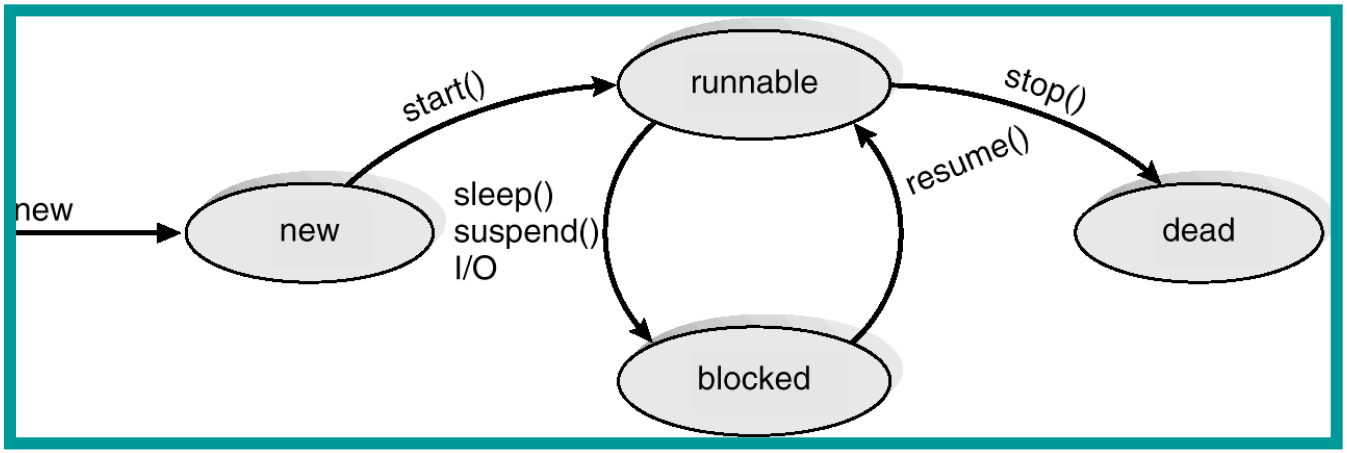
Various Implementations

Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through clone() system call
- clone() allows a child task to share the address space of the parent task (process)

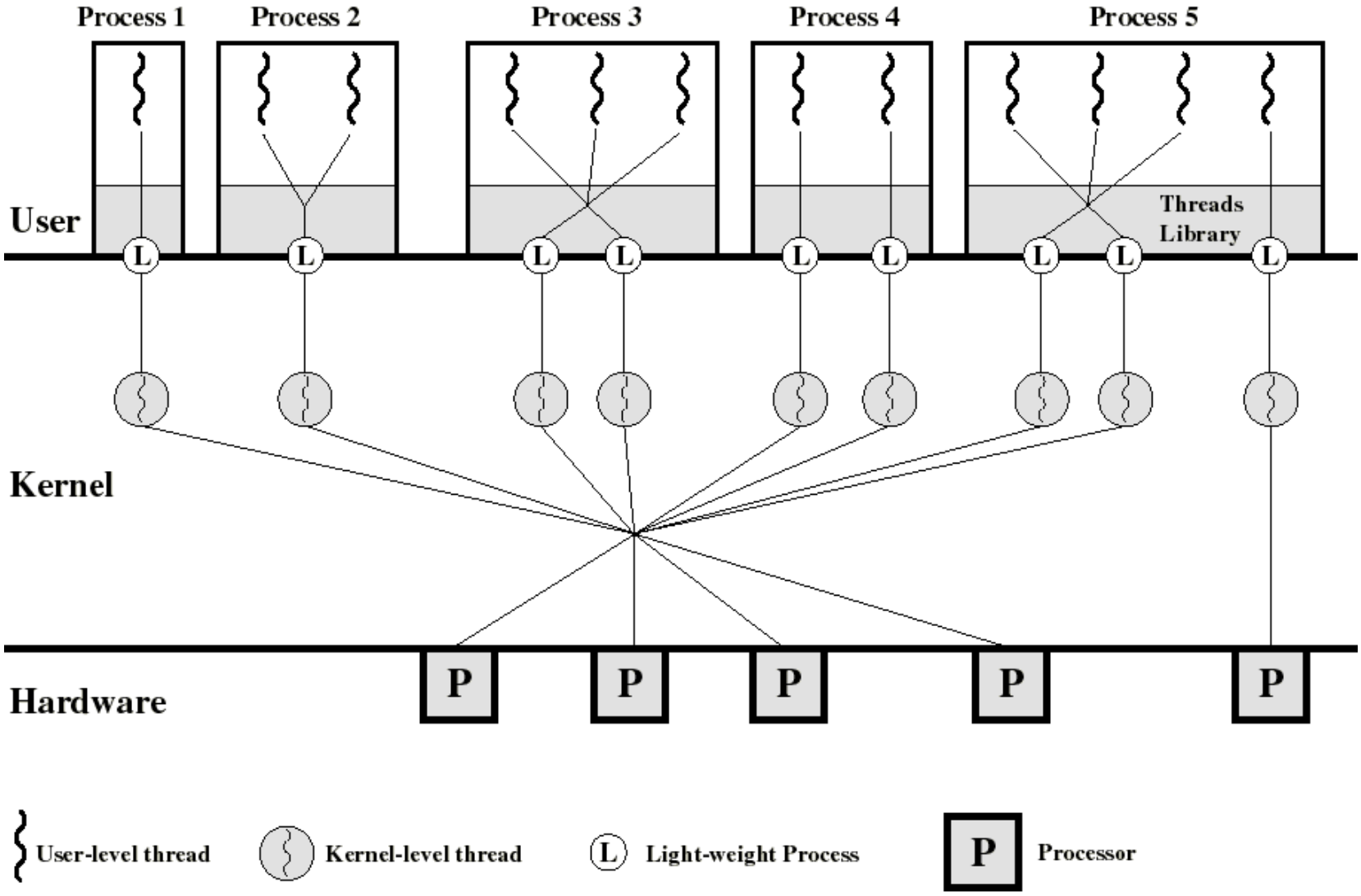
Java Threads

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- Java threads are



Solaris

- **Process includes the user's address space, stack, and process control block**
- **User-level threads (threads library)**
  - invisible to the OS
  - are the interface for application parallelism
- **Kernel threads**
  - the unit that can be dispatched on a processor and it's structures are maintain by the kernel
- **Lightweight processes (LWP)**
  - each LWP supports one or more ULTs and maps to exactly one KLT
  - each LWP is visible to the application



Process 2 is equivalent to a pure ULT approach
Process 4 is equivalent to a pure KLT approach
We can specify a different degree of parallelism (process 3 and 5)

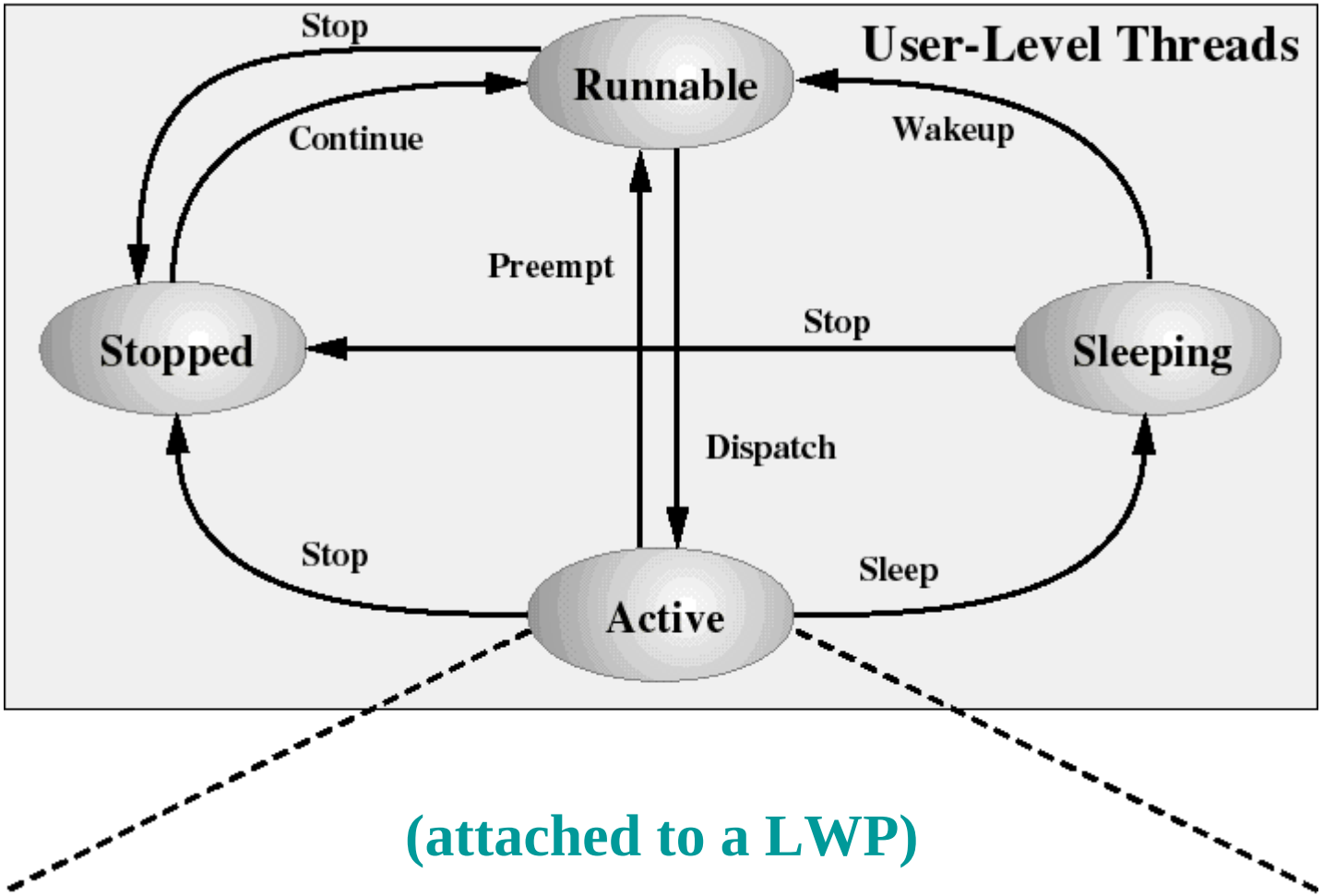
Solaris: versatility

- **We can use ULTs when logical parallelism does not need to be supported by hardware parallelism (we save mode switching)**
  - Ex: Multiple windows but only one is active at any one time
- **If threads may block then we can specify two or more LWPs to avoid blocking the whole application**

Solaris: user-level thread execution

- **Transitions among these states is under the exclusive control of the application**
  - a transition can occur only when a call is made to a function of the thread library
- **It's only when a ULT is in the active state that it is attached to a LWP (so that it will run when the kernel level thread runs)**
  - a thread may transfer to the sleeping state by invoking a synchronization primitive (chap 5) and later transfer to the runnable state when the event waited for occurs
  - A thread may force another thread to go to the stop state...

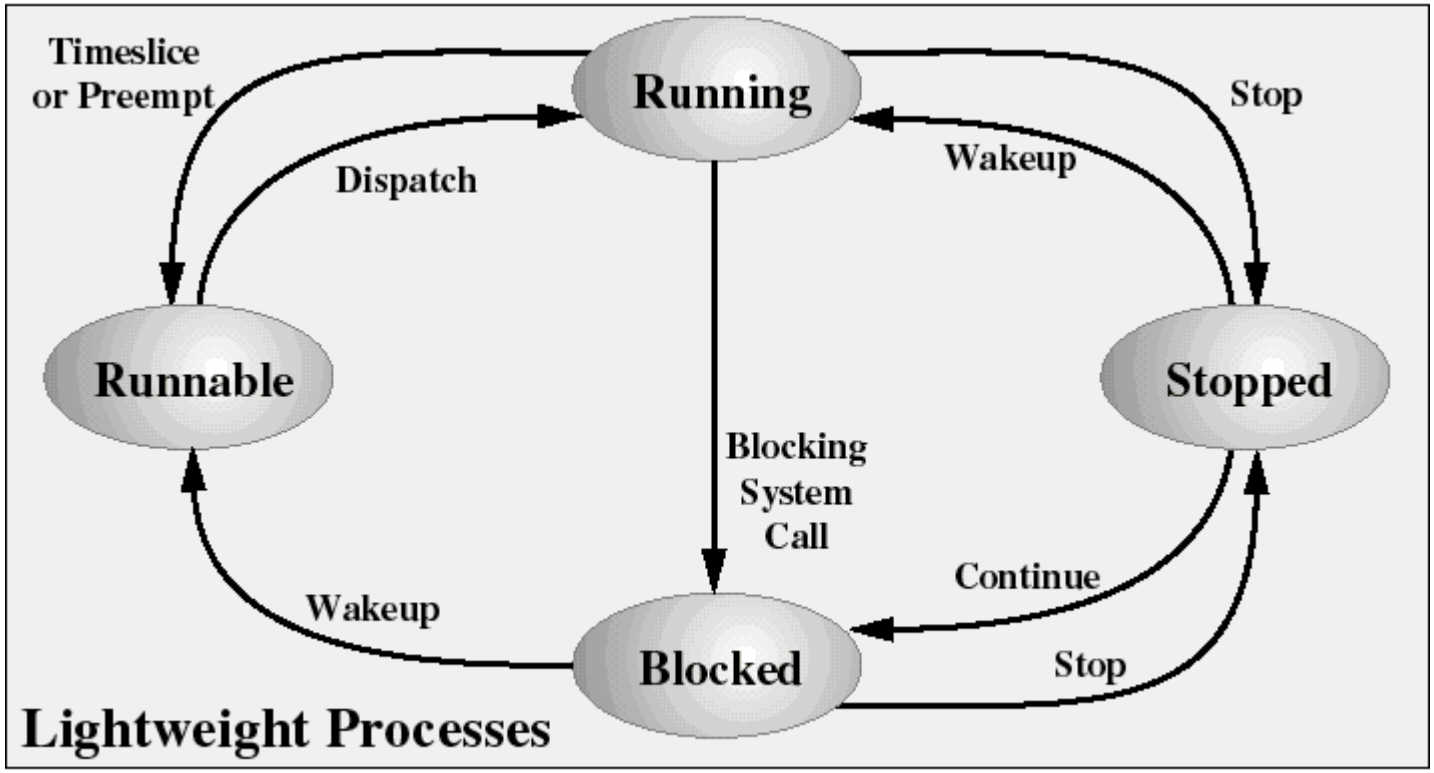
Solaris: user-level thread states



Decomposition of user-level *Active* state

- **When a ULT is Active, it is associated to a LWP and, thus, to a KLT**
- **Transitions among the LWP states is under the exclusive control of the kernel**
- **A LWP can be in the following states:**
  - running: when the KLT is executing
  - blocked: because the KLT issued a blocking system call (but the ULT remains bound to that LWP and remains active)
  - runnable: waiting to be dispatched to CPU

Solaris: Lightweight Process States



LWP states are independent of ULT states (except for bound ULTs)