Sistemi Operativi I

Corso di Laurea in Informatica

2025-2026

Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

SAPIENZA UNIVERSITÀ DI ROMA

Beyond Single-Threaded Processes

- The process model assumes that a process is an executing program with a **single thread** of control

Beyond Single-Threaded Processes

- The process model assumes that a process is an executing program with a **single thread** of control
- All modern operating systems provide features enabling a process to contain **multiple threads** of control

Beyond Single-Threaded Processes

- The process model assumes that a process is an executing program with a **single thread** of control
- All modern operating systems provide features enabling a process to contain **multiple threads** of control
- We introduce many concepts associated with multi-threaded computer systems

Beyond Single-Threaded Processes

- The process model assumes that a process is an executing program with a **single thread** of control
- All modern operating systems provide features enabling a process to contain **multiple threads** of control
- We introduce many concepts associated with multi-threaded computer systems
- We look at a number of issues related to multi-threaded programming and its effect on the design of operating systems

Threads: Overview

- A **thread** is a basic unit of CPU utilization, consisting of a program counter, a stack, a set of registers and a thread ID

Threads: Overview

- A **thread** is a basic unit of CPU utilization, consisting of a program counter, a stack, a set of registers and a thread ID
- Traditional (heavyweight) processes have a single thread of control
  - There is only one program counter, and one sequence of instructions that can be carried out at any given time

Threads: Overview

- A **thread** is a basic unit of CPU utilization, consisting of a program counter, a stack, a set of registers and a thread ID
- Traditional (heavyweight) processes have a single thread of control
  - There is only one program counter, and one sequence of instructions that can be carried out at any given time
- Multi-threaded applications have multiple threads within a single process, each having their own program counter, stack, and set of registers
  - But sharing common code, data, and certain structures, such as open files

Process vs. Thread

- A **process** defines the address space, text (code), data, resources, etc.

Process vs. Thread

- A **process** defines the address space, text (code), data, resources, etc.
- A **thread** defines a single sequential execution stream *within* a process (i.e., program counter, stack, registers)

Process vs. Thread

- A **process** defines the address space, text (code), data, resources, etc.
- A **thread** defines a single sequential execution stream *within* a process (i.e., program counter, stack, registers)
- A thread is bound to a specific process

Process vs. Thread

- Each process may have several threads of control within it

Process vs. Thread

- Each process may have several threads of control within it
- The process' address space is shared among all its threads

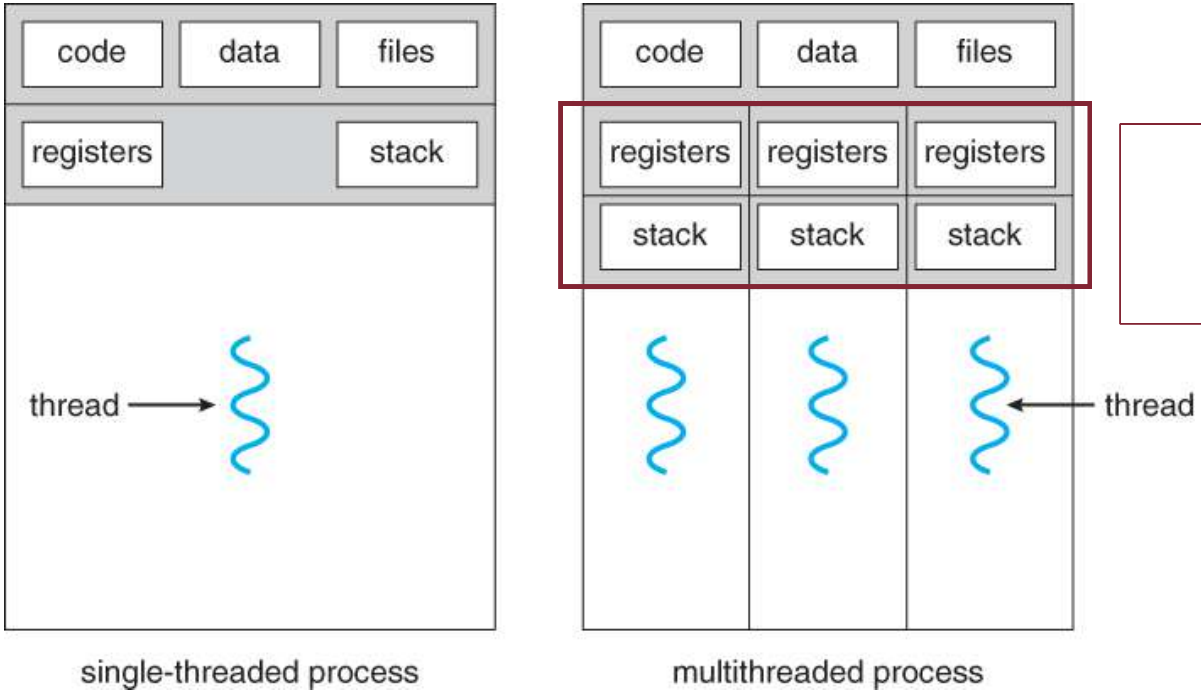
Process vs. Thread

- Each process may have several threads of control within it
- The process' address space is shared among all its threads
- No system calls are required for threads to cooperate with each other

Process vs. Thread

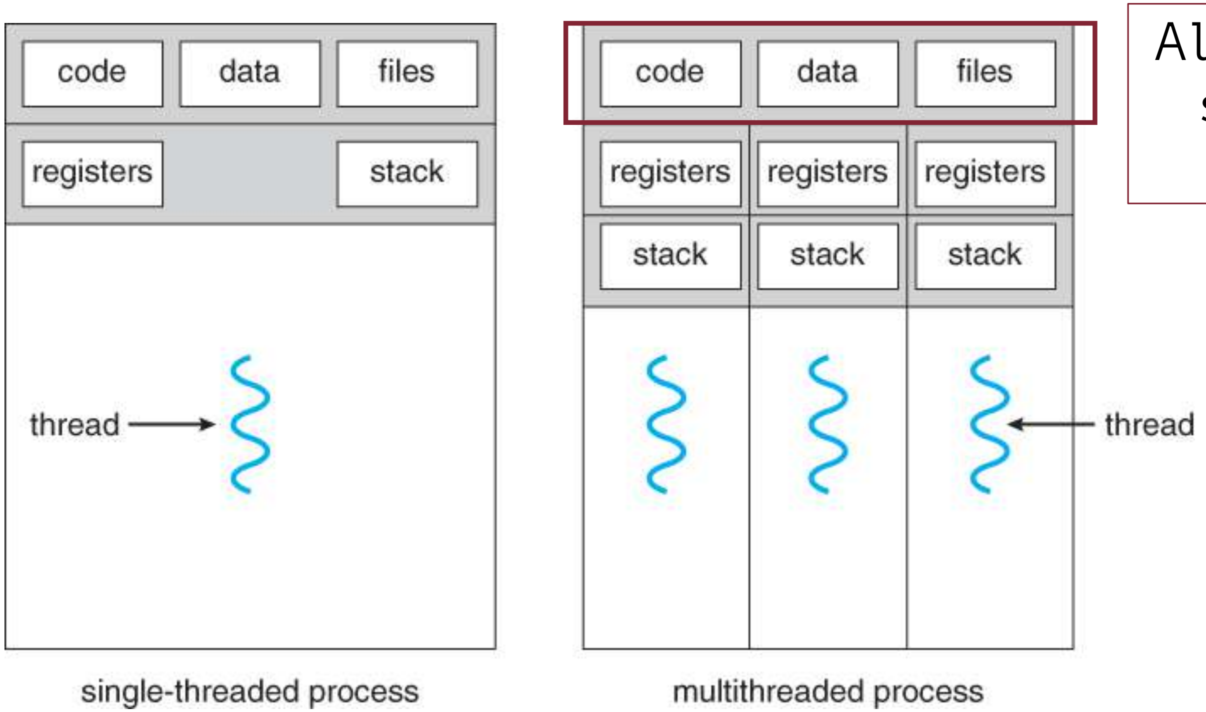
- Each process may have several threads of control within it
- The process' address space is shared among all its threads
- No system calls are required for threads to cooperate with each other
- Simpler than message passing and shared memory

Single- vs. Multi-Threaded Process



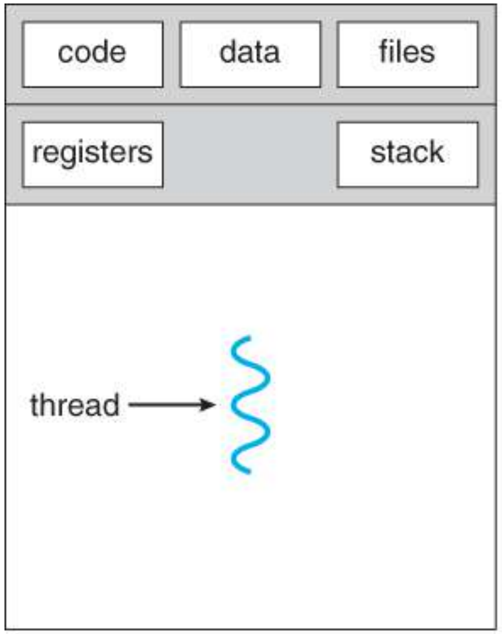
Each thread has its own independent set of registers and "state"

Single- vs. Multi-Threaded Process

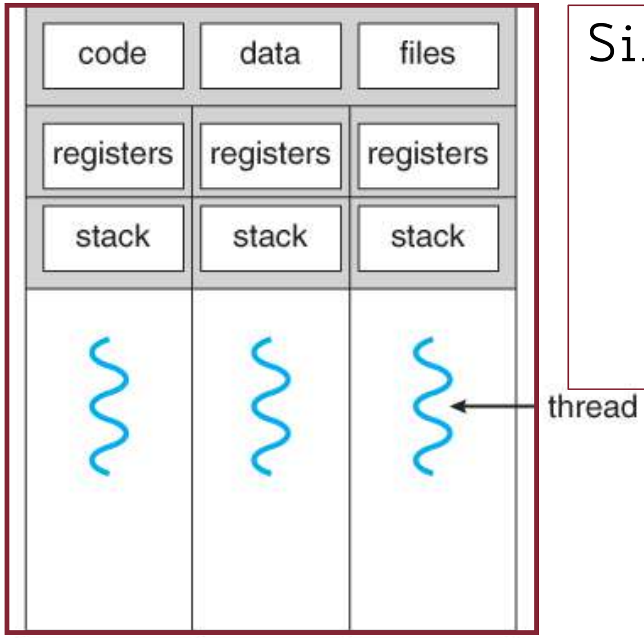


All the threads of a process share the same code and "global" resources

Single- vs. Multi-Threaded Process



single-threaded process



multithreaded process

Since all the threads live in the same address space, communication between them is easier than communication between processes

Threads: Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently

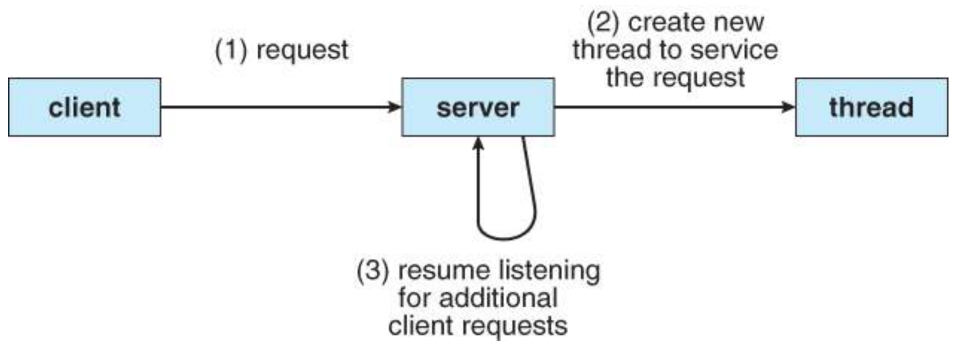
Threads: Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently
- Threading allows overlap of I/O with other tasks within the same program
  - Similar to what multiprogramming does across different processes

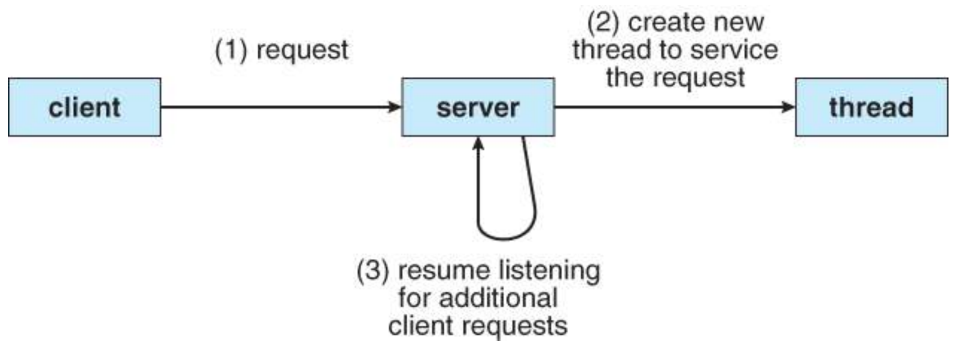
Threads: Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently
- Threading allows overlap of I/O with other tasks within the same program
  - Similar to what multiprogramming does across different processes
- **Example: word processor**
  - a thread may check grammar while another thread handles user input (keystrokes), and a third does periodic backups of the file being edited

Multi-threaded Web Server

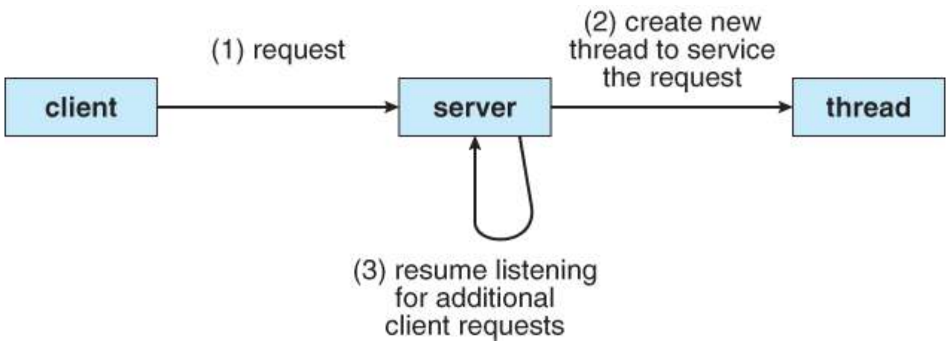


Multi-threaded Web Server



Multiple threads allow for multiple requests to be satisfied simultaneously, without having to serve requests sequentially or to fork off separate processes for every incoming request

Multi-threaded Web Server



What if the server process spawns off a new process for each incoming request rather than a thread?

Multiple Processes vs. Multiple Threads

- Theoretically, each sub-task of an application could be implemented as a new single-threaded process rather than a multi-threaded process

Multiple Processes vs. Multiple Threads

- Theoretically, each sub-task of an application could be implemented as a new single-threaded process rather than a multi-threaded process
- There are at least **2 reasons** why this is not the best choice:

Multiple Processes vs. Multiple Threads

- Theoretically, each sub-task of an application could be implemented as a new single-threaded process rather than a multi-threaded process
- There are at least **2 reasons** why this is not the best choice:
    - Inter-thread communication is significantly quicker than inter-process one

Multiple Processes vs. Multiple Threads

- Theoretically, each sub-task of an application could be implemented as a new single-threaded process rather than a multi-threaded process
- There are at least **2 reasons** why this is not the best choice:
    - Inter-thread communication is significantly quicker than inter-process one
    - Context-switches between threads is a lot faster than between processes

Threads: Benefits

- 4 main benefits:

Threads: Benefits

- 4 main benefits:
  - **Responsiveness** → one thread may provide rapid response while other threads are blocked or slowed down doing intensive computations

Threads: Benefits

- 4 main benefits:
  - Responsiveness → one thread may provide rapid response while other threads are blocked or slowed down doing intensive computations
  - Resource sharing → threads share common code, data, and address space

Threads: Benefits

- 4 main benefits:
  - **Responsiveness** → one thread may provide rapid response while other threads are blocked or slowed down doing intensive computations
  - **Resource sharing** → threads share common code, data, and address space
  - **Economy** → creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes

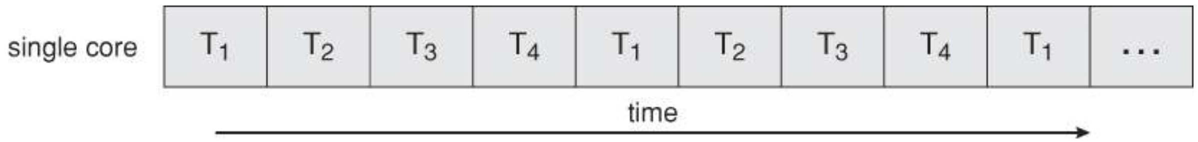
Threads: Benefits

- 4 main benefits:
  - Responsiveness → one thread may provide rapid response while other threads are blocked or slowed down doing intensive computations
  - Resource sharing → threads share common code, data, and address space
  - Economy → creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes
  - Scalability (multi-processor architectures) → A single threaded process can only run on one CPU, whereas a multi-threaded process may be split amongst all available processors/cores

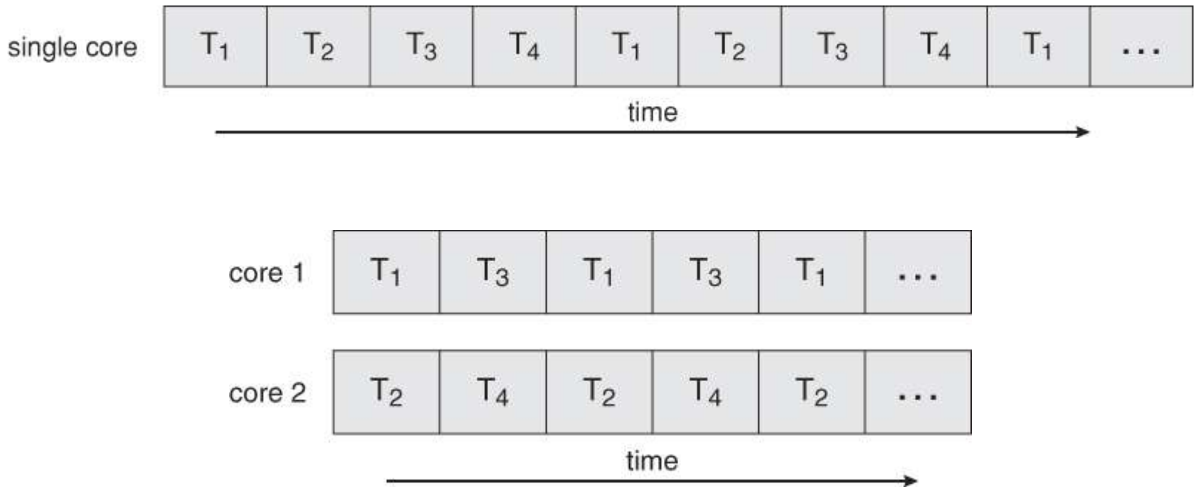
Multi-core Programming

- A recent trend in computer architecture is to produce chips with multiple cores, or CPUs on a single chip
- A multi-threaded application running on a traditional single-core chip would have to interleave the threads
- On a multi-core chip, however, threads could be spread across the available cores, allowing **true parallel processing**!

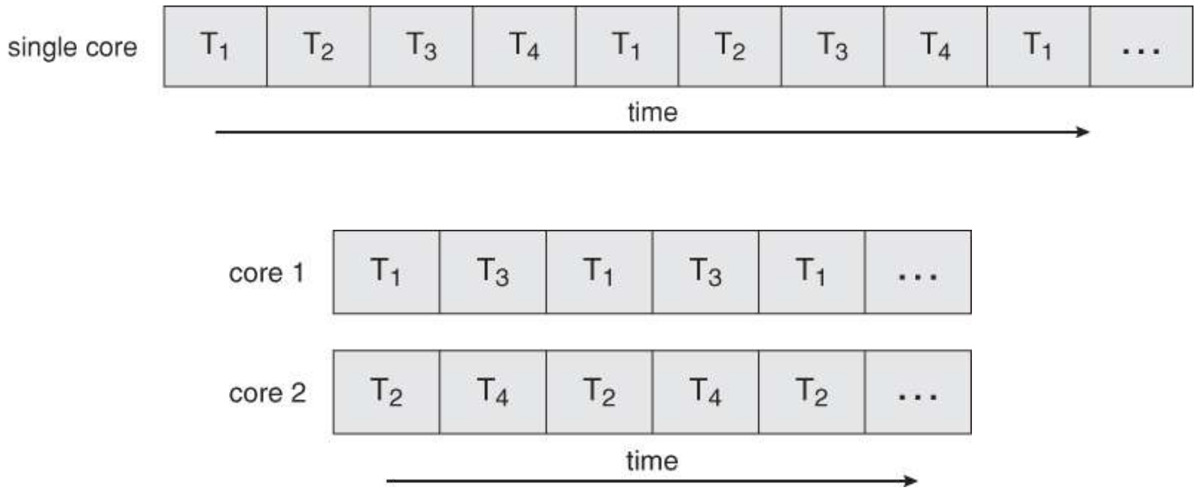
Single- vs. Multi-core Programming



Single- vs. Multi-core Programming

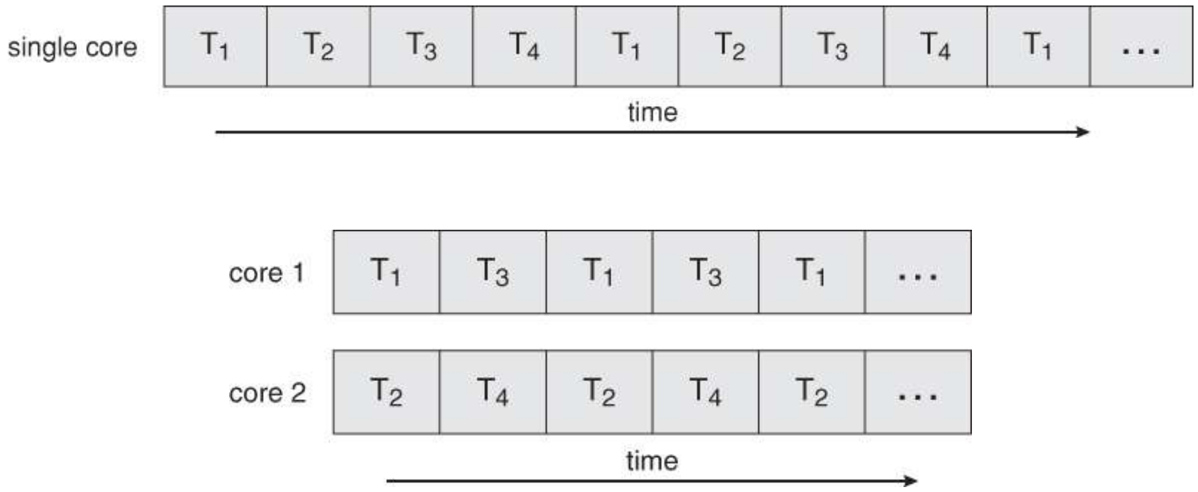


Single- vs. Multi-core Programming



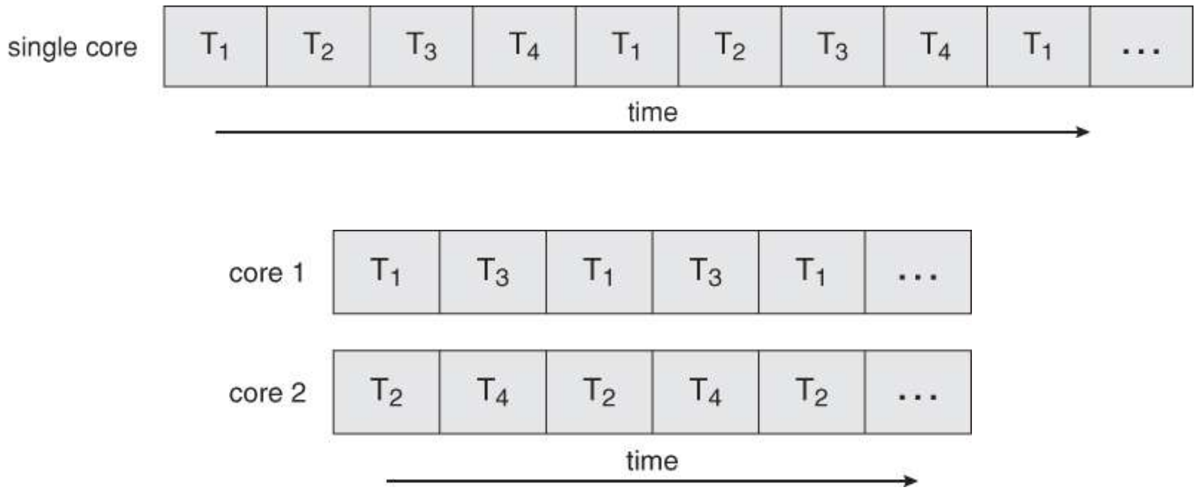
Multi-core chips require new OS scheduling algorithms to make better use of the multiple cores available

Single- vs. Multi-core Programming



CPUs have been developed to support more simultaneous threads per core in hardware (e.g., Intel's hyper-threading)

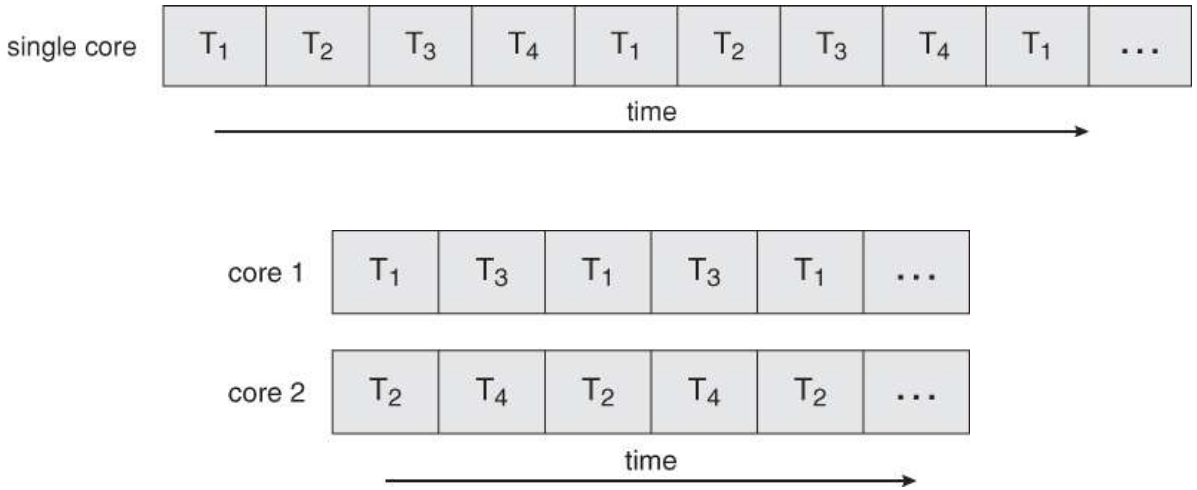
Single- vs. Multi-core Programming



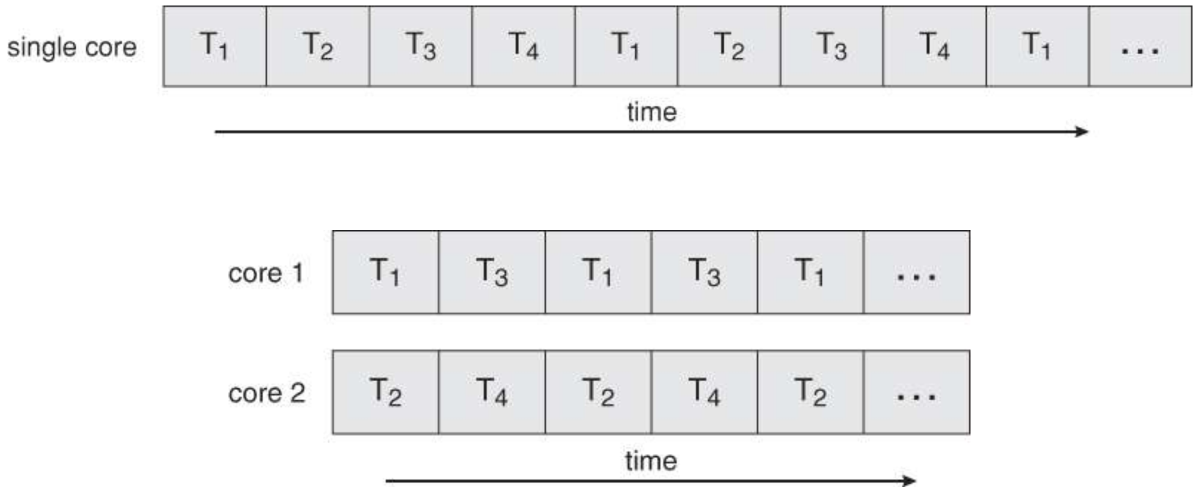
Hyper-threading

Each physical core appears as two processors to the OS, allowing concurrent scheduling of two threads per core

Single- vs. Multi-core Programming



Single- vs. Multi-core Programming



Concurrency

vs.

Parallelism

Types of Parallelism

- In theory, there are **2 ways** to parallelize the workload:

Types of Parallelism

- In theory, there are **2 ways** to parallelize the workload:
  - **Data parallelism:** divides the data up amongst multiple cores (threads), and performs the same task on each chunk of the data

Types of Parallelism

- In theory, there are **2 ways** to parallelize the workload:
  - **Data parallelism:** divides the data up amongst multiple cores (threads), and performs the same task on each chunk of the data
  - **Task parallelism:** divides the different tasks to be performed among the different cores and performs them simultaneously

Example: A Pure CPU-bound Task

- Suppose you are asked to implement a simple program that:
  - Takes as input a positive integer N
  - Produces as output the total sum from 1 to N

Example: A Pure CPU-bound Task

- Suppose you are asked to implement a simple program that:
  - Takes as input a positive integer N
  - Produces as output the total sum from 1 to N
- The easiest solution is something as follows:

```
int sum = 0;
for (int i=1; i <= N; ++i) {
    sum += i;
}
return sum;
```

Example: A Pure CPU-bound Task

- Suppose you are asked to implement a simple program that:
  - Takes as input a positive integer N
  - Produces as output the total sum from 1 to N
- The easiest solution is something as follows:

```
int sum = 0;
for (int i=1; i <= N; ++i) {
    sum += i;
}
return sum;
```

CPU-bound

Example: A Pure CPU-bound Task

- If N grows large it may take a while…

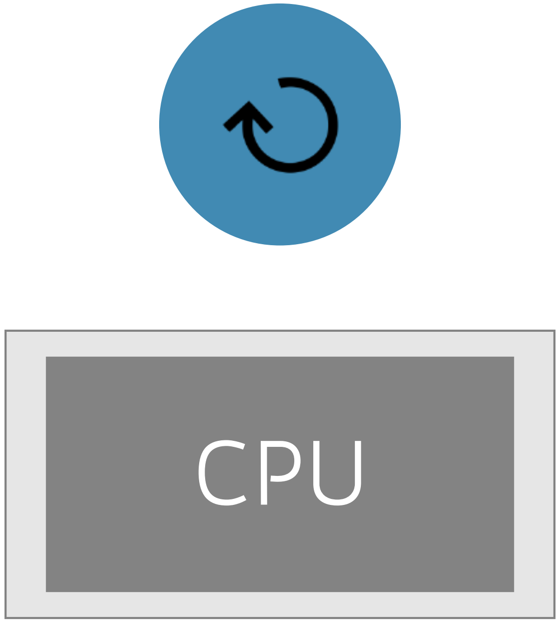
Example: A Pure CPU-bound Task

- If N grows large it may take a while…
- Based on the underlying HW, can we improve the performance of the previously single-threaded process?

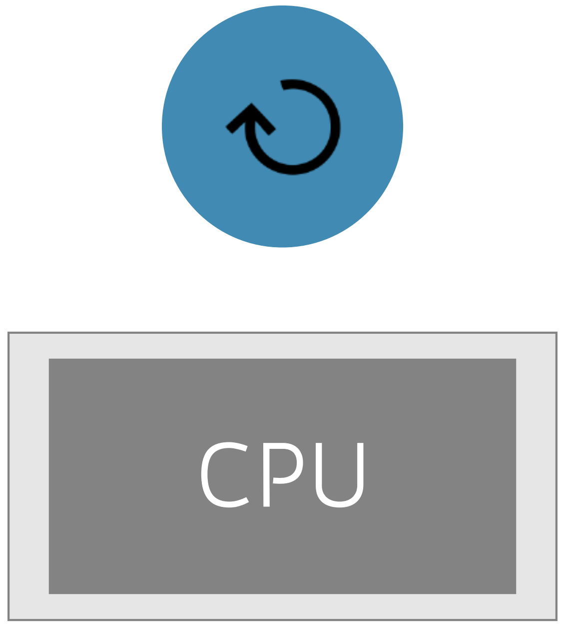
Example: A Pure CPU-bound Task

- If N grows large it may take a while…
- Based on the underlying HW, can we improve the performance of the previously single-threaded process?
- We will consider the following setups:
  - Number of CPU cores: 1 vs. M
  - Processes/Threads: 1/1 vs. M/1 vs. 1/M

1 CPU Core, 1 Process, 1 Thread



1 CPU Core, 1 Process, 1 Thread

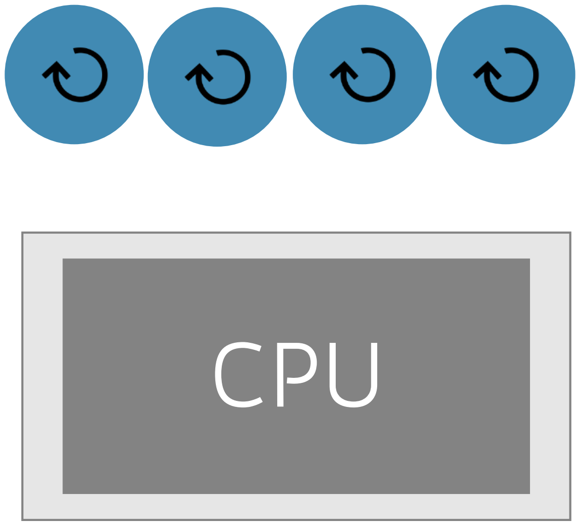


No Parallelism

No Concurrency

1 CPU Core, M Processes, 1 Thread

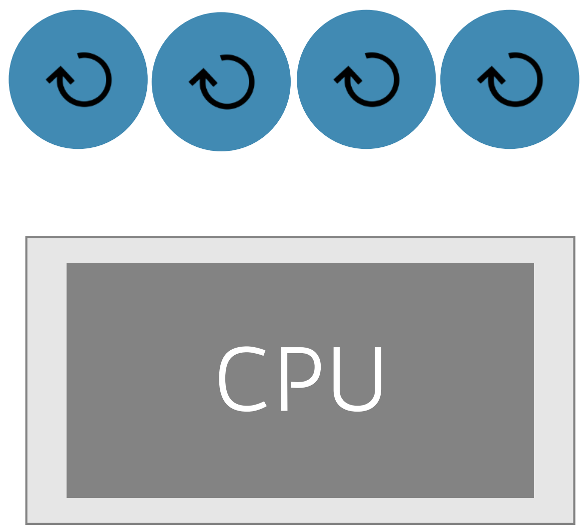
Divide N into M chunks: {[1, ..., N/M], [(N/M)+1, ... , 2N/M], ..., [(M-1)(N/M)+1, ..., N/M]}



1 CPU Core, M Processes, 1 Thread

Divide N into M chunks: {[1, …, N/M], [(N/M)+1, … , 2N/M], …, [(M-1)(N/M)+1, …, N/M]}

e.g., N = 1000; M=8: {[1, …, 125], [126, …, 250], …, [876, …, 1000]}

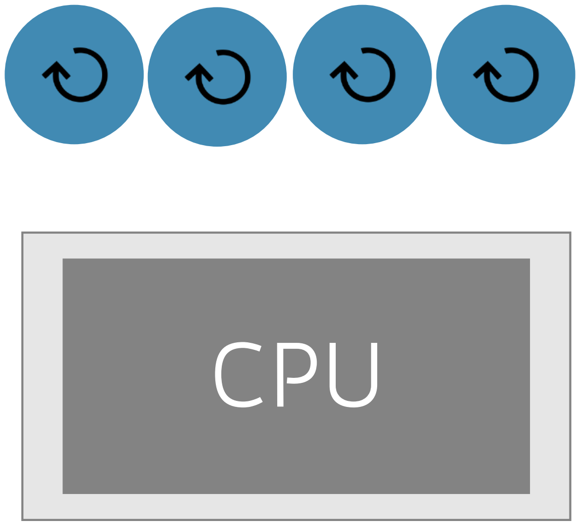


1 CPU Core, M Processes, 1 Thread

Divide N into M chunks: {[1, …, N/M], [(N/M)+1, … , 2N/M], …, [(M-1)(N/M)+1, …, N/M]}

e.g., N = 1000; M=8: {[1, …, 125], [126, …, 250], …, [876, …, 1000]}

The i-th process computes the sum of the numbers contained in the i-th chunk

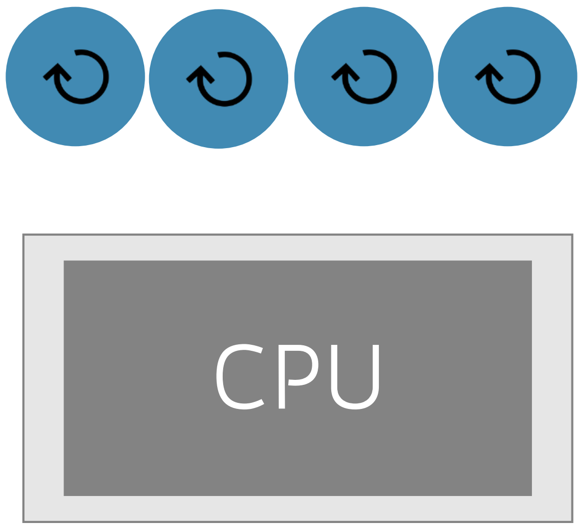


1 CPU Core, M Processes, 1 Thread

Divide N into M chunks: {[1, ..., N/M], [(N/M)+1, ... , 2N/M], ..., [(M-1)(N/M)+1, ..., N/M]}

e.g., N = 1000; M=8: {[1, ..., 125], [126, ..., 250], ..., [876, ..., 1000]}

The i-th process computes the sum of the numbers contained in the i-th chunk

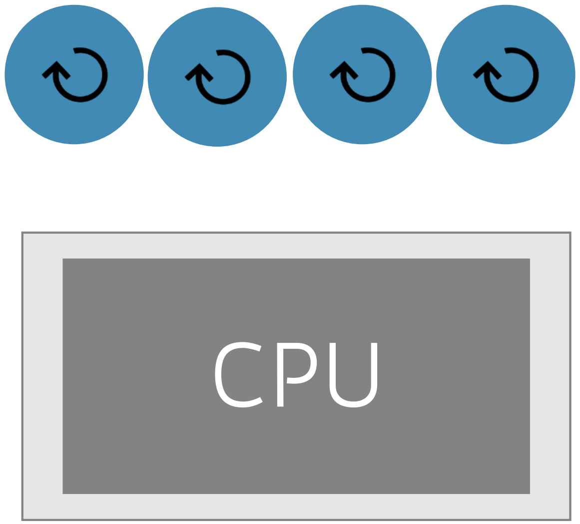


No Parallelism

Concurrency (among processes)

1 CPU Core, M Processes, 1 Thread

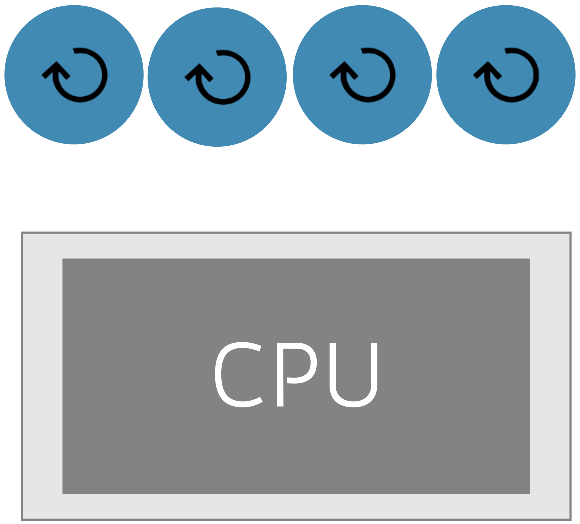
Will this solution get any speedup to the whole computation?



1 CPU Core, M Processes, 1 Thread

Will this solution get any speedup to the whole computation?

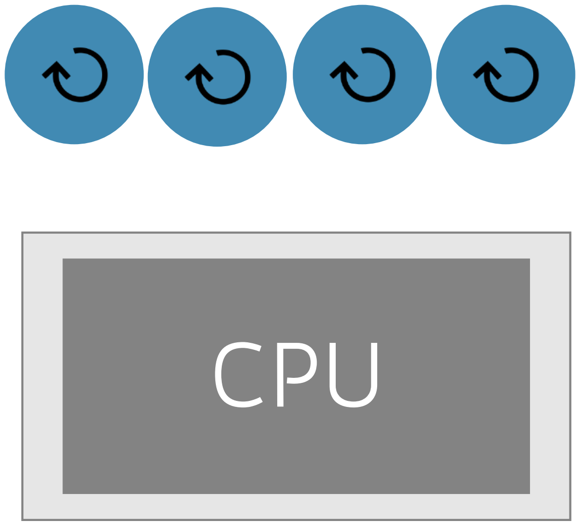
NO!



1 CPU Core, M Processes, 1 Thread

Will this solution get any speedup to the whole computation?

Only one process is running on a single CPU core

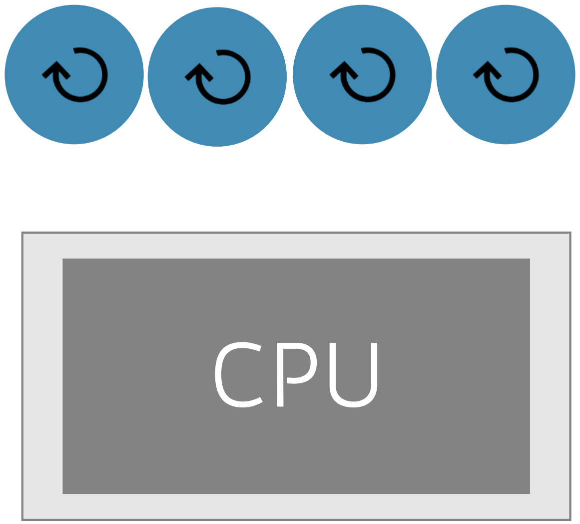


1 CPU Core, M Processes, 1 Thread

Will this solution get any speedup to the whole computation?

Only one process is running on a single CPU core

All the M processes must finish to get the final result

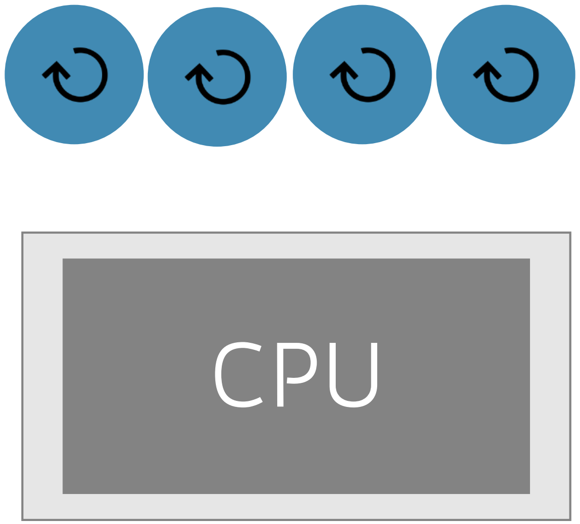


1 CPU Core, M Processes, 1 Thread

Will this solution get any speedup to the whole computation?

Only one process is running on a single CPU core

All the M processes must finish to get the final result

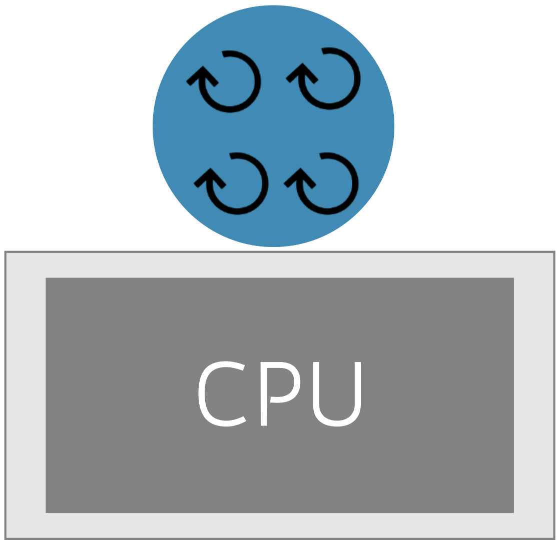


Eventually, each process must communicate its partial sum to the others

Inter-Process Communication

1 CPU Core, 1 Process, M Threads

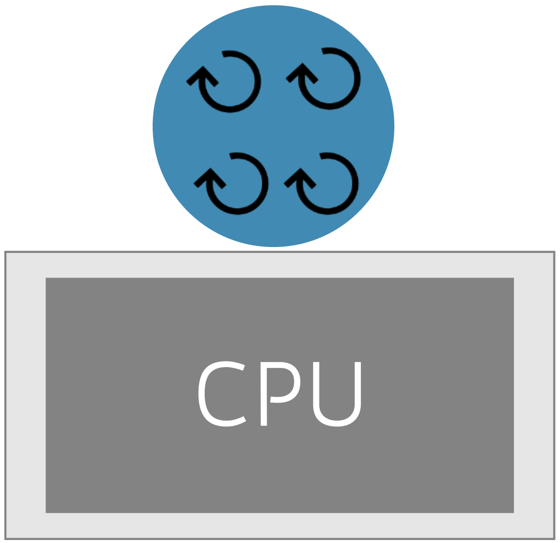
Will this solution get any speedup to the whole computation?



1 CPU Core, 1 Process, M Threads

Will this solution get any speedup to the whole computation?

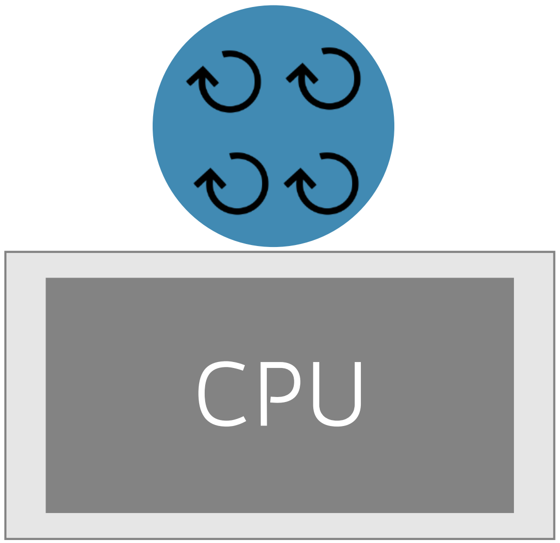
NO!



1 CPU Core, 1 Process, M Threads

Will this solution get any speedup to the whole computation?

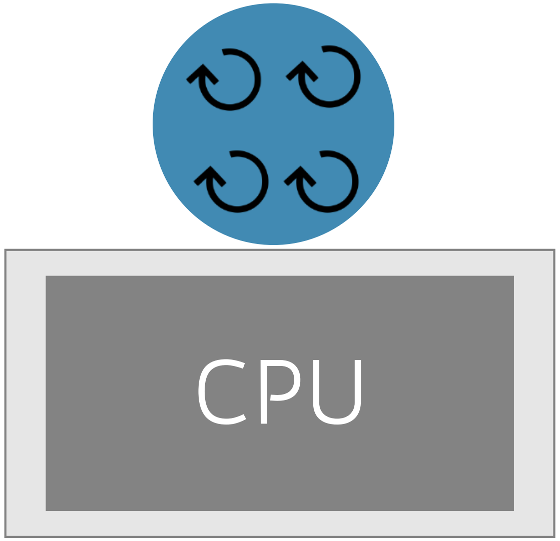
Only one thread is running on a single CPU core



1 CPU Core, 1 Process, M Threads

Will this solution get any speedup to the whole computation?

Only one thread is running on a single CPU core

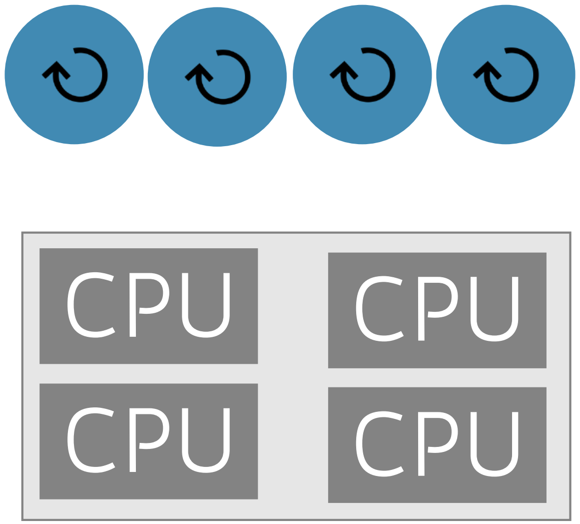


The only advantage is that each thread can easily share its partial sum with the others!

No Inter-Process Communication

M CPU Cores, M Processes, 1 Thread

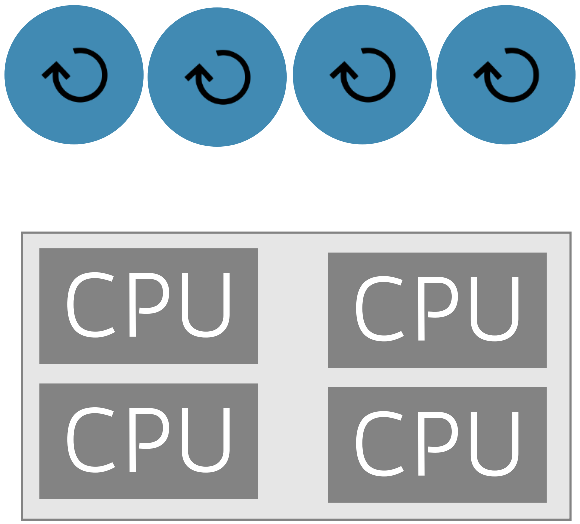
Will this solution get any speedup to the whole computation?



M CPU Cores, M Processes, 1 Thread

Will this solution get any speedup to the whole computation?

YES!

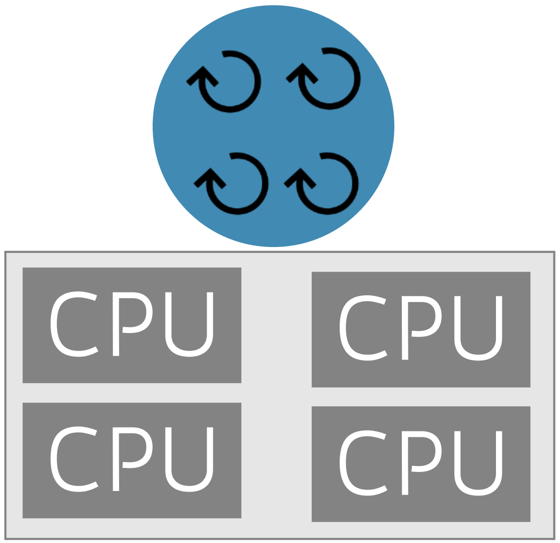


True Parallelism

Still, each process must communicate its partial sum to the others

M CPU Cores, 1 Process, M Threads

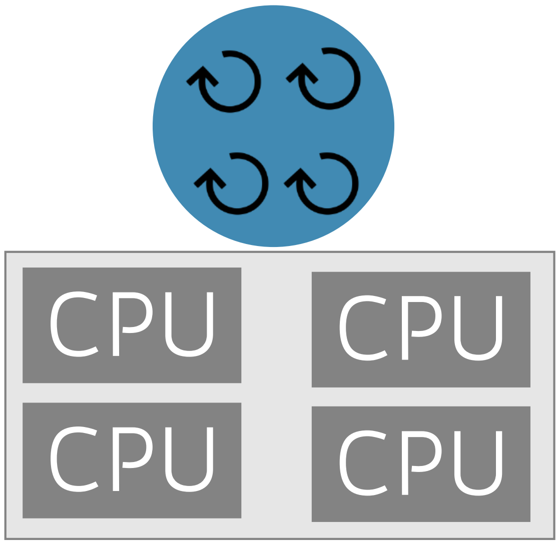
Will this solution get any speedup to the whole computation?



M CPU Cores, 1 Process, M Threads

Will this solution get any speedup to the whole computation?

YES!



True Parallelism

No Inter-Process Communication

A Mixed CPU- and IO-bound Task

- There are a lot of complex problems that are **both** CPU and I/O intensive **in phases**

A Mixed CPU- and IO-bound Task

- There are a lot of complex problems that are **both** CPU and I/O intensive **in phases**
- I/O-intensive when CPU is free, CPU-intensive when I/O finished or side-by-side

A Mixed CPU- and IO-bound Task

- There are a lot of complex problems that are **both** CPU and I/O intensive **in phases**
- I/O-intensive when CPU is free, CPU-intensive when I/O finished or side-by-side
- For instance:
  - Distributed processing of high volumes of data

A Mixed CPU- and IO-bound Task

- There are a lot of complex problems that are **both** CPU and I/O intensive **in phases**
- I/O-intensive when CPU is free, CPU-intensive when I/O finished or side-by-side
- For instance:
  - Distributed processing of high volumes of data
  - Disk defragmentation

A Mixed CPU- and IO-bound Task

- There are a lot of complex problems that are **both** CPU and I/O intensive **in phases**
- I/O-intensive when CPU is free, CPU-intensive when I/O finished or side-by-side
- For instance:
  - Distributed processing of high volumes of data
  - Disk defragmentation
  - Compression/Decompression algorithms (side-by-side)

A Mixed CPU- and I/O-bound Task

- In all these cases, multi-threading can be useful **even on a single-core CPU**

A Mixed CPU- and I/O-bound Task

- In all these cases, multi-threading can be useful **even on a single-core CPU**
- Indeed, it might pay to split CPU- and I/O-intensive tasks of an application into separate threads

A Mixed CPU- and I/O-bound Task

- In all these cases, multi-threading can be useful **even on a single-core CPU**
- Indeed, it might pay to split CPU- and I/O-intensive tasks of an application into separate threads
- This way the CPU- and I/O-bound threads can alternate on the CPU

A Mixed CPU- and I/O-bound Task

- In all these cases, multi-threading can be useful **even on a single-core CPU**
- Indeed, it might pay to split CPU- and I/O-intensive tasks of an application into separate threads
- This way the CPU- and I/O-bound threads can alternate on the CPU
- This slows down the CPU-bound thread a little, but reduces or eliminates the I/O-bound gap

To Wrap Up

- A **thread** is a single execution stream within a process

To Wrap Up

- A **thread** is a single execution stream within a process
- **Thread** vs. **Process**:
  - common vs. separate address spaces → **quicker communication**
  - lightweight vs. heavyweight → **faster context switching**

To Wrap Up

- A **thread** is a single execution stream within a process
- **Thread** vs. **Process**:
  - common vs. separate address spaces → **quicker communication**
  - lightweight vs. heavyweight → **faster context switching**
- On a single core:
  - Fully CPU-bound processes do not take advantage of multi-threading
  - Concurrency between threads in mixed CPU- and I/O-bound processes

Multi-threading: Support and Management

- Support for (multiple) threads can be provided in 2 ways:
  - at the kernel level → kernel threads
  - at the user level → user threads

Multi-threading: Support and Management

- Support for (multiple) threads can be provided in 2 ways:
  - at the kernel level → kernel threads
  - at the user level → user threads
- Kernel threads
  - managed directly by the OS kernel itself

Multi-threading: Support and Management

- Support for (multiple) threads can be provided in 2 ways:
  - at the kernel level → kernel threads
  - at the user level → user threads
- Kernel threads
  - managed directly by the OS kernel itself
- User threads
  - managed in user space by a user-level thread library, without OS intervention

Kernel Threads

- The smallest unit of execution that can be scheduled by the OS

Kernel Threads

- The smallest unit of execution that can be scheduled by the OS
- The OS is responsible for supporting and managing all threads

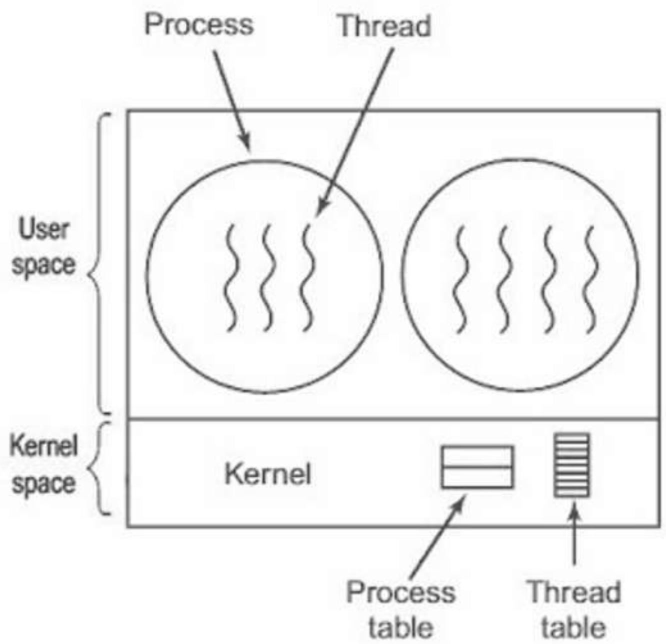
Kernel Threads

- The smallest unit of execution that can be scheduled by the OS
- The OS is responsible for supporting and managing all threads
- One Process Control Block (PCB) for each process, one Thread Control Block (TCB) for each thread

Kernel Threads

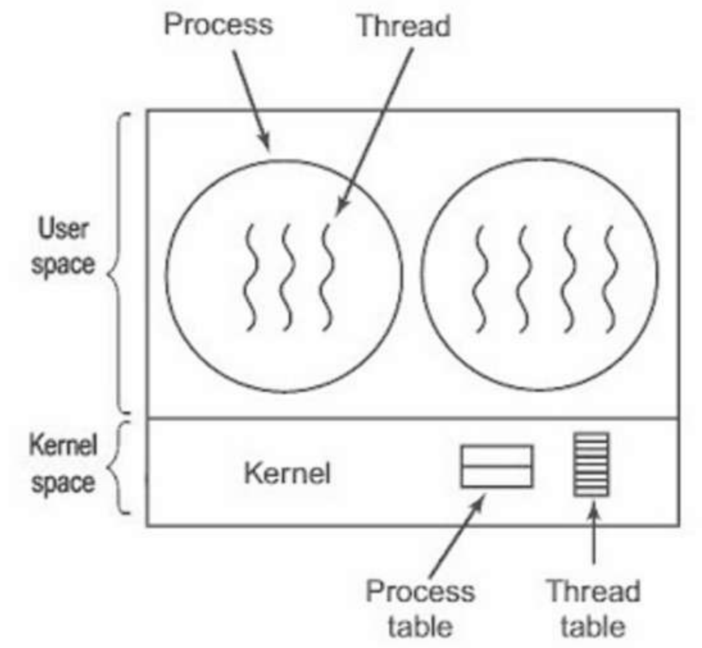
- The smallest unit of execution that can be scheduled by the OS
- The OS is responsible for supporting and managing all threads
- One Process Control Block (PCB) for each process, one Thread Control Block (TCB) for each thread
- The OS usually provides system calls to create and manage threads from user space

Kernel Threads: PROs



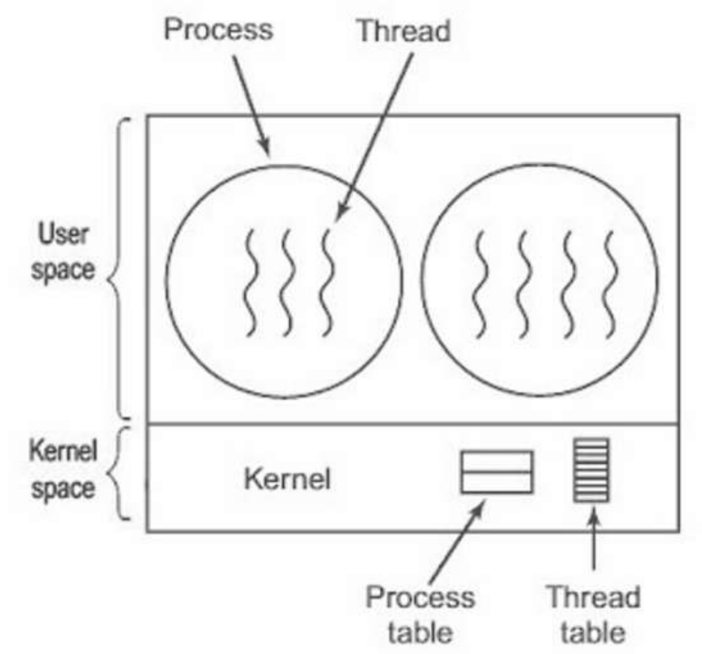
- PROs
  - The kernel has full knowledge of all threads
  - Scheduler may decide to give more CPU time to a process having a large numer of threads
  - Good for applications that frequently block
  - Switching between threads is faster than switching between processes

Kernel Threads: PROs



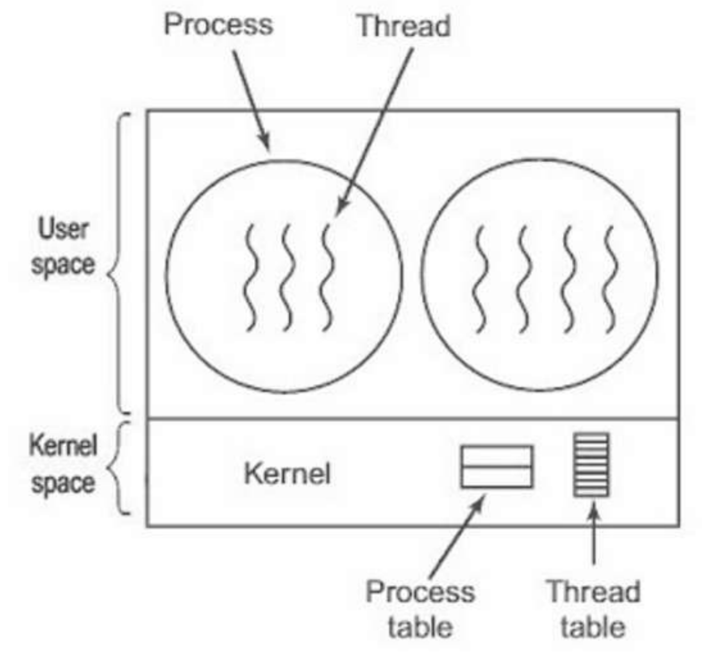
- PROs
  - The kernel has full knowledge of all threads
  - Scheduler may decide to give more CPU time to a process having a large numer of threads
  - Good for applications that frequently block
  - Switching between threads is faster than switching between processes

Kernel Threads: PROs



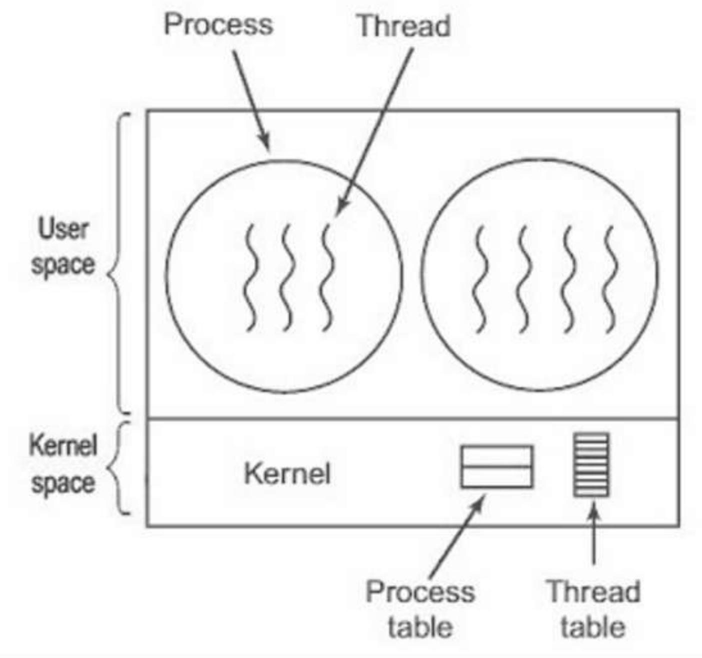
- PROs
  - The kernel has full knowledge of all threads
  - Scheduler may decide to give more CPU time to a process having a large numer of threads
  - Good for applications that frequently block
  - Switching between threads is faster than switching between processes

Kernel Threads: PROs



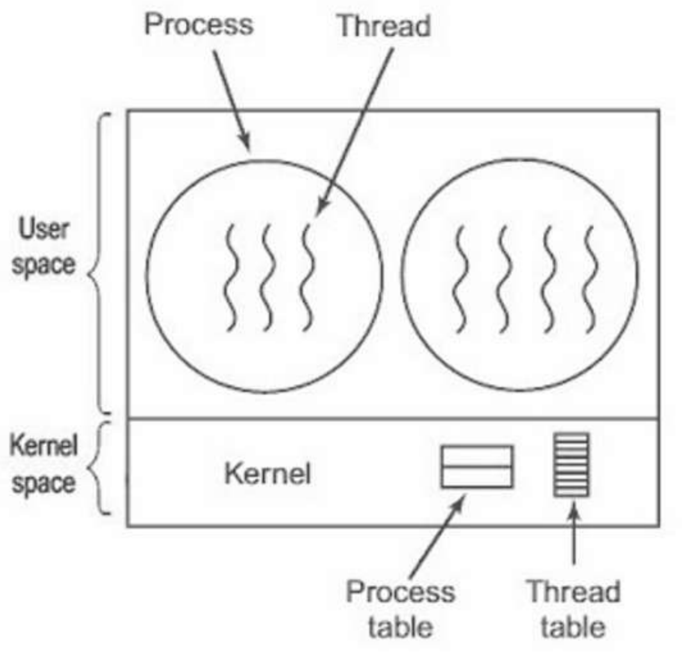
- PROs
  - The kernel has full knowledge of all threads
  - Scheduler may decide to give more CPU time to a process having a large numer of threads
  - Good for applications that frequently block
  - Switching between threads is faster than switching between processes

Kernel Threads: PROs



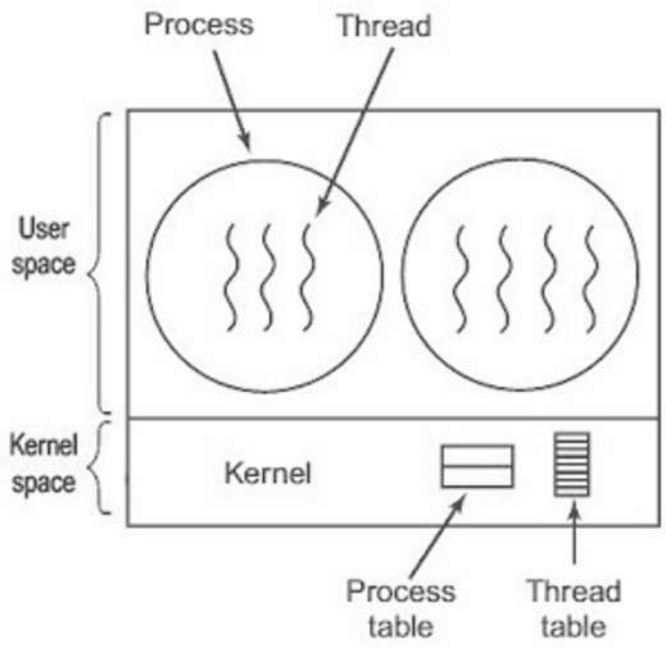
- PROs
  - The kernel has full knowledge of all threads
  - Scheduler may decide to give more CPU time to a process having a large numer of threads
  - Good for applications that frequently block
  - Switching between threads is faster than switching between processes

Kernel Threads: CONs



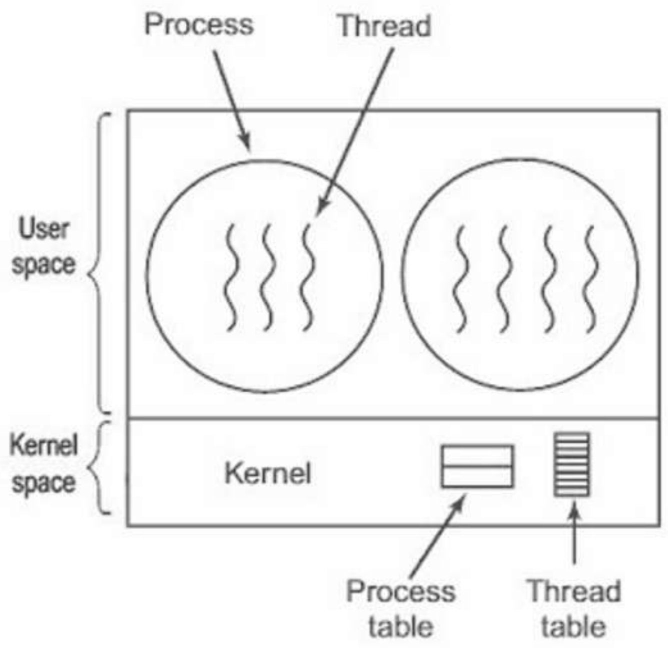
- CONs
  - Significant overhead and increase in kernel complexity
  - Slow and inefficient (need kernel invocations)
  - Context switching, although lighter, is managed by the kernel

Kernel Threads: CONs



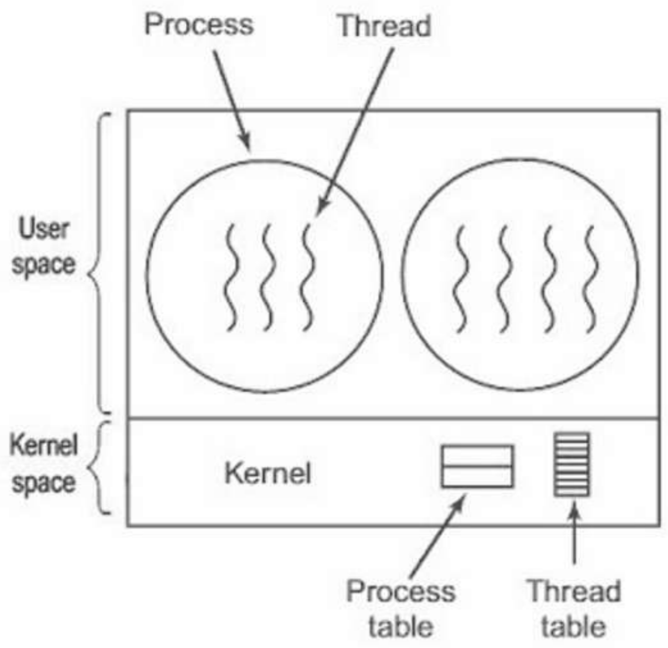
- CONs
  - Significant overhead and increase in kernel complexity
  - Slow and inefficient (need kernel invocations)
  - Context switching, although lighter, is managed by the kernel

Kernel Threads: CONs



- CONs
  - Significant overhead and increase in kernel complexity
  - Slow and inefficient (need kernel invocations)
  - Context switching, although lighter, is managed by the kernel

Kernel Threads: CONs



- CONs
  - Significant overhead and increase in kernel complexity
  - Slow and inefficient (need kernel invocations)
  - Context switching, although lighter, is managed by the kernel

User Threads

- Managed entirely by the run-time system (user-level thread library)

User Threads

- Managed entirely by the run-time system (user-level thread library)
- The OS kernel knows nothing about user-level threads

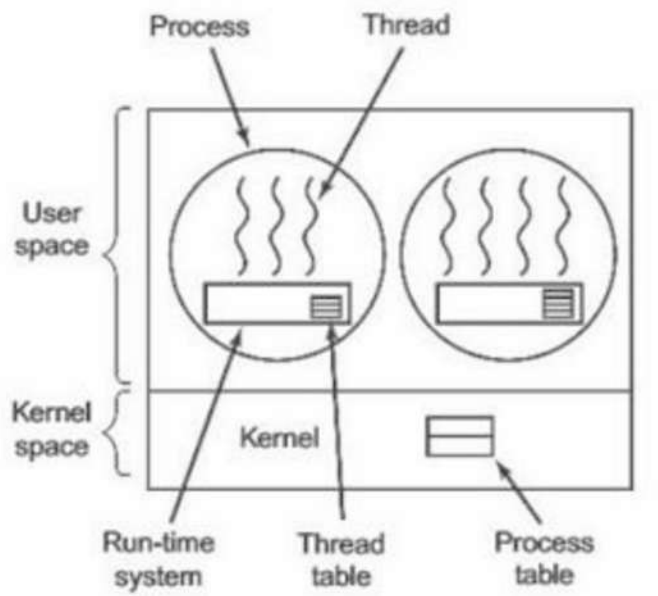
User Threads

- Managed entirely by the run-time system (user-level thread library)
- The OS kernel knows nothing about user-level threads
- The OS kernel manages user-level threads as if they where single-threaded processes

User Threads

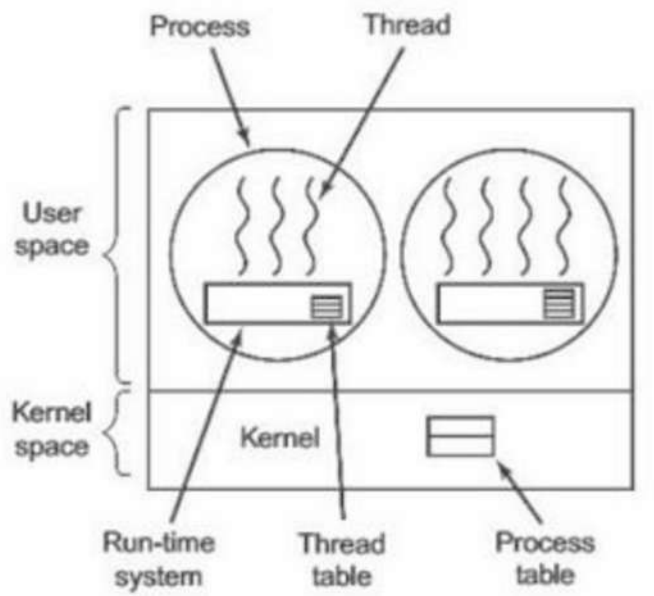
- Managed entirely by the run-time system (user-level thread library)
- The OS kernel knows nothing about user-level threads
- The OS kernel manages user-level threads as if they where single-threaded processes
- Ideally, thread operations should be as fast as a function call

User Threads: PROs



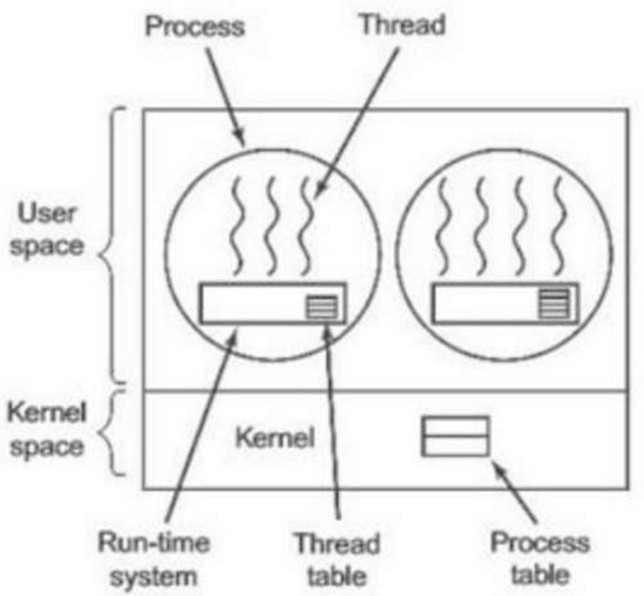
- PROs
  - Really fast and lightweight
  - Scheduling policies are more flexible
  - Can be implemented in OSs that do not support threading
  - No system calls involved, just user-space function calls
  - No actual context switch

User Threads: PROs



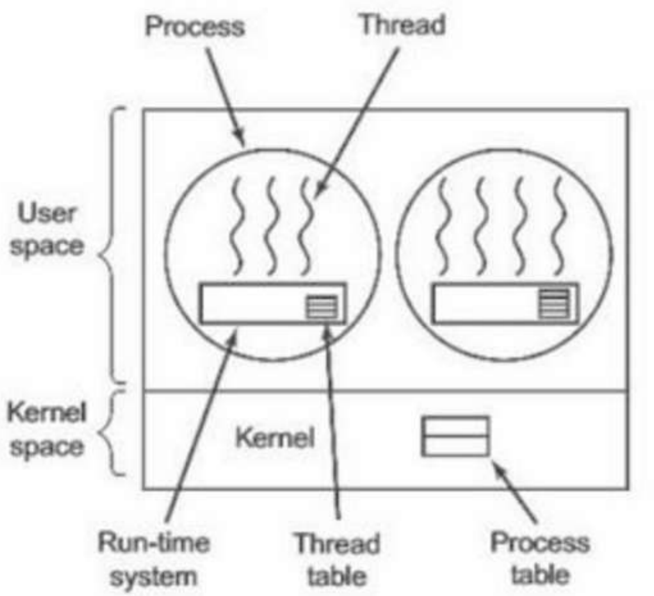
- PROs
  - Really fast and lightweight
  - Scheduling policies are more flexible
  - Can be implemented in OSs that do not support threading
  - No system calls involved, just user-space function calls
  - No actual context switch

User Threads: PROs



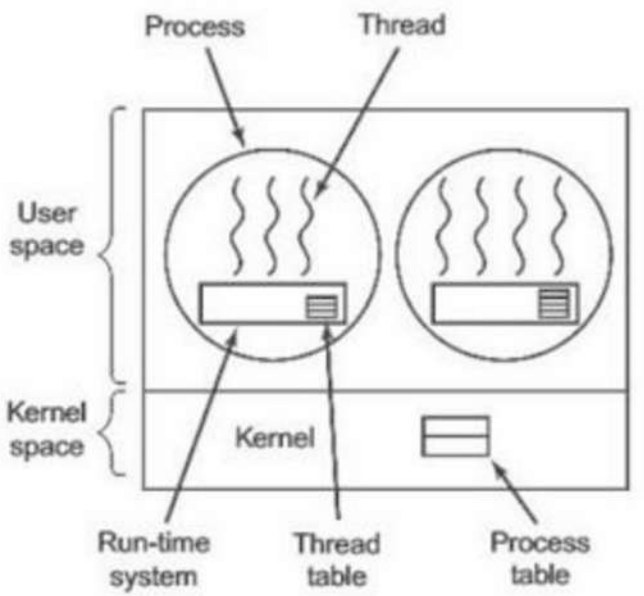
- PROs
  - Really fast and lightweight
  - Scheduling policies are more flexible
  - Can be implemented in OSs that do not support threading
  - No system calls involved, just user-space function calls
  - No actual context switch

User Threads: PROs



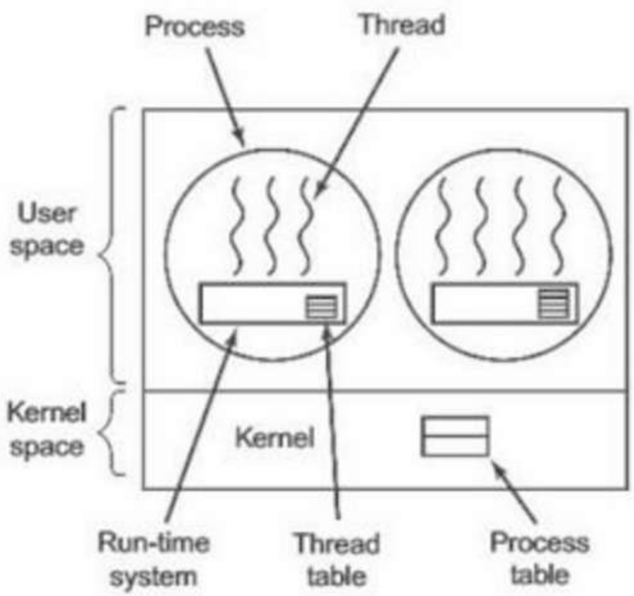
- PROs
  - Really fast and lightweight
  - Scheduling policies are more flexible
  - Can be implemented in OSs that do not support threading
  - No system calls involved, just user-space function calls
  - No actual context switch

User Threads: PROs



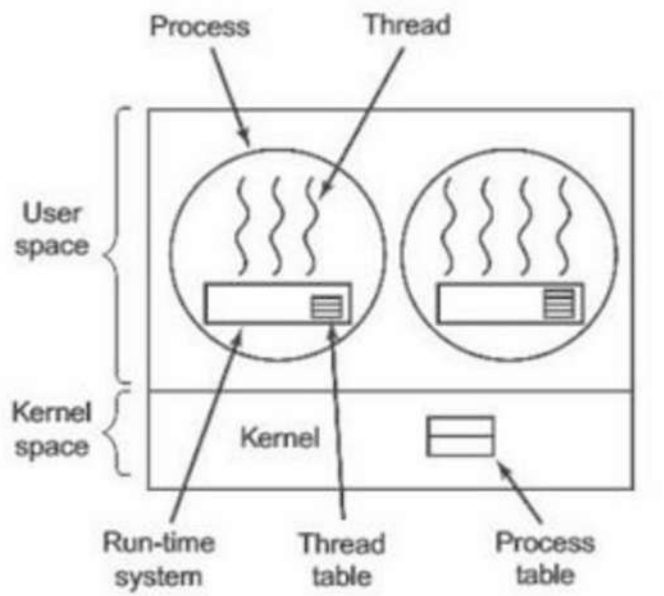
- PROs
  - Really fast and lightweight
  - Scheduling policies are more flexible
  - Can be implemented in OSs that do not support threading
  - No system calls involved, just user-space function calls
  - No actual context switch

User Threads: PROs



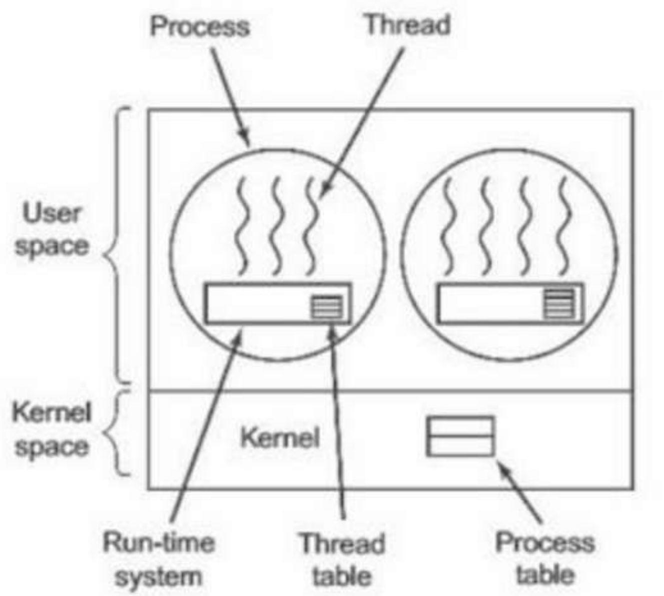
- PROs
  - Really fast and lightweight
  - Scheduling policies are more flexible
  - Can be implemented in OSs that do not support threading
  - No system calls involved, just user-space function calls
  - No actual context switch

User Threads: CONs



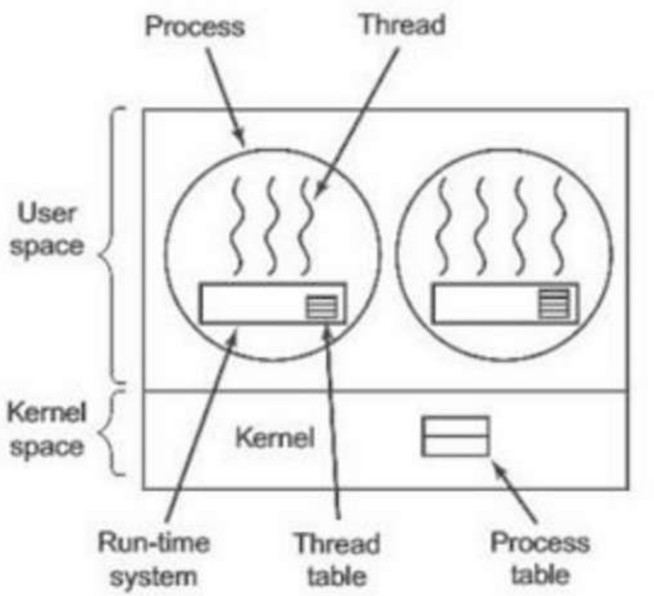
- CONs
  - No true concurrency of multi-threaded processes
  - Poor scheduling decisions
  - Lack of coordination between kernel and threads
    - A process with 100 threads competes for a time slice with a process with just 1 thread
  - Requires non-blocking system calls, otherwise all threads within a process have to wait

User Threads: CONs



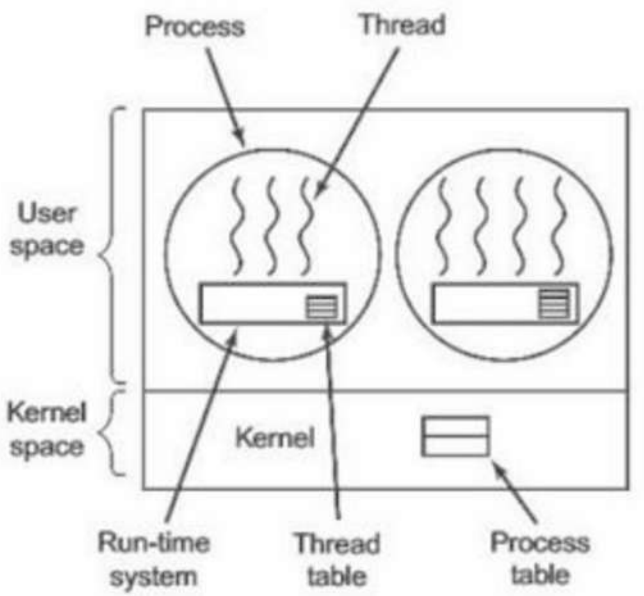
- CONs
  - No true concurrency of multi-threaded processes
  - Poor scheduling decisions
  - Lack of coordination between kernel and threads
    - A process with 100 threads competes for a time slice with a process with just 1 thread
  - Requires non-blocking system calls, otherwise all threads within a process have to wait

User Threads: CONs



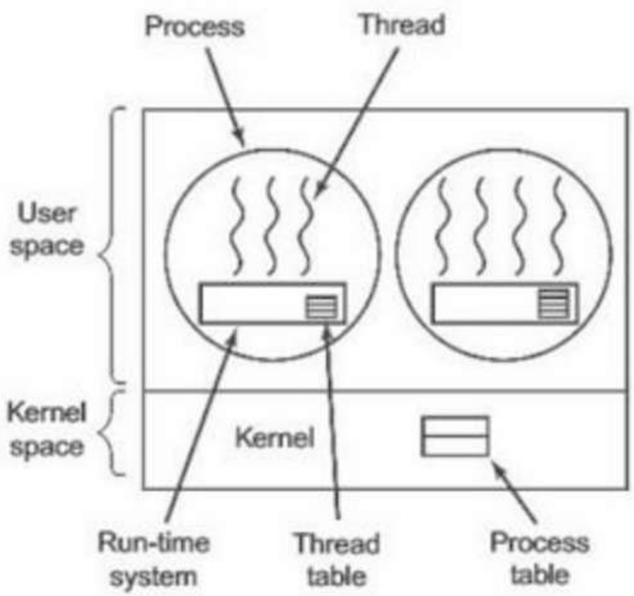
- CONs
  - No true concurrency of multi-threaded processes
  - Poor scheduling decisions
  - Lack of coordination between kernel and threads
    - A process with 100 threads competes for a time slice with a process with just 1 thread
  - Requires non-blocking system calls, otherwise all threads within a process have to wait

User Threads: CONs



- CONs
  - No true concurrency of multi-threaded processes
  - Poor scheduling decisions
  - Lack of coordination between kernel and threads
    - A process with 100 threads competes for a time slice with a process with just 1 thread
  - Requires non-blocking system calls, otherwise all threads within a process have to wait

User Threads: CONs



- CONs
  - No true concurrency of multi-threaded processes
  - Poor scheduling decisions
  - Lack of coordination between kernel and threads
    - A process with 100 threads competes for a time slice with a process with just 1 thread
  - Requires non-blocking system calls, otherwise all threads within a process have to wait

Multi-threading Models

- In a specific implementation, user threads must be mapped to kernel threads in one of the following ways:
  - Many-to-One (N:1)
  - One-to-One (1:1)
  - Many-to-Many (M:N)
  - Two-level

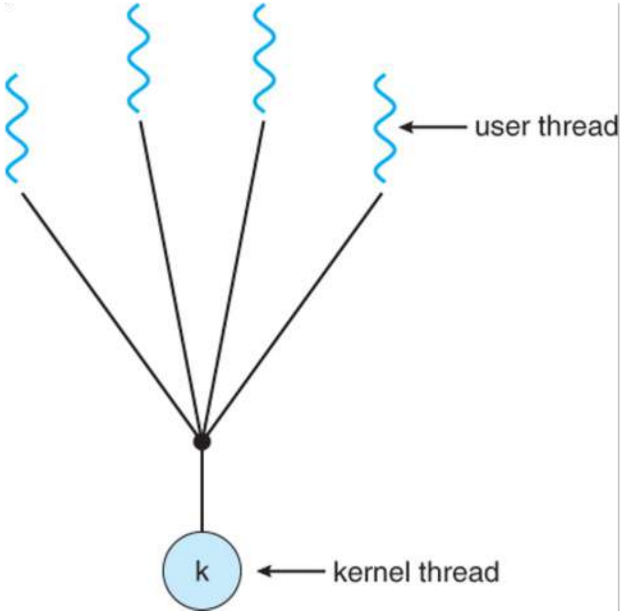
Multi-threading Models

- In a specific implementation, user threads must be mapped to kernel threads in one of the following ways:
  - Many-to-One (N:1)
  - One-to-One (1:1)
  - Many-to-Many (M:N)
  - Two-level

Remember:

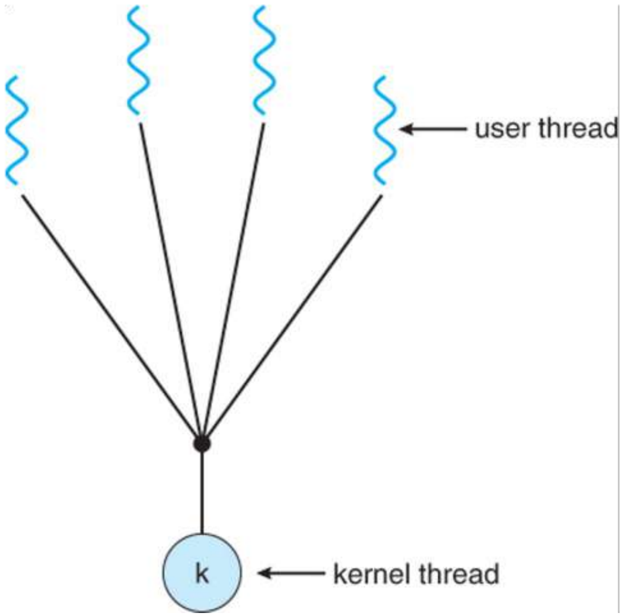
A kernel thread is the unit of execution that is scheduled by the OS to run on the CPU (similar to single-threaded process)

Many-to-One Model (N:1)



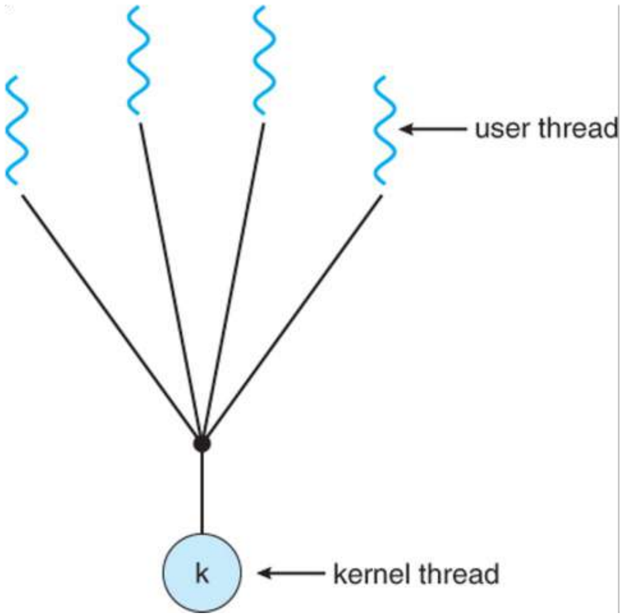
- Many user threads are all mapped onto a single kernel thread
- The process can only run one user thread at a time because there is only one kernel thread associated with it
- As single kernel thread can operate on a single CPU, multi-user-thread processes cannot be split across multiple CPUs
- If a blocking system call is made, the entire process blocks, even if other user threads would be able to continue

Many-to-One Model (N:1)



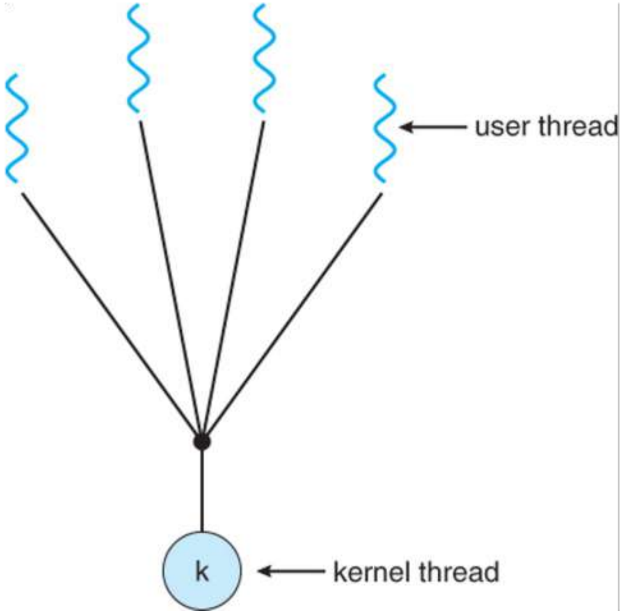
- Many user threads are all mapped onto a single kernel thread
- The process can only run one user thread at a time because there is only one kernel thread associated with it
- As single kernel thread can operate on a single CPU, multi-user-thread processes cannot be split across multiple CPUs
- If a blocking system call is made, the entire process blocks, even if other user threads would be able to continue

Many-to-One Model (N:1)



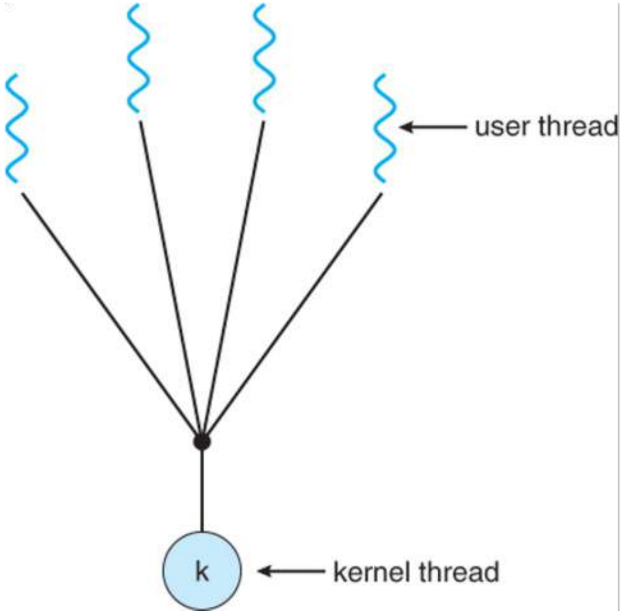
- Many user threads are all mapped onto a single kernel thread
- The process can only run one user thread at a time because there is only one kernel thread associated with it
- As single kernel thread can operate on a single CPU, multi-user-thread processes cannot be split across multiple CPUs
- If a blocking system call is made, the entire process blocks, even if other user threads would be able to continue

Many-to-One Model (N:1)



- Many user threads are all mapped onto a single kernel thread
- The process can only run one user thread at a time because there is only one kernel thread associated with it
- As single kernel thread can operate on a single CPU, multi-user-thread processes cannot be split across multiple CPUs
- If a blocking system call is made, the entire process blocks, even if other user threads would be able to continue

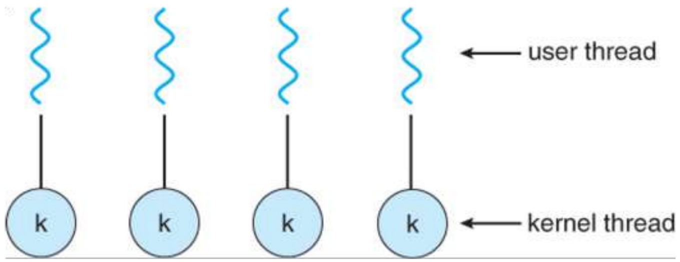
Many-to-One Model (N:1)



- Many user threads are all mapped onto a single kernel thread
- The process can only run one user thread at a time because there is only one kernel thread associated with it
- As single kernel thread can operate on a single CPU, multi-user-thread processes cannot be split across multiple CPUs
- If a blocking system call is made, the entire process blocks, even if other user threads would be able to continue

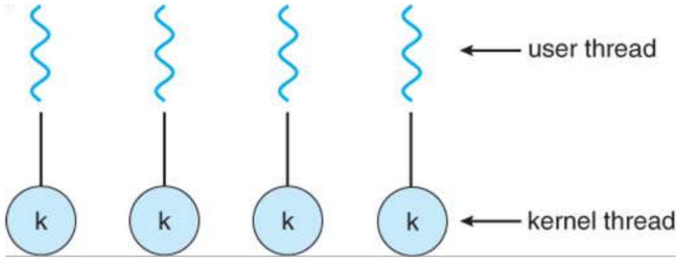
pure user-level

One-to-One Model (1:1)



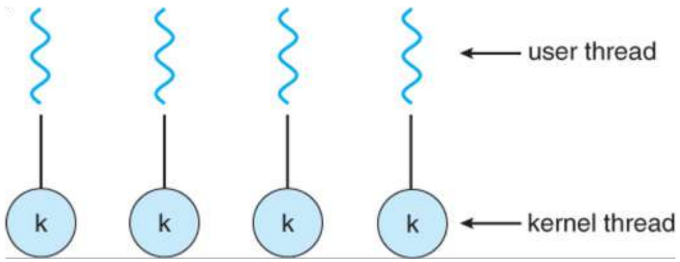
- A separate kernel thread to handle each user thread
- Overcomes the limitations of blocking system calls and splitting of processes across multiple CPUs
- The overhead of managing the one-to-one model is more significant and may slow down the system
- Most implementations of this model place a limit on how many threads can be created

One-to-One Model (1:1)



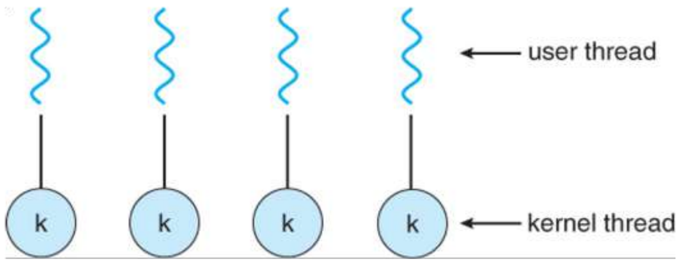
- A separate kernel thread to handle each user thread
- Overcomes the limitations of blocking system calls and splitting of processes across multiple CPUs
- The overhead of managing the one-to-one model is more significant and may slow down the system
- Most implementations of this model place a limit on how many threads can be created

One-to-One Model (1:1)



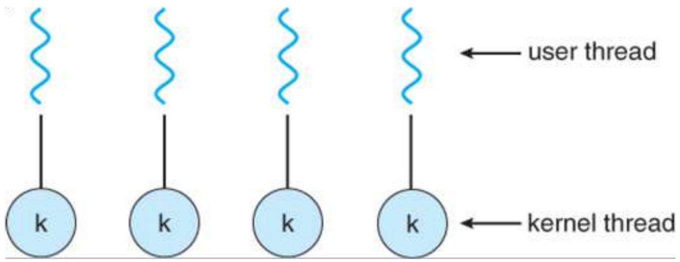
- A separate kernel thread to handle each user thread
- Overcomes the limitations of blocking system calls and splitting of processes across multiple CPUs
- The overhead of managing the one-to-one model is more significant and may slow down the system
- Most implementations of this model place a limit on how many threads can be created

One-to-One Model (1:1)



- A separate kernel thread to handle each user thread
- Overcomes the limitations of blocking system calls and splitting of processes across multiple CPUs
- The overhead of managing the one-to-one model is more significant and may slow down the system
- Most implementations of this model place a limit on how many threads can be created

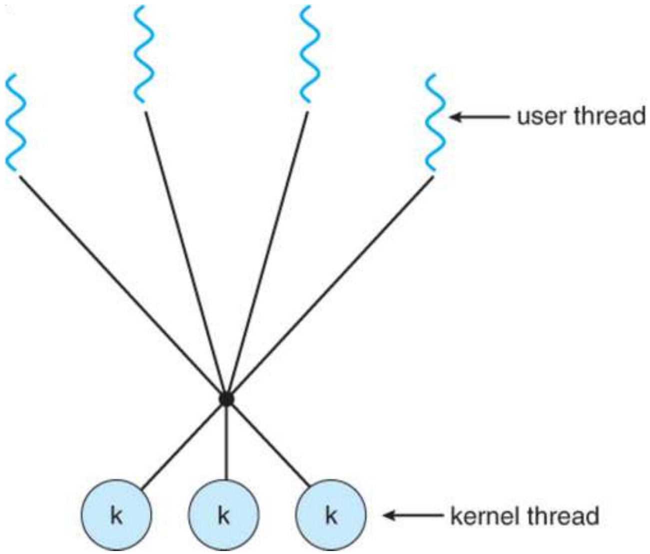
One-to-One Model (1:1)



- A separate kernel thread to handle each user thread
- Overcomes the limitations of blocking system calls and splitting of processes across multiple CPUs
- The overhead of managing the one-to-one model is more significant and may slow down the system
- Most implementations of this model place a limit on how many threads can be created

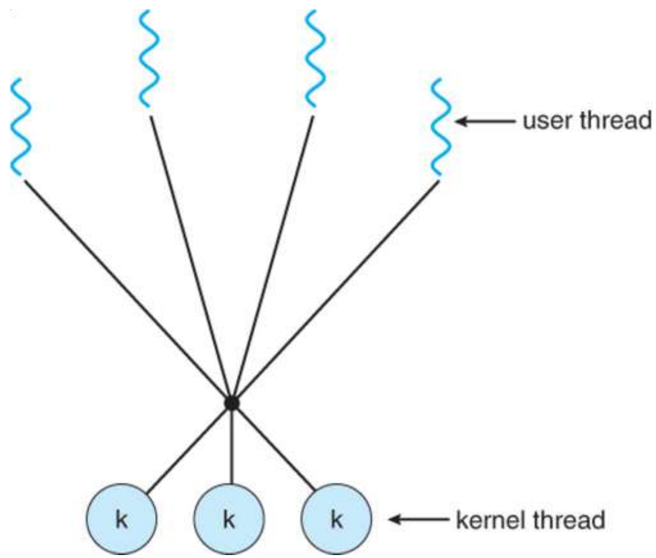
pure kernel-level

Many-to-Many Model (M:N)



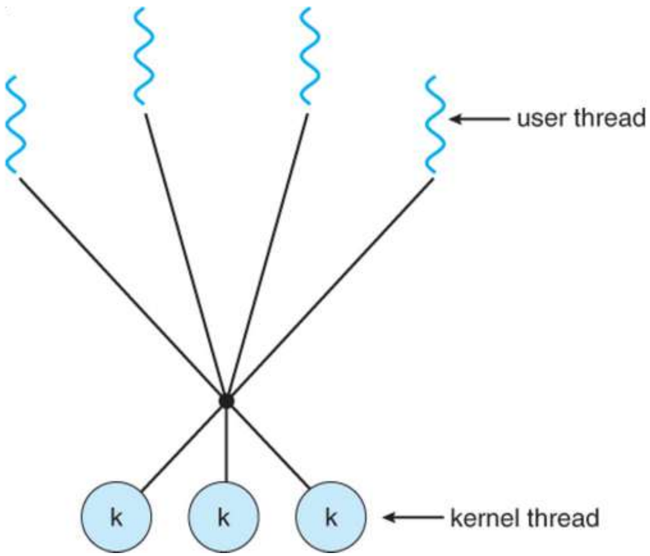
- Multiplexes any number of user threads onto an equal or smaller number of kernel threads
- Users have no restrictions on the number of threads created
- Processes can be split across multiple processors
- Blocking kernel system calls do not block the entire process

Many-to-Many Model (M:N)



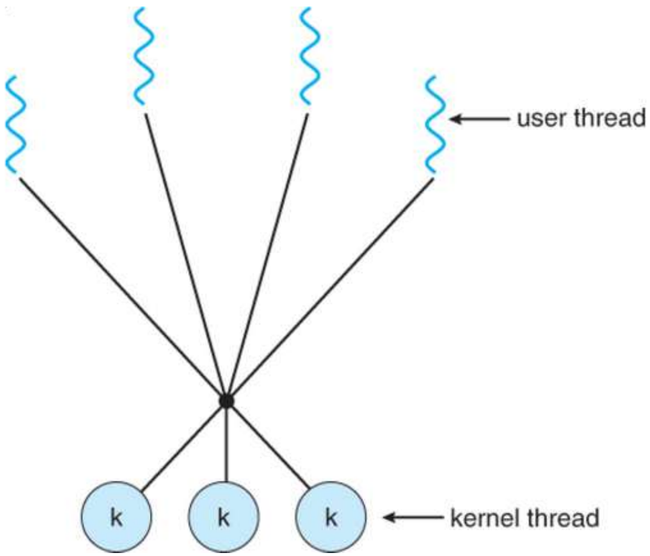
- Multiplexes any number of user threads onto an equal or smaller number of kernel threads
- Users have no restrictions on the number of threads created
- Processes can be split across multiple processors
- Blocking kernel system calls do not block the entire process

Many-to-Many Model (M:N)



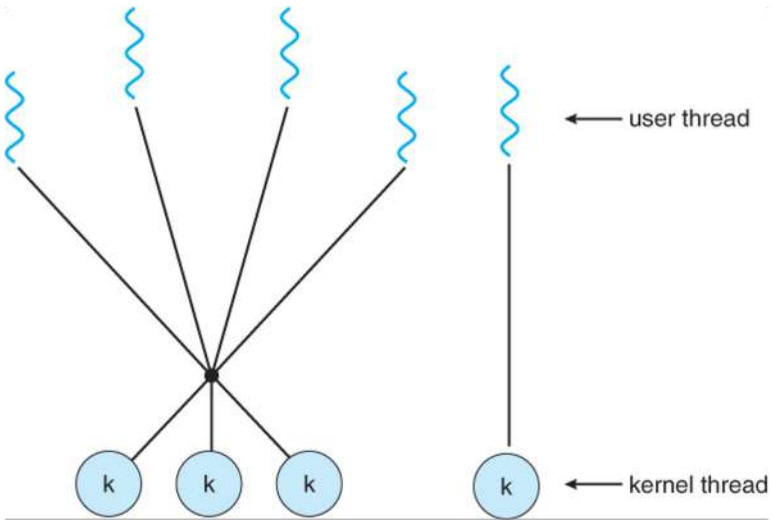
- Multiplexes any number of user threads onto an equal or smaller number of kernel threads
- Users have no restrictions on the number of threads created
- Processes can be split across multiple processors
- Blocking kernel system calls do not block the entire process

Many-to-Many Model (M:N)



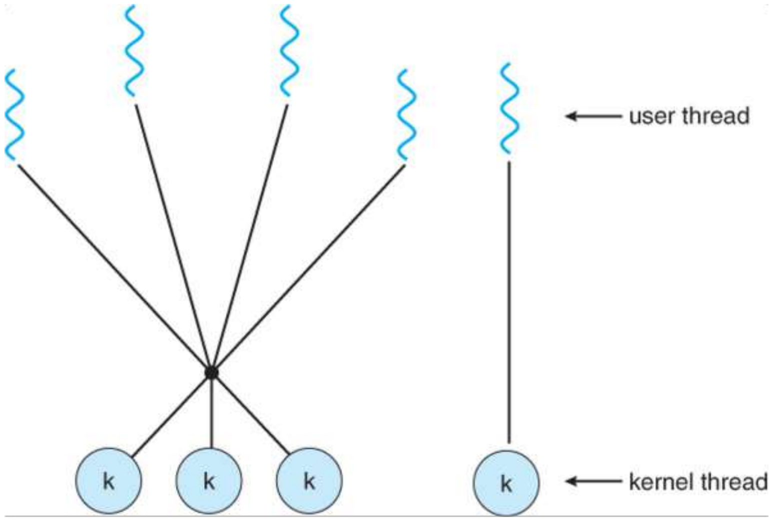
- Multiplexes any number of user threads onto an equal or smaller number of kernel threads
- Users have no restrictions on the number of threads created
- Processes can be split across multiple processors
- Blocking kernel system calls do not block the entire process

Two-Level Model



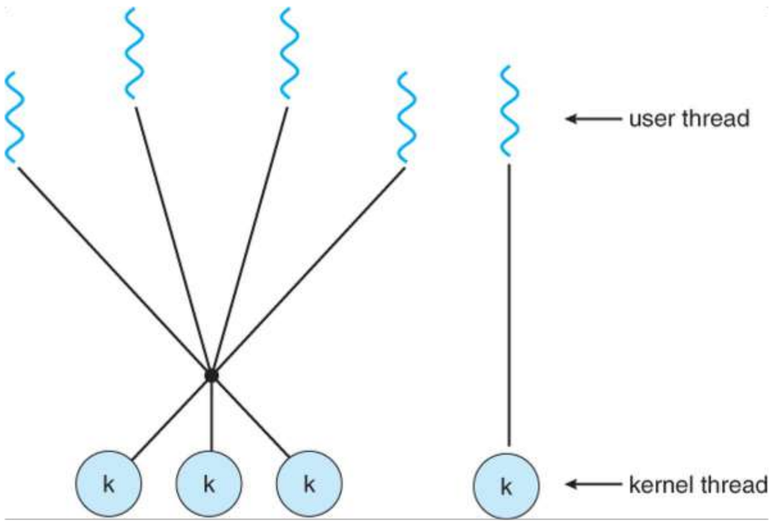
- A variant of the many-to-many model
- Mixes many-to-many with one-to-one
- Increases the flexibility of scheduling policies

Two-Level Model



- A variant of the many-to-many model
- Mixes many-to-many with one-to-one
- Increases the flexibility of scheduling policies

Two-Level Model



- A variant of the many-to-many model
- Mixes many-to-many with one-to-one
- Increases the flexibility of scheduling policies

Thread Libraries

- Provides programmers with an API for creating and managing threads

Thread Libraries

- Provides programmers with an API for creating and managing threads
- 2 primary ways of implementing it:
  - user space → API functions implemented entirely in user space (function calls)

Thread Libraries

- Provides programmers with an API for creating and managing threads
- 2 primary ways of implementing it:
  - user space → API functions implemented entirely in user space (function calls)
  - kernel space → implemented in kernel space within a kernel that supports threads (system calls)

Thread Libraries: Examples

- There are 3 main thread libraries in use today:
  - POSIX Pthreads → may be provided as either a user or kernel library, as an extension to the POSIX standard

Thread Libraries: Examples

- There are 3 main thread libraries in use today:
  - POSIX Pthreads → may be provided as either a user or kernel library, as an extension to the POSIX standard
  - Win32 threads → provided as a kernel-level library on Windows systems

Thread Libraries: Examples

- There are 3 main thread libraries in use today:
  - POSIX Pthreads → may be provided as either a user or kernel library, as an extension to the POSIX standard
  - Win32 threads → provided as a kernel-level library on Windows systems
  - Java threads → the implementation of threads is based upon whatever OS and hardware the JVM is running on, e.g., either Pthreads or Win32 threads

Thread Pools: Idea

- A specific number of threads are created when the process starts

Thread Pools: Idea

- A specific number of threads are created when the process starts
- Those threads are placed in the "pool" waiting for some work to do

Thread Pools: Idea

- A specific number of threads are created when the process starts
- Those threads are placed in the "pool" waiting for some work to do
- When the main thread must serve a request it awakens a thread from the pool

Thread Pools: Idea

- A specific number of threads are created when the process starts
- Those threads are placed in the "pool" waiting for some work to do
- When the main thread must serve a request it awakens a thread from the pool
- The worker thread processes the request and goes back to the pool once terminated

Thread Pools: Idea

- A specific number of threads are created when the process starts
- Those threads are placed in the "pool" waiting for some work to do
- When the main thread must serve a request it awakens a thread from the pool
- The worker thread processes the request and goes back to the pool once terminated
- If no threads are available in the pool the server waits for one

To Wrap Up

Threading Model	Where Threads Run	Scheduling Responsibility	Preemption Possible?	Key Notes	Real-world Examples
N:1 (User-level threads)	All threads run in user space on a single kernel thread	Managed entirely by the user-level library	No preemption between user threads within the same kernel thread – if one thread blocks (e.g., I/O), all threads block	Fast context switches, low kernel involvement, limited concurrency on multiprocessors	Early versions of GNU Portable Threads, green threads in older Java VMs

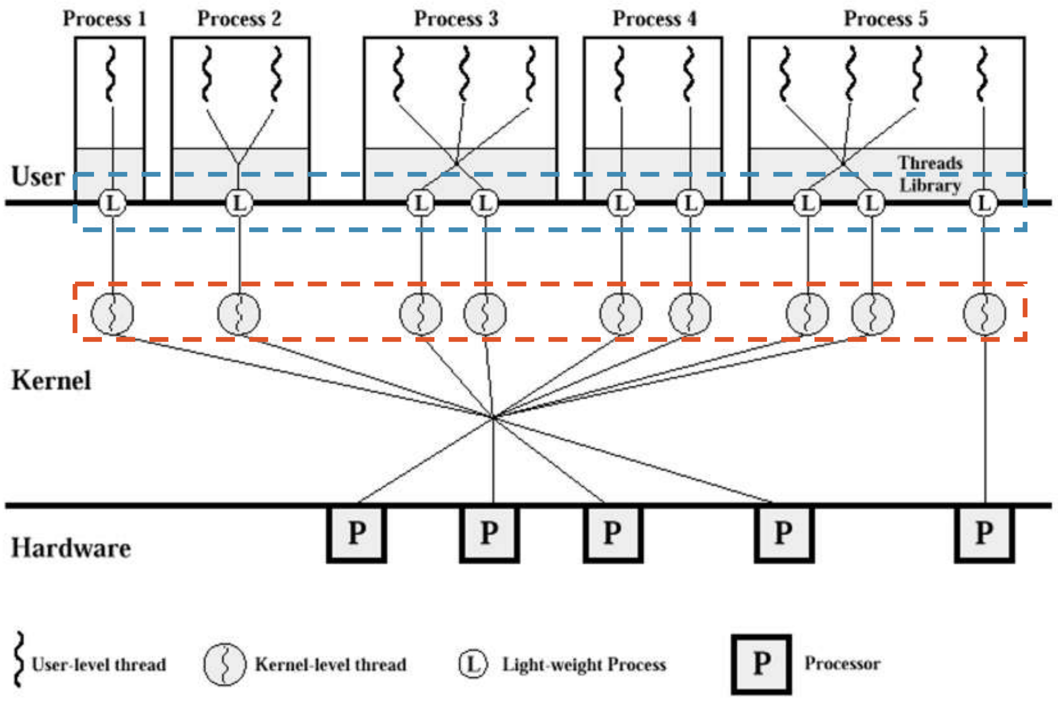
To Wrap Up

Threading Model	Where Threads Run	Scheduling Responsibility	Preemption Possible?	Key Notes	Real-world Examples
N:1 (User-level threads)	All threads run in user space on a single kernel thread	Managed entirely by the user-level library	No preemption between user threads within the same kernel thread – if one thread blocks (e.g., I/O), all threads block	Fast context switches, low kernel involvement, limited concurrency on multiprocessors	Early versions of GNU Portable Threads, green threads in older Java VMs
1:1 (Kernel-level threads)	Each user thread maps to one kernel thread	Managed by the OS kernel scheduler	Fully preemptive – kernel can interrupt and schedule any thread	Higher overhead (kernel context switch), true concurrency on multiprocessors	Windows threads, Linux Pthreads (NPTL), Solaris threads

To Wrap Up

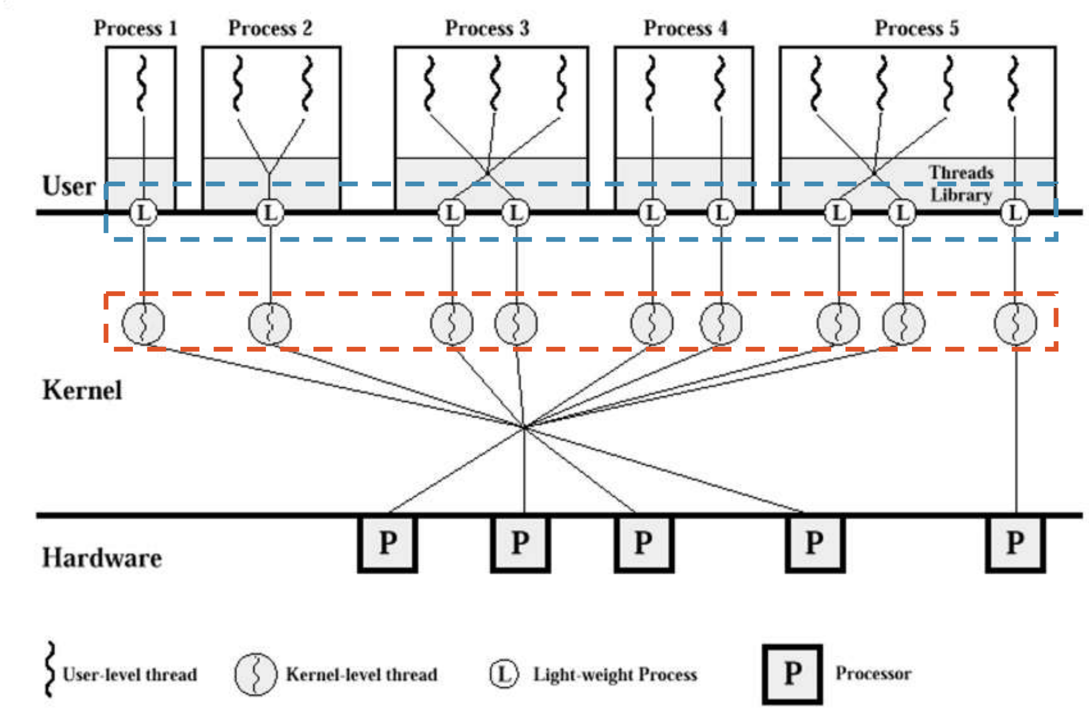
Threading Model	Where Threads Run	Scheduling Responsibility	Preemption Possible?	Key Notes	Real-world Examples
N:1 (User-level threads)	All threads run in user space on a single kernel thread	Managed entirely by the user-level library	No preemption between user threads within the same kernel thread – if one thread blocks (e.g., I/O), all threads block	Fast context switches, low kernel involvement, limited concurrency on multiprocessors	Early versions of GNU Portable Threads, green threads in older Java VMs
1:1 (Kernel-level threads)	Each user thread maps to one kernel thread	Managed by the OS kernel scheduler	Fully preemptive – kernel can interrupt and schedule any thread	Higher overhead (kernel context switch), true concurrency on multiprocessors	Windows threads, Linux Pthreads (NPTL), Solaris threads
M:N (Hybrid)	Multiple user threads mapped onto multiple kernel threads	User-level library schedules user threads onto kernel threads; kernel schedules kernel threads on CPU	Preemption possible for kernel threads; user-level library can implement additional scheduling policies	Combines flexibility of user-level scheduling with kernel-level concurrency; complexity in coordination	Solaris Scheduler Activations, older versions of GNU Portable Threads

Thread Scheduling (M:N)



M:N thread implementations provide a virtual processor (L) as an interface between user and kernel threads

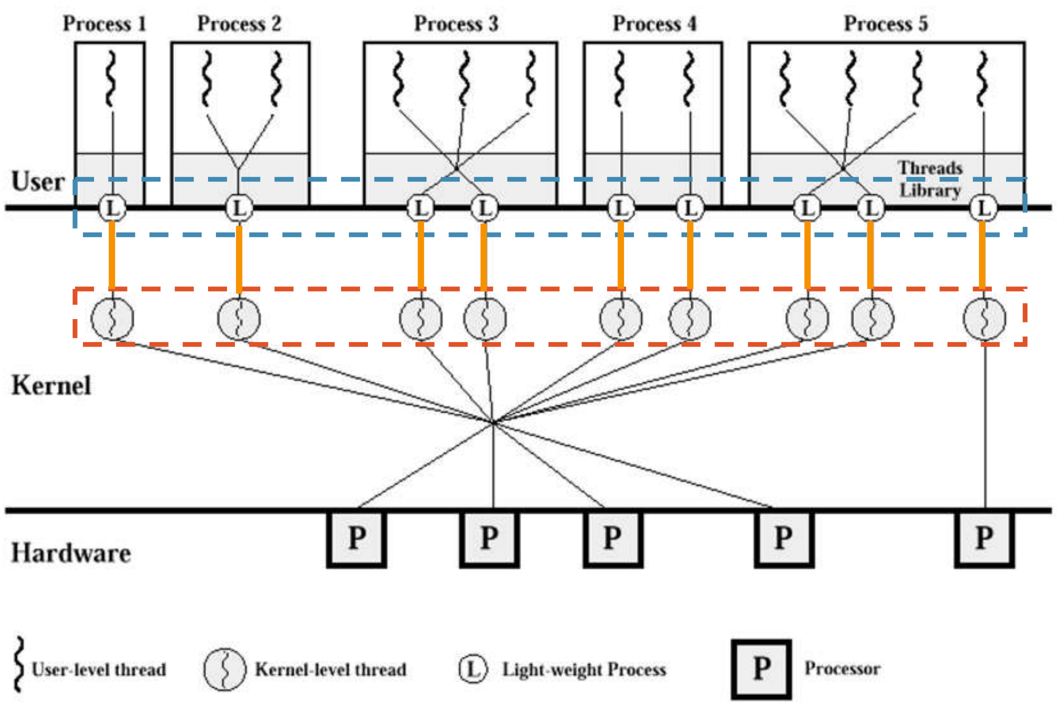
Thread Scheduling (M:N)



M:N thread implementations provide a virtual processor (L) as an interface between user and kernel threads

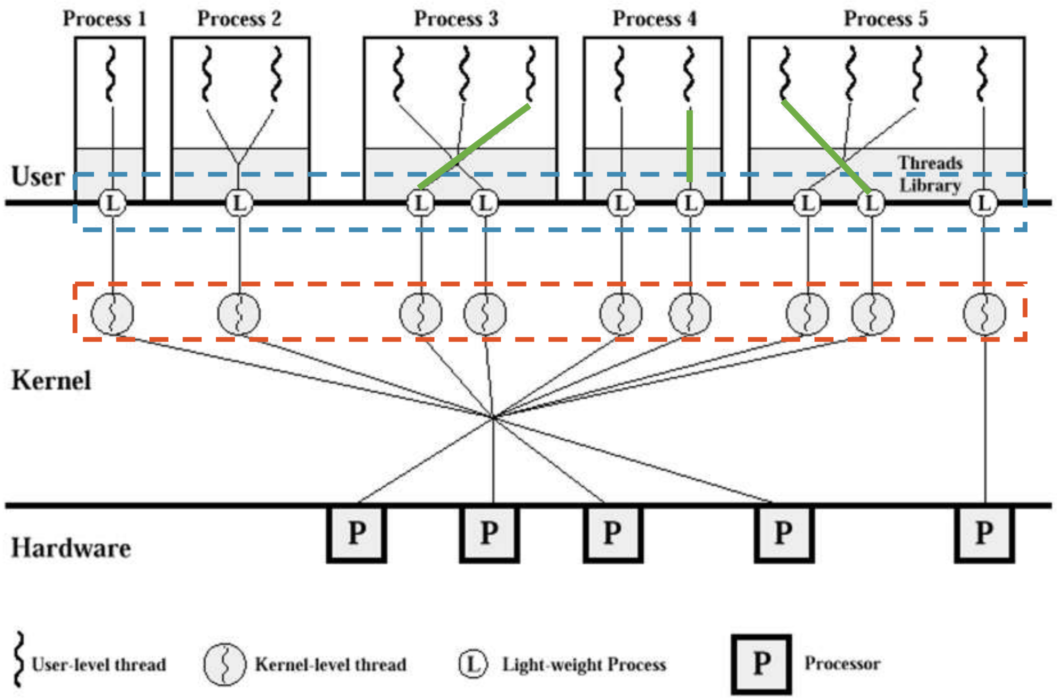
Light-Weight Process (LWP)

Thread Scheduling (M:N)



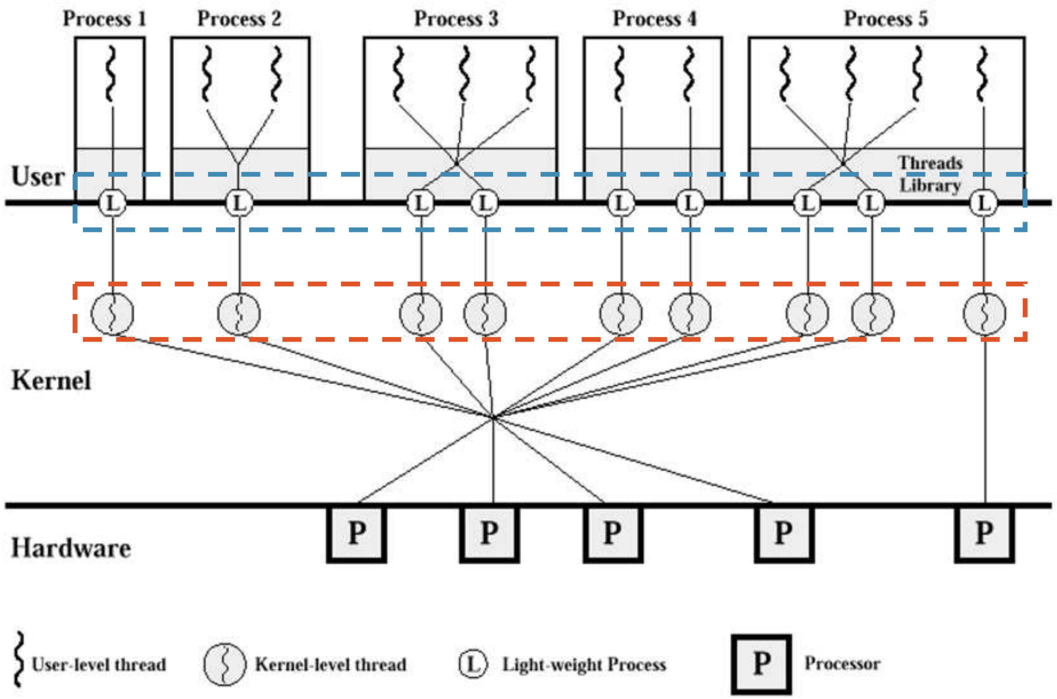
1:1 correspondence between LWPs and kernel threads

Thread Scheduling (M:N)



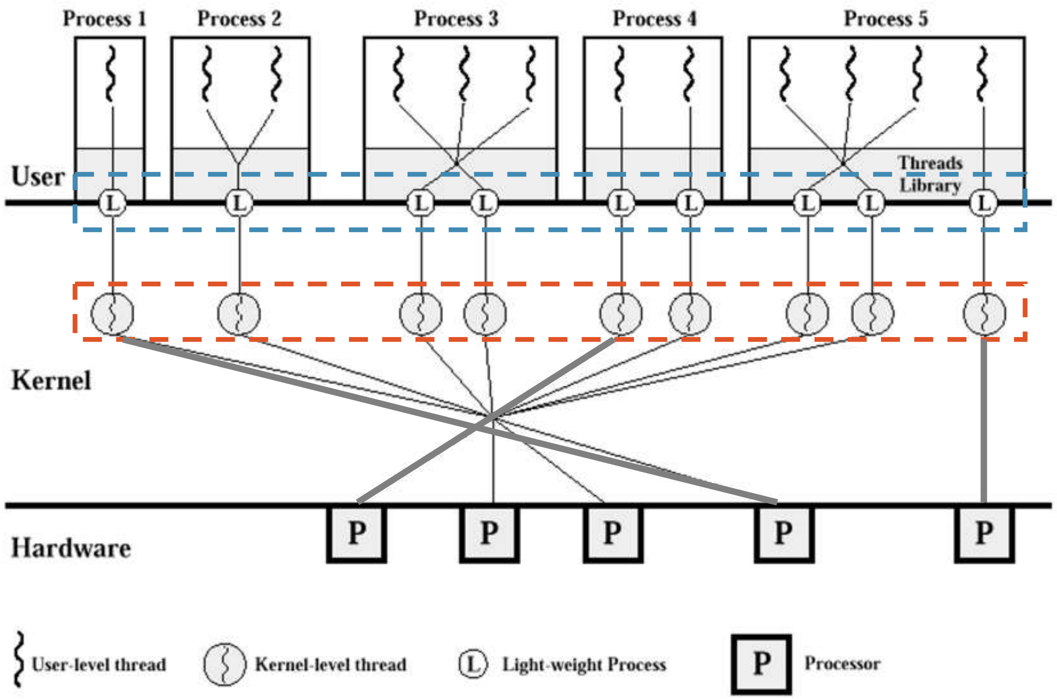
The application (user-level thread library) maps user threads onto available LWPs

Thread Scheduling (M:N)



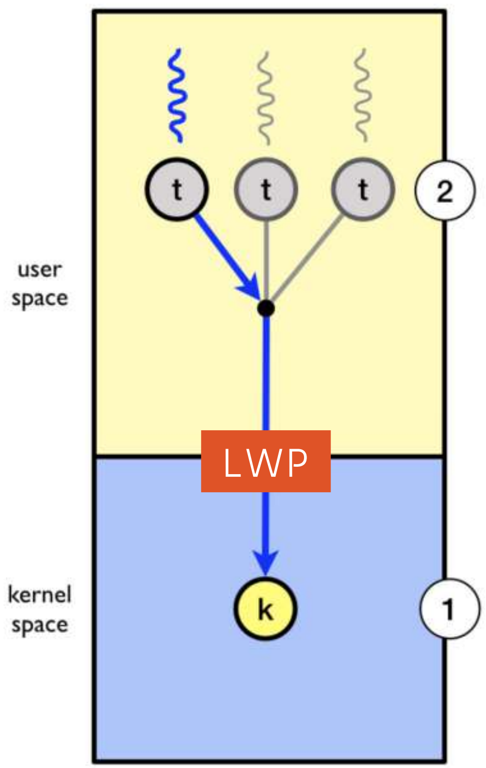
The number of kernel threads available in the system may change dynamically

Thread Scheduling (M:N)



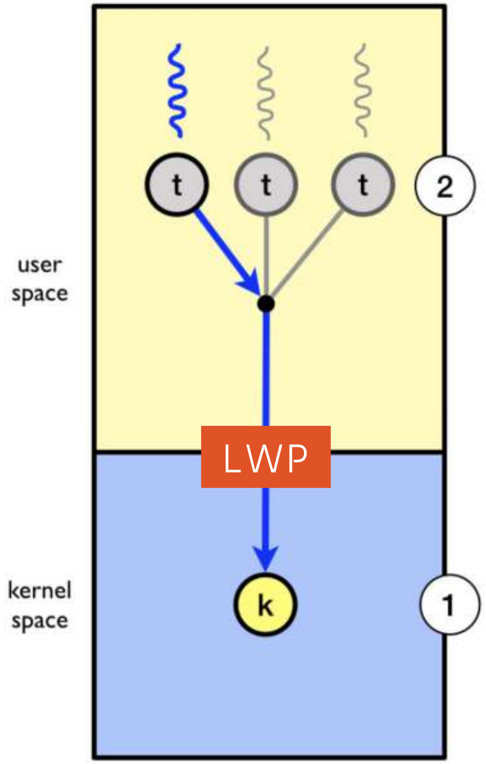
Kernel threads are scheduled onto the real processor(s) by the OS

Scheduler Activations: Example



The kernel has allocated one kernel thread (1) to a process (i.e., an LWP) with three user-level threads (2)

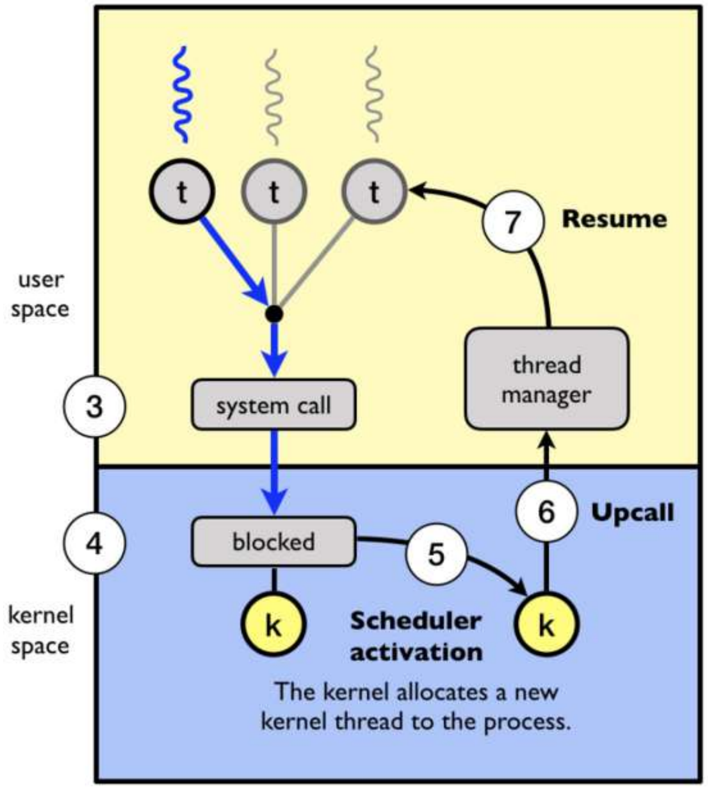
Scheduler Activations: Example



The kernel has allocated one kernel thread (1) to a process (i.e., an LWP) with three user-level threads (2)

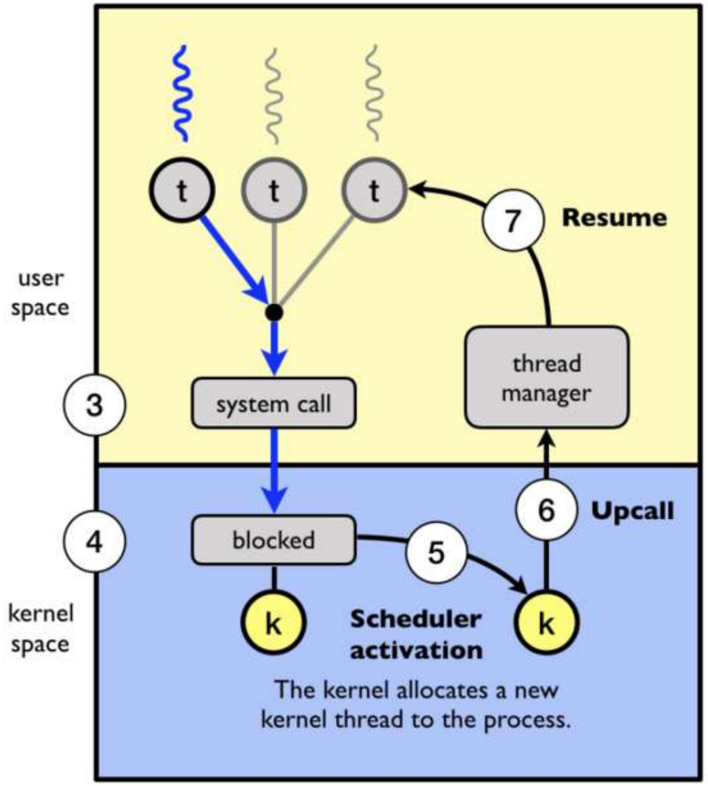
The three user level threads take turn executing on the single kernel-level thread

Scheduler Activations: Example



The executing thread makes a blocking system call (3)

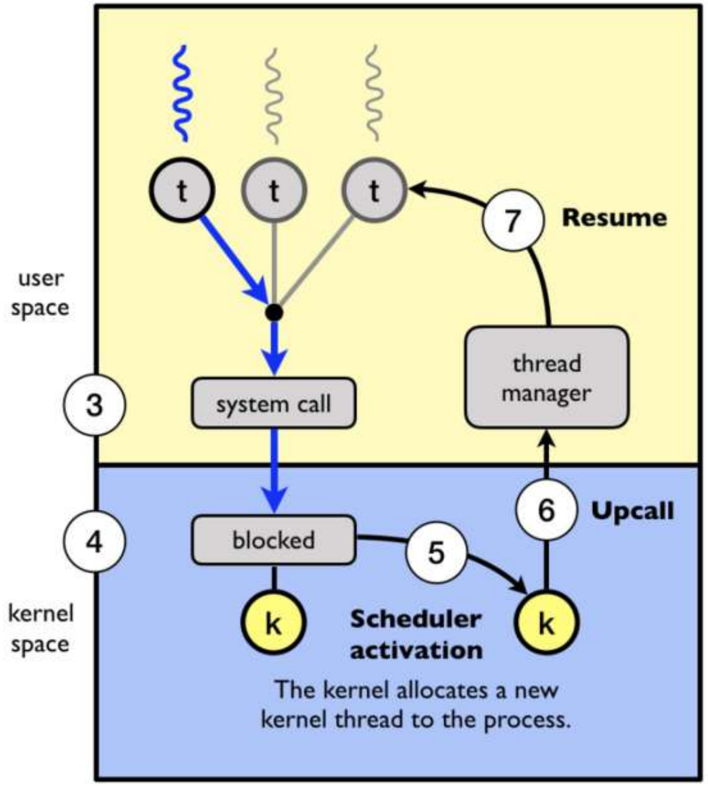
Scheduler Activations: Example



The executing thread makes a blocking system call (3)

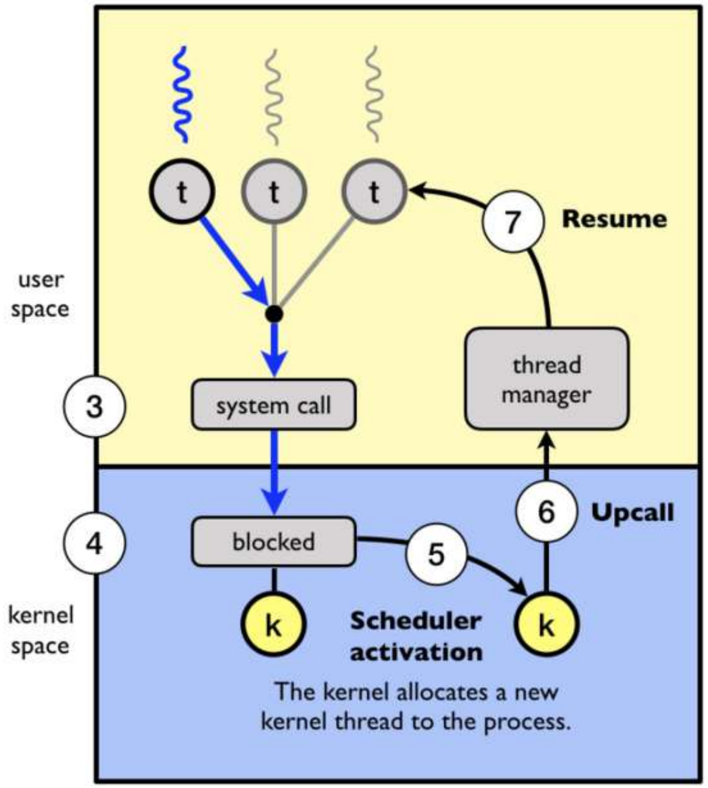
The kernel blocks the calling user-level thread and the kernel-level thread (LWP) used to execute the user-level thread (4)

Scheduler Activations: Example



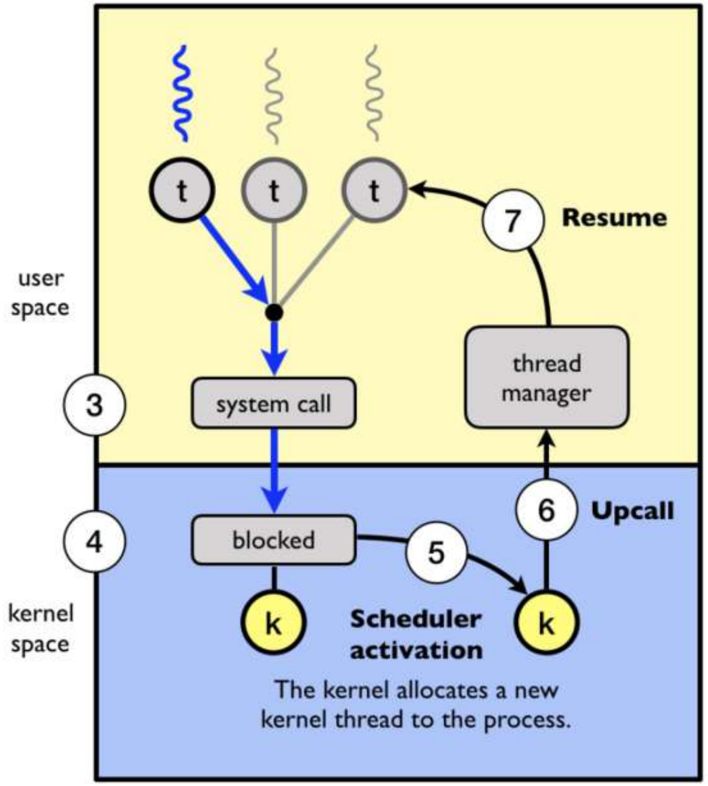
Scheduler activation: the kernel decides to allocate a new kernel-level thread to the process (5)

Scheduler Activations: Example



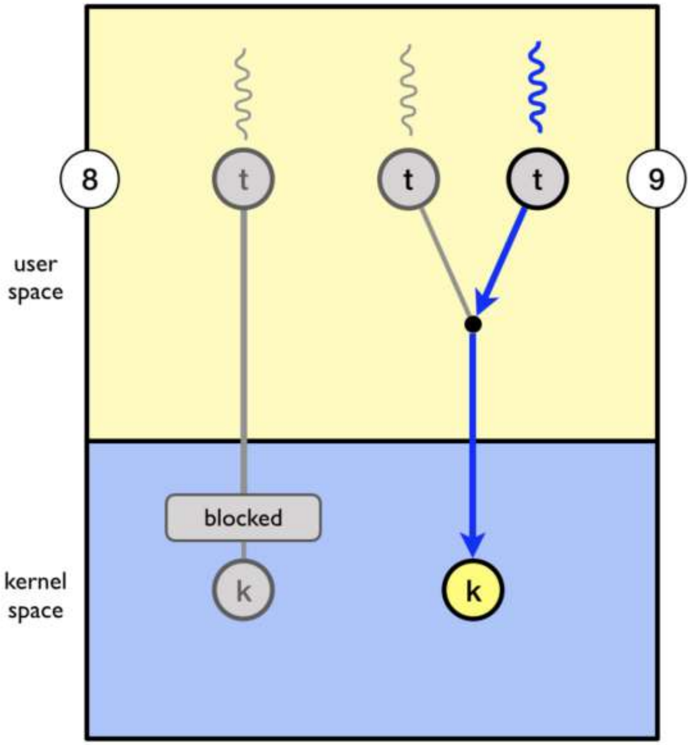
Upcall: The kernel notifies the user-level thread library which user-level thread that is now blocked and that a new kernel-level thread is available (6)

Scheduler Activations: Example



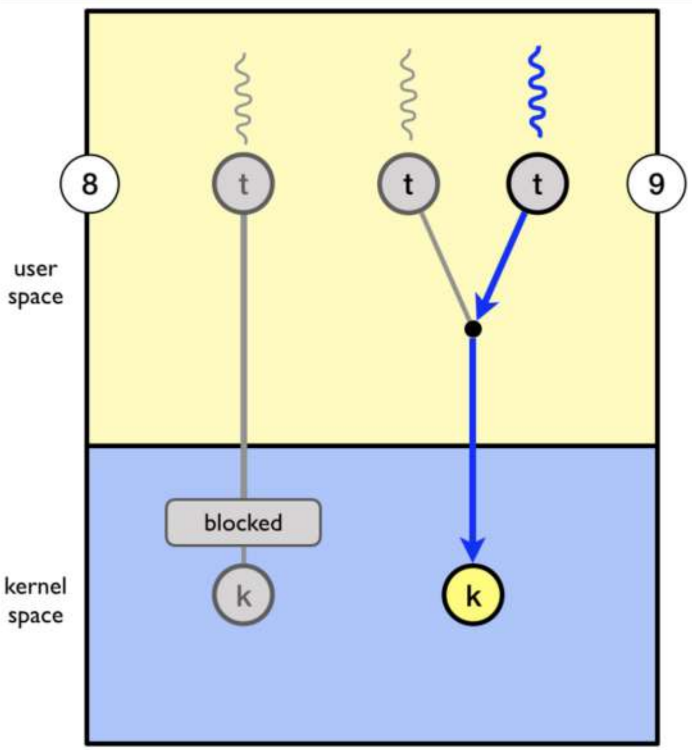
Upcall handler: The user-level thread library resumes one of the ready threads on to the new kernel thread (7)

Scheduler Activations: Example



While one user-level thread is blocked (8) the other threads can take turn executing on the new kernel thread (9)

Scheduler Activations: Example



When the first thread wakes up, the kernel will notify the user thread library via another upcall

User-Level Thread Scheduling

- Scheduling user-level threads on the available kernel-level threads (via LWPs)

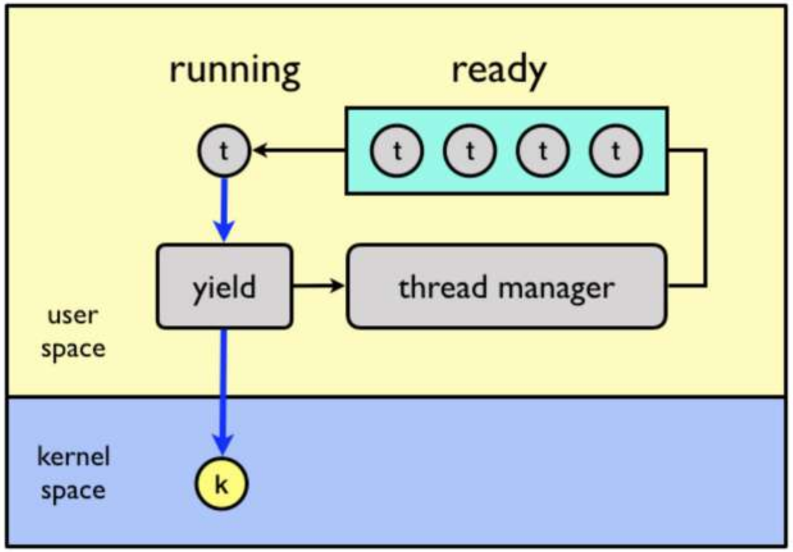
User-Level Thread Scheduling

- Scheduling user-level threads on the available kernel-level threads (via LWPs)
- Implemented within the user-level thread library in user space (no kernel privileges!)

User-Level Thread Scheduling

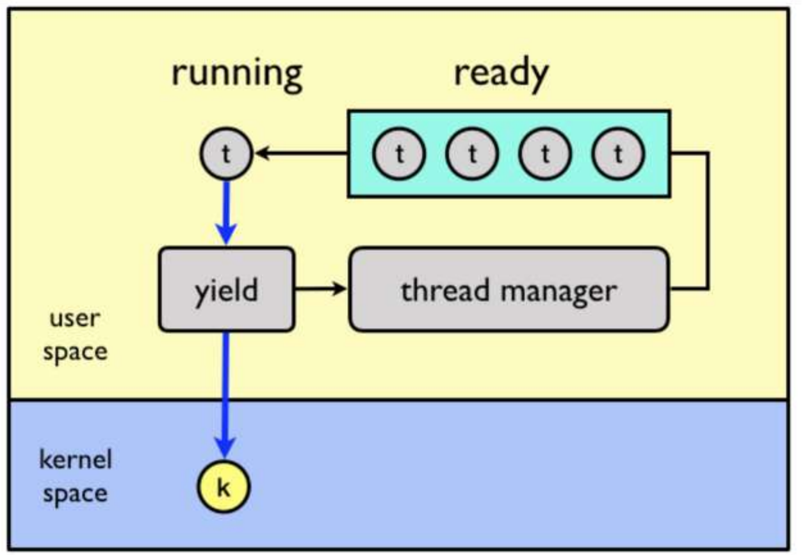
- Scheduling user-level threads on the available kernel-level threads (via LWPs)
- Implemented within the user-level thread library in user space (no kernel privileges!)
- Two main scheduling methods:
  - Cooperative
  - Preemptive

Cooperative Thread Scheduling



Similar to multiprogramming where a process executes on the CPU until making a I/O request

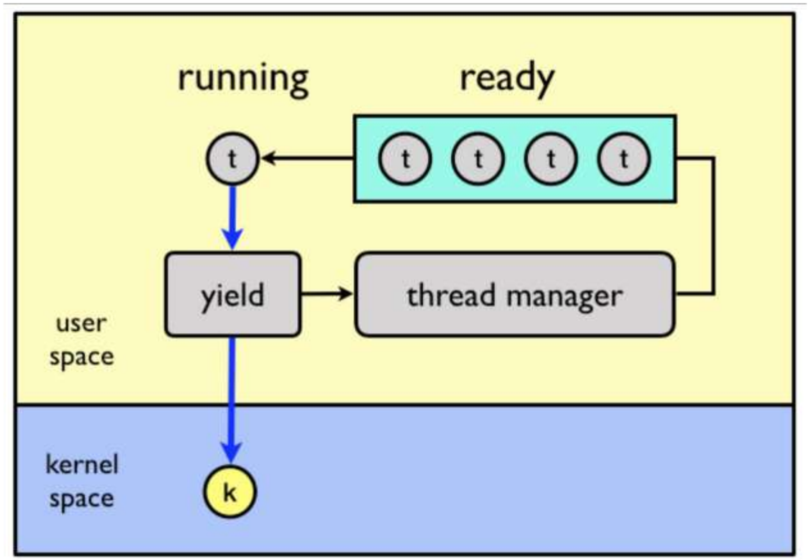
Cooperative Thread Scheduling



Similar to multiprogramming where a process executes on the CPU until making a I/O request

Cooperative user-level threads execute on the assigned kernel-level thread until they voluntarily give back the kernel thread to the library

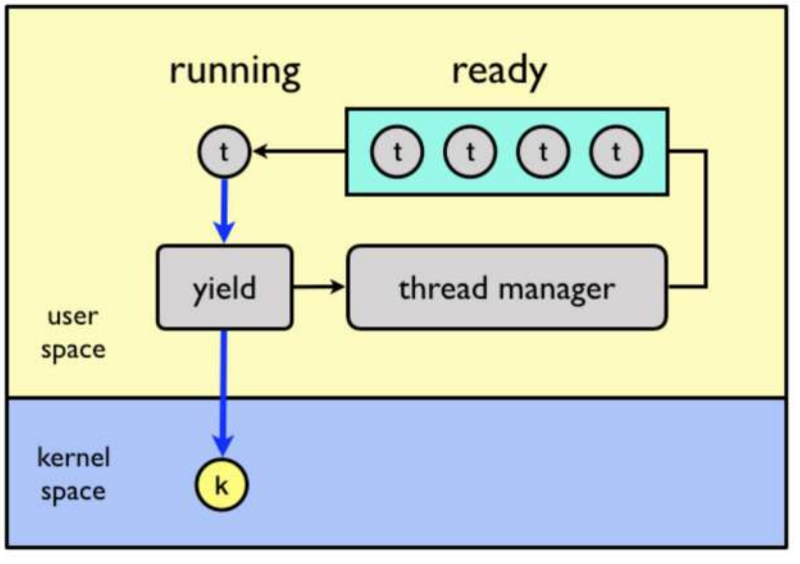
Cooperative Thread Scheduling



Threads yield to each other, either

- explicitly (e.g., by calling a yield() provided by the user-level thread library) or

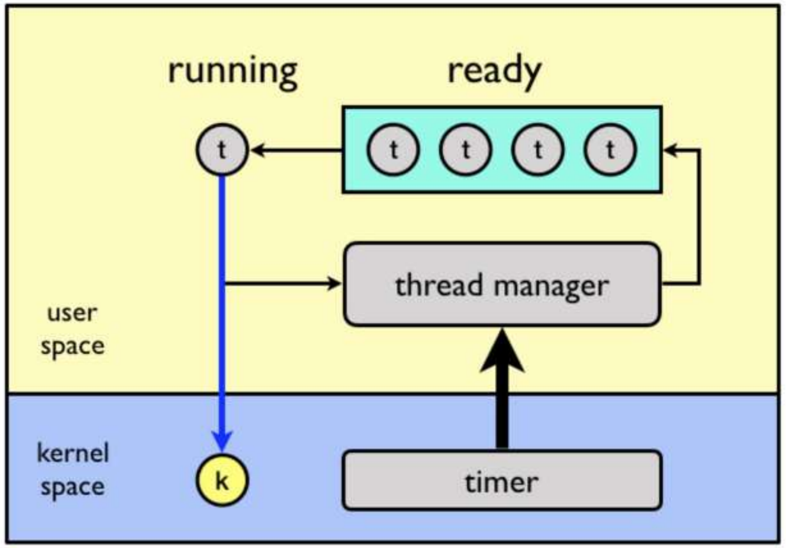
Cooperative Thread Scheduling



Threads yield to each other, either

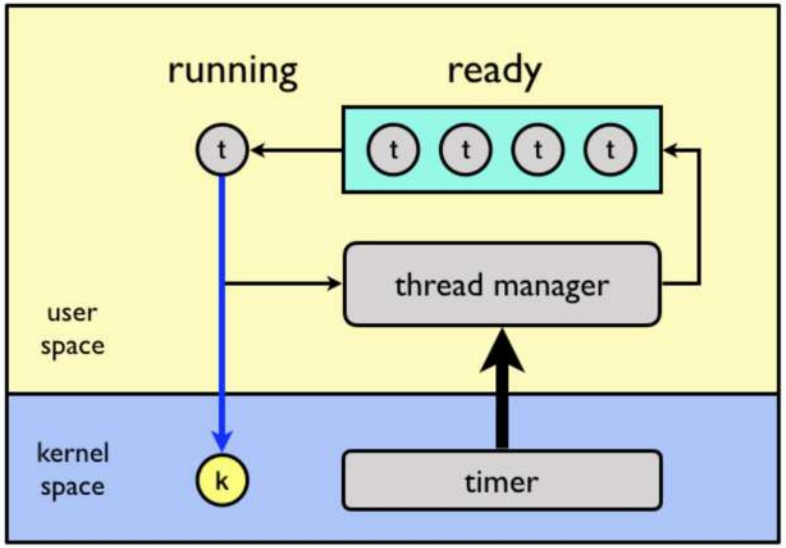
- explicitly (e.g., by calling a yield() provided by the user-level thread library) or
- implicitly (e.g., requesting a lock held by another thread)

Preemptive Thread Scheduling



Similar to multitasking (a.k.a. time sharing), where a timer is set to cause an interrupt at a regular time interval

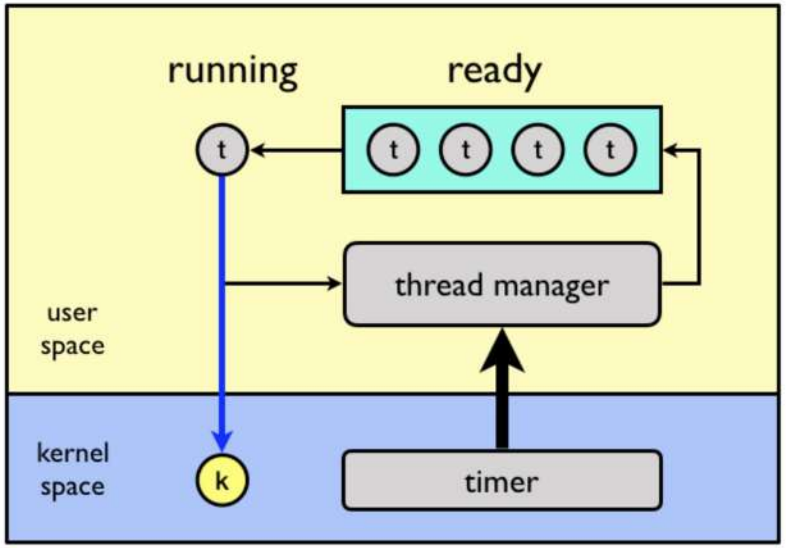
Preemptive Thread Scheduling



Similar to multitasking (a.k.a. time sharing), where a timer is set to cause an interrupt at a regular time interval

The running process is replaced if the job requests I/O or if the job is interrupted by the timer

Preemptive Thread Scheduling

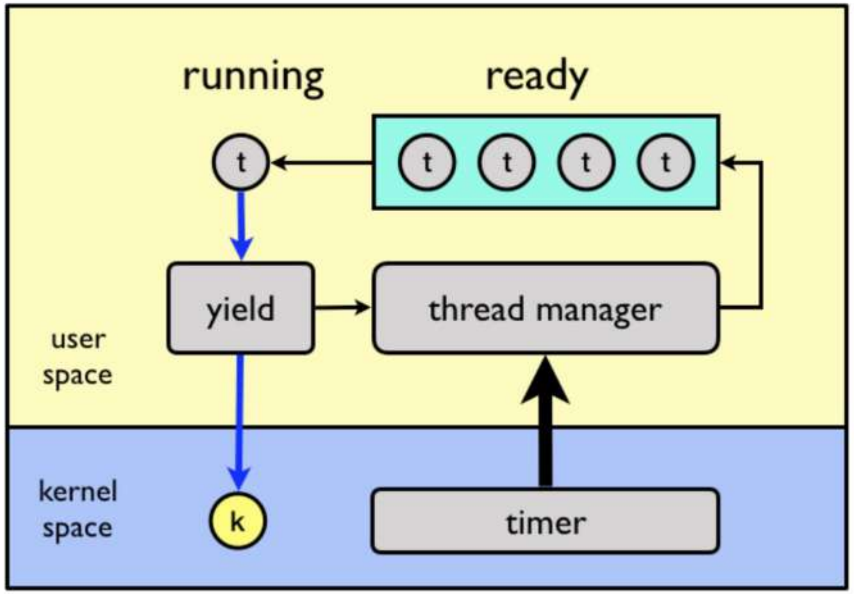


Similar to multitasking (a.k.a. time sharing), where a timer is set to cause an interrupt at a regular time interval

The running process is replaced if the job requests I/O or if the job is interrupted by the timer

The timer is used to cause execution flow to jump to a central dispatcher thread (in the user-level library), which chooses the next thread to run

Hybrid Thread Scheduling



Cooperative + Preemptive

Summary

- A thread is a single execution stream within a process

Summary

- A thread is a single execution stream within a process
- User- vs. Kernel-level threads

Summary

- A thread is a single execution stream within a process
- User- vs. Kernel-level threads
- Mapping user- to kernel-level threads
  - N:1/1:1/M:N

Summary

- A thread is a single execution stream within a process
- User- vs. Kernel-level threads
- Mapping user- to kernel-level threads
    - N:1/1:1/M:N
- Scheduling user-level threads vs. kernel-level threads