
Threads

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscg@mail.sysu.edu.cn





■ Contents

- Overview
- Multicore Programming
- User and Kernel Level Threads
- Multithreading Models
- Threads Libraries
 - POSIX Pthreads
- Implicit Threading
 - Threads Pools
 - OpenMP
- Threading Issues
- Linux clone()
- Example: Windows Threads

■ Review of Process Characteristics

- Unit of resource ownership
 - A process is allocated an *address space* to hold the process image.
 - control of some resources (files, I/O devices ...)
- Unit of dispatching
 - A 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.
- These characteristics are treated independently:
 - The unit of resource ownership is usually referred to as a *Task* or (for historical reasons) also as a *Process*.
 - The unit of dispatching is usually referred to a *Thread* or a Light-Weight Process (*LWP*).
 - A traditional Heavy-Weight Process (*HWP*) is equal to a task with a single thread.
- *Multithreading* - Several threads can exist in the same task
 - Multithreading is the ability of an operating system to support multiple, concurrent paths of execution within a single process.

■ Motivation for Threads

- Most modern applications are multithreaded.
 - Threads run within one application.
- Multiple tasks with one application can be implemented by separate threads. For example:
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request.
- Process creation is *heavy-weight*, while thread creation is *light-weight*.
- Use of threads can simplify code, increase efficiency.
- Kernels are generally multithreaded.

■ Tasks/Processes vs. Threads

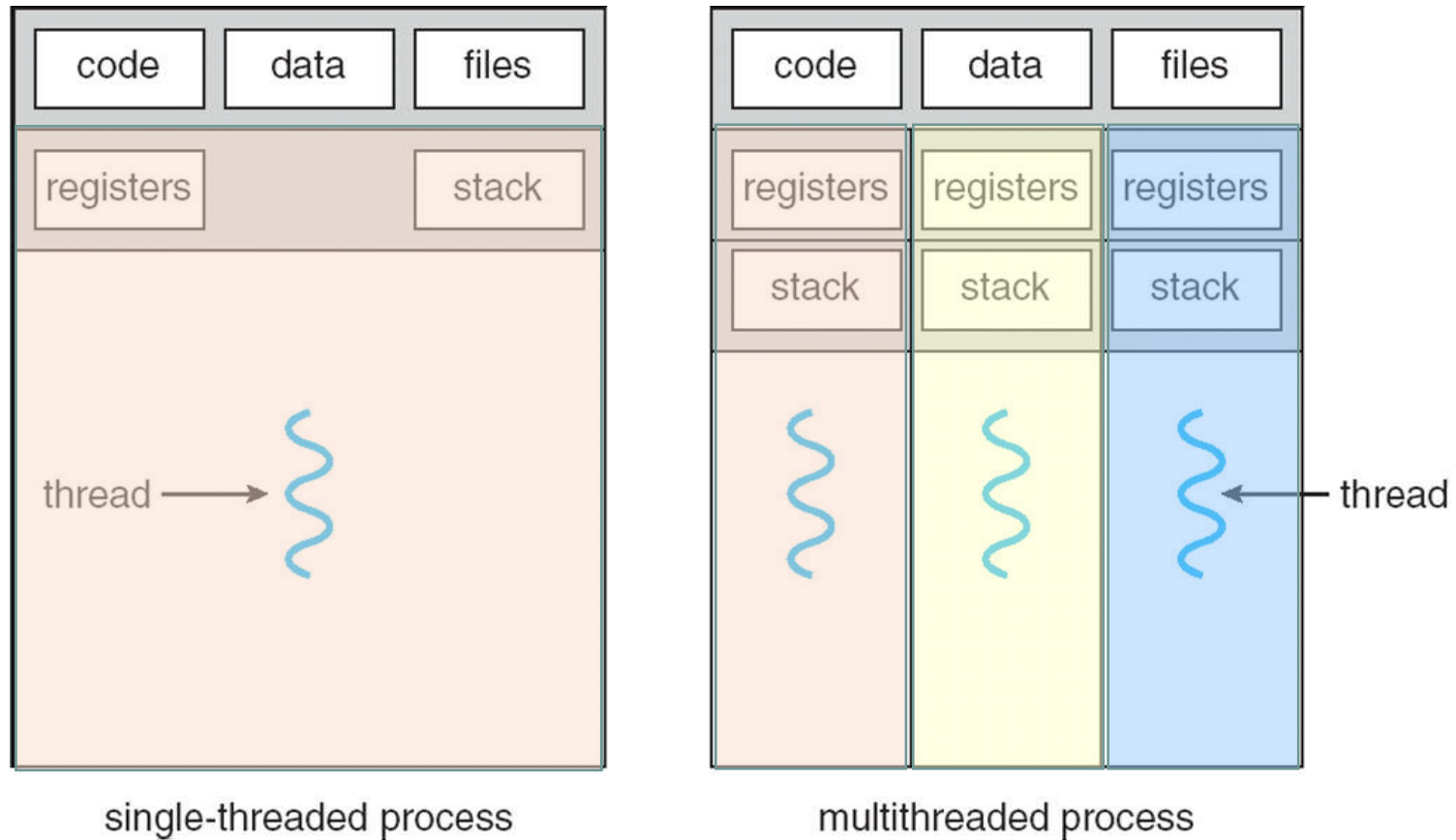
- Per task/process items (shared by all threads of the task):
 - Address space which holds the process image
 - Global variables
 - Protected access to files, I/O and other resources
 - Children processes
 - Pending alarms
 - Signals and signal handlers
 - Accounting information.
- Per thread items:
 - An execution context/state (Running, Ready, etc.)
 - saved when not in Running state
 - Program counter (程序计数器)
 - Register set
 - Execution stack
 - Some per-thread static storage for local variables (TLS).



■ **Tasks/Processes vs. Threads**

- A thread has access to the address space and resources of its task:
 - The memory address space and resources are shared by all threads of the task.
 - When one thread alters a (non-private) memory item, all other threads of the task can see that.
 - A file opened with one thread is available to other threads of the task.

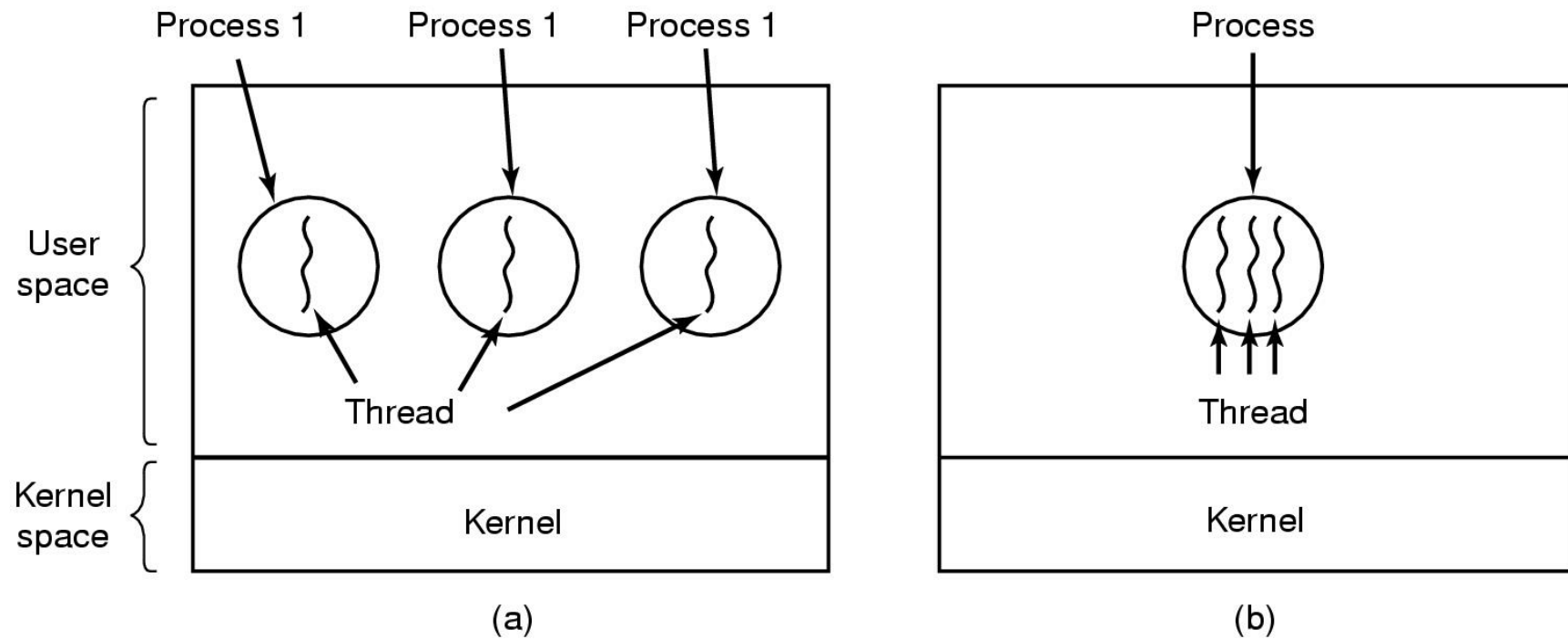
■ Tasks/Processes vs. Threads



Single and Multithreaded Processes



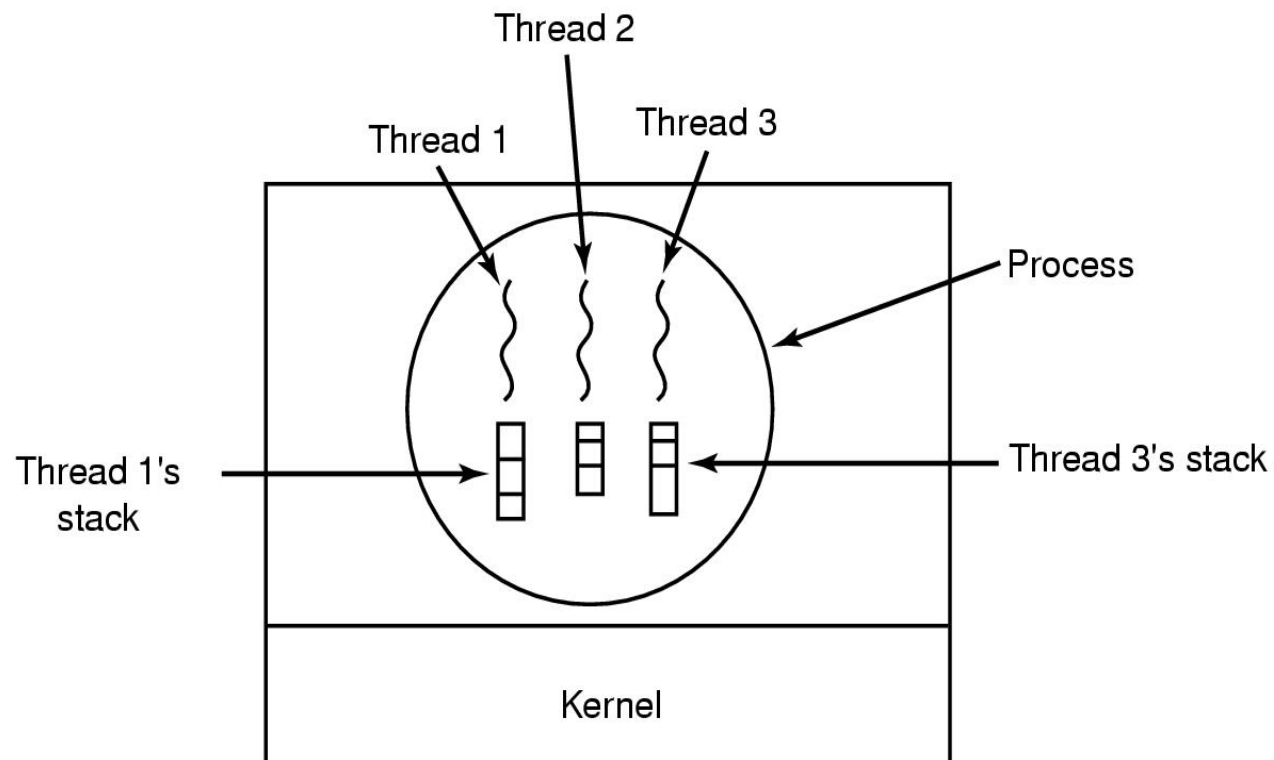
■ Tasks/Processes vs. Threads



(a) Three processes each with one thread.

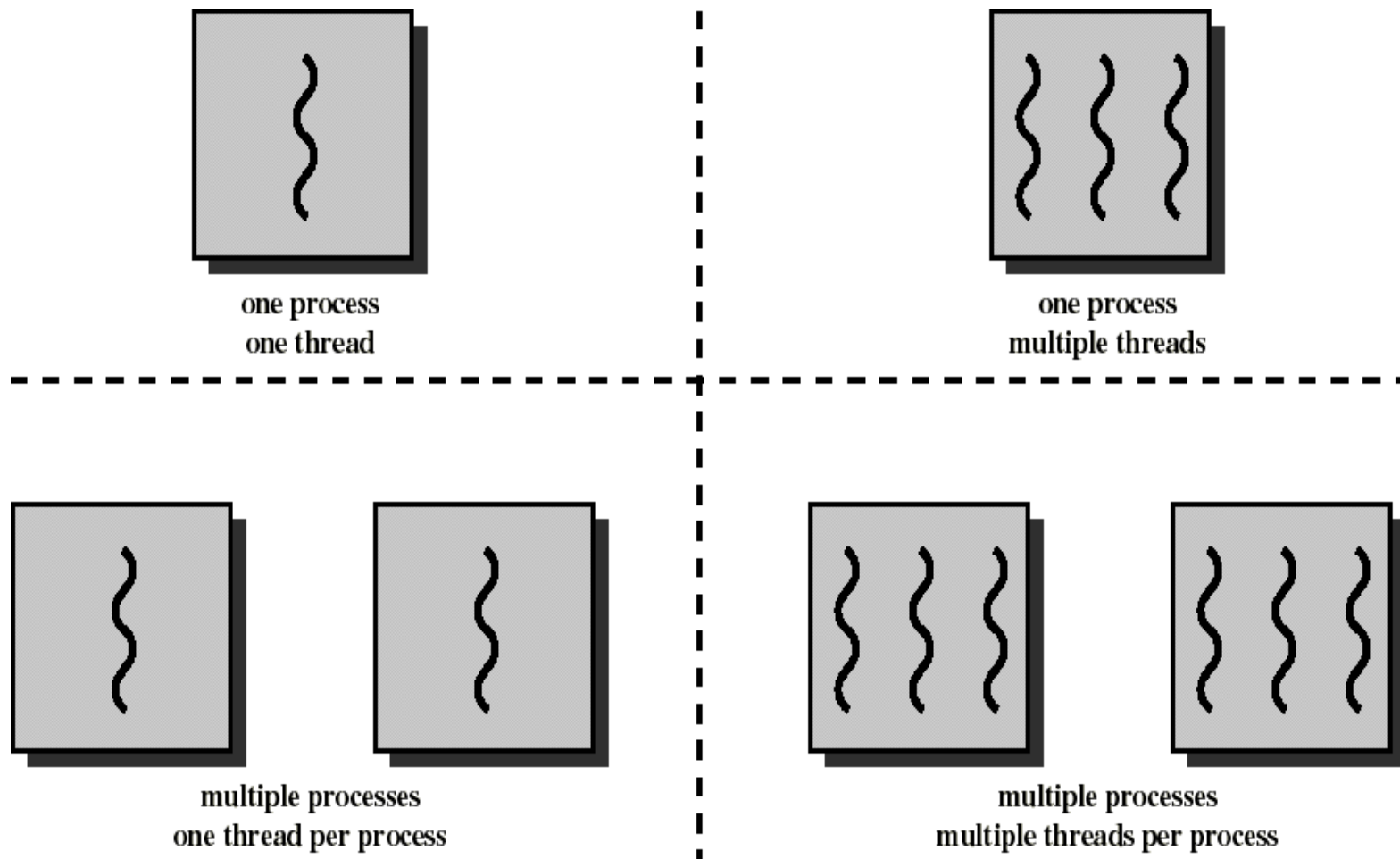
(b) One process with three threads.

■ Tasks/Processes vs. Threads



Each thread has its own stack

■ Tasks/Processes vs. Threads



Combinations of Threads and Processes



■ Tasks/Processes vs. Threads

- Creating and managing processes is generally regarded as an expensive task.
 - E.g., `fork()` system call.
- Making sure all the processes peacefully co-exist on the system is not easy.
 - as concurrency transparency comes at a price
- Rather than make the operating system responsible for concurrency transparency, it is left to the individual application to manage the creation and scheduling of each thread.
 - Threads can be thought of as an “execution of a part of a program (in user-space)”.

■ Tasks/Processes vs. Threads

- Relationship between threads and processes
 - One-to-One, Many-to-One and Many-to-Many

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(CMU,1985)
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,1989), Emeralds (Extensible Microkernel for Embedded, Real time, Distributed System, 1999)
M:N	Combines attributes of M:1 and 1:M cases.	TRIX (MIT,1970s)



■ Benefits of Threads

- The benefits of multithreaded programming can be broken down into four major categories:
 - *Responsiveness* (响应能力)
 - Multithreading may allow continued execution if **part of** process is blocked, especially important for user interfaces.
 - While one server thread is blocked and waiting, a second thread in the same task can run.
 - *Resource Sharing* (资源共享)
 - Threads share process resources, easier than IPC like shared-memory or message-passing.
 - Inter-thread communication and synchronization is faster.

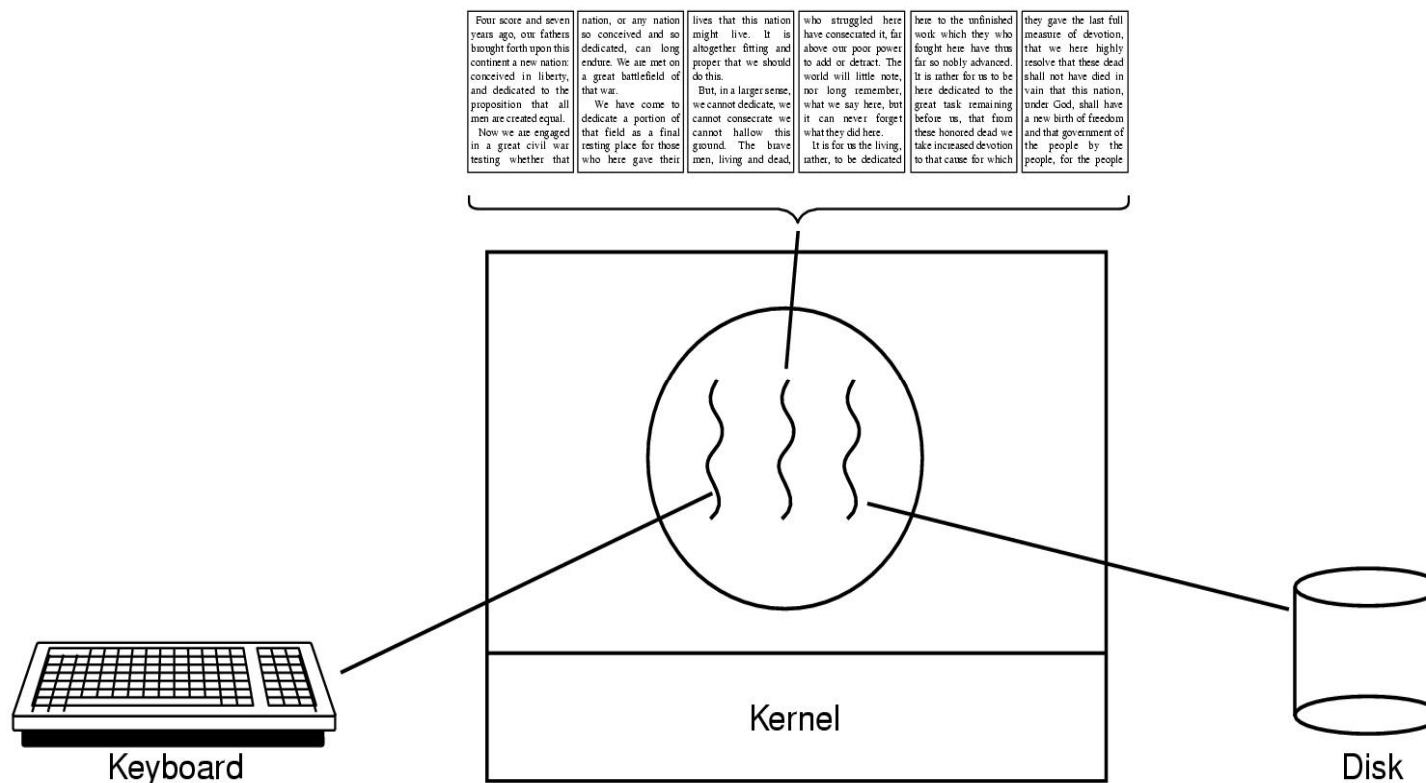
■ Benefits of Threads

- The benefits of multithreaded programming can be broken down into four major categories: (cont.)
 - *Economy* (经济性/低系统开销)
 - Thread creating and context switching are lower overhead than process.
 - Less time to create and terminate a thread than a process because we do not need another address space
 - Less time to switch between two threads than between processes because we need not to deal with the address space.
 - *Scalability* (可扩展性/可伸缩性)
 - Threads may be running in parallel on different processing cores, taking advantage of multiprocessor/multicore architectures and networked/distributed systems

■ Benefits of Threads

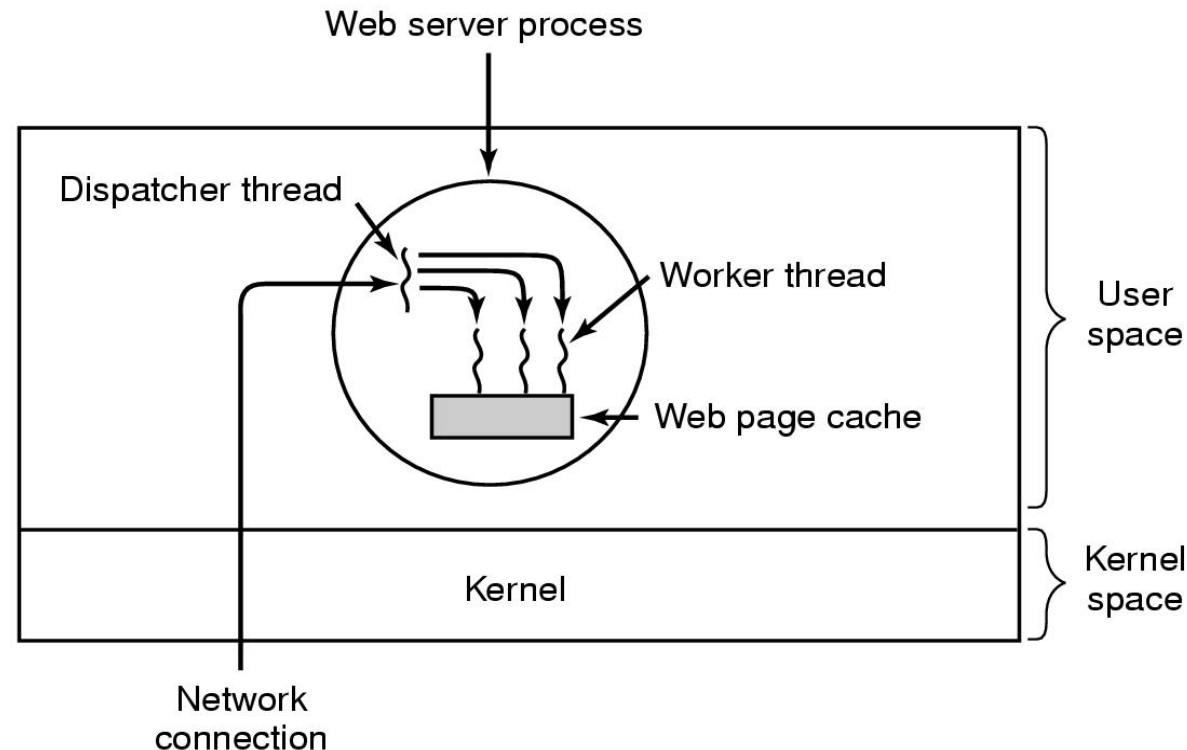
■ Example 1: Word Processor

- One thread displays menu and reads user input while another thread executes user commands and a third one writes to disk.
 - The application is more responsive.



■ Benefits of Threads

- Example 2: A File/Web server
 - The server needs to handle several files/pages requests over a short period. Hence it is more efficient to create (and destroy) a single thread for each request.
 - On a SMP (Symmetric Multi-Processor) machine: multiple threads can possibly be executing simultaneously on different processors.



■ Benefits of Threads

■ Example 2: A File/Web server

■ Dispatcher thread

```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

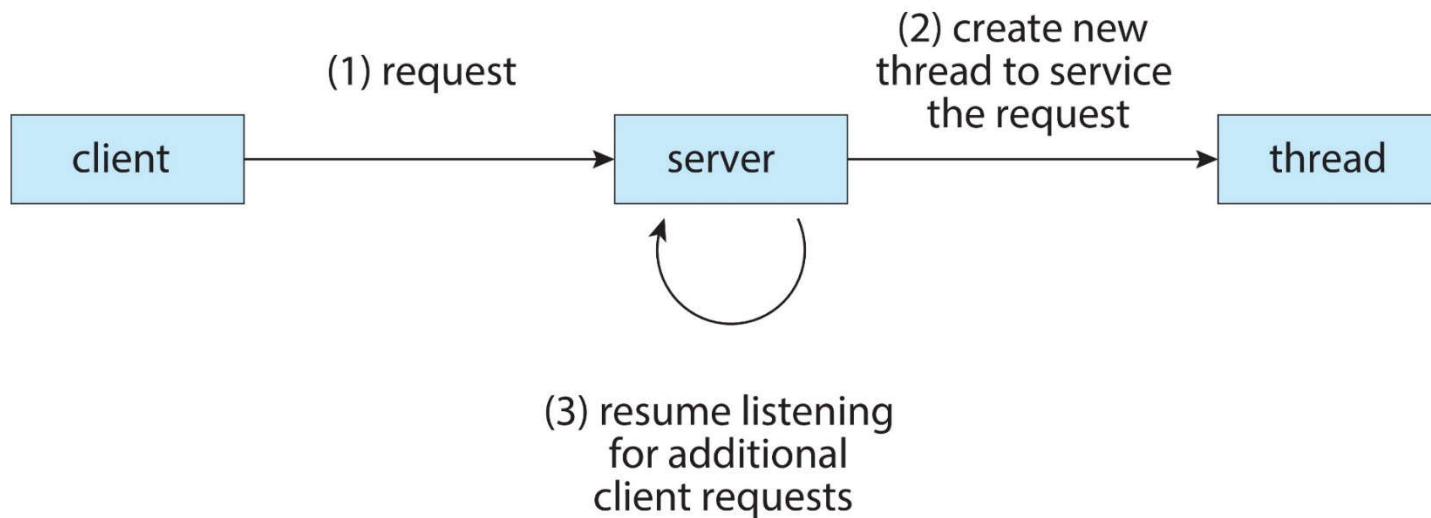
■ Worker thread

```
while (TRUE) {  
    wait_for_work(&buf);  
    Look_for_page_in_cache(&buf,&page);  
    if (page_not_in_cache(&page))  
        read_page_from_disk(&buf,&page);  
    return_page(&page);  
}
```



■ Benefits of Threads

- Multithreaded Server Architecture.





■ Benefits of Threads

- Two Important Implications
 - Threaded applications often run *faster* than non-threaded applications.
 - as context-switches between kernel and user-space are avoided
 - Threaded applications are *harder* to develop.
 - although simple, *clean designs* can help here
- Additionally, the assumption is that the development environment provides a *Threads Library* for developers to use.
 - most modern environments do

■ 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 blocking it and switching to another application
- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel.
- Concurrency requirement
 - It is necessary to synchronize the activities of various threads so that they do not obtain inconsistent views of the current data.



■ Thread States

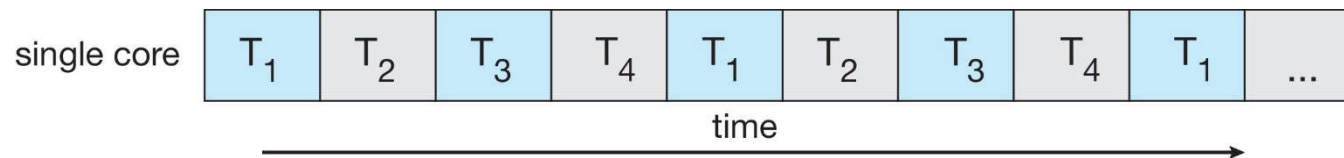
- There are three key states for a thread.
 - Running State
 - Ready State
 - Blocked State.
- They have *no suspend state* because all threads within the same task share the same address space.
 - Indeed, suspending (e.g., swapping) a single thread involves suspending all threads of the same task.
- Termination of a task will terminate all threads within the task.

■ Multicore Systems

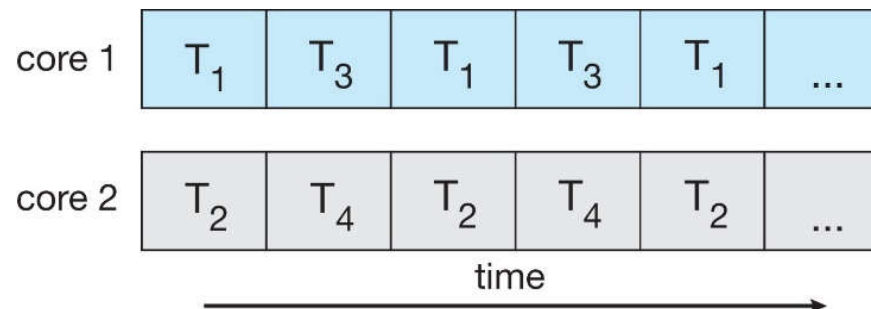
- Multicore architectures place multiple computing cores on a single processing chip.
 - Each core appears as a separate processor to the operating system.
 - Whether the cores appear across CPU chips or within CPU chips, we call these systems *multicore* or *multiprocessor* systems.
- Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and has concurrency improved.
- *Parallelism* and *Concurrency*
 - A parallel system can *perform* more than one task simultaneously.
 - A concurrent system supports more than one task by allowing all the tasks to *make progress*. Thus, it is possible to have concurrency without parallelism.
 - Before the advent of SMP and multicore architectures, most computer systems had only a single processor. CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes in the system, thereby allowing each process to make progress. Such processes were running concurrently, but not in parallel.

■ Multicore Systems

- On a system with multiple cores, concurrency means that the threads can run *in parallel*, because the system can assign a separate thread to each core.
- Example: Threads T_1 , T_2 , T_3 and T_4 execute concurrently on a single-core system.



- Example: Threads T_1 , T_2 , T_3 and T_4 execute in parallel on a 2-core system.



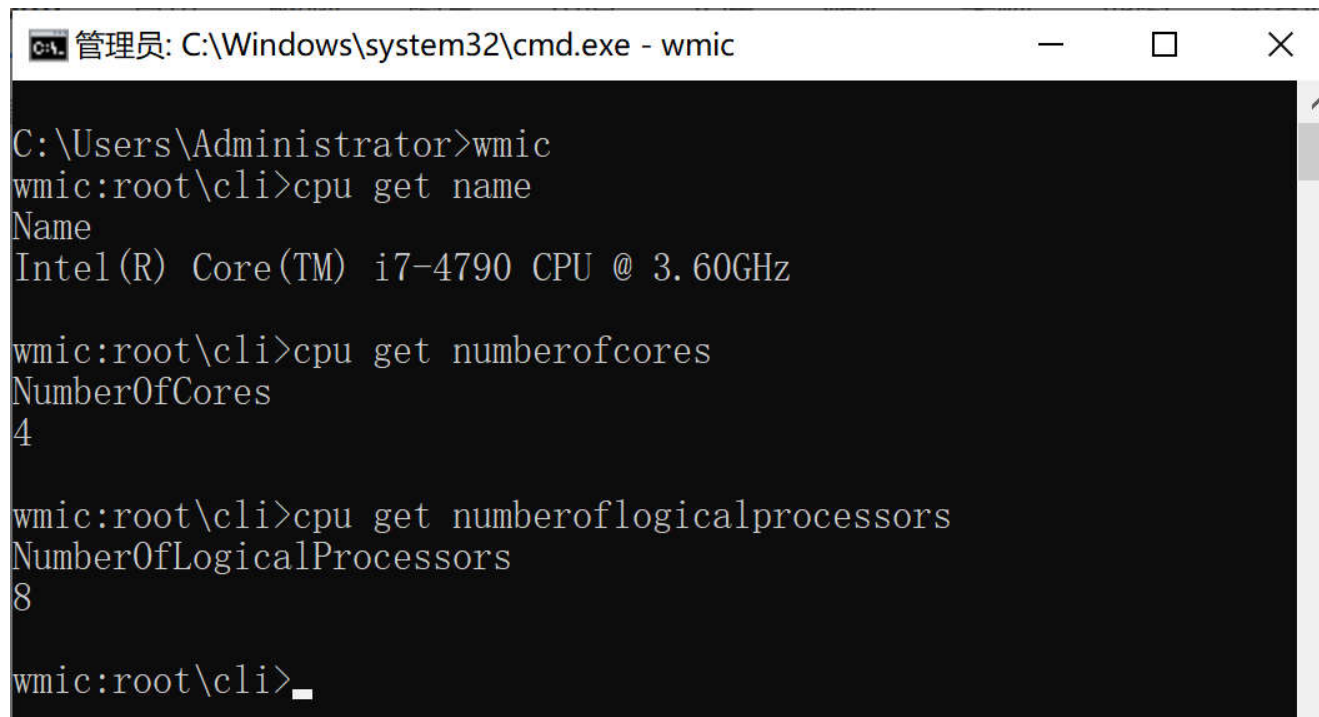
■ Multithreading

- As systems have grown from tens of threads to thousands of threads, CPU designers have improved system performance by adding *hardware* to improve thread performance.
 - Modern Intel CPUs frequently support two threads (as logical processors) per core.
 - Oracle T4 CPU supports eight threads per core. This support means that multiple threads can be loaded into the core for fast switching.
- Multicore computers will no doubt continue to increase in core counts and hardware thread support.

■ Multithreading

■ Example

- In Windows Command Line Interface (CLI), inquire about CPU's information.



```
C:\Users\Administrator>wmic
wmic:root\cli>cpu get name
Name
Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz

wmic:root\cli>cpu get numberofcores
NumberOfCores
4

wmic:root\cli>cpu get numberoflogicalprocessors
NumberOfLogicalProcessors
8

wmic:root\cli>_
```

- NumberOfLogicalProcessors is the maximal number of threads being supported by the Intel i7-4790 CPU, two threads (logical processors) for each physical core.

■ Multithreading

■ Programming Challenges

- For operating system designer
 - implementing system models to support parallel execution with multiple processing cores
- For application programmer
 - to modify or design programs that are multithreaded
- In general, five areas present challenges in programming for multicore systems:
 - (1) *Identifying tasks*
 - examine applications to find areas that can be divided into separate, concurrent tasks which ideally are independent of one another and thus can run in parallel on individual cores
 - (2) *Balance.*
 - ensure that the identified tasks perform equal work of equal value

■ Multithreading

■ Programming Challenges

- In general, five areas present challenges in programming for multicore systems: (cont.)

(3) *Data splitting*

- The data accessed and manipulated by the separate tasks must be divided to run on separate cores.

(4) *Data dependency*

- examine the data for dependencies between two or more tasks. Synchronization is used to accommodate the data dependency

(5) *Testing and debugging*

- Testing and debugging concurrent programs running in parallel on multiple cores are difficult because there are too many possible execution paths.

■ Multithreading

■ Types of Parallelism

- In general, there are two types of parallelism including data parallelism and task parallelism.
- *Data parallelism*
 - focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core
- *Task parallelism*
 - involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or on different data

■ Multithreading

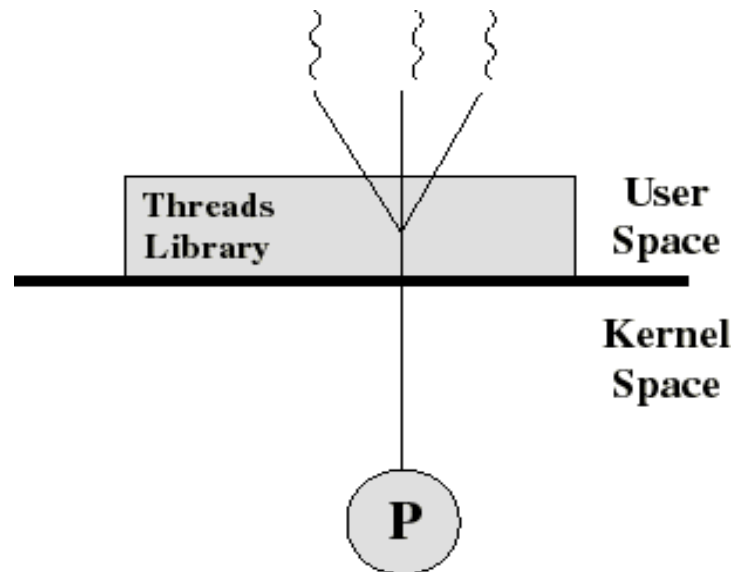
- Multithreading is a type of programming that takes advantage of a CPU's capability to process many threads at the same time across *multiple processing cores*.
 - tasks or instructions executing simultaneously
- The *main thread* runs at the start of a program by default and creates new threads to handle tasks.
 - These new threads run in parallel to one another, and usually *synchronize* their results with the main thread once completed.

■ Multithreading Levels

- A *Threads Library* contains code and provides *APIs* for:
 - creating and destroying threads
 - passing messages and data between threads
 - scheduling thread execution
 - saving and restoring thread contexts
- Three primary thread libraries:
 - POSIX Pthreads
 - Win32 threads
 - Java threads
- There are three multithreading levels:
 - User-level Threads (ULT)
 - threads library entirely in user space
 - Kernel-level Threads (KLT)
 - kernel-level threads library supported by the OS.
 - Hybrid ULT/KLT Approach

■ User-level Threads (ULTs)

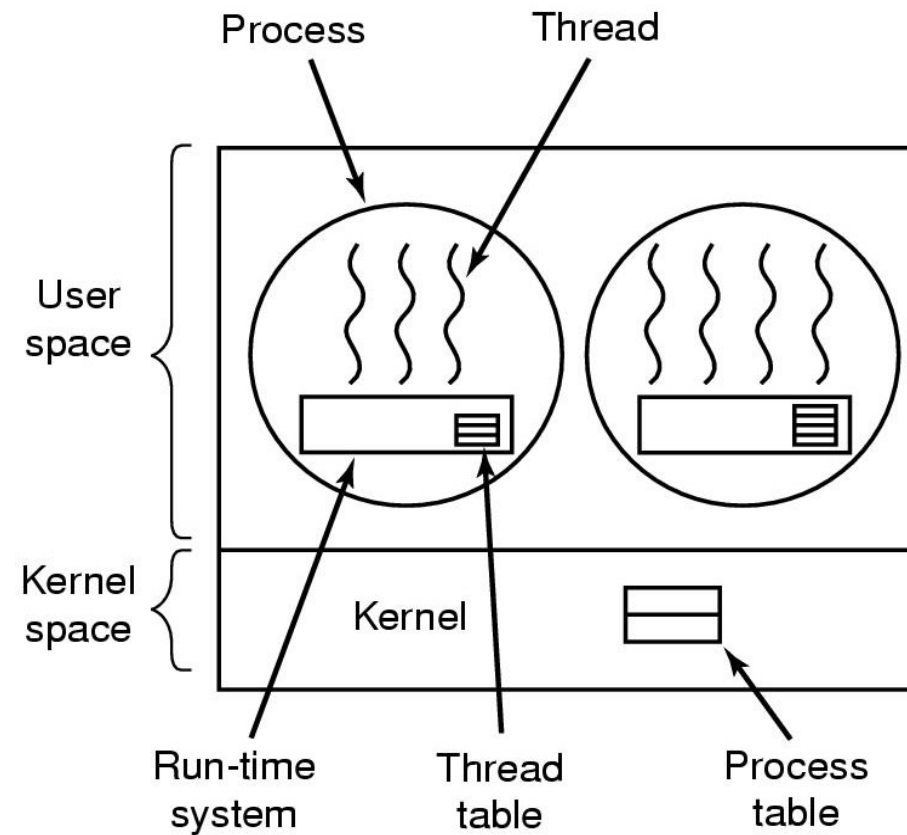
- Management for ULTs are done by the application with user-level threads library.
- The kernel is not aware of the existence of ULTs.
- Thread switching does not require kernel mode privileges.
- Scheduling is application specific.





■ User-level Threads (ULTs)

- Implementing ULTs in user space.





■ User-level Threads (ULTs)

- Kernel activity for ULTs
 - The kernel is not aware of thread activity but it is still managing *process* activity.
 - Thread states are independent of process states.
 - When a thread makes a system call, the whole task will be blocked.
 - But for the user-level threads library, that thread is still in the running state.

■ User-level Threads (ULTs)

■ Advantages and inconveniences of ULTs

■ Advantages

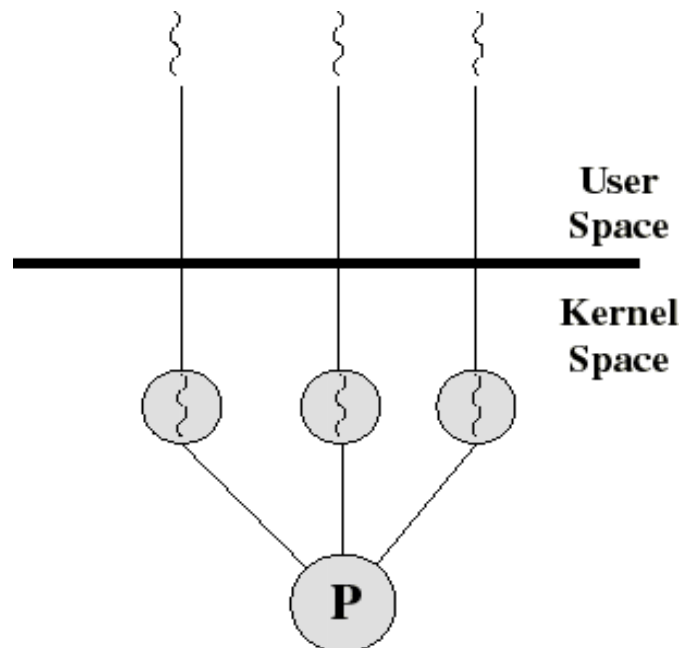
- Thread switching does not involve the kernel.
 - with no dual-mode switching
- Scheduling can be application specific.
 - choosing the best algorithm
- ULTs can run on any OS.
 - only needing support of a user-level thread library

■ Inconveniences

- One thread being blocked may cause the kernel blocking the process.
 - all threads within the process will be blocked
- The kernel can only assign processes to processors.
 - E.g., two threads within the same process cannot run simultaneously on two processors

■ Kernel-level Threads (KLTs)

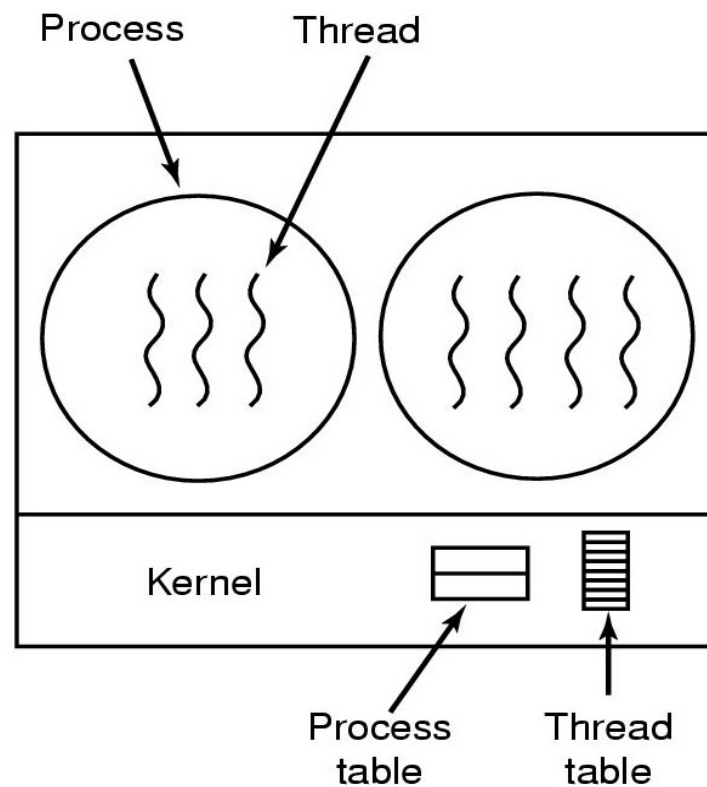
- Management for KLTs are done by the kernel.
- No threads library but an **API** provided for users to the kernel thread facility.
- Kernel maintains context information for the process and the threads.
- Switching between threads requires the kernel.
- Scheduling on a thread basis





■ Kernel-level Threads (KLTs)

- Implementing KLTs in the Kernel.





■ Kernel-level Threads (KLTs)

- KLTs are virtually included in all general purpose operating systems.
 - Windows
 - OS/2
 - Linux
 - Solaris
 - Tru64 UNIX
 - Mac OS X.
- Linux threads
 - Linux refers to threads as tasks.
 - Thread creation is done through `clone()` system call.
 - `clone()` allows a child task to share the address space of the parent task (process).
 - This sharing of the address space allows the cloned child task to behave much like a separate thread.



■ Kernel-level Threads (KLTs)

■ Advantages and inconveniences of KLTs

■ Advantages

- The kernel can simultaneously schedule many threads of the same process on many processors.
- Blocking is done on a thread level.
- Kernel routines can be multithreaded.

■ Inconveniences

- Thread switching within the same process involves the kernel.
 - This would result in a significant slow down.

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

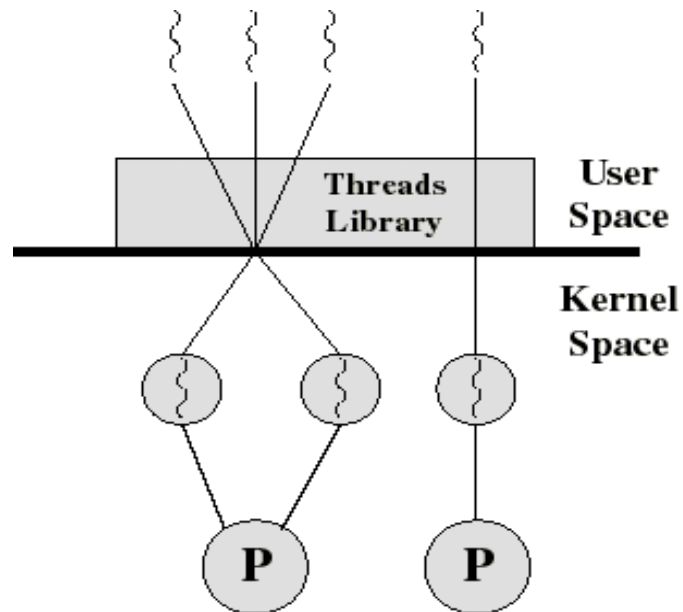
Thread operation latencies (μ s)

From *Anderson, T. et al*, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", ACM TOCS, February 1992.



■ Hybrid ULT/KLT Approaches

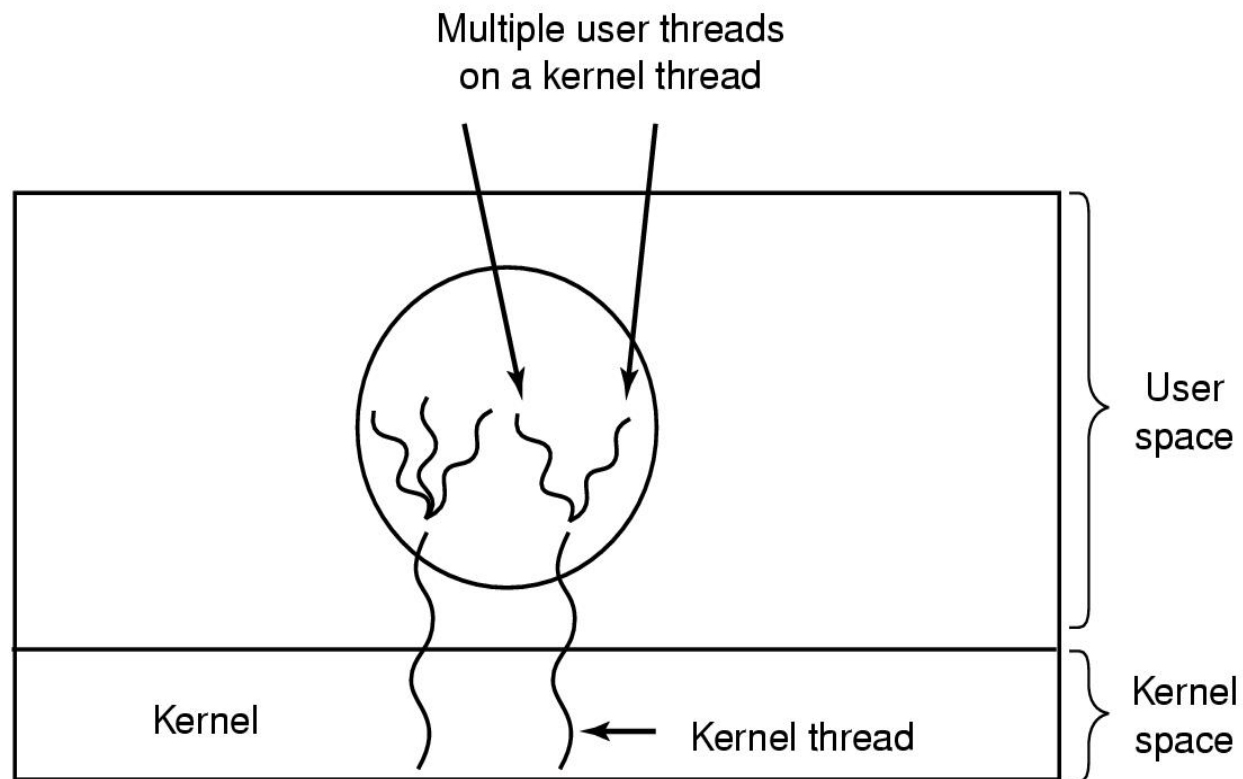
- Thread creation is done in the user space.
- Bulk of scheduling and synchronization of threads are done in the user space.
- The programmer may adjust the number of KLTs.
- It may combine the best of both approaches.
- Example: Solaris prior to version 9.





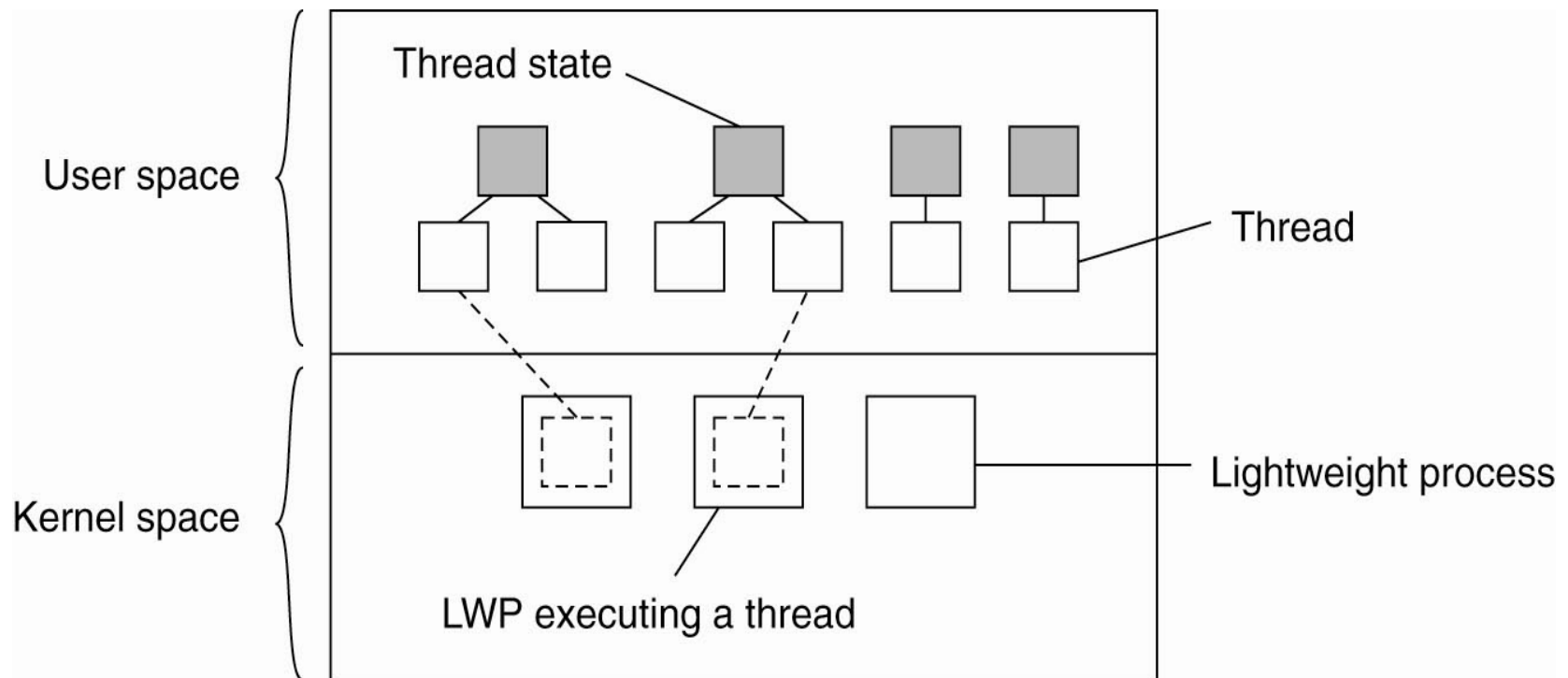
■ Hybrid ULT/KLT Approaches

- Hybrid Implementation.



Hybrid ULT/KLT Approaches

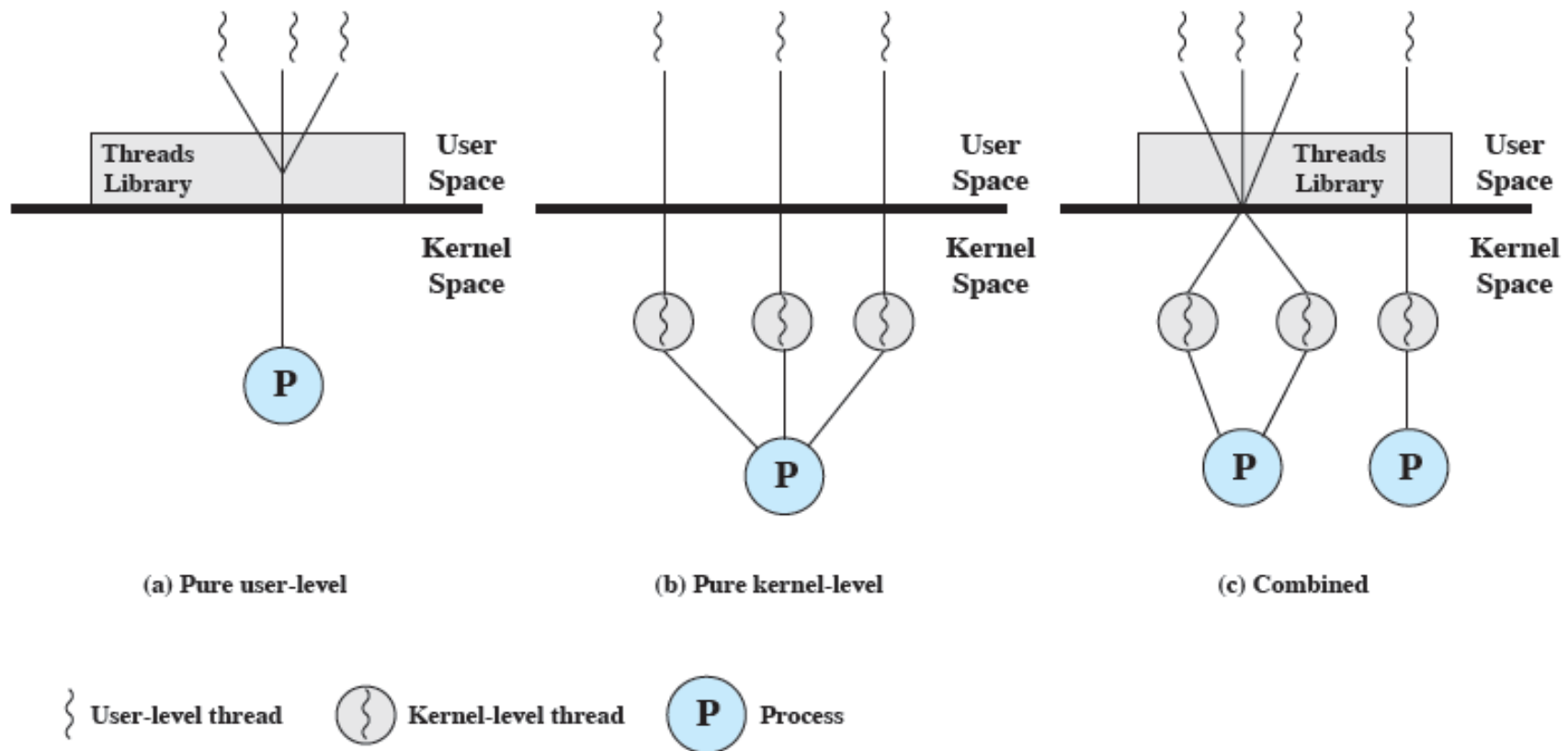
- Hybrid Implementation.





Hybrid ULT/KLT Approaches

- ULT, KLT and Combined Approaches.



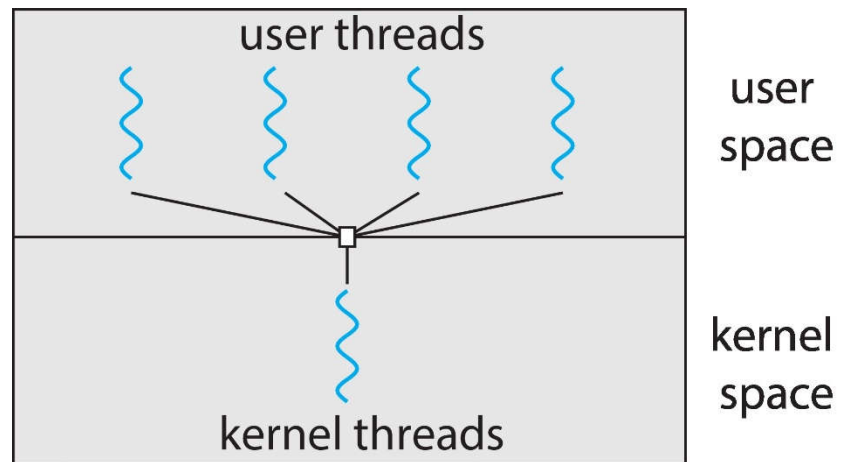
■ Multithreading Models

- Support for threads may be provided either at the user level, for *user threads* (ULTs) or by the kernel, for *kernel threads* (KLTs).
 - User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.
- Relationship between user threads and kernel threads are
 - Many-to-One model,
 - *One-to-One model*, and
 - Many-to-Many model.

■ Multithreading Models

■ Many-to-One Model

- Many user-level threads mapped to single kernel-level thread.





■ Multithreading Models

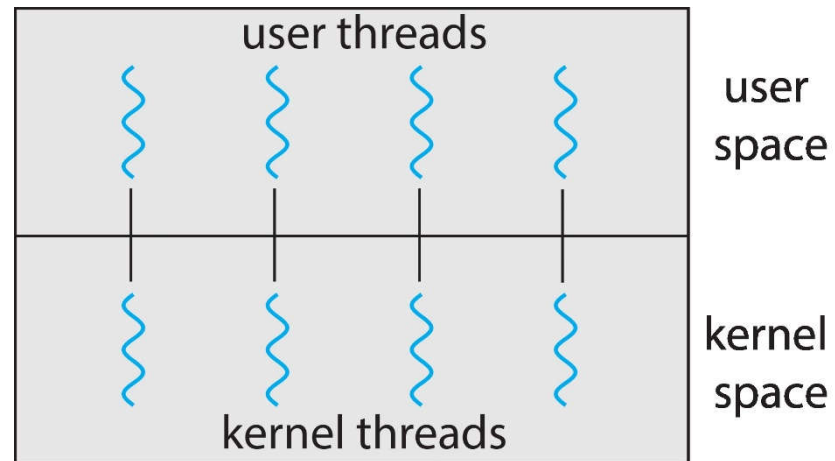
■ Many-to-One Model

- Efficiently, threads are managed by the thread library in user space.
- One thread blocking causes the process (and all its threads) to block.
- Multiple threads may not run in parallel on multicore system because only one thread may be in kernel at a time.
 - Few systems currently use this model.
- Examples
 - Solaris Green Threads
 - GNU Portable Threads

■ Multithreading Models

■ One-to-One Model

- Each user-level thread maps to a kernel-level thread.





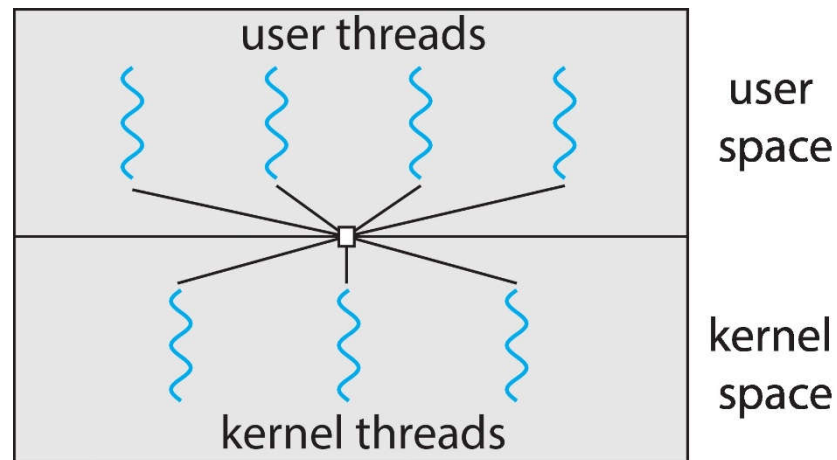
■ Multithreading Models

■ One-to-One Model

- Creating a user-level thread creates a kernel thread.
- More concurrency than many-to-one.
 - allowing another thread to run when a thread is blocking
 - allowing multiple threads to run in parallel on multiprocessors
- Number of threads per process sometimes restricted due to the overhead of creating kernel threads which may burden the performance of the system.
- Examples
 - Windows
 - *Linux*
 - Solaris 9 and later.

■ Multithreading Models

- Many-to-Many Model
 - Multiplexes many user-level threads to a smaller or equal number of kernel threads.



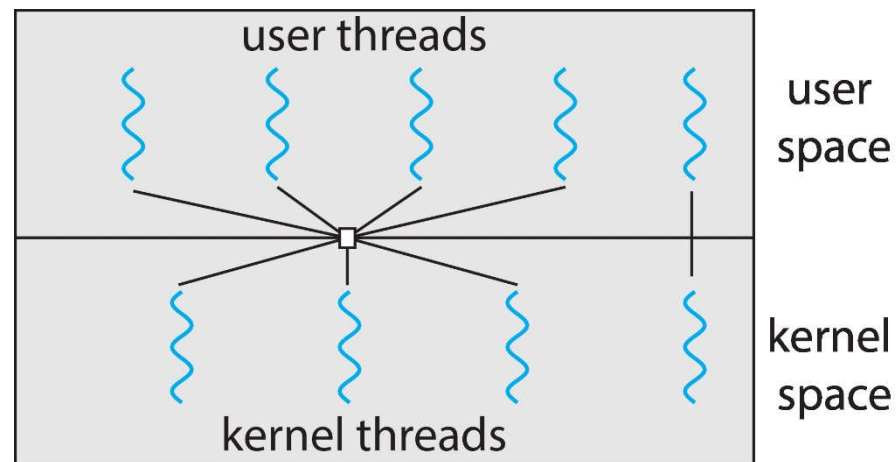
■ Multithreading Models

- Many-to-Many Model
 - Developers can create a sufficient number of user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
 - When a thread blocking, the kernel can schedule another thread for execution.
 - Examples
 - Solaris prior to version 9.
 - Windows with the *ThreadFiber* package.

■ Multithreading Models

■ Two-level Model

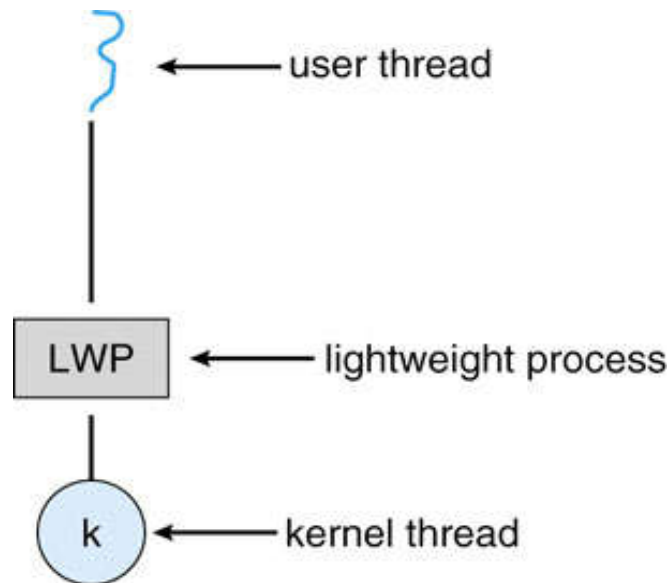
- Similar to Many-to-Many model, except that it allows a user thread to be **bound** to a kernel thread.
- Examples
 - IRIX (by SGI for graphic workstation)
 - **HP-UX**
 - Tru64 UNIX
 - Solaris 8 and earlier.





■ Multithreading Models

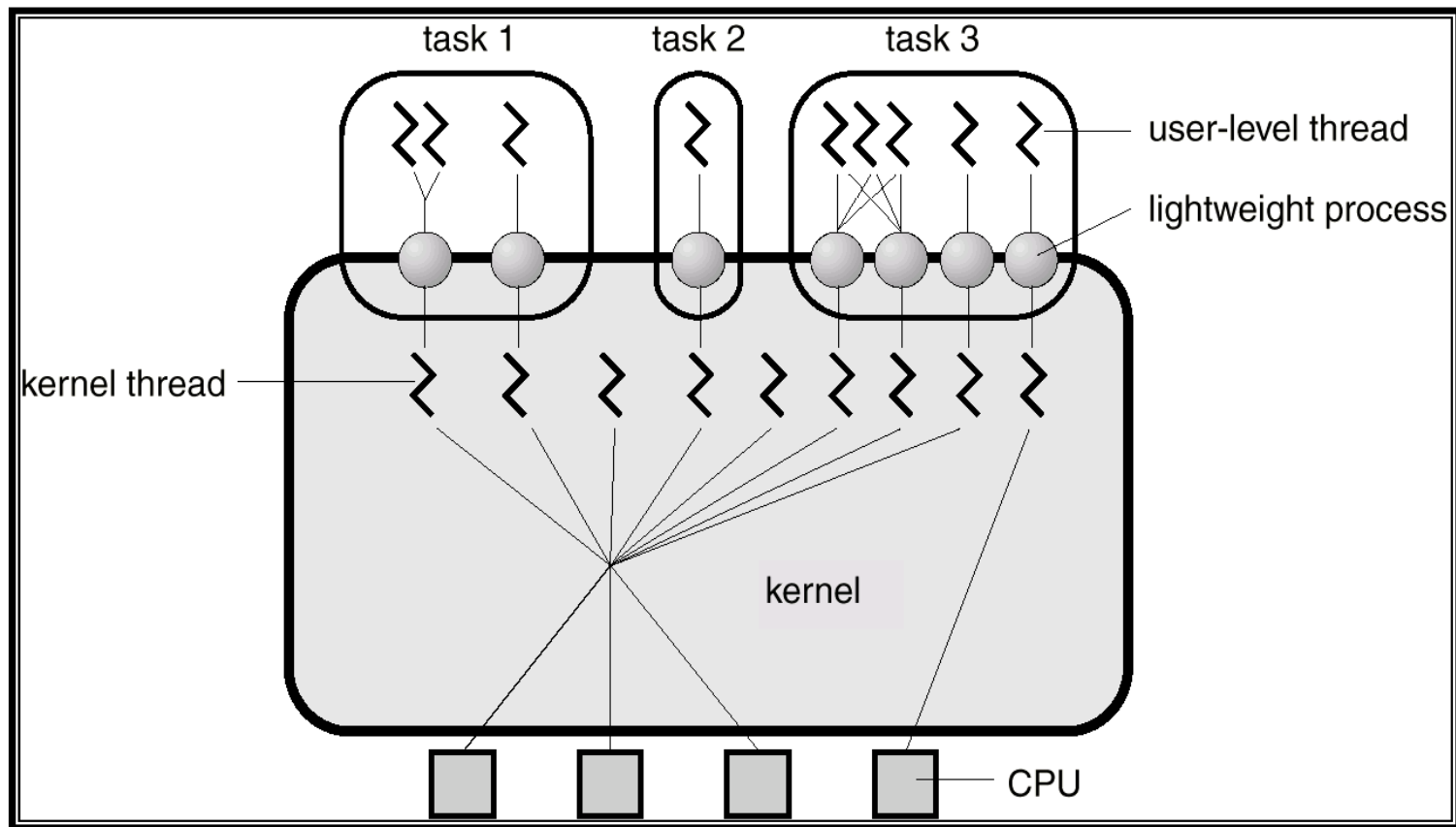
- Lightweight Process (LWP)





■ Multithreading Models

■ Example: Solaris 2 Threads



■ Multithreading Models

- Example: Java Threads
 - Java threads are managed by the JVM.
 - Typically implemented using the threads model provided by underlying OS.
 - Java threads may be created by:
 - Extending Thread class (at language-level)
 - Implementing the Runnable interface.

