Sistemi Operativi I

Corso di Laurea in Informatica

2024-2025

Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

Recap from Previous Lecture

- 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

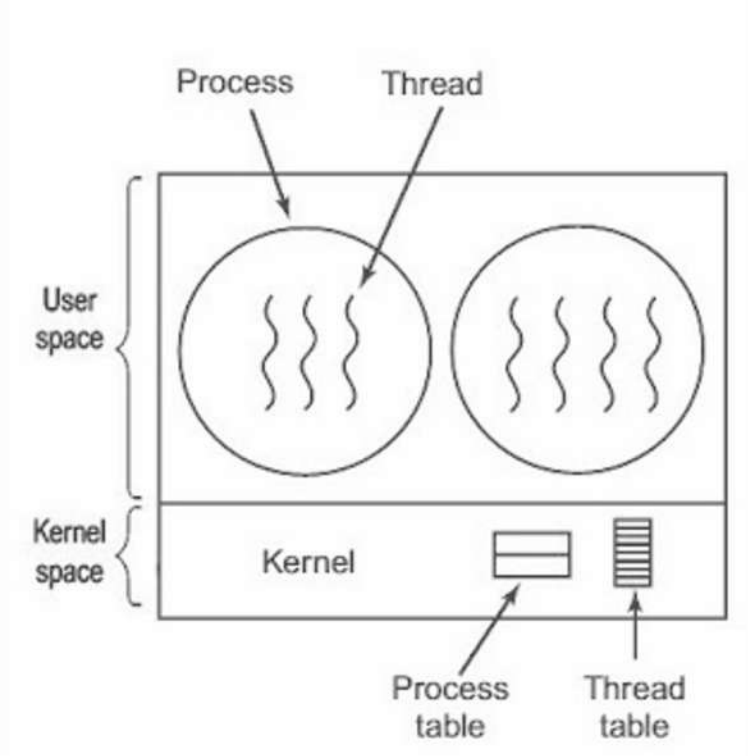
Kernel Threads

- 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

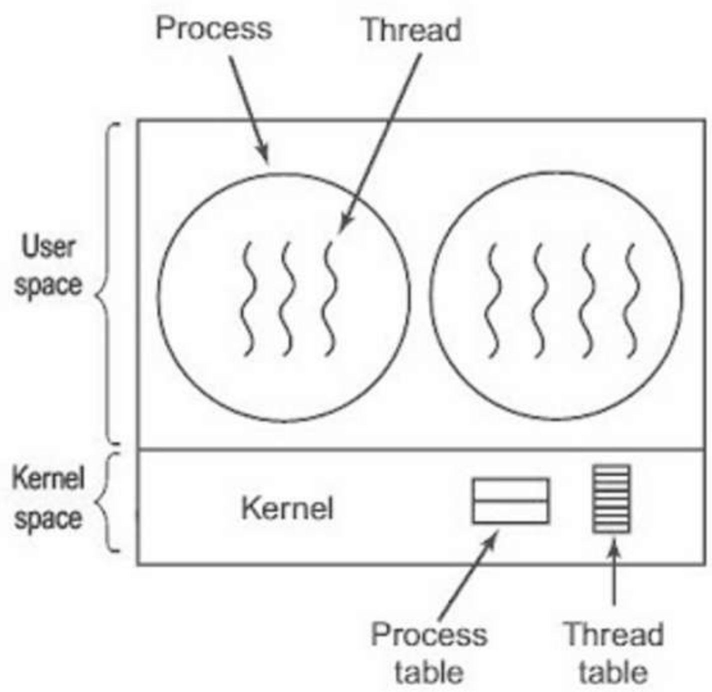
- The smallest unit of execution that can be scheduled by the OS
- The OS is responsible for supporting and managing all threads
- One Process Control Block (PCB) for each process, one Thread Control Block (TCB) for each thread
- The OS usually provides system calls to create and manage threads from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

Kernel Threads: PROs



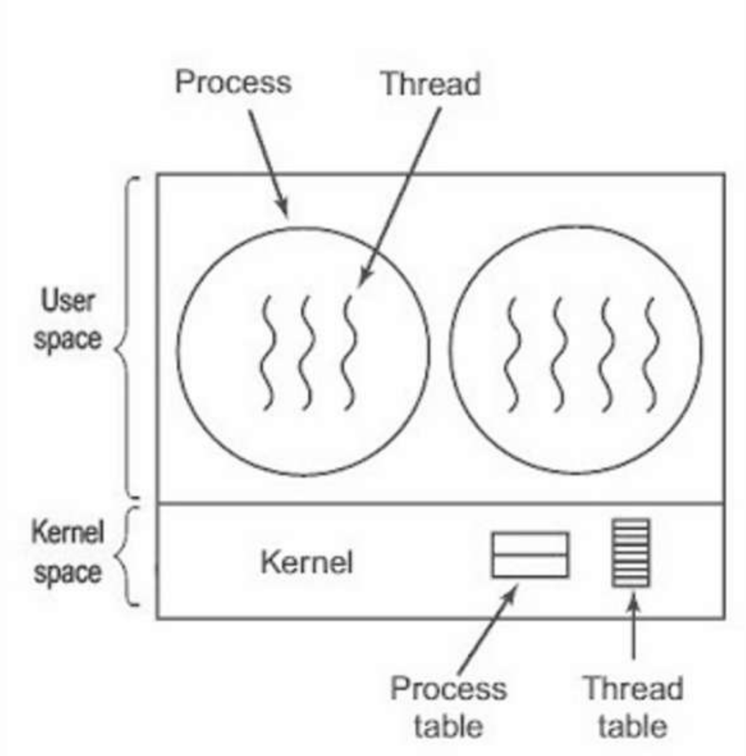
- PROs
  - The kernel has full knowledge of all threads
  - Scheduler may decide to give more CPU time to a process having a large numer of threads
  - Good for applications that frequently block
  - Switching between threads is faster than switching between processes

Kernel Threads: PROs



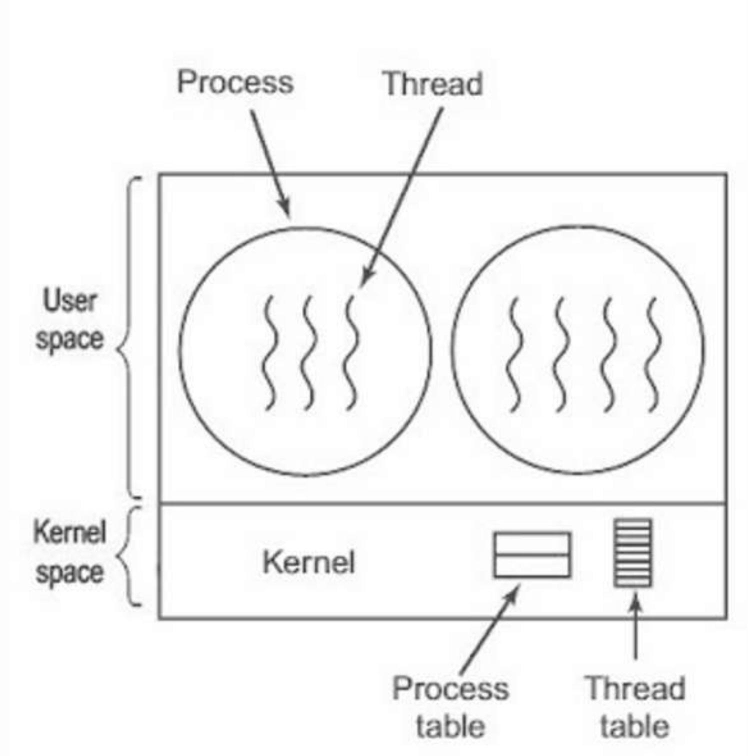
- PROs
  - The kernel has full knowledge of all threads
  - Scheduler may decide to give more CPU time to a process having a large numer of threads
  - Good for applications that frequently block
  - Switching between threads is faster than switching between processes

Kernel Threads: PROs



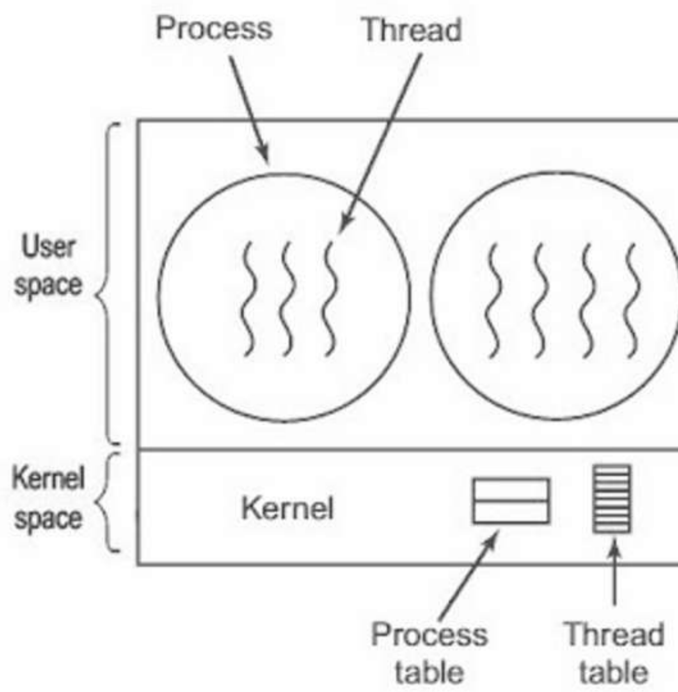
- PROs
  - The kernel has full knowledge of all threads
  - Scheduler may decide to give more CPU time to a process having a large numer of threads
  - Good for applications that frequently block
  - Switching between threads is faster than switching between processes

Kernel Threads: PROs



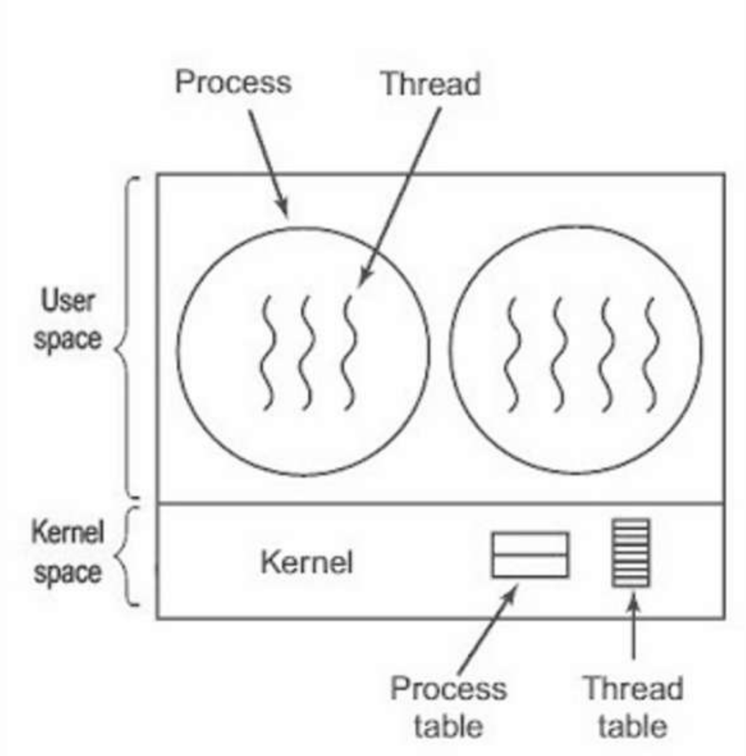
- PROs
  - The kernel has full knowledge of all threads
  - Scheduler may decide to give more CPU time to a process having a large numer of threads
  - Good for applications that frequently block
  - Switching between threads is faster than switching between processes

Kernel Threads: PROs



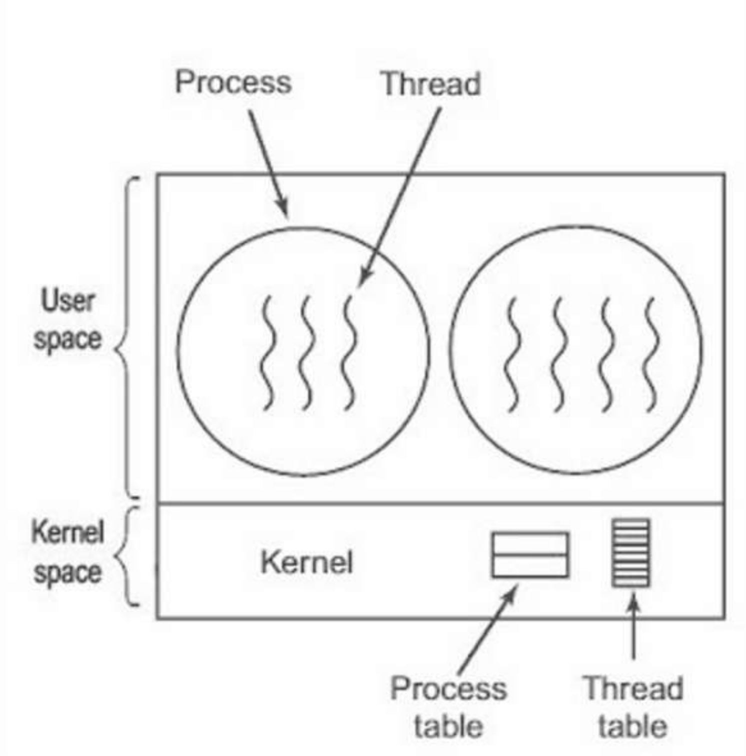
- PROs
  - The kernel has full knowledge of all threads
  - Scheduler may decide to give more CPU time to a process having a large numer of threads
  - Good for applications that frequently block
  - Switching between threads is faster than switching between processes

Kernel Threads: CONs



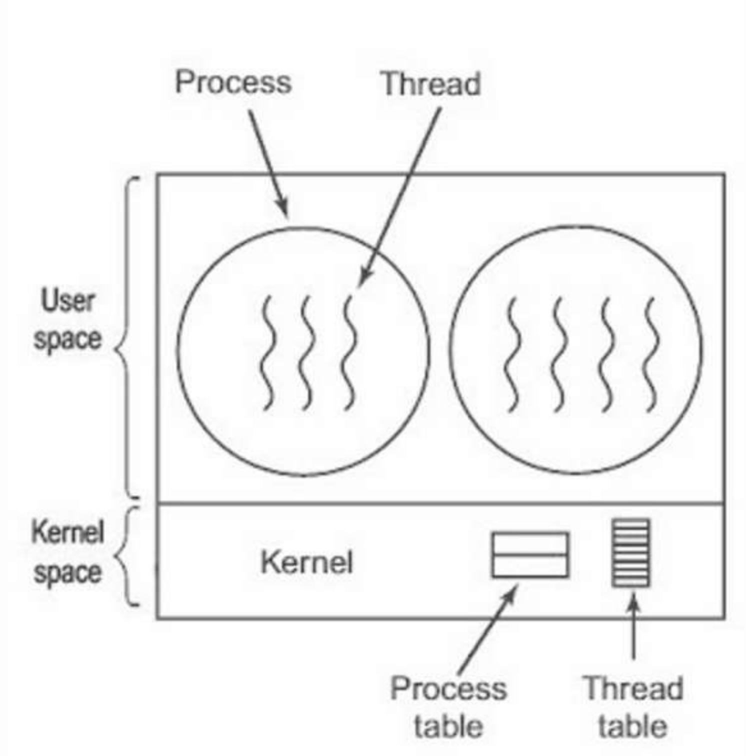
- CONs
  - Significant overhead and increase in kernel complexity
  - Slow and inefficient (need kernel invocations)
  - Context switching, although lighter, is managed by the kernel

Kernel Threads: CONs



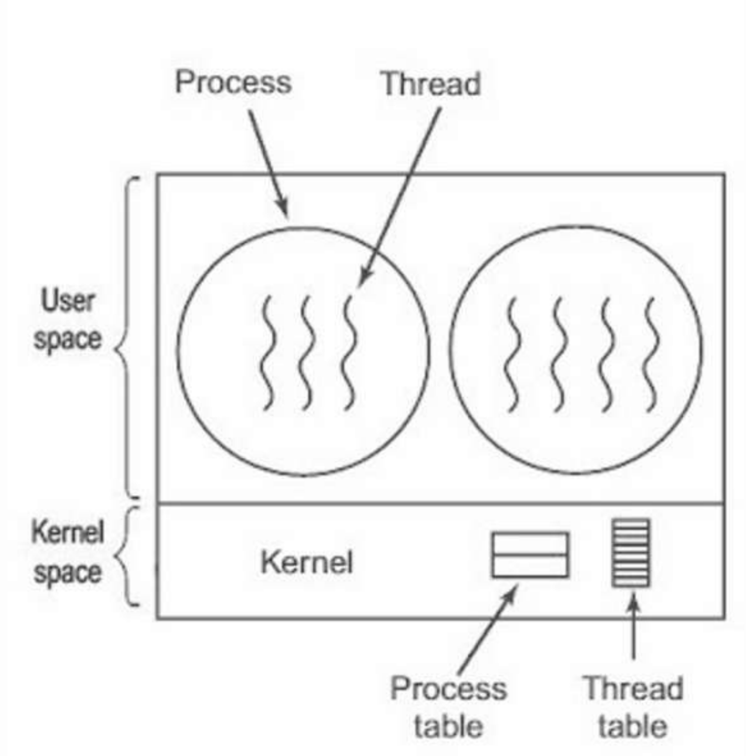
- CONs
  - Significant overhead and increase in kernel complexity
  - Slow and inefficient (need kernel invocations)
  - Context switching, although lighter, is managed by the kernel

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

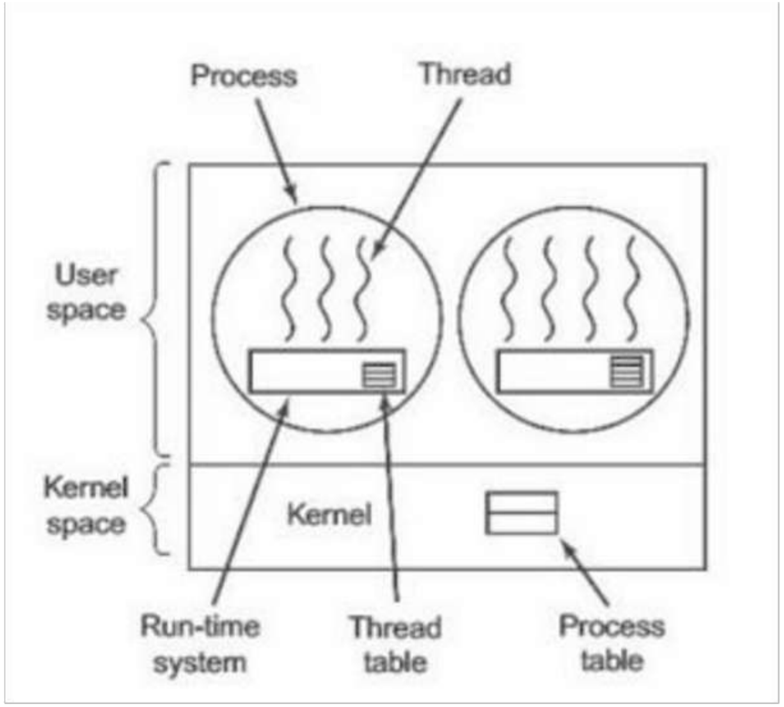
User Threads

- 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

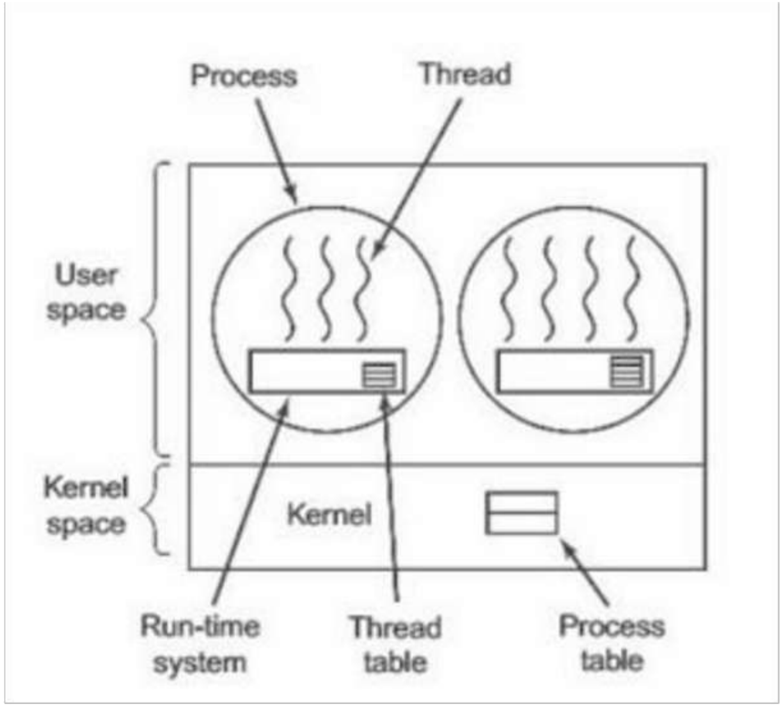
- Managed entirely by the run-time system (user-level thread library)
- The OS kernel knows nothing about user-level threads
- The OS kernel manages user-level threads as if they where single-threaded processes
- Ideally, thread operations should be as fast as a function call

User Threads: PROs



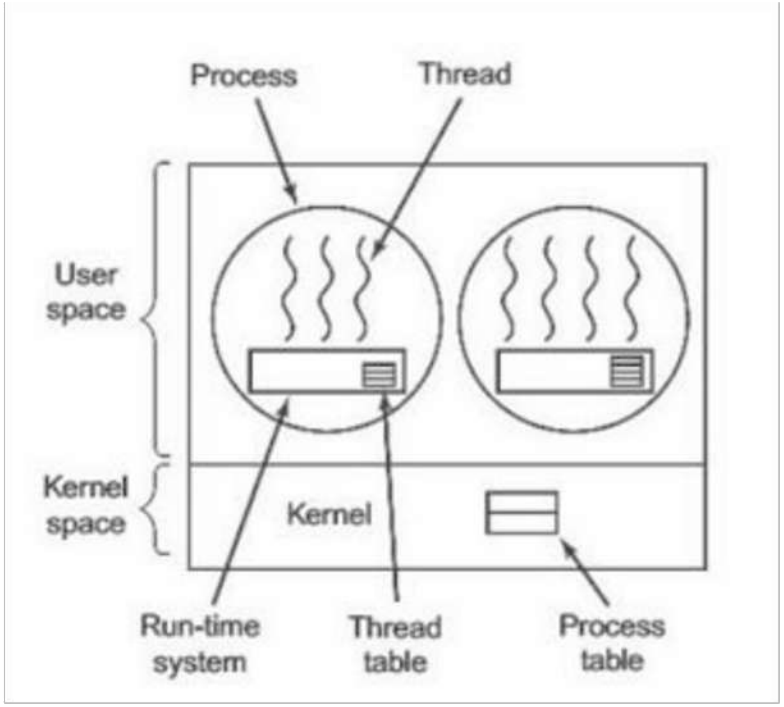
- PROs
  - Really fast and lightweight
  - Scheduling policies are more flexible
  - Can be implemented in OSs that do not support threading
  - No system calls involved, just user-space function calls
  - No actual context switch

User Threads: PROs



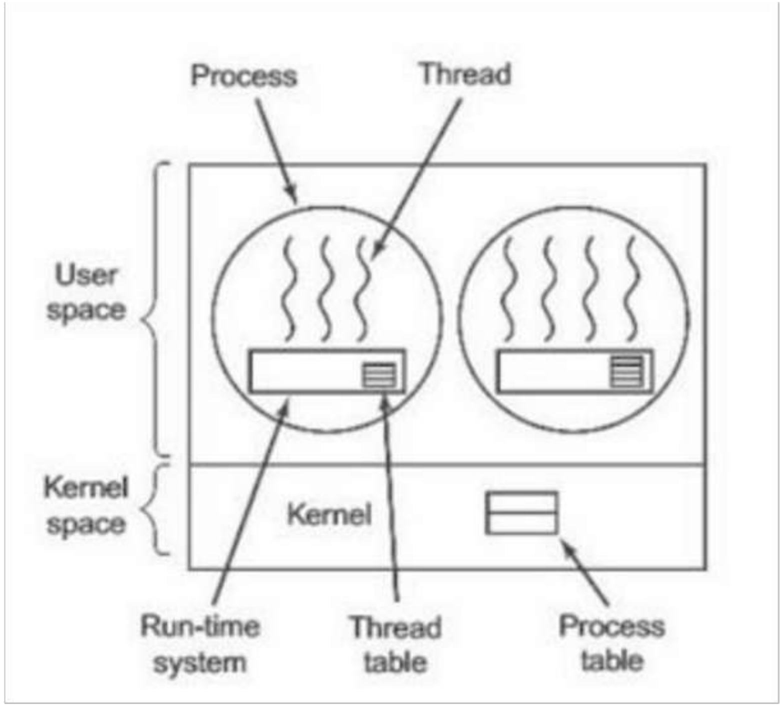
- PROs
  - Really fast and lightweight
  - Scheduling policies are more flexible
  - Can be implemented in OSs that do not support threading
  - No system calls involved, just user-space function calls
  - No actual context switch

User Threads: PROs



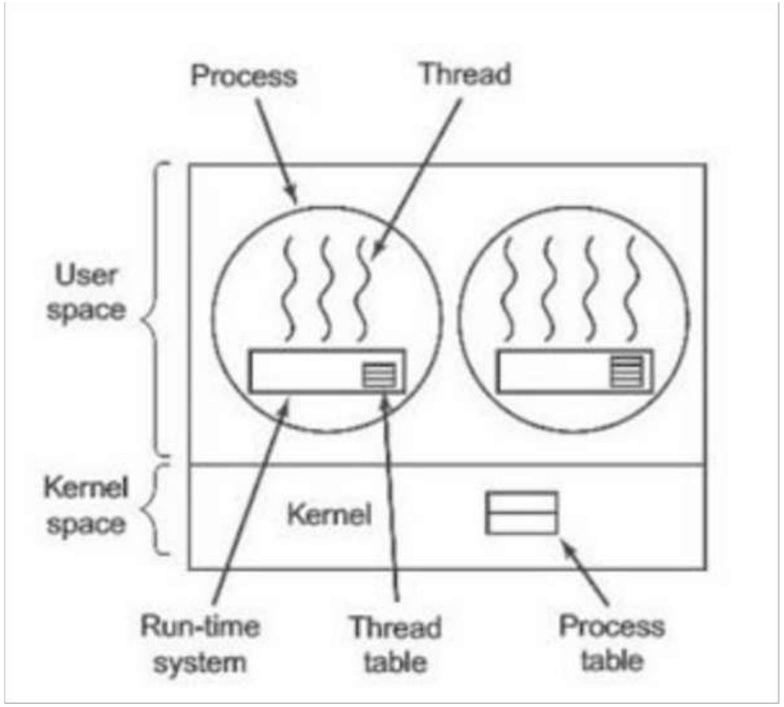
- PROs
  - Really fast and lightweight
  - Scheduling policies are more flexible
  - Can be implemented in OSs that do not support threading
  - No system calls involved, just user-space function calls
  - No actual context switch

User Threads: PROs



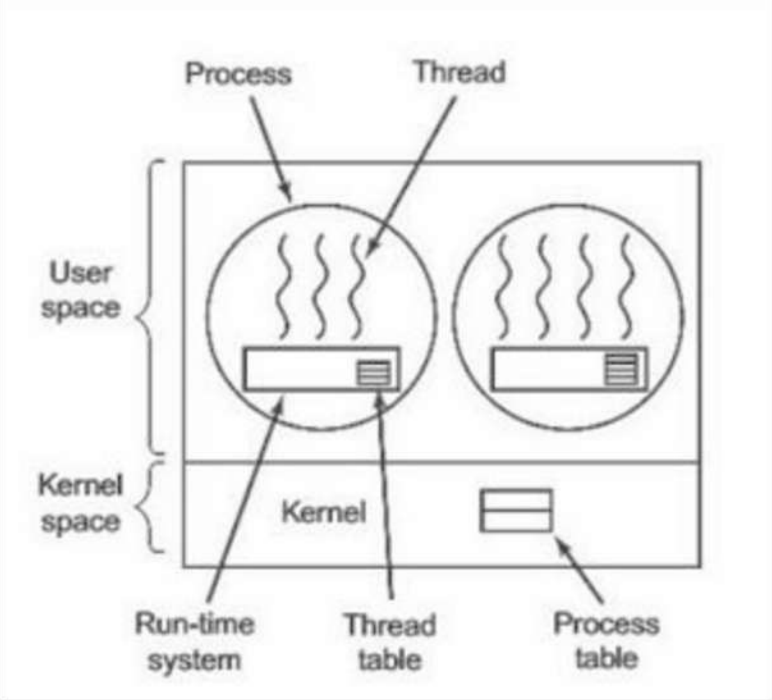
- PROs
  - Really fast and lightweight
  - Scheduling policies are more flexible
  - Can be implemented in OSs that do not support threading
  - No system calls involved, just user-space function calls
  - No actual context switch

User Threads: PROs



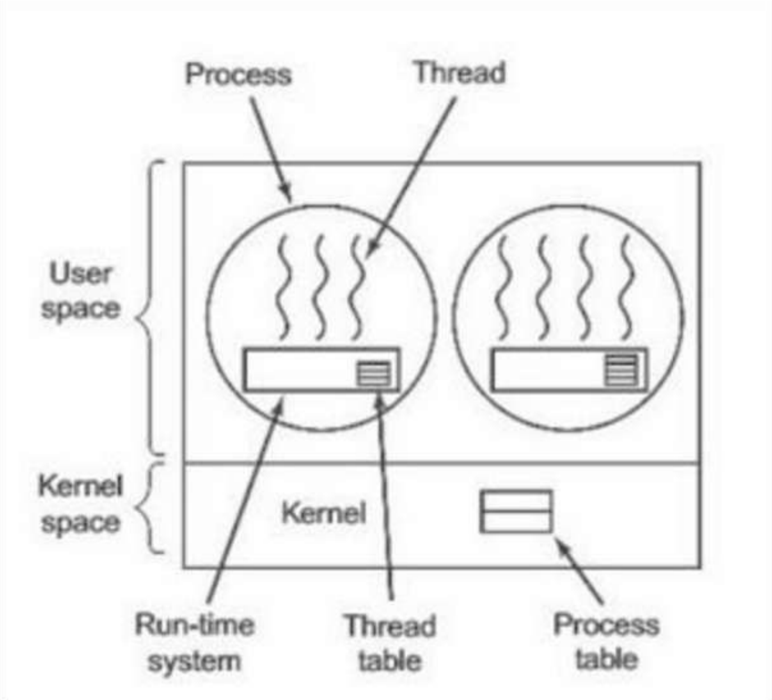
- PROs
  - Really fast and lightweight
  - Scheduling policies are more flexible
  - Can be implemented in OSs that do not support threading
  - No system calls involved, just user-space function calls
  - No actual context switch

User Threads: PROs



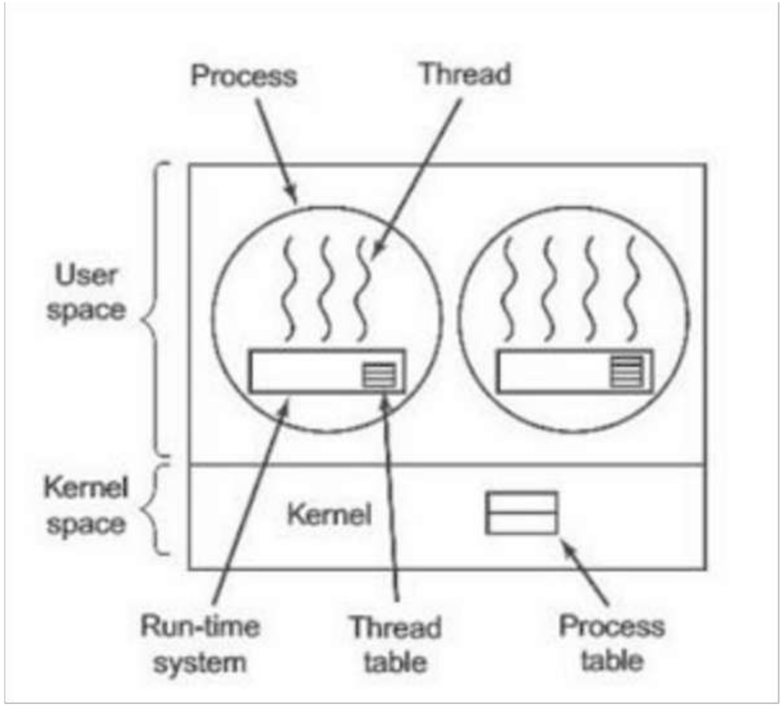
- PROs
  - Really fast and lightweight
  - Scheduling policies are more flexible
  - Can be implemented in OSs that do not support threading
  - No system calls involved, just user-space function calls
  - No actual context switch

User Threads: CONs



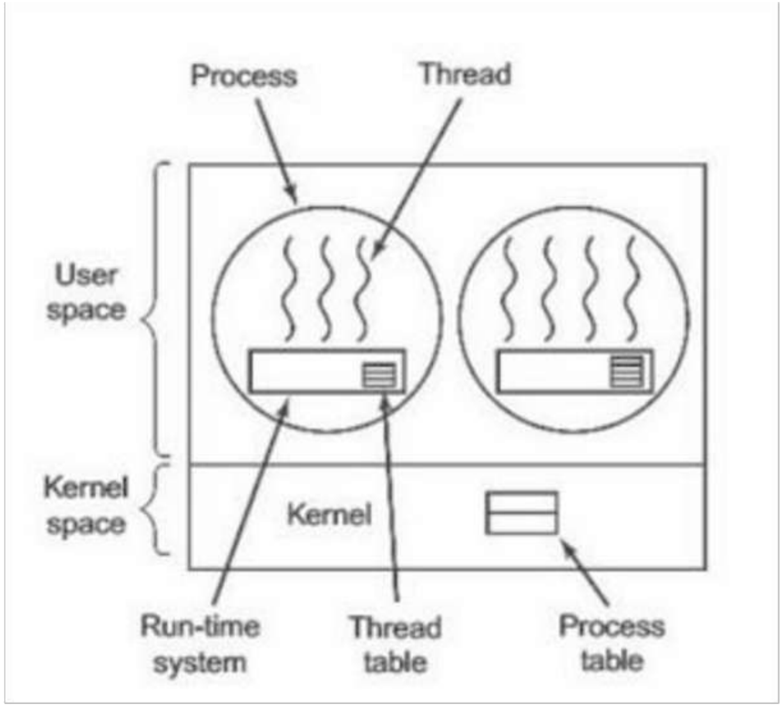
- CONs
  - No true concurrency of multi-threaded processes
  - Poor scheduling decisions
  - Lack of coordination between kernel and threads
    - A process with 100 threads competes for a time slice with a process with just 1 thread
  - Requires non-blocking system calls, otherwise all threads within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

User Threads: CONs



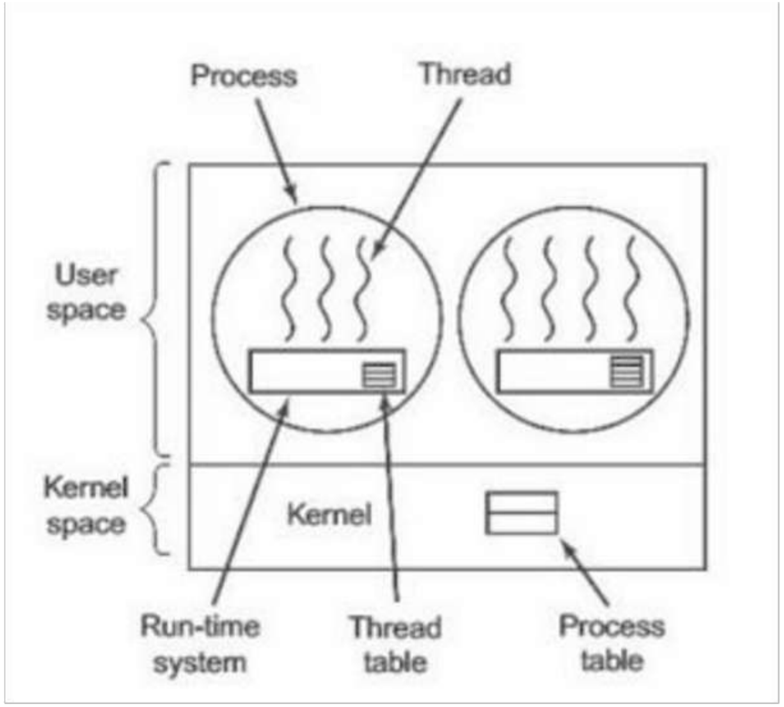
- CONs
  - No true concurrency of multi-threaded processes
  - Poor scheduling decisions
  - Lack of coordination between kernel and threads
    - A process with 100 threads competes for a time slice with a process with just 1 thread
  - Requires non-blocking system calls, otherwise all threads within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

User Threads: CONs



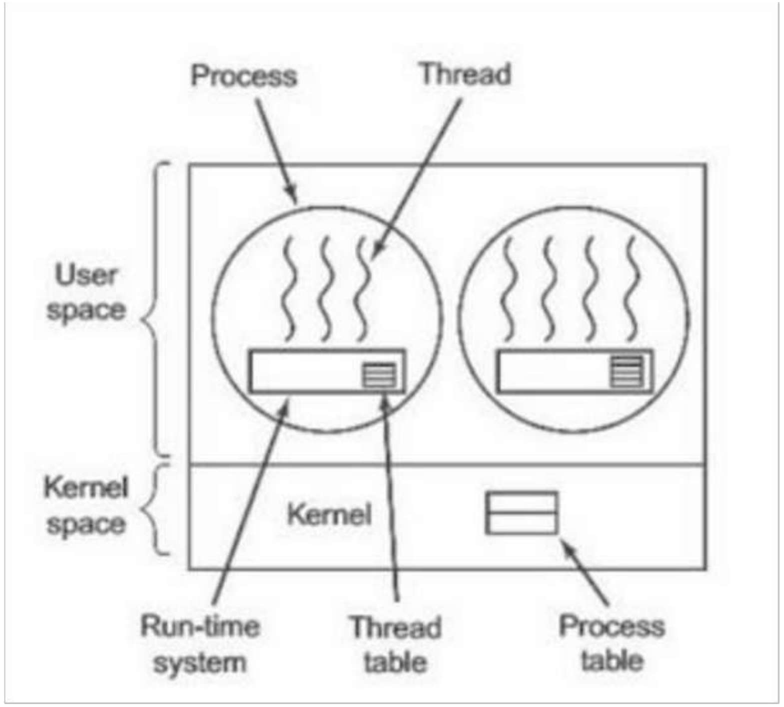
- CONs
  - No true concurrency of multi-threaded processes
  - Poor scheduling decisions
  - Lack of coordination between kernel and threads
    - A process with 100 threads competes for a time slice with a process with just 1 thread
  - Requires non-blocking system calls, otherwise all threads within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

User Threads: CONs



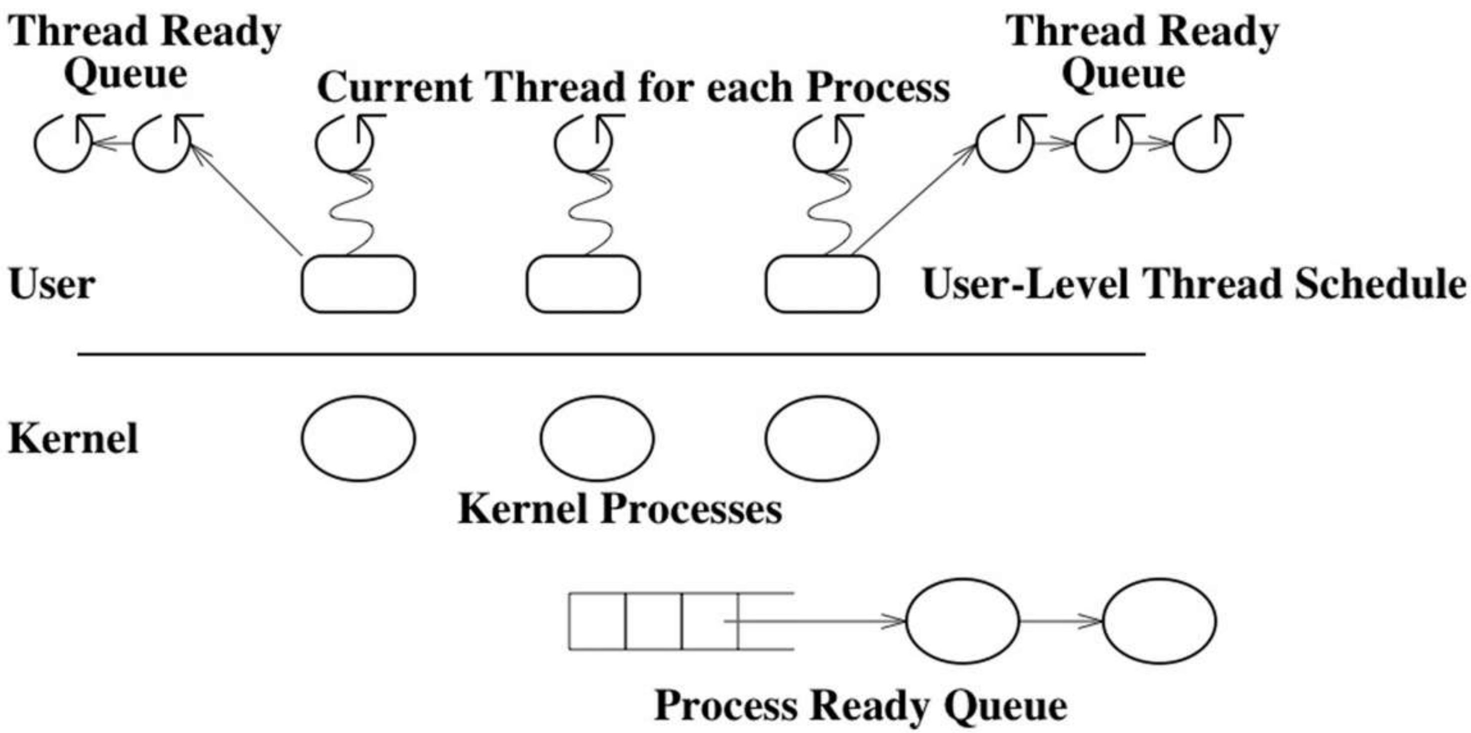
- CONs
  - No true concurrency of multi-threaded processes
  - Poor scheduling decisions
  - Lack of coordination between kernel and threads
    - A process with 100 threads competes for a time slice with a process with just 1 thread
  - Requires non-blocking system calls, otherwise all threads within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

User Threads: CONs

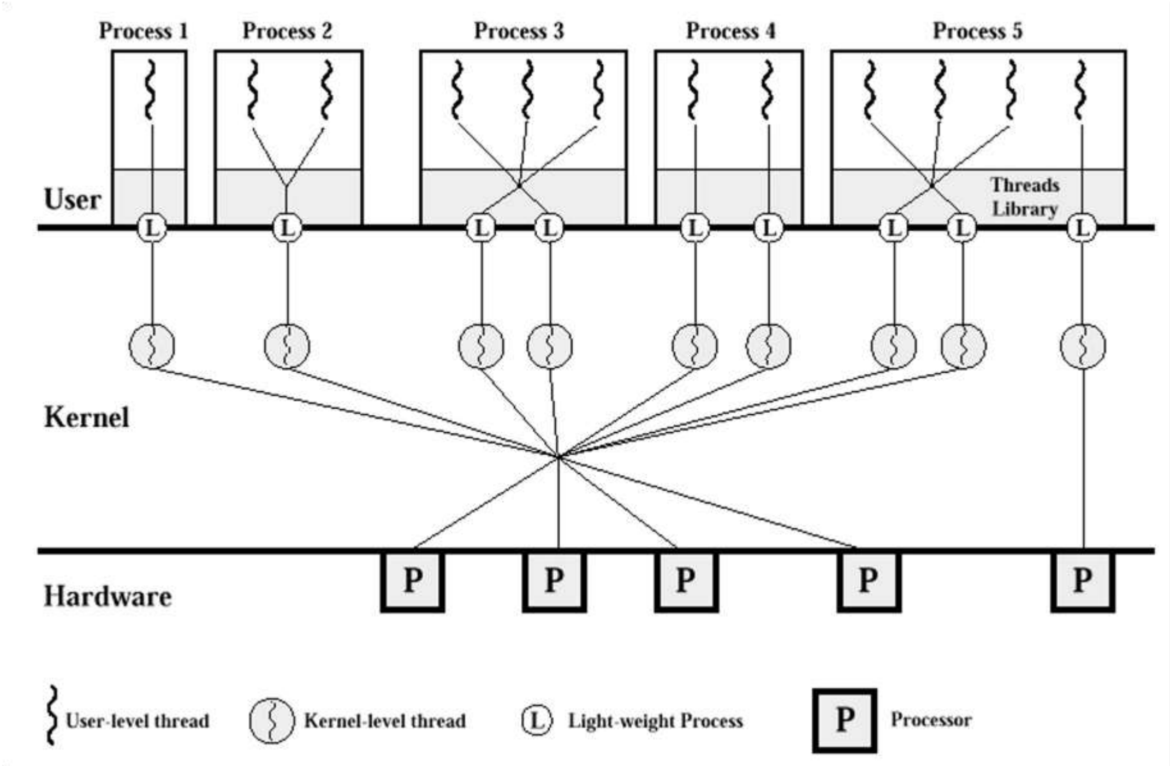


- CONs
  - No true concurrency of multi-threaded processes
  - Poor scheduling decisions
  - Lack of coordination between kernel and threads
    - A process with 100 threads competes for a time slice with a process with just 1 thread
  - Requires non-blocking system calls, otherwise all threads within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

User Threads



Hybrid Management: Lightweight Processes



Multi-threading Models

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

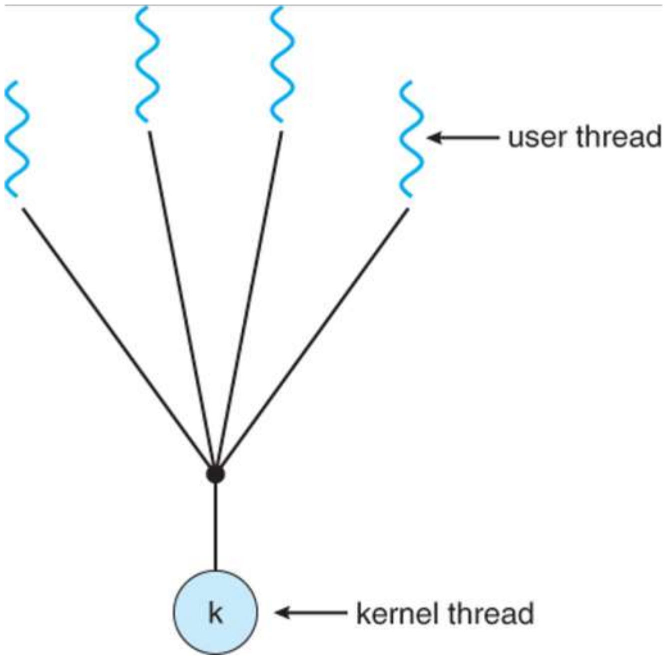
Multi-threading Models

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

Remember:

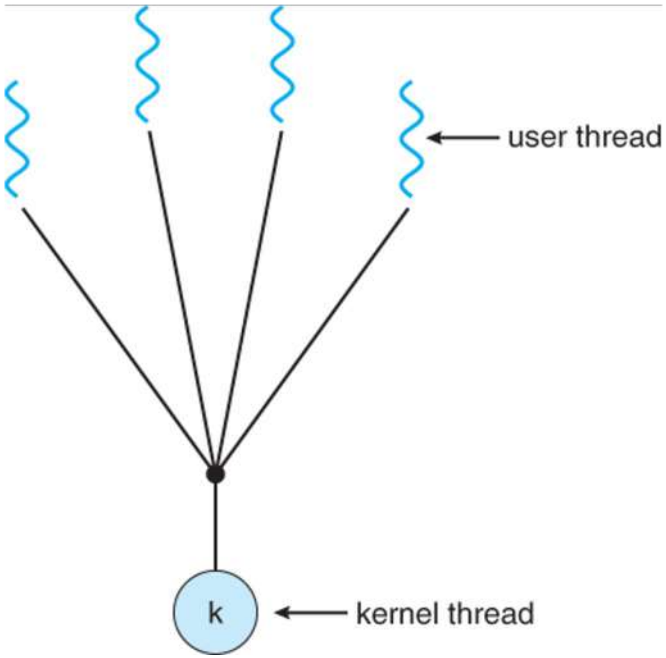
A kernel thread is the unit of execution that is scheduled by the OS to run on the CPU (similar to single-threaded process)

Many-to-One Model



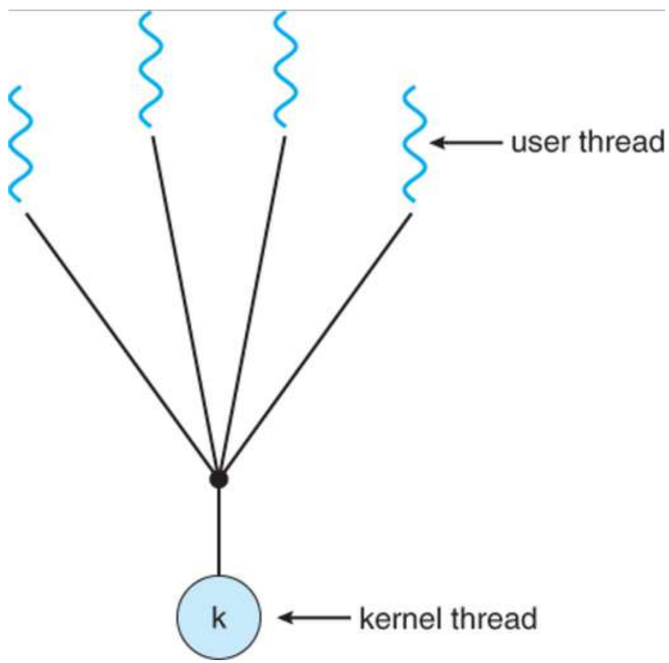
- Many user threads are all mapped onto a single kernel thread
- The process can only run one user thread at a time because there is only one kernel thread associated with it
- As single kernel thread can operate on a single CPU, multi-user-thread processes cannot be split across multiple CPUs
- If a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

Many-to-One Model



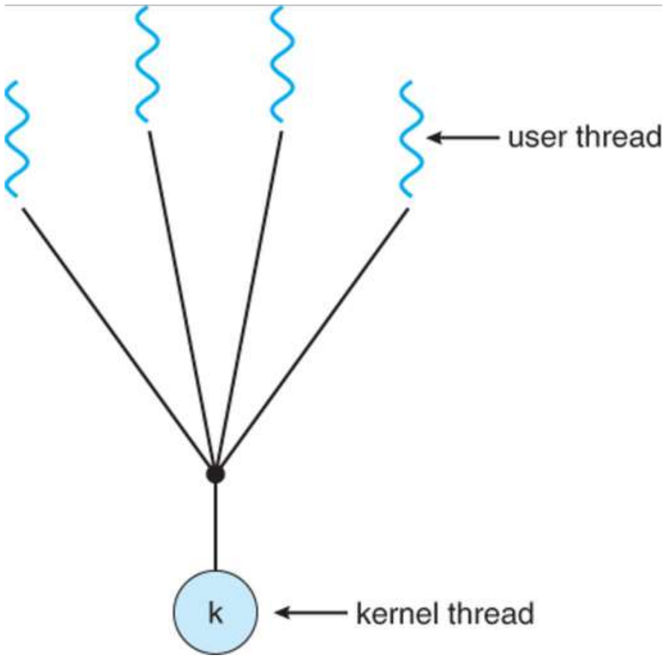
- Many user threads are all mapped onto a single kernel thread
- The process can only run one user thread at a time because there is only one kernel thread associated with it
- As single kernel thread can operate on a single CPU, multi-user-thread processes cannot be split across multiple CPUs
- If a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

Many-to-One Model



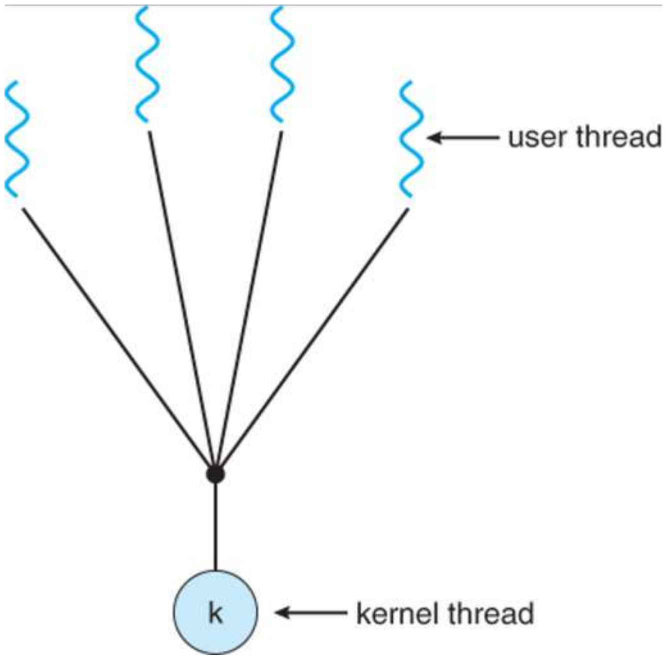
- Many user threads are all mapped onto a single kernel thread
- The process can only run one user thread at a time because there is only one kernel thread associated with it
- As single kernel thread can operate on a single CPU, multi-user-thread processes cannot be split across multiple CPUs
- If a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

Many-to-One Model



- 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

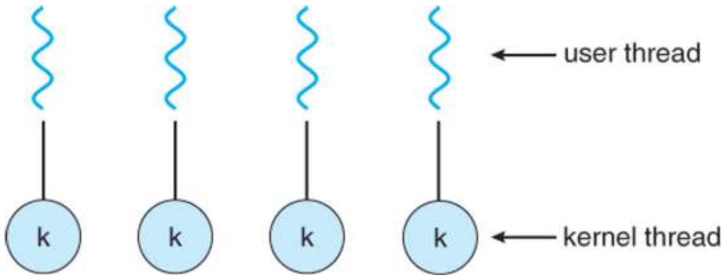
Many-to-One Model



- 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

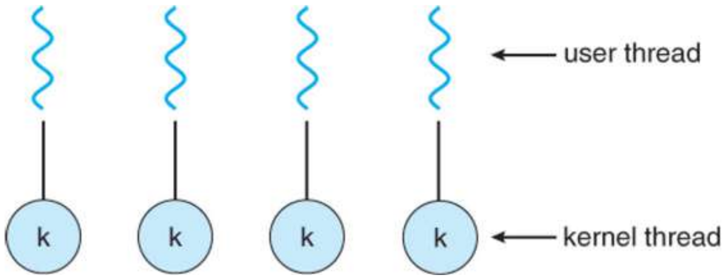
pure user-level

One-to-One Model



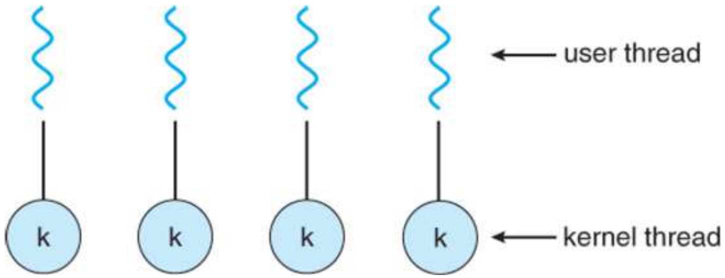
- A separate kernel thread to handle each user thread
- Overcomes the limitations of blocking system calls and splitting of processes across multiple CPUs
- The overhead of managing the one-to-one model is more significant and may slow down the system
- Most implementations of this model place a limit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

One-to-One Model



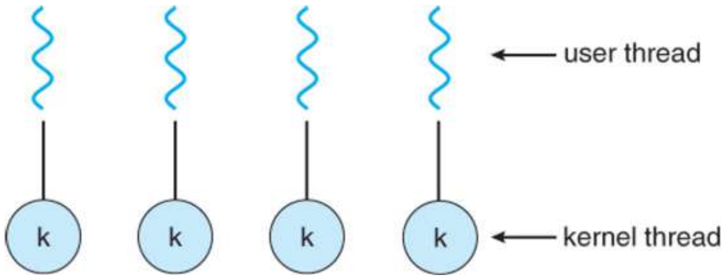
- A separate kernel thread to handle each user thread
- Overcomes the limitations of blocking system calls and splitting of processes across multiple CPUs
- The overhead of managing the one-to-one model is more significant and may slow down the system
- Most implementations of this model place a limit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

One-to-One Model



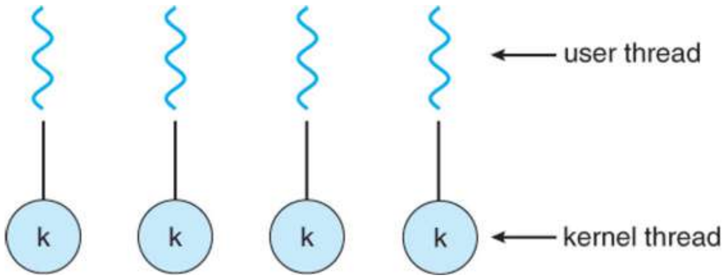
- A separate kernel thread to handle each user thread
- Overcomes the limitations of blocking system calls and splitting of processes across multiple CPUs
- The overhead of managing the one-to-one model is more significant and may slow down the system
- Most implementations of this model place a limit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

One-to-One Model



- 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

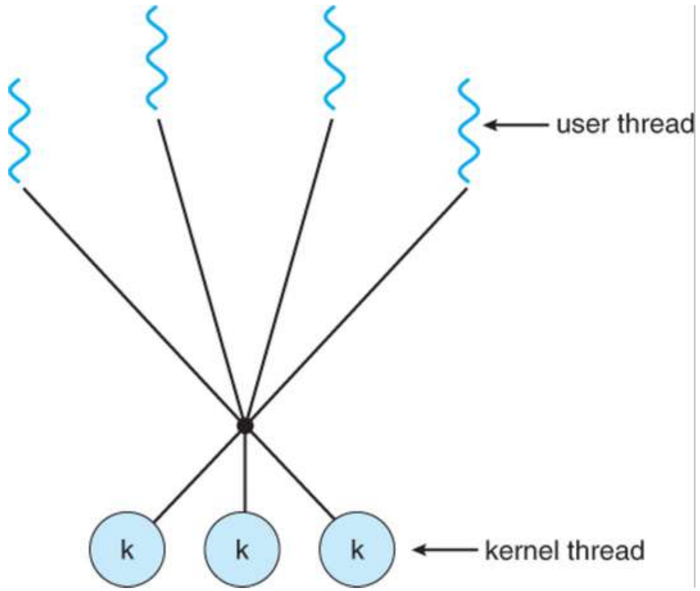
One-to-One Model



- 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

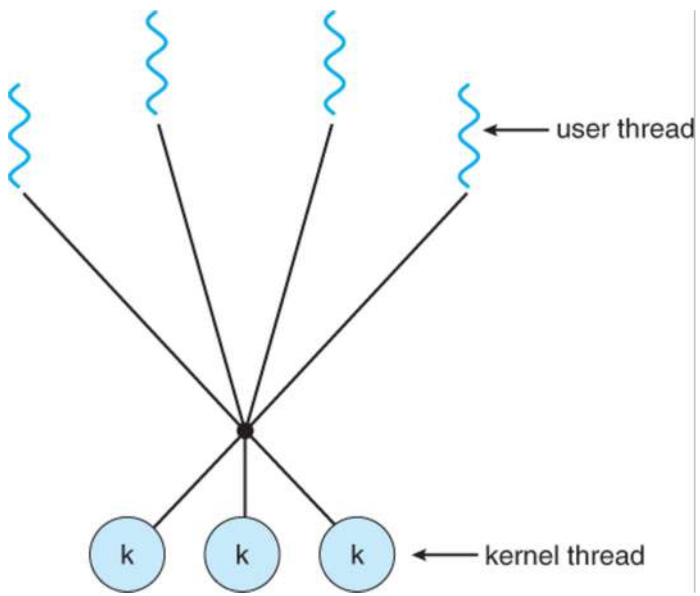
pure kernel-level

Many-to-Many Model



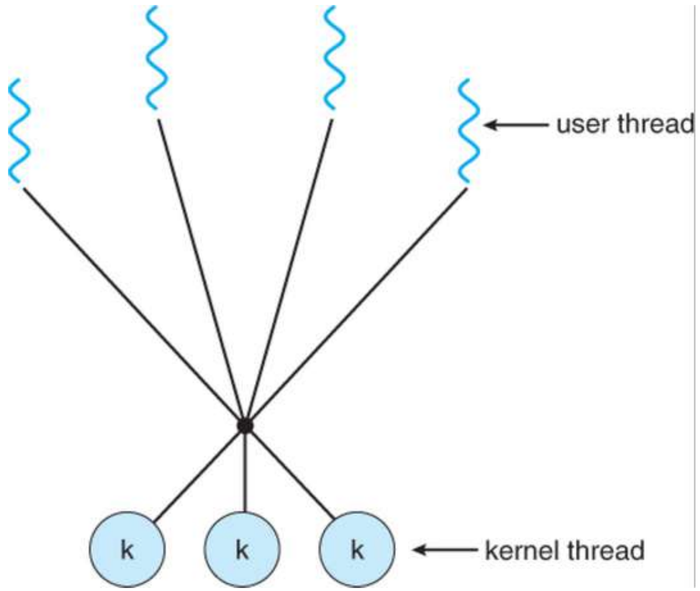
- Multiplexes any number of user threads onto an equal or smaller number of kernel threads
- Users have no restrictions on the number of threads created
- Processes can be split across multiple processors
- Blocking kernel system calls do not block the entire process

Many-to-Many Model



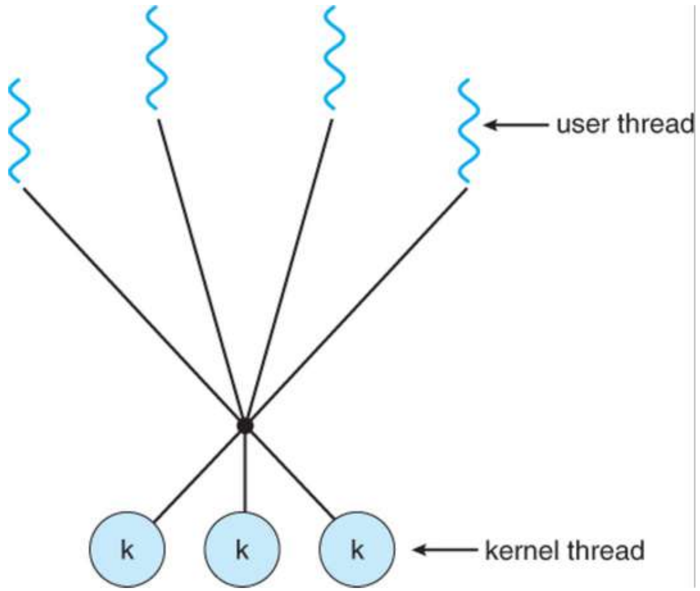
- Multiplexes any number of user threads onto an equal or smaller number of kernel threads
- Users have no restrictions on the number of threads created
- Processes can be split across multiple processors
- Blocking kernel system calls do not block the entire process

Many-to-Many Model



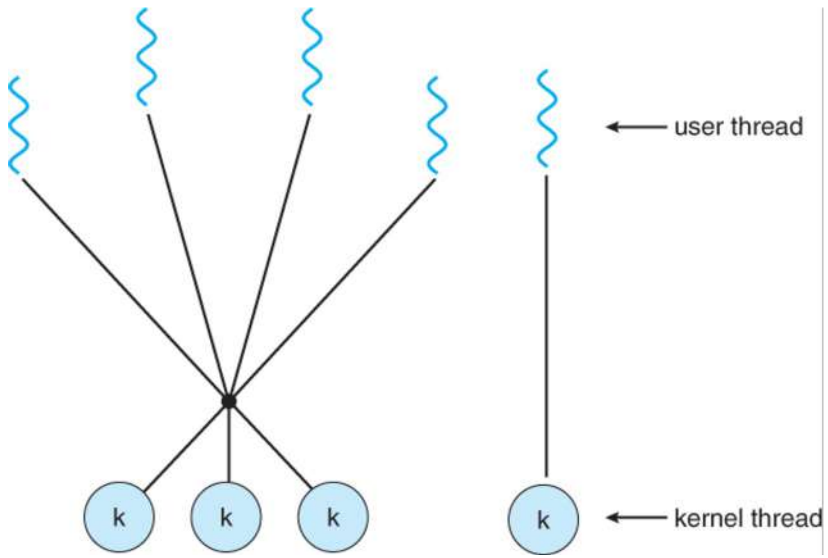
- Multiplexes any number of user threads onto an equal or smaller number of kernel threads
- Users have no restrictions on the number of threads created
- Processes can be split across multiple processors
- Blocking kernel system calls do not block the entire process

Many-to-Many Model



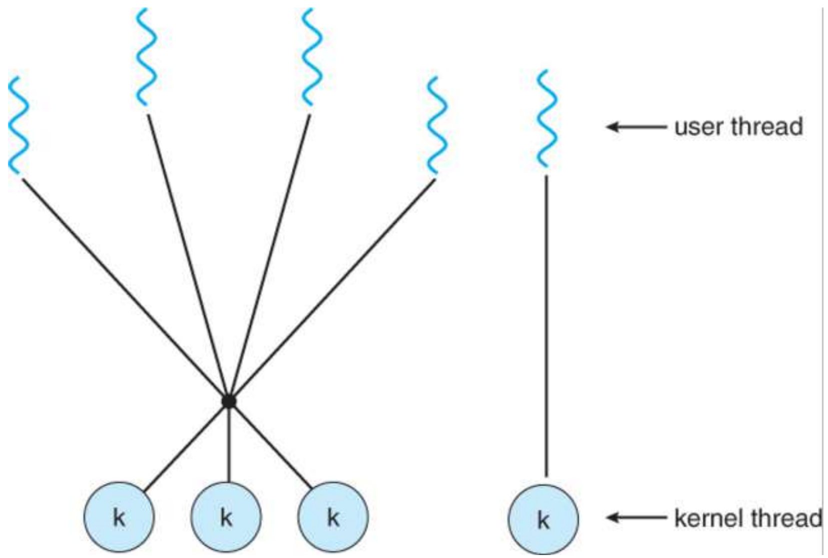
- Multiplexes any number of user threads onto an equal or smaller number of kernel threads
- Users have no restrictions on the number of threads created
- Processes can be split across multiple processors
- Blocking kernel system calls do not block the entire process

Two-Level Model



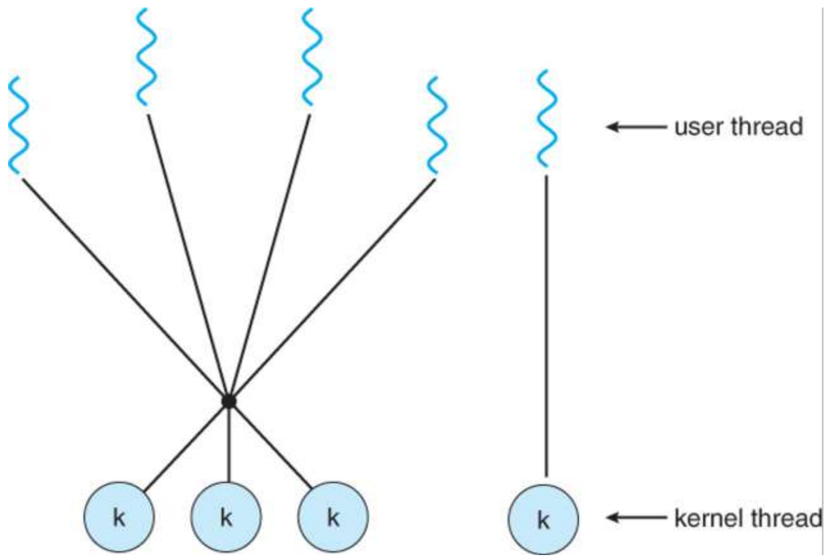
- A variant of the many-to-many model
- Mixes many-to-many with one-to-one
- Increases the flexibility of scheduling policies

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

Pthreads

- The POSIX standard (IEEE 1003.1c) defines the specification for Pthreads, not the implementation

Pthreads

- The POSIX standard (IEEE 1003.1c) defines the specification for Pthreads, not the implementation
- Available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows

Pthreads

- The POSIX standard (IEEE 1003.1c) defines the specification for Pthreads, not the implementation
- Available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows
- Global variables are shared amongst all threads

Pthreads

- The POSIX standard (IEEE 1003.1c) defines the specification for Pthreads, not the implementation
- Available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows
- Global variables are shared amongst all threads
- One thread can wait for the others to rejoin before continuing

Java Threads

- Java Threads can be created in 2 ways:
  - Extending the Thread class
  - Implementing the Runnable Interface

Java Threads

- Java Threads can be created in 2 ways:
  - Extending the Thread class
  - Implementing the Runnable Interface
- Both solutions require to override the run() method

Java Threads

- Java Threads can be created in 2 ways:
  - Extending the Thread class
  - Implementing the Runnable Interface
- Both solutions require to override the run() method
- Note that Java doesn't support multiple inheritance!
  - If your class extends the Thread class, it cannot extend any other class
  - In such a situation, implementing Runnable is preferable

Java Threads: Single-Threaded Web Server

```java
public class SingleThreadedServer implements Runnable {

    protected int          serverPort   = 8080;
    protected ServerSocket serverSocket = null;
    protected boolean      isStopped    = false;

    public SingleThreadedServer(int port){
        this.serverPort = port;
    }

    public void run() {

        try {
            this.serverSocket = new ServerSocket(this.serverPort);
        }
        catch (IOException e) {
            throw new RuntimeException("Cannot open port " + this.serverPort, e);
        }

        while(!this.isStopped) {
            Socket clientSocket = null;
            try {
                clientSocket = this.serverSocket.accept();
            } catch (IOException e) {
                if(this.isStopped) {
                    System.out.println("Server Stopped.") ;
                    return;
                }
                throw new RuntimeException(
                    "Error accepting client connection", e);
            }
            try {
                processClientRequest(clientSocket);
            } catch (Exception e) {
                //log exception and go on to the next request.
            }
        }

        System.out.println("Server Stopped.");
    }

    private void processClientRequest(Socket clientSocket) throws Exception {
        // Process client request here ...
    }

}
```

This is the simplest (not optimal) single-threaded implementation of a Java web server

Java Threads: Single-Threaded Web Server

```java
public class SingleThreadedServer implements Runnable {

    protected int          serverPort   = 8080;
    protected ServerSocket serverSocket = null;
    protected boolean      isStopped    = false;

    public SingleThreadedServer(int port){
        this.serverPort = port;
    }

    public void run() {

        try {
            this.serverSocket = new ServerSocket(this.serverPort);
        }
        catch (IOException e) {
            throw new RuntimeException("Cannot open port " + this.serverPort, e);
        }

        while(!this.isStopped) {
            Socket clientSocket = null;
            try {
                clientSocket = this.serverSocket.accept();
            } catch (IOException e) {
                if(this.isStopped) {
                    System.out.println("Server Stopped.") ;
                    return;
                }
                throw new RuntimeException(
                    "Error accepting client connection", e);
            }
            try {
                processClientRequest(clientSocket);
            } catch (Exception e) {
                //log exception and go on to the next request.
            }
        }

        System.out.println("Server Stopped.");
    }

    private void processClientRequest(Socket clientSocket) throws Exception {
        // Process client request here ...
    }

}
```

The Server Loop

1. Wait for a client request
2. Process a single client request
3. Repeat from 1

Java Threads: Single-Threaded Web Server

```java
public class SingleThreadedServer implements Runnable {

    protected int          serverPort   = 8080;
    protected ServerSocket serverSocket = null;
    protected boolean      isStopped    = false;

    public SingleThreadedServer(int port){
        this.serverPort = port;
    }

    public void run() {

        try {
            this.serverSocket = new ServerSocket(this.serverPort);
        }
        catch (IOException e) {
            throw new RuntimeException("Cannot open port " + this.serverPort, e);
        }

        while(!this.isStopped) {
            Socket clientSocket = null;
            try {
                clientSocket = this.serverSocket.accept();
            } catch (IOException e) {
                if(this.isStopped) {
                    System.out.println("Server Stopped.") ;
                    return;
                }
                throw new RuntimeException(
                    "Error accepting client connection", e);
            }
            try {
                processClientRequest(clientSocket);
            } catch (Exception e) {
                //log exception and go on to the next request.
            }
        }

        System.out.println("Server Stopped.");
    }

    private void processClientRequest(Socket clientSocket) throws Exception {
        // Process client request here ...
    }

}
```

The Server Loop

1. Wait for a client request
2. Process a single client request
3. Repeat from 1

A single-threaded server processes incoming requests in the very same thread that accepts connections

Java Threads: Single-Threaded Web Server

```java
public class SingleThreadedServer implements Runnable {

    protected int          serverPort   = 8080;
    protected ServerSocket serverSocket = null;
    protected boolean      isStopped    = false;

    public SingleThreadedServer(int port){
        this.serverPort = port;
    }

    public void run() {

        try {
            this.serverSocket = new ServerSocket(this.serverPort);
        }
        catch (IOException e) {
            throw new RuntimeException("Cannot open port " + this.serverPort, e);
        }

        while(!this.isStopped) {
            Socket clientSocket = null;
            try {
                clientSocket = this.serverSocket.accept();
            } catch (IOException e) {
                if(this.isStopped) {
                    System.out.println("Server Stopped.") ;
                    return;
                }
                throw new RuntimeException(
                    "Error accepting client connection", e);
            }
            try {
                processClientRequest(clientSocket);
            } catch (Exception e) {
                //log exception and go on to the next request.
            }
        }

        System.out.println("Server Stopped.");
    }

    private void processClientRequest(Socket clientSocket) throws Exception {
        // Process client request here ...
    }

}
```

The Server Loop

1. Wait for a client request
2. Process a single client request
3. Repeat from 1

A single-threaded server processes incoming requests in the very same thread that accepts connections

This is not a good idea as clients can connect to the server only when this is inside the serverSocket.accept() method call

Java Threads: Multi-Threaded Web Server

```java
public class MultiThreadedServer implements Runnable {

    protected int          serverPort    = 8080;
    protected ServerSocket serverSocket  = null;
    protected boolean      isStopped     = false;

    public MultiThreadedServer(int port){
        this.serverPort = port;
    }

    public void run() {

        try {
            this.serverSocket = new ServerSocket(this.serverPort);
        }
        catch (IOException e) {
            throw new RuntimeException("Cannot open port " + this.serverPort, e);
        }

        while(!this.isStopped) {
            Socket clientSocket = null;
            try {
                clientSocket = this.serverSocket.accept();
            } catch (IOException e) {
                if(this.isStopped) {
                    System.out.println("Server Stopped.") ;
                    return;
                }
                throw new RuntimeException(
                    "Error accepting client connection", e);
            }
            new Thread(new WorkerRunnable(clientSocket, "Multithreaded Server")).start();
        }

        System.out.println("Server Stopped.");
    }

}
```

The server loop is very similar to that of a single-threaded server

Java Threads: Multi-Threaded Web Server

```java
public class MultiThreadedServer implements Runnable {

    protected int          serverPort   = 8080;
    protected ServerSocket serverSocket = null;
    protected boolean      isStopped    = false;

    public MultiThreadedServer(int port){
        this.serverPort = port;
    }

    public void run() {

        try {
            this.serverSocket = new ServerSocket(this.serverPort);
        }
        catch (IOException e) {
            throw new RuntimeException("Cannot open port " + this.serverPort, e);
        }

        while(!this.isStopped) {
            Socket clientSocket = null;
            try {
                clientSocket = this.serverSocket.accept();
            } catch (IOException e) {
                if(this.isStopped) {
                    System.out.println("Server Stopped.") ;
                    return;
                }
                throw new RuntimeException(
                    "Error accepting client connection", e);
            }
            new Thread(new WorkerRunnable(clientSocket, "Multithreaded Server")).start();
        }

        System.out.println("Server Stopped.");
    }

}
```

The server loop is very similar to that of a single-threaded server

A multi-threaded server passes the connection on to a worker thread that processes the request

Java Threads: Multi-Threaded Web Server

```
public class MultiThreadedServer implements Runnable {

    protected int          serverPort   = 8080;
    protected ServerSocket serverSocket = null;
    protected boolean      isStopped    = false;

    public MultiThreadedServer(int port){
        this.serverPort = port;
    }

    public void run() {

        try {
            this.serverSocket = new ServerSocket(this.serverPort);
        }
        catch (IOException e) {
            throw new RuntimeException("Cannot open port " + this.serverPort, e);
        }

        while(!this.isStopped) {
            Socket clientSocket = null;
            try {
                clientSocket = this.serverSocket.accept();
            } catch (IOException e) {
                if(this.isStopped) {
                    System.out.println("Server Stopped.") ;
                    return;
                }
                throw new RuntimeException(
                    "Error accepting client connection", e);
            }
            new Thread(new WorkerRunnable(clientSocket, "Multithreaded Server")).start();
        }

        System.out.println("Server Stopped.");
    }

}
```

The server loop is very similar to that of a single-threaded server

A multi-threaded server passes the connection on to a worker thread that processes the request

This way the thread listening for incoming requests spends as much time as possible in the serverSocket.accept() call

Java Threads: Multi-Threaded Web Server

```java
public class MultiThreadedServer implements Runnable {

    protected int          serverPort   = 8080;
    protected ServerSocket serverSocket = null;
    protected boolean      isStopped    = false;

    public MultiThreadedServer(int port){
        this.serverPort = port;
    }

    public void run() {

        try {
            this.serverSocket = new ServerSocket(this.serverPort);
        }
        catch (IOException e) {
            throw new RuntimeException("Cannot open port " + this.serverPort, e);
        }

        while(!this.isStopped) {
            Socket clientSocket = null;
            try {
                clientSocket = this.serverSocket.accept();
            } catch (IOException e) {
                if(this.isStopped) {
                    System.out.println("Server Stopped.") ;
                    return;
                }
                throw new RuntimeException(
                    "Error accepting client connection", e);
            }
            new Thread(new WorkerRunnable(clientSocket, "Multithreaded Server")).start();
        }

        System.out.println("Server Stopped.");
    }

}
```

The server loop is very similar to that of a single-threaded server

A multi-threaded server passes the connection on to a worker thread that processes the request

This way the thread listening for incoming requests spends as much time as possible in the serverSocket.accept() call

The risk of clients being denied access to the server because the listening thread is outside the accept() call is minimized

Java Threads: Multi-Threaded Web Server

```java
public class WorkerRunnable implements Runnable{

    protected Socket clientSocket = null;
    protected String serverText   = null;

    public WorkerRunnable(Socket clientSocket, String serverText) {
        this.clientSocket = clientSocket;
        this.serverText   = serverText;
    }

    public void run() {
        // process client request here ...
    }
}
```

The server loop is very similar to that of a single-threaded server

A multi-threaded server passes the connection on to a worker thread that processes the request

This way the thread listening for incoming requests spends as much time as possible in the serverSocket.accept() call

The risk of clients being denied access to the server because the listening thread is outside the accept() call is minimized

Thread Pools

- Let's go back to the multi-threaded web server example

Thread Pools

- Let's go back to the multi-threaded web server example
- Upon each request the web server receives it spawns off a new thread

Thread Pools

- Let's go back to the multi-threaded web server example
- Upon each request the web server receives it spawns off a new thread
- Creating new threads rather than new process is surely less expensive

Thread Pools

- Let's go back to the multi-threaded web server example
- Upon each request the web server receives it spawns off a new thread
- Creating new threads rather than new process is surely less expensive
- Still, some major problems might occur:
  - overhead to create a thread for each request
  - number of concurrent threads active on the system is possibly unbound

Thread Pools

- Let's go back to the multi-threaded web server example
- Upon each request the web server receives it spawns off a new thread
- Creating new threads rather than new process is surely less expensive
- Still, some major problems might occur:
  - overhead to create a thread for each request
  - number of concurrent threads active on the system is possibly unbound
- Solution: use a thread pool

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 web server gets 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 web server gets 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 web server gets 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

Thread Pools: Benefits

- Servicing a request with an existing thread is faster than waiting to create a thread

Thread Pools: Benefits

- Servicing a request with an existing thread is faster than waiting to create a thread
- A thread pool limits the number of threads that exist at any one point

Thread Pools: Benefits

- Servicing a request with an existing thread is faster than waiting to create a thread
- A thread pool limits the number of threads that exist at any one point
- Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task
  - Example: the task could be scheduled to execute after a time delay or to execute periodically

Threading Issues: fork() and exec()

- Q: If one thread forks, is the entire process copied, or is the new process single-threaded?

Threading Issues: fork() and exec()

- Q: If one thread forks, is the entire process copied, or is the new process single-threaded?
- A1: System dependent

Threading Issues: fork() and exec()

- Q: If one thread forks, is the entire process copied, or is the new process single-threaded?
- A1: System dependent
- A2: If the new process execs right away, there is no need to copy all the other threads, otherwise the entire process should be copied

Threading Issues: fork() and exec()

- Q: If one thread forks, is the entire process copied, or is the new process single-threaded?
- A1: System dependent
- A2: If the new process execs right away, there is no need to copy all the other threads, otherwise the entire process should be copied
- A3: Many versions of UNIX provide multiple versions of the fork call for this purpose

Threading Issues: Signal Handling

- Q: When a multi-threaded process receives a signal, to what thread should that signal be delivered?

Threading Issues: Signal Handling

- Q: When a multi-threaded process receives a signal, to what thread should that signal be delivered?
- A: There are 4 major options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals in a process

Threading Issues: Signal Handling (UNIX)

- UNIX allows individual threads to indicate which signals they are accepting and which they are ignoring
- Provides 2 separate system calls for delivering signals to process/threads, respectively:
  - kill(pid, signal)
  - pthread_kill(tid, signal)

Thread Scheduling: Contention Scope

- The scope in which threads compete for the use of physical CPUs

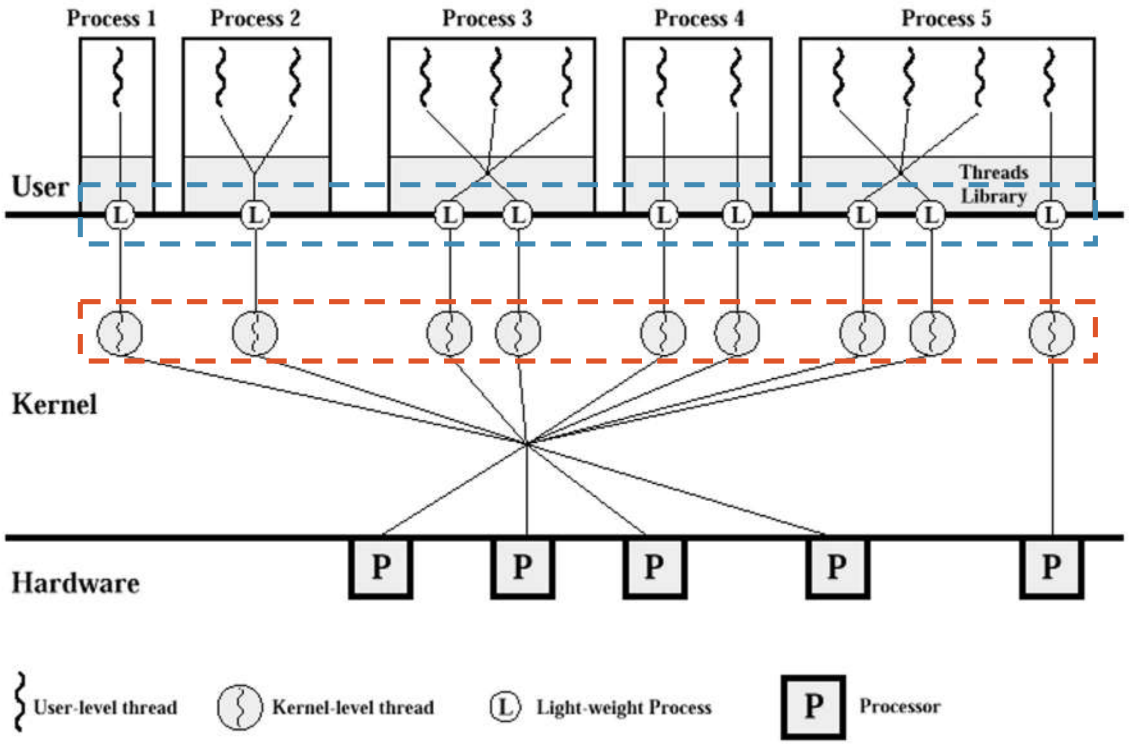
Thread Scheduling: Contention Scope

- The scope in which threads compete for the use of physical CPUs
- Process Contention Scope (PCS)
  - competition occurs between threads that are part of the same process (multiple user threads mapped to a single kernel thread, managed by the thread library)
  - on systems implementing many-to-one and many-to-many threads

Thread Scheduling: Contention Scope

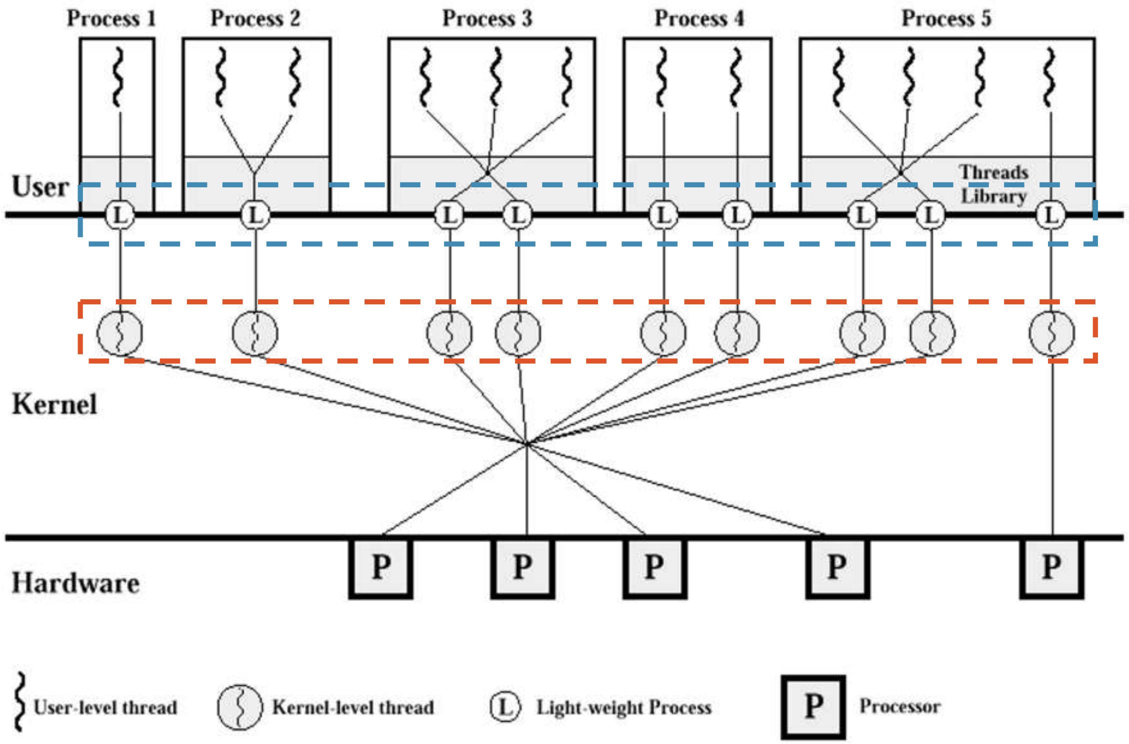
- The scope in which threads compete for the use of physical CPUs
- System Contention Scope (SCS)
  - involves the system scheduler scheduling kernel threads to run on one or more CPUs
  - on systems implementing one-to-one threads

Thread Scheduling: Activation



Many implementations of threads provide a virtual processor (L) as an interface between user and kernel thread (many-to-many or two-tier)

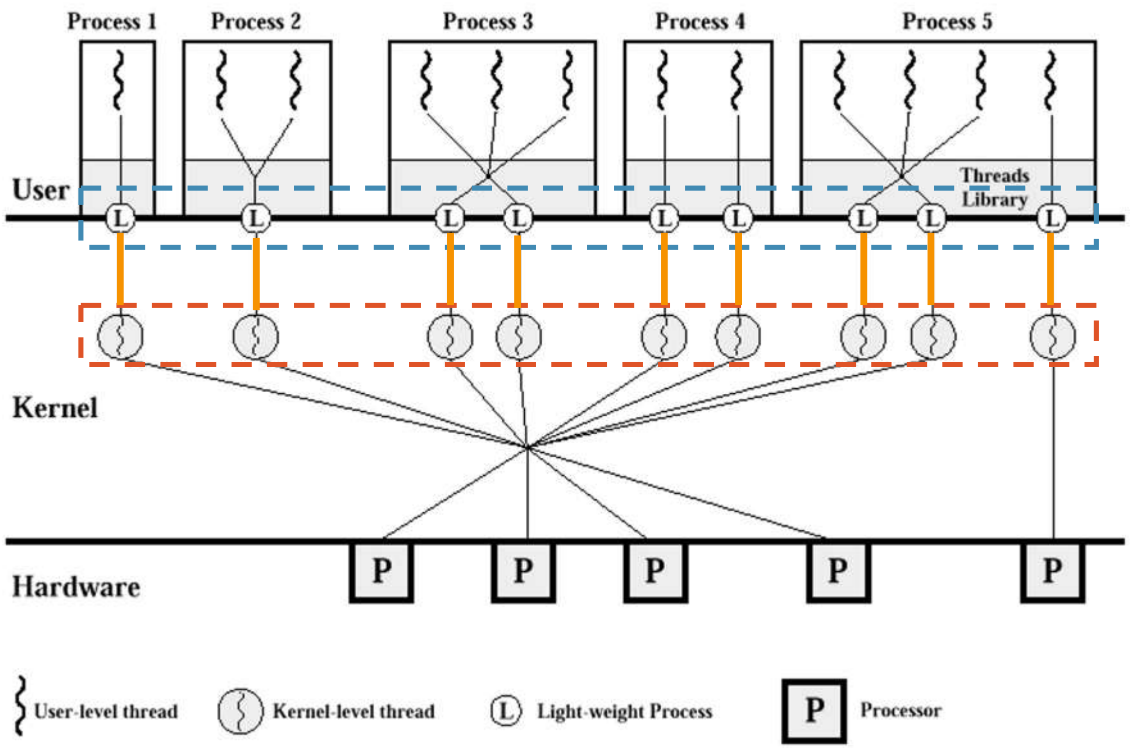
Thread Scheduling: Activation



Many implementations of threads provide a virtual processor (L) as an interface between user and kernel thread (many-to-many or two-tier)

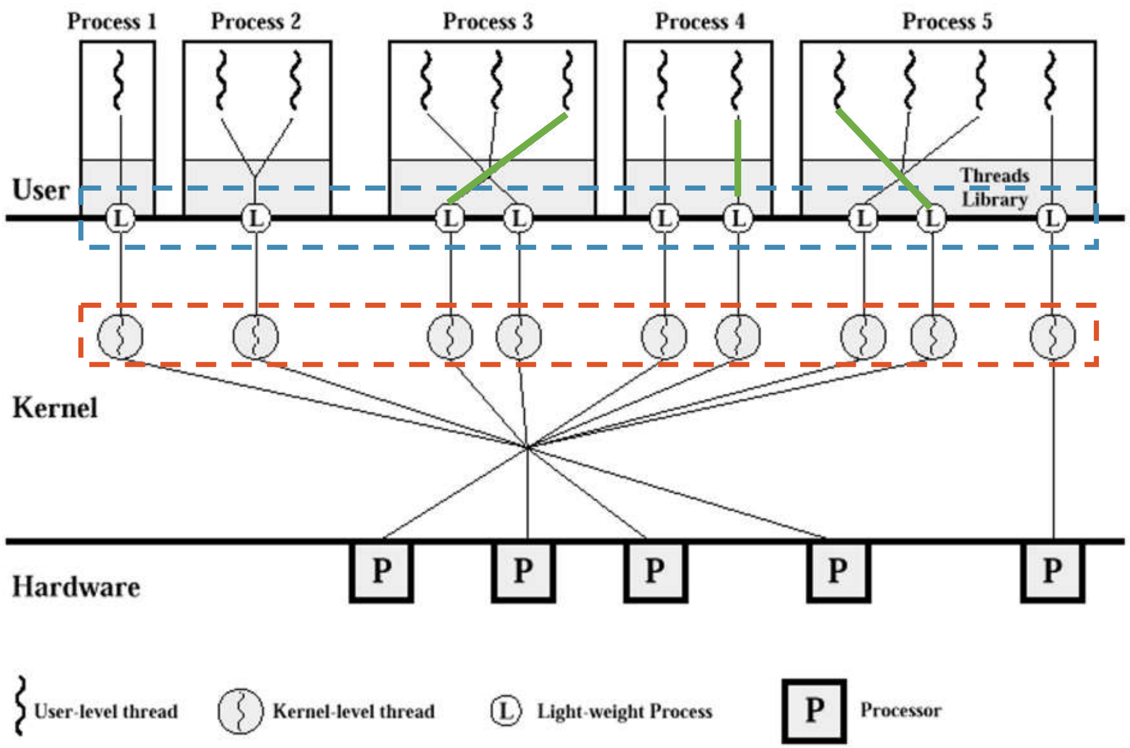
Light-Weight Process (LWP)

Thread Scheduling: Activation



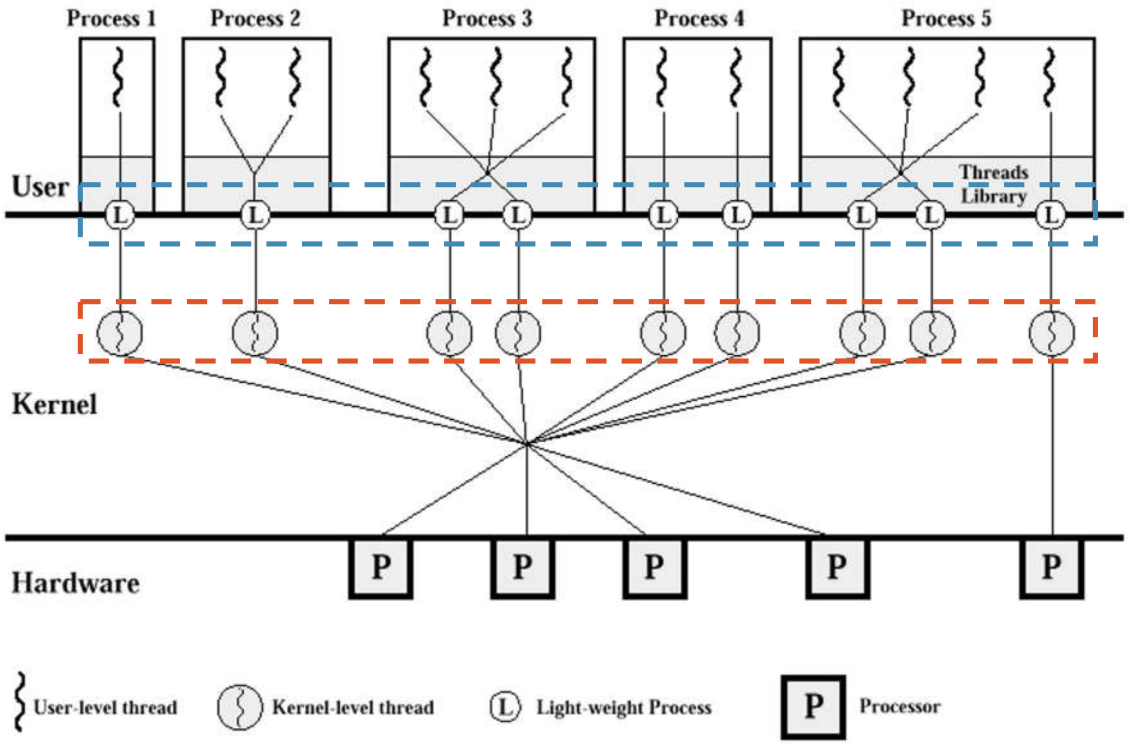
1:1 correspondence between LWPs and kernel threads

Thread Scheduling: Activation



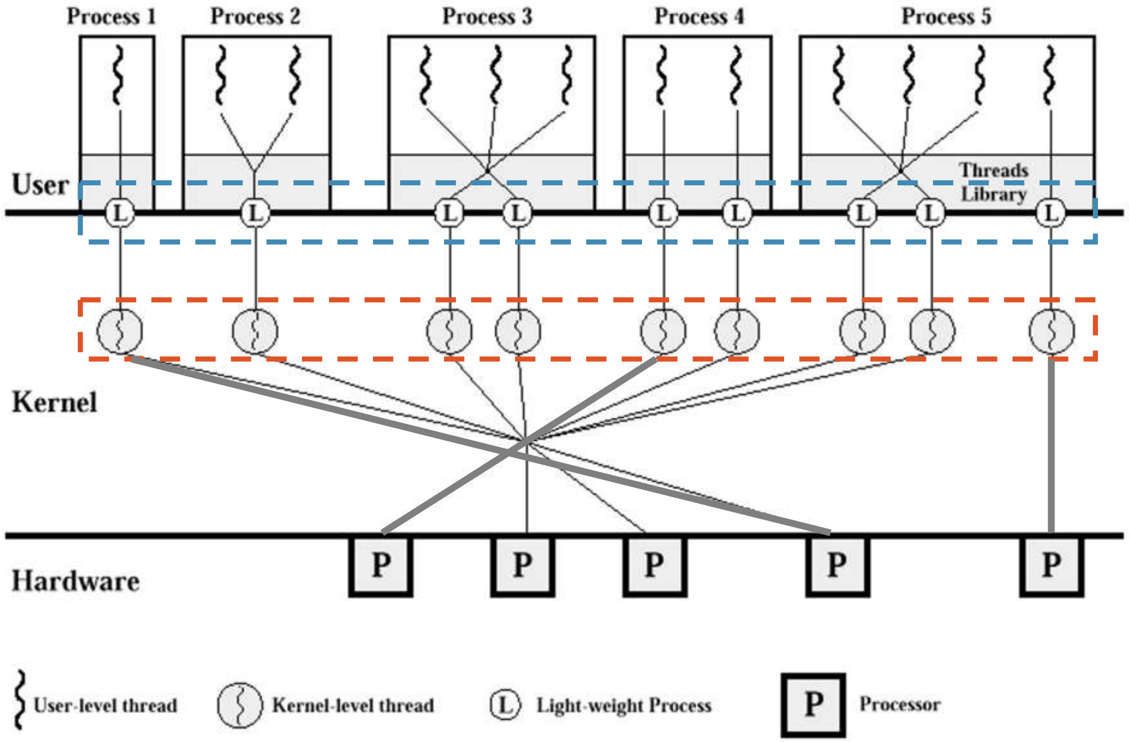
The application (user-level thread library) maps user threads onto available LWPs

Thread Scheduling: Activation



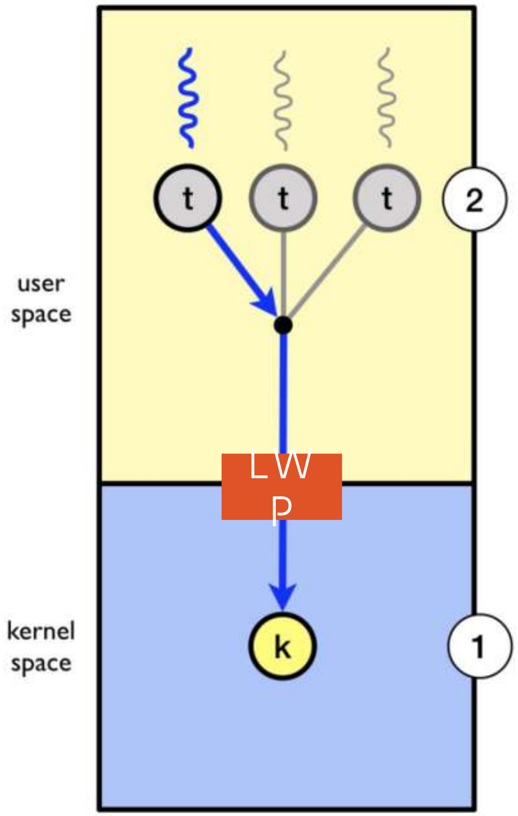
The number of kernel threads available in the system may change dynamically

Thread Scheduling: Activation



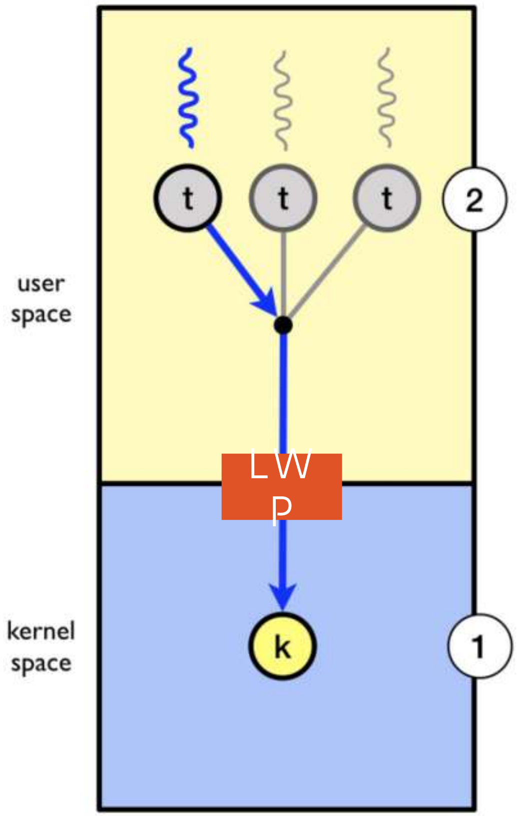
Kernel threads are scheduled onto the real processor(s) by the OS

Scheduler Activations: Example



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

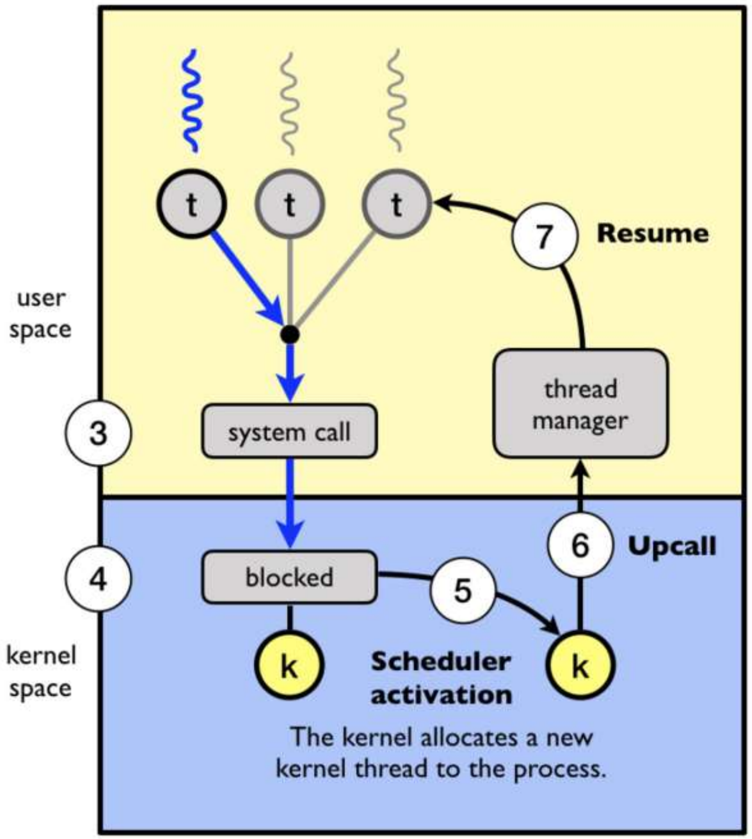
Scheduler Activations: Example



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

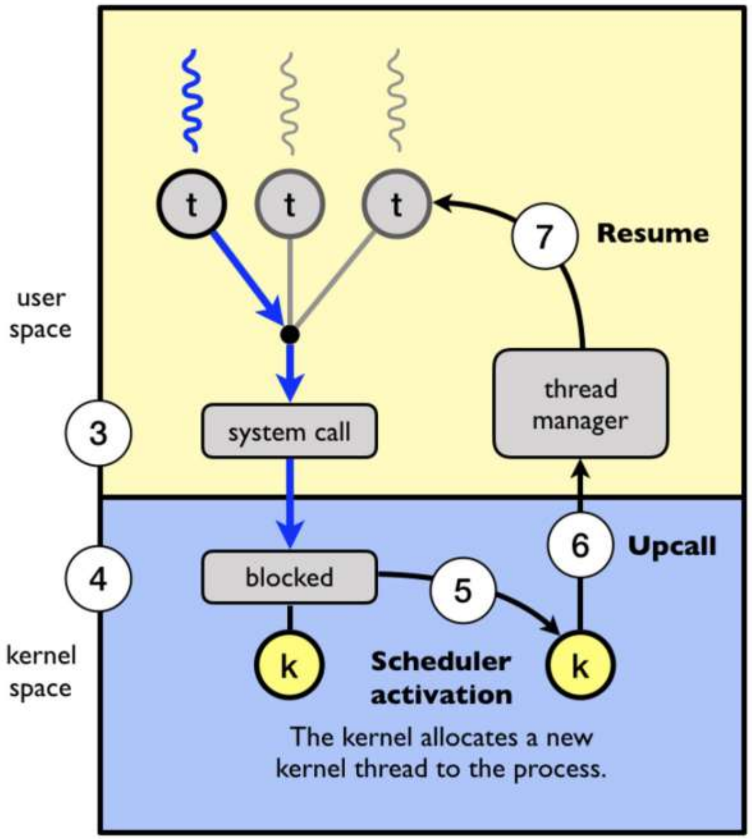
The three user level threads take turn executing on the single kernel-level thread

Scheduler Activations: Example



The executing thread makes a blocking system call (3)

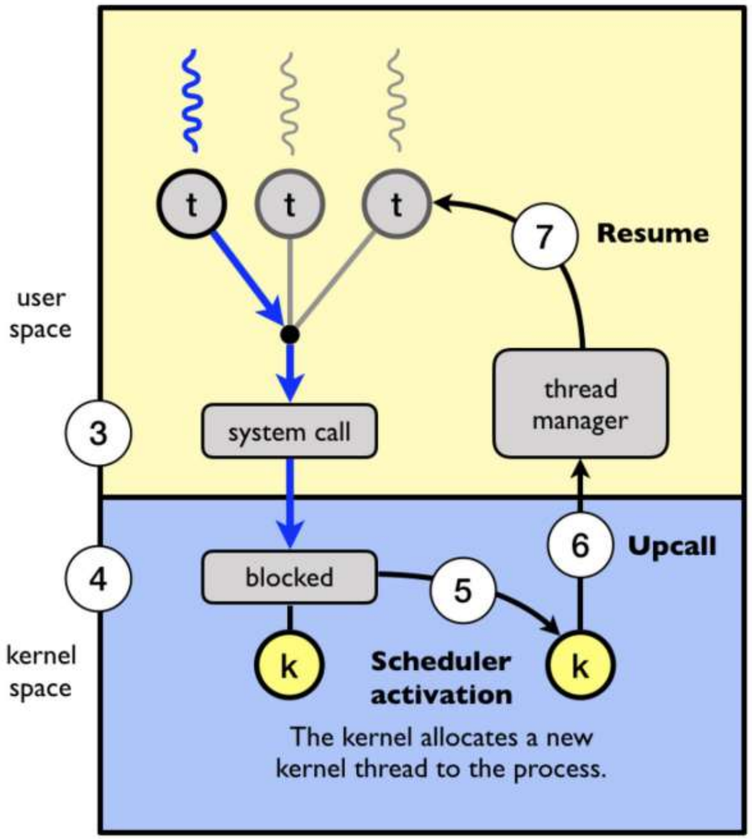
Scheduler Activations: Example



The executing thread makes a blocking system call (3)

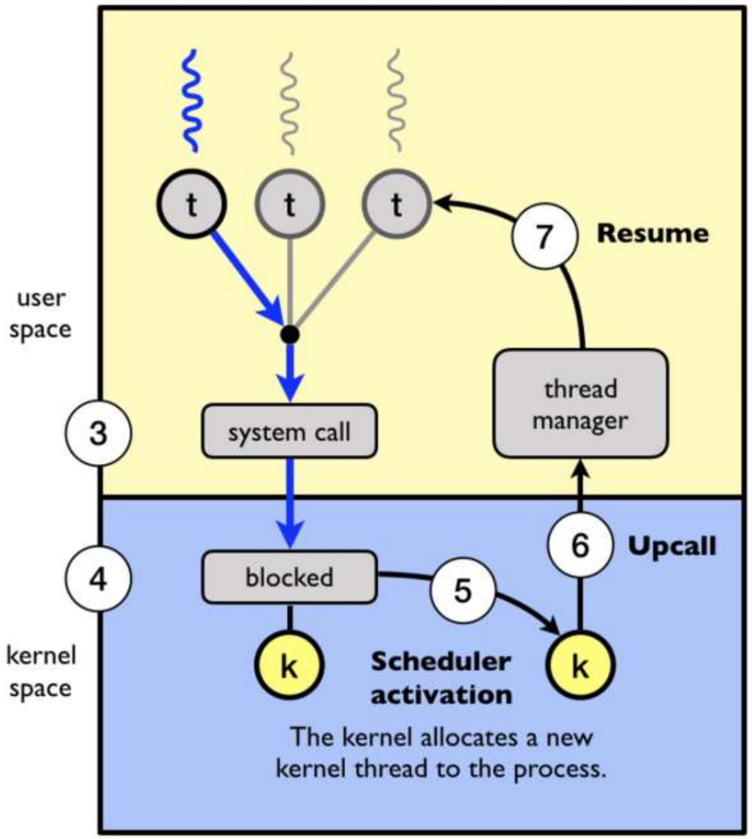
The kernel blocks the calling user-level thread and the kernel-level thread (LWP) used to execute the user-level thread (4)

Scheduler Activations: Example



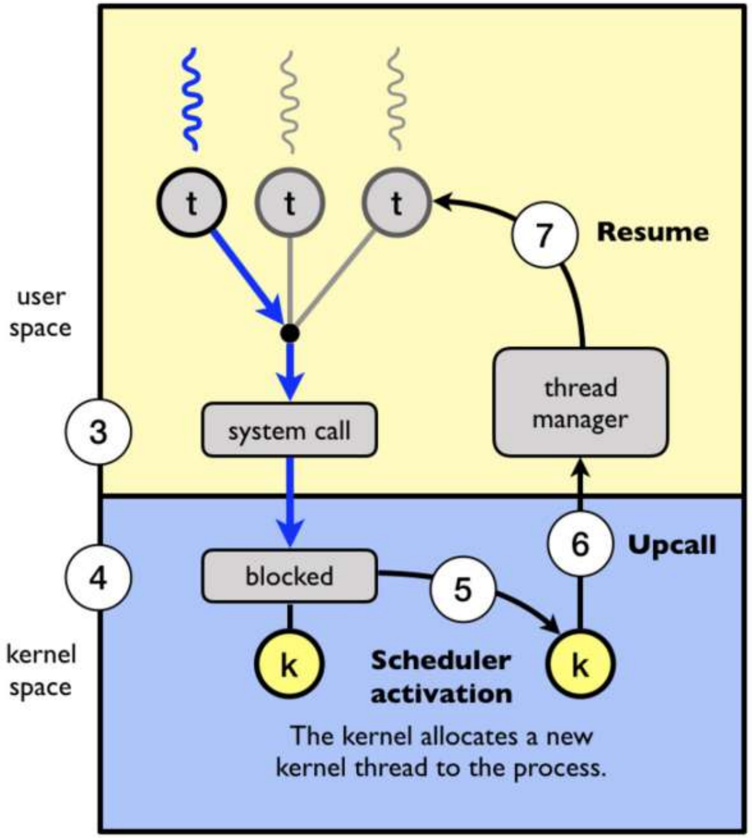
Scheduler activation: the kernel decides to allocate a new kernel-level thread to the process (5)

Scheduler Activations: Example



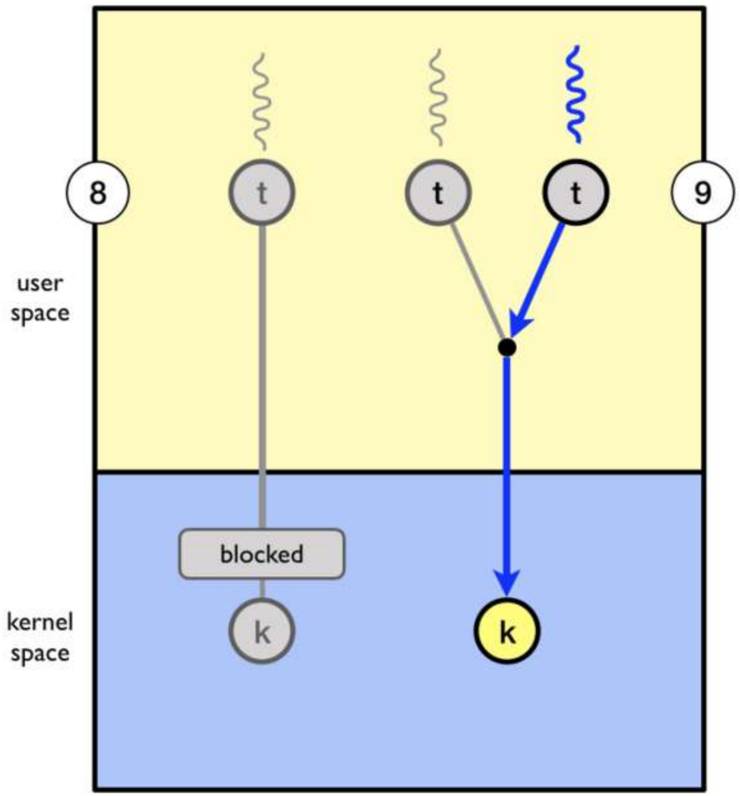
Upcall: The kernel notifies the user-level thread library which user-level thread that is now blocked and that a new kernel-level thread is available (6)

Scheduler Activations: Example



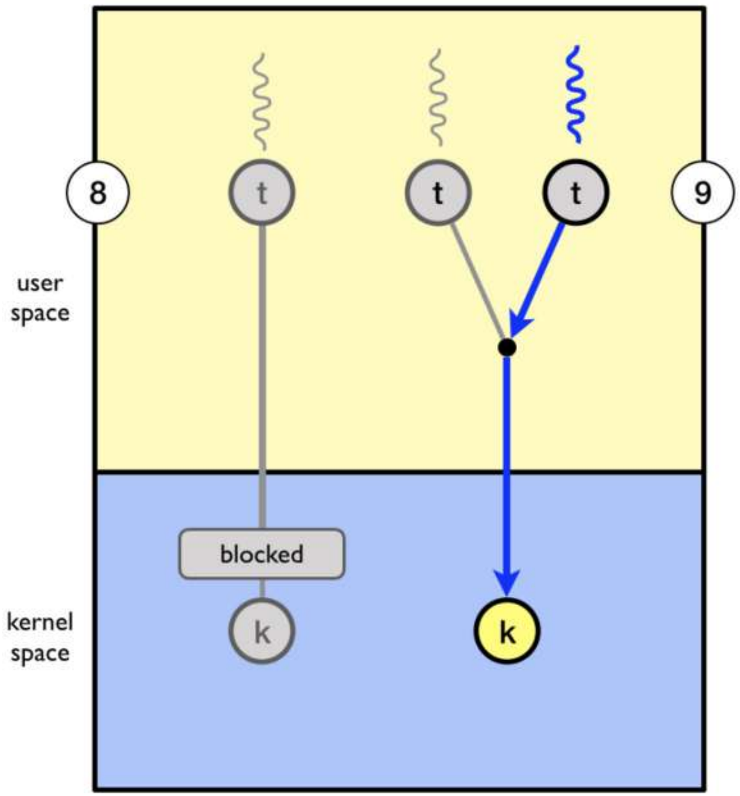
Upcall handler: The user-level thread library resumes one of the ready threads on to the new kernel thread (7)

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)

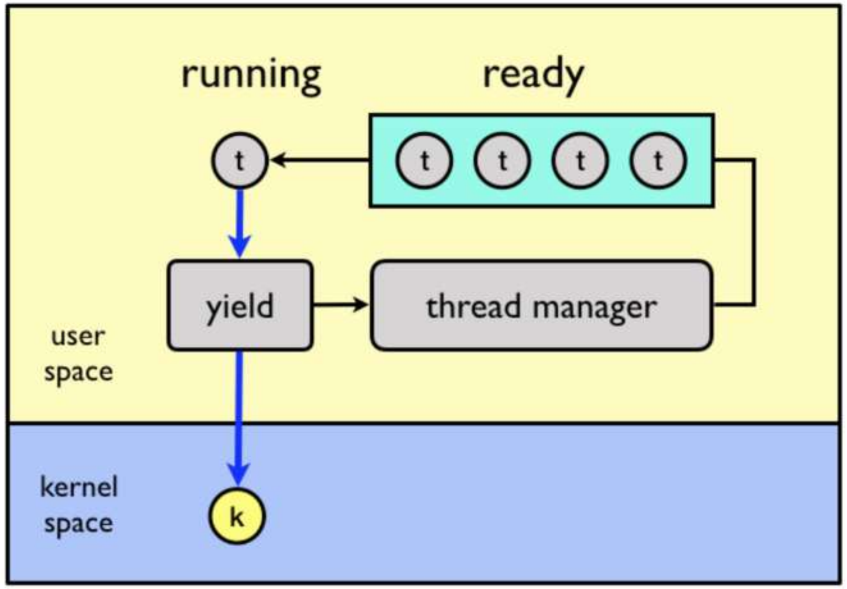
User-Level Thread Scheduling

- 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

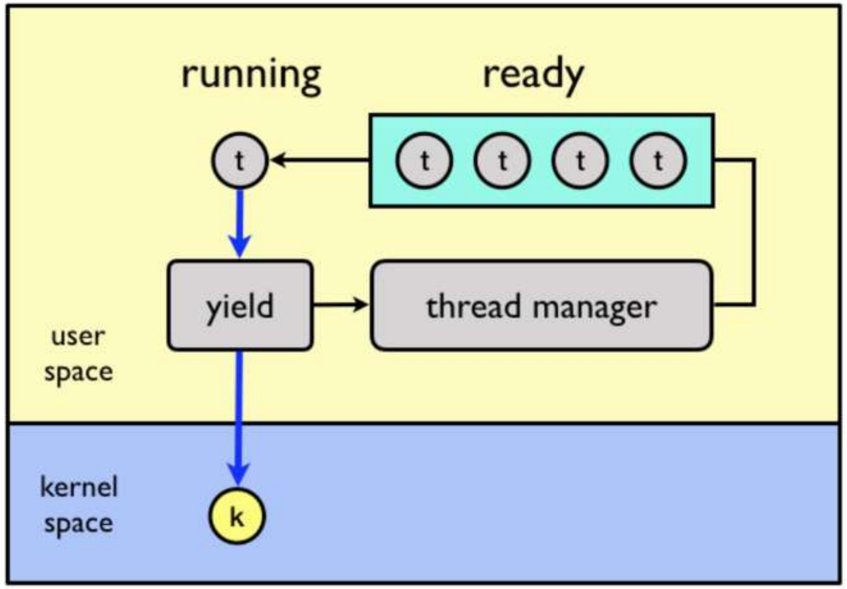
- Scheduling user-level threads on the available kernel-level threads (via LWPs)
- Implemented within the user-level thread library in user space (no kernel privileges!)
- Two main scheduling methods:
  - Cooperative
  - Preemptive

Cooperative Thread Scheduling



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

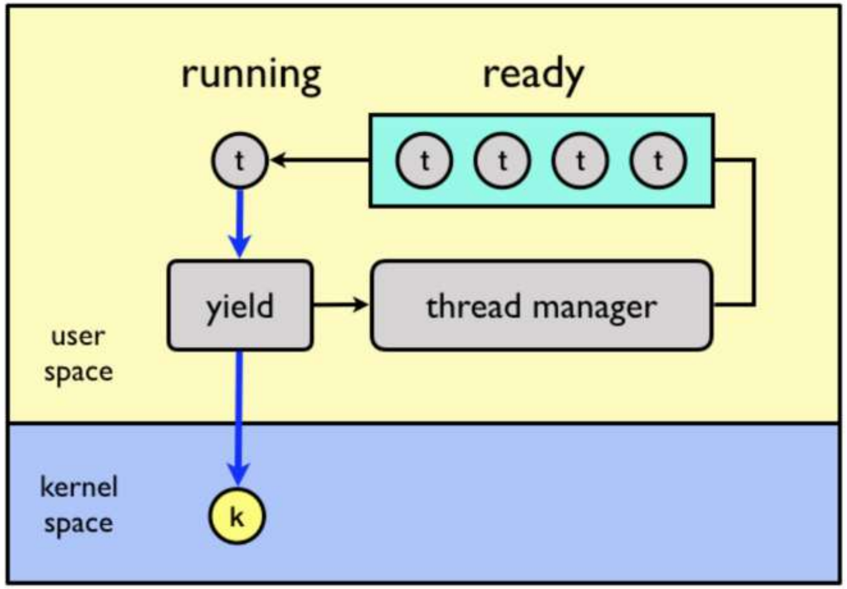
Cooperative Thread Scheduling



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

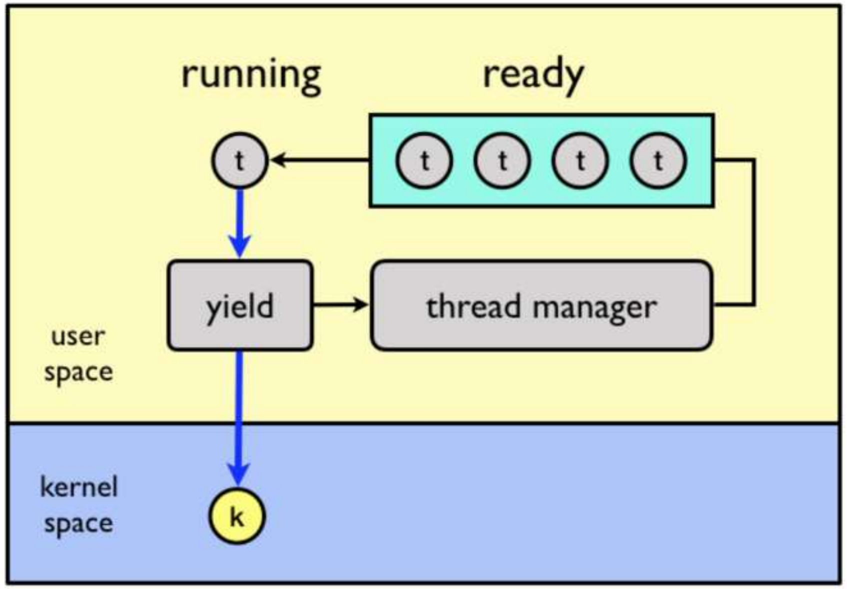
Cooperative Thread Scheduling



Threads yield to each other, either

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

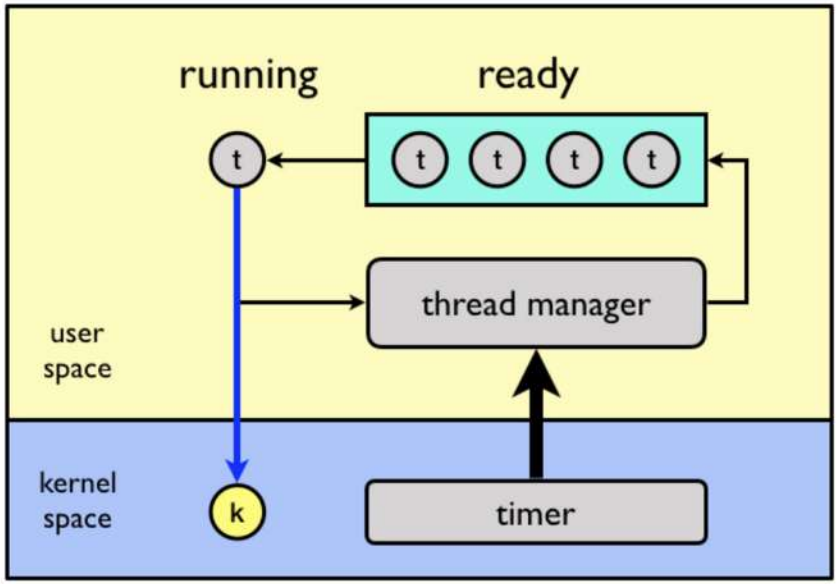
Cooperative Thread Scheduling



Threads yield to each other, either

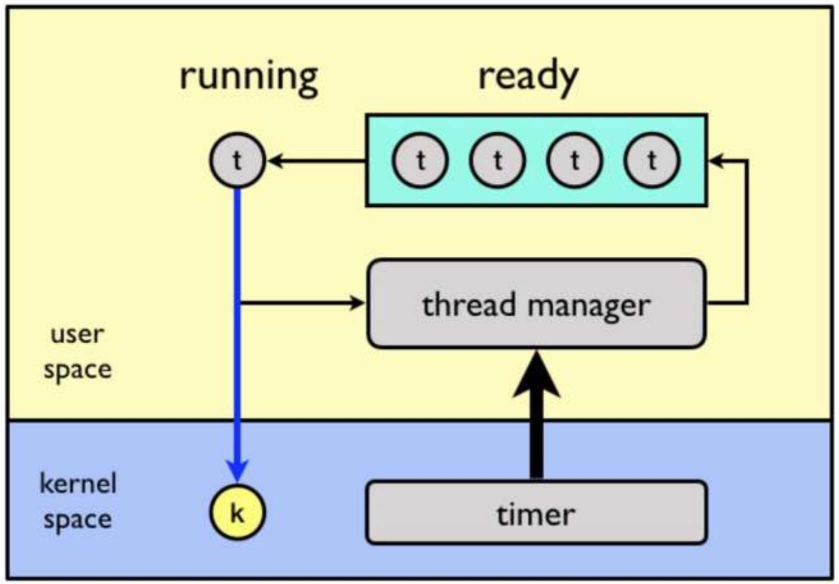
- explicitly (e.g., by calling a yield() provided by the user-level thread library) or
- implicitly (e.g., requesting a lock held by another thread)

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

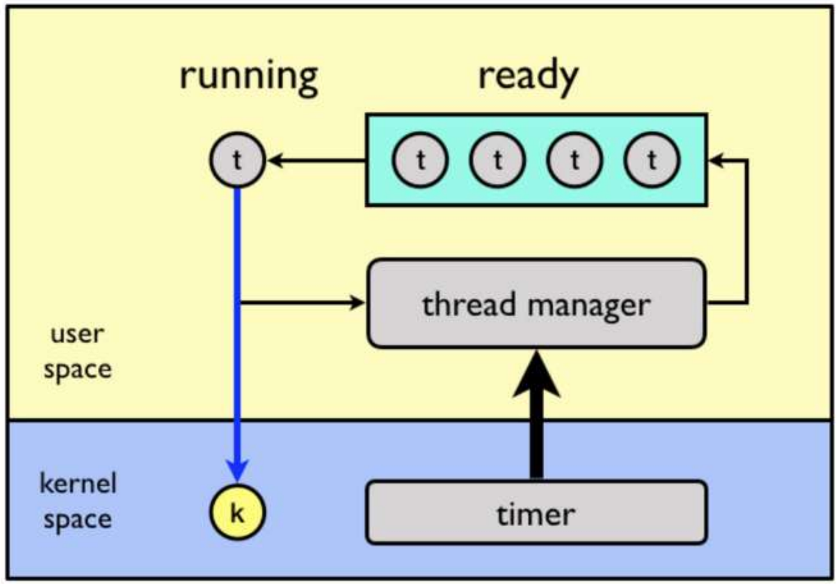
Preemptive Thread Scheduling



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

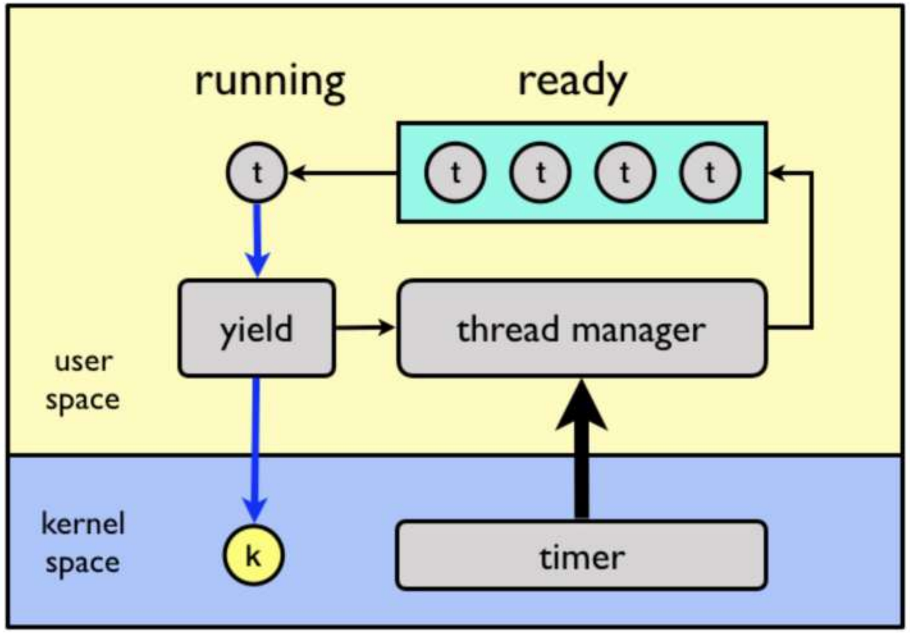


Similar to multitasking (a.k.a. time sharing), where a timer is set to cause an interrupt at a regular time interval

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

Summary

- A thread is a single execution stream within a process
- User- vs. Kernel-level threads
- Mapping user- to kernel-level threads
- Example of user-level thread library: Java Threads

Summary

- A thread is a single execution stream within a process
- User- vs. Kernel-level threads
- Mapping user- to kernel-level threads
- Example of user-level thread library: Java Threads
- Scheduling user-level threads vs. kernel-level threads