Systems and Networking – Unit I

B.Sc. in Applied Computer Science and Artificial Intelligence

2022-2023

Gabriele Tolomei

Department of Computer Science

Sapienza Università di Roma

tolomei@di.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

Today: Beyond Single-Threaded Processes

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

Today: 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

Today: 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

Today: 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, and 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, and 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, and 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)

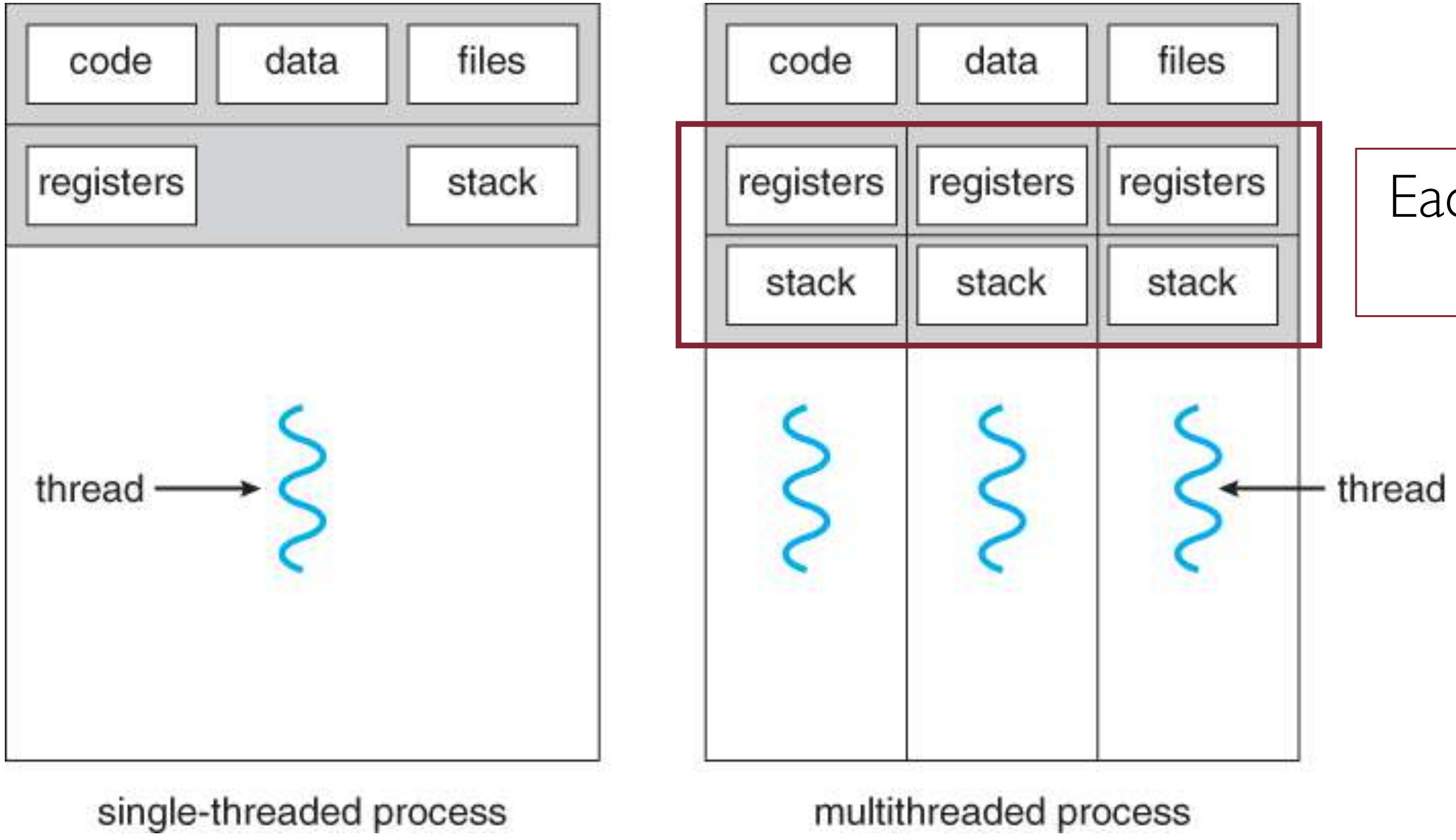
Process vs. Thread

- 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

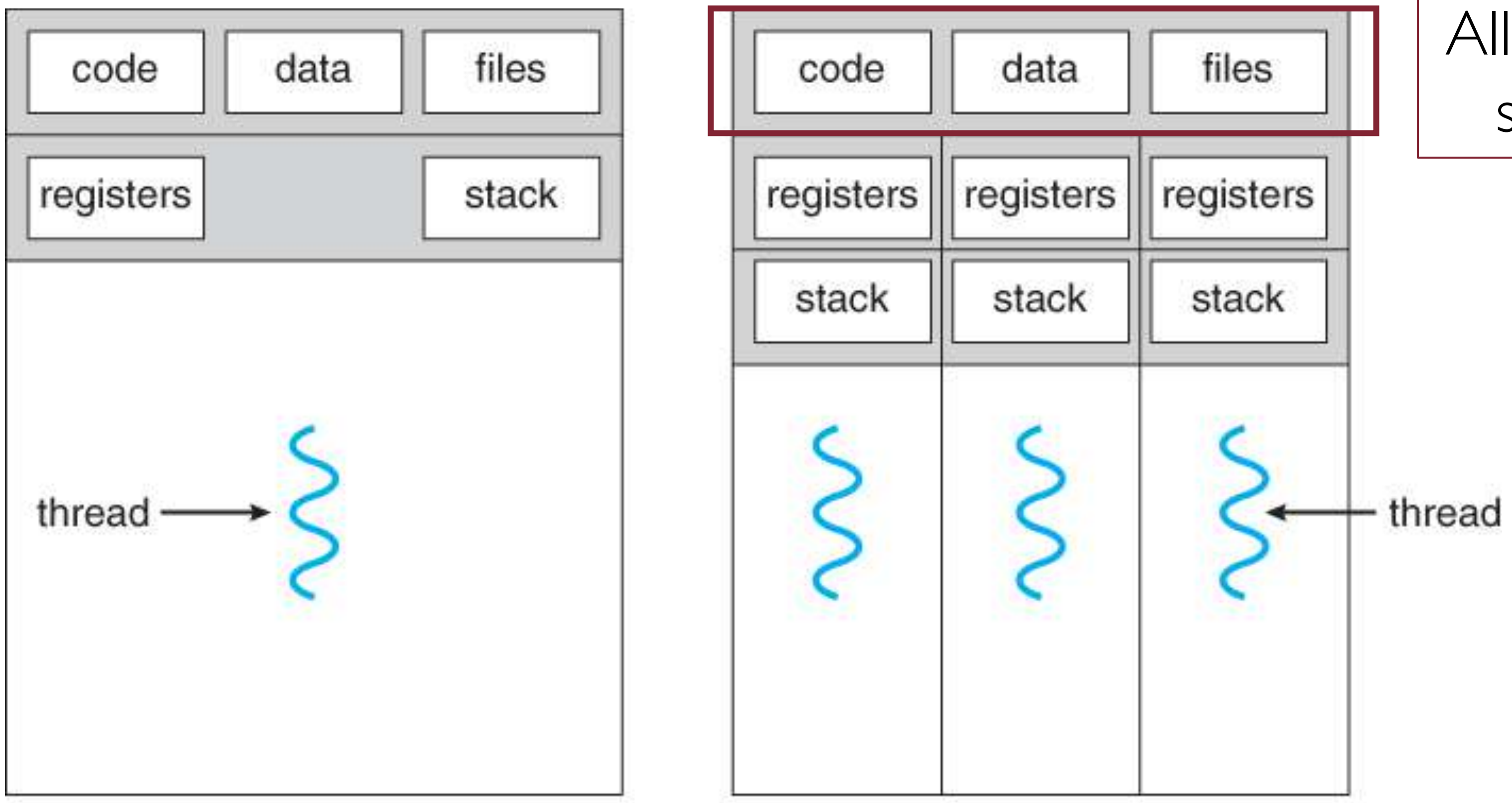
- A process defines the address space, text (code), data, resources, etc.
- A thread defines a single sequential execution stream *within* a process (i.e., program counter, stack, registers)
- A thread is bound to a specific process
- Each process may have several threads of control within it
  - The 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

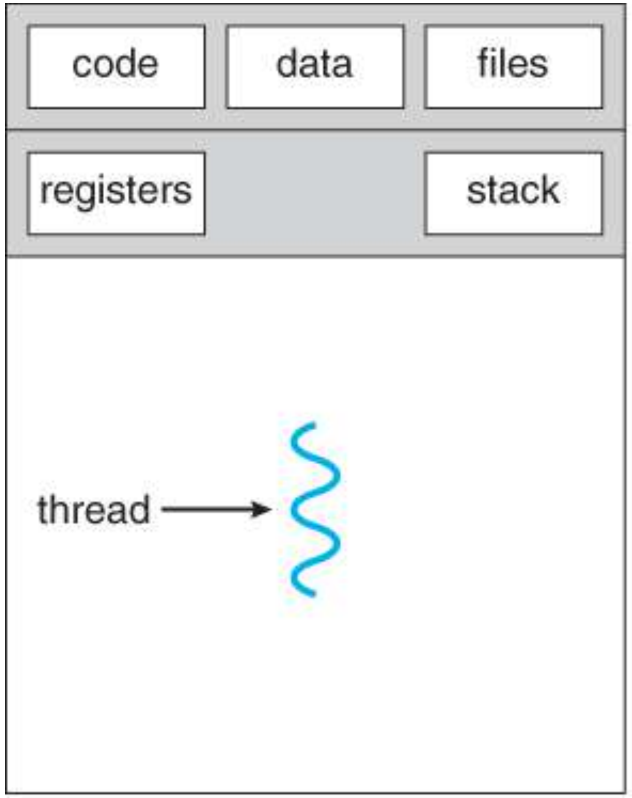


single-threaded process

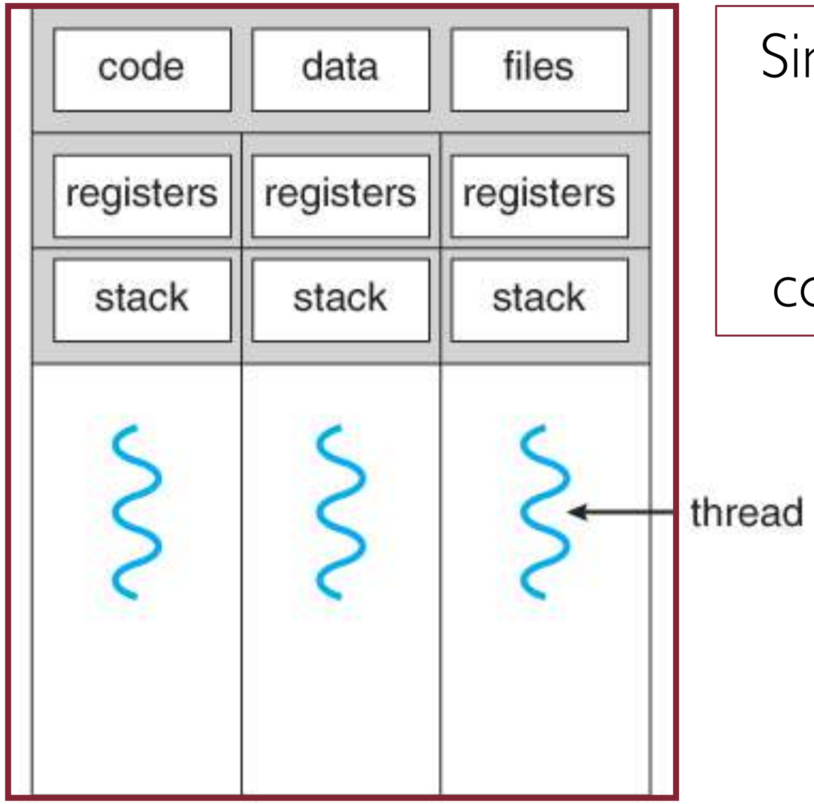
multithreaded process

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 of the others

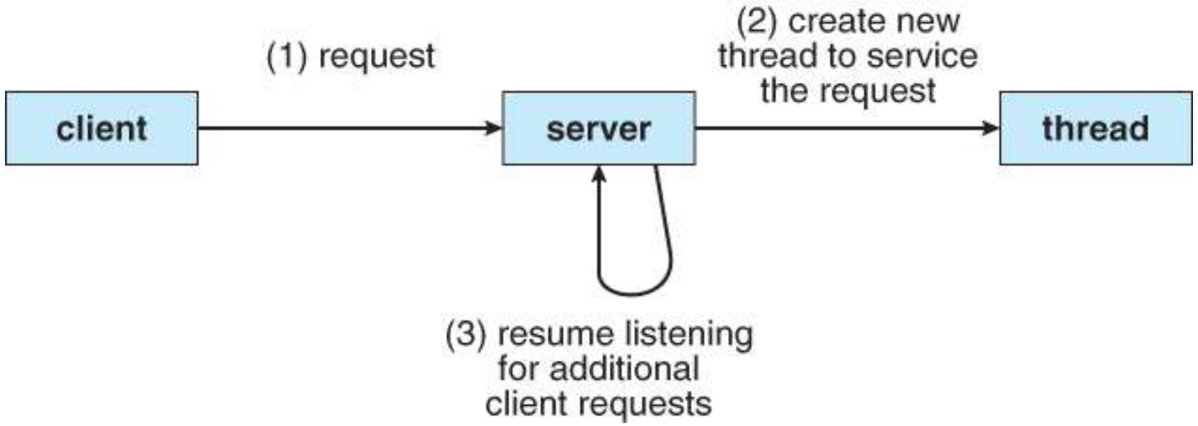
Threads: Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others
- This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking

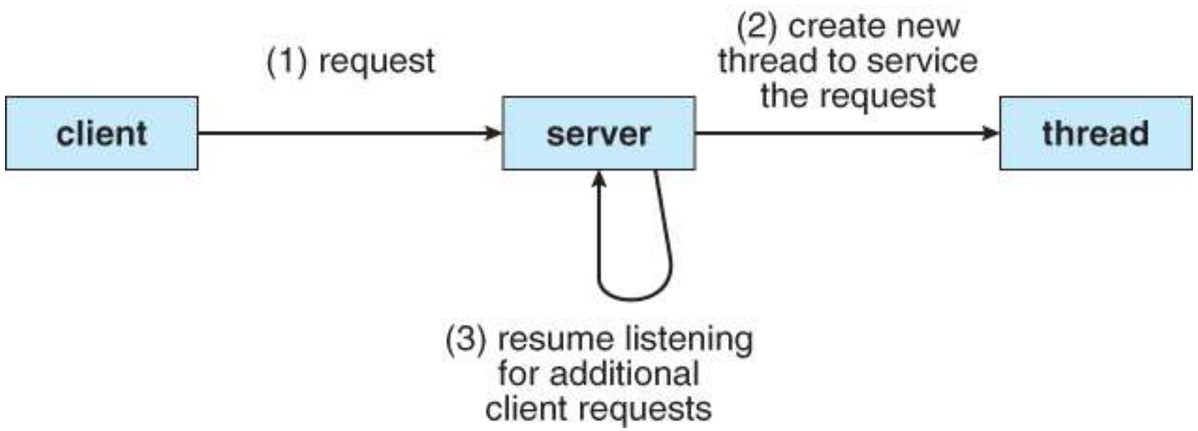
Threads: Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others
- This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking
- Example: word processor
  - a thread may check spelling and grammar while another thread handles user input (keystrokes), and a third does periodic backups of the file being edited

Multi-threaded Web Server

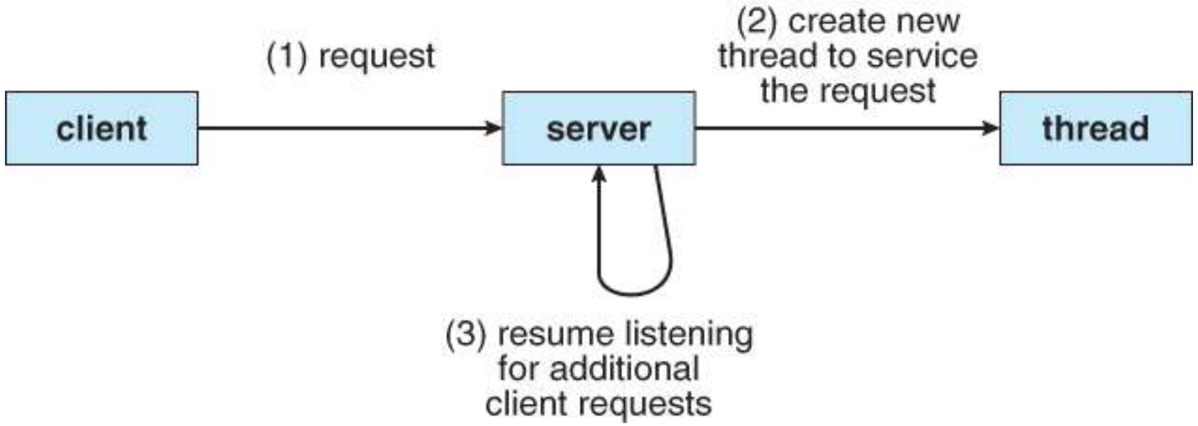


Multi-threaded Web Server



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:
  - 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

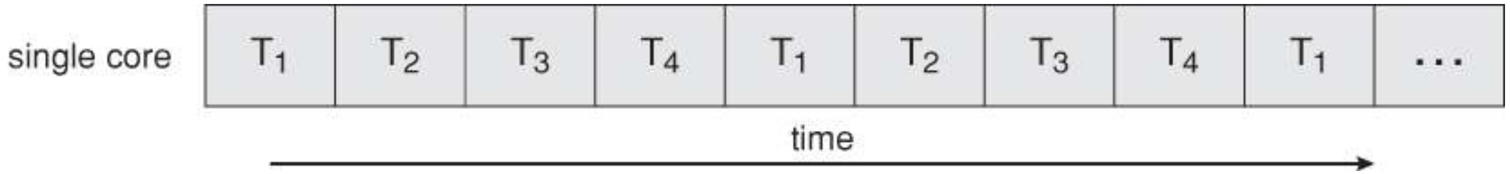
Threads: Benefits

- 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

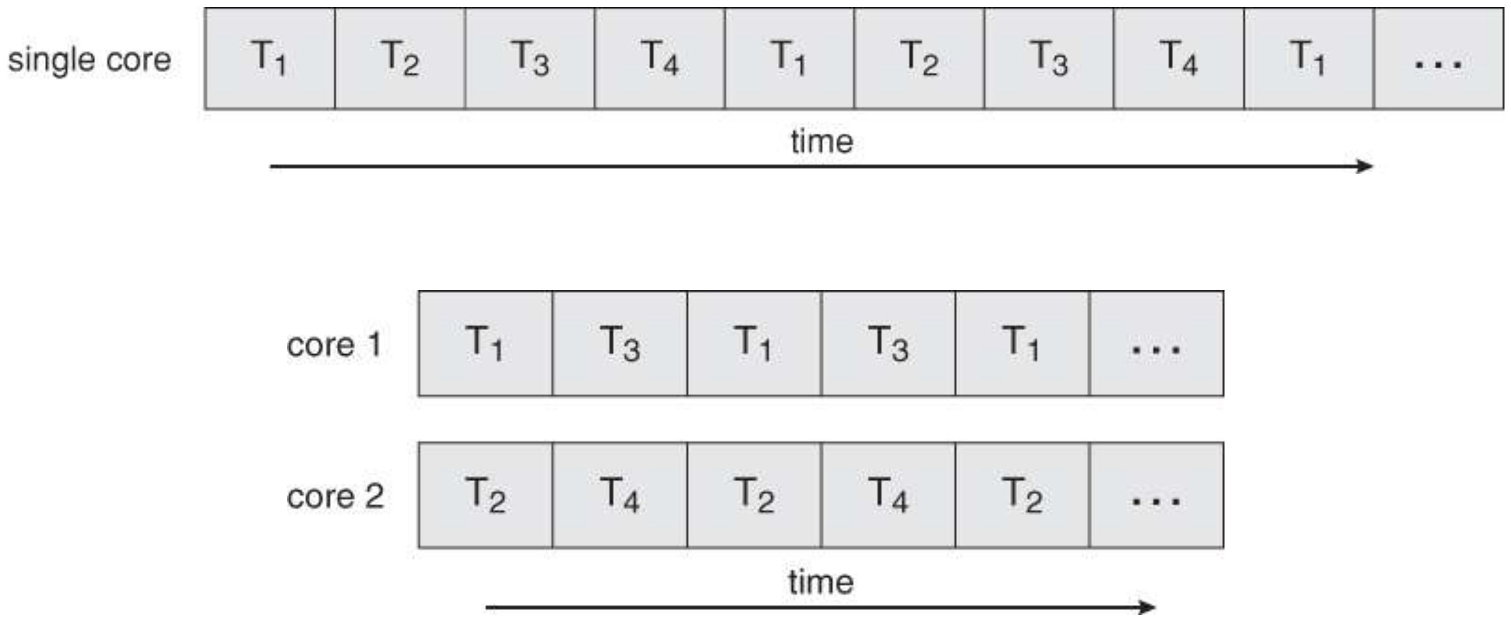
Multi-core Programming

- 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!

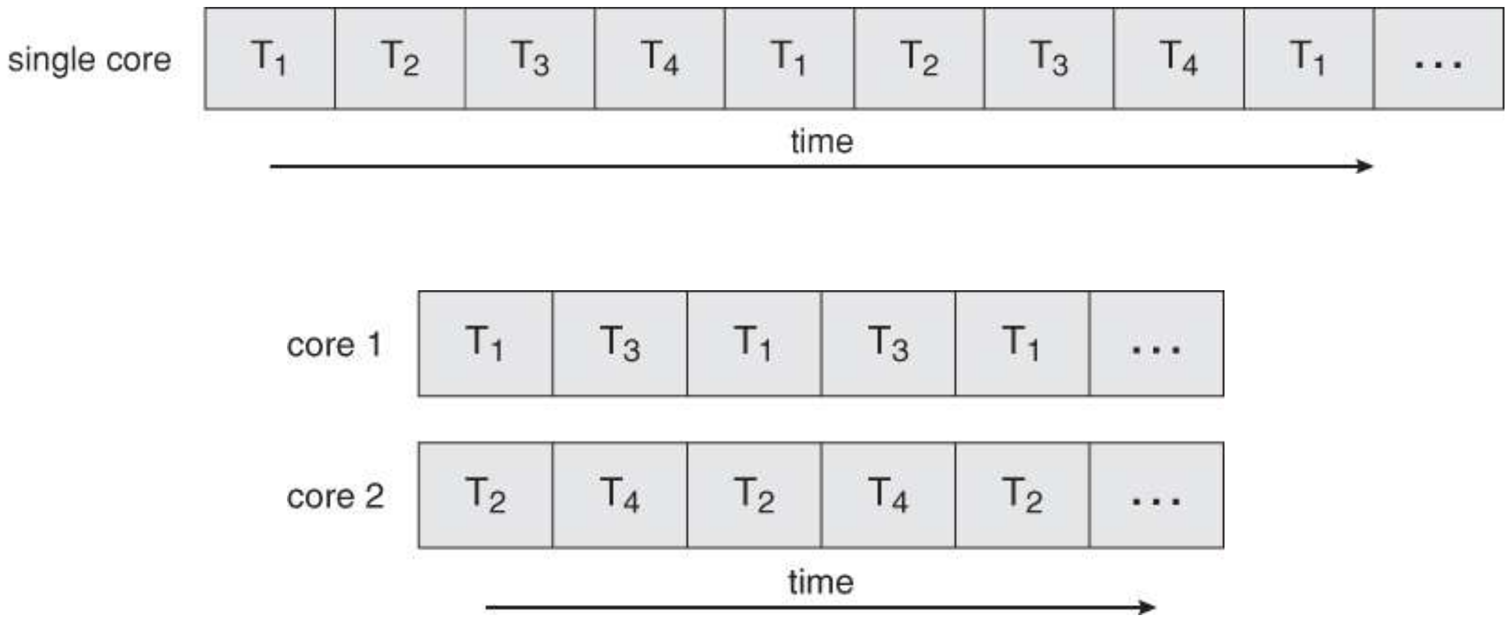
Single- vs. Multi-core Programming



Single- vs. Multi-core Programming

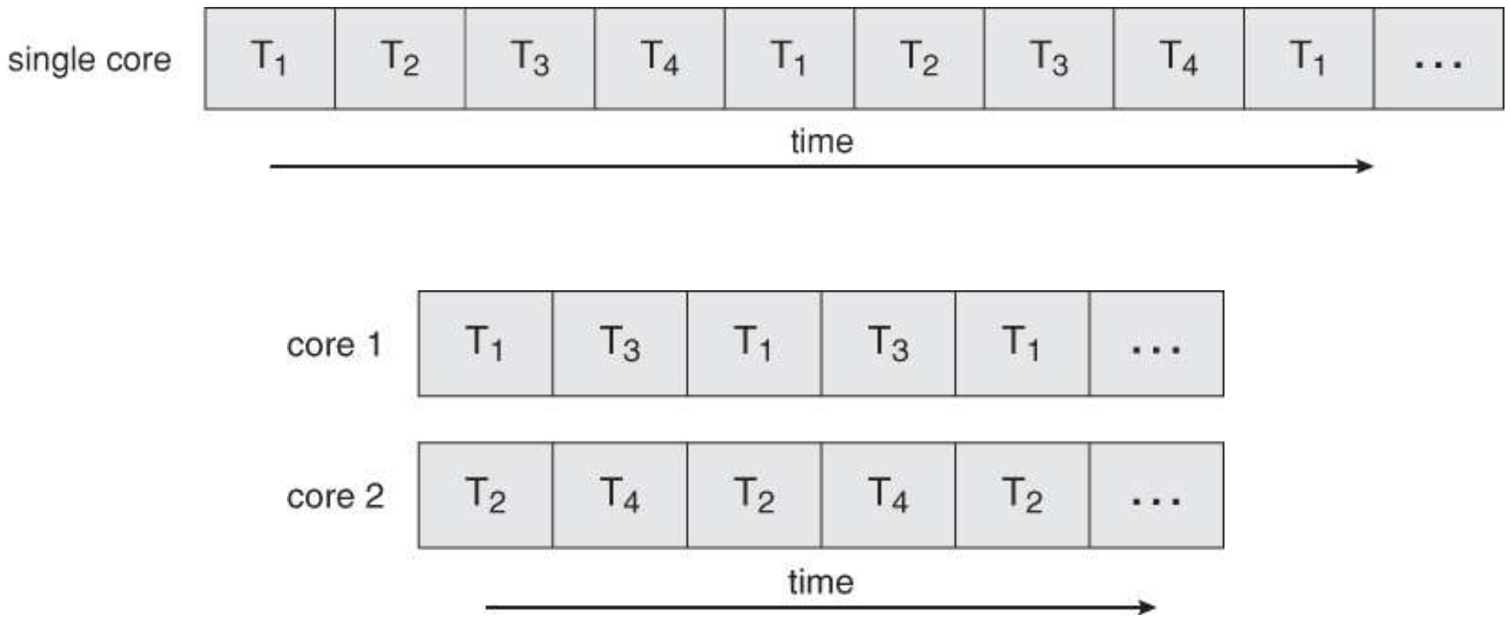


Single- vs. Multi-core Programming



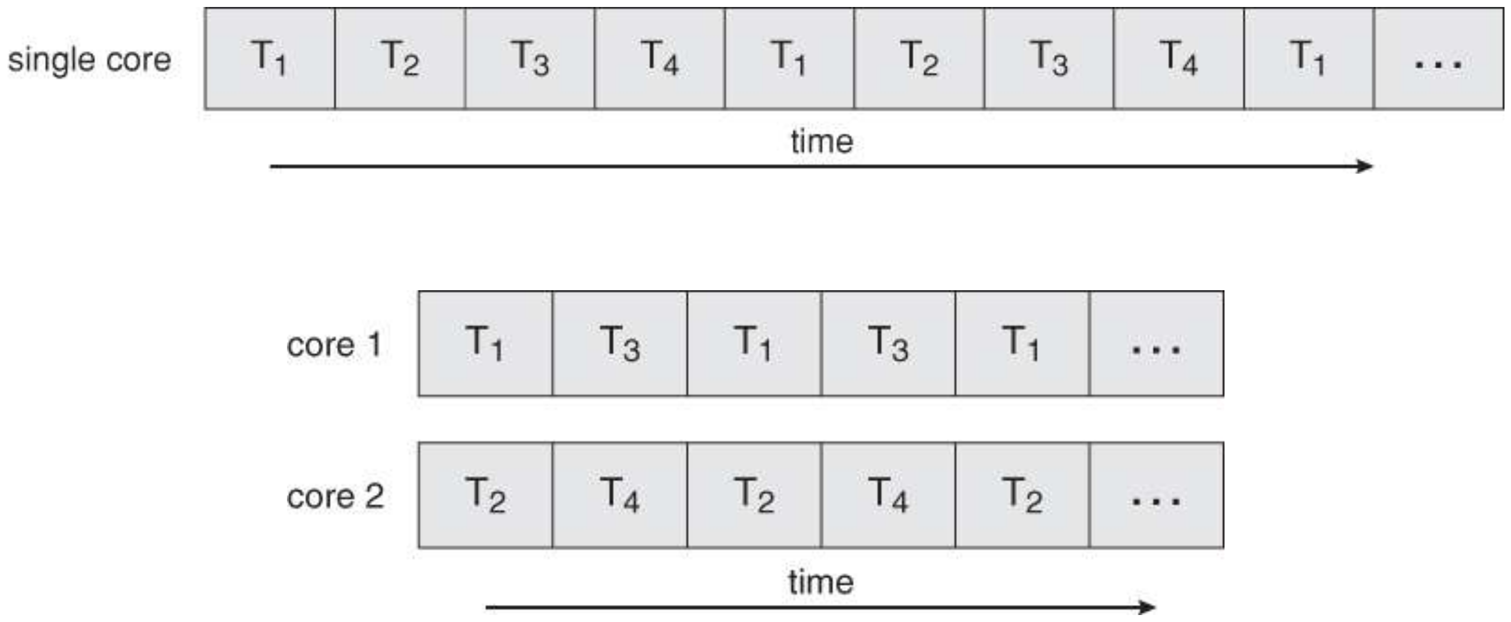
Multi-core chips require new OS scheduling algorithms to make better use of the multiple cores available

Single- vs. Multi-core Programming



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

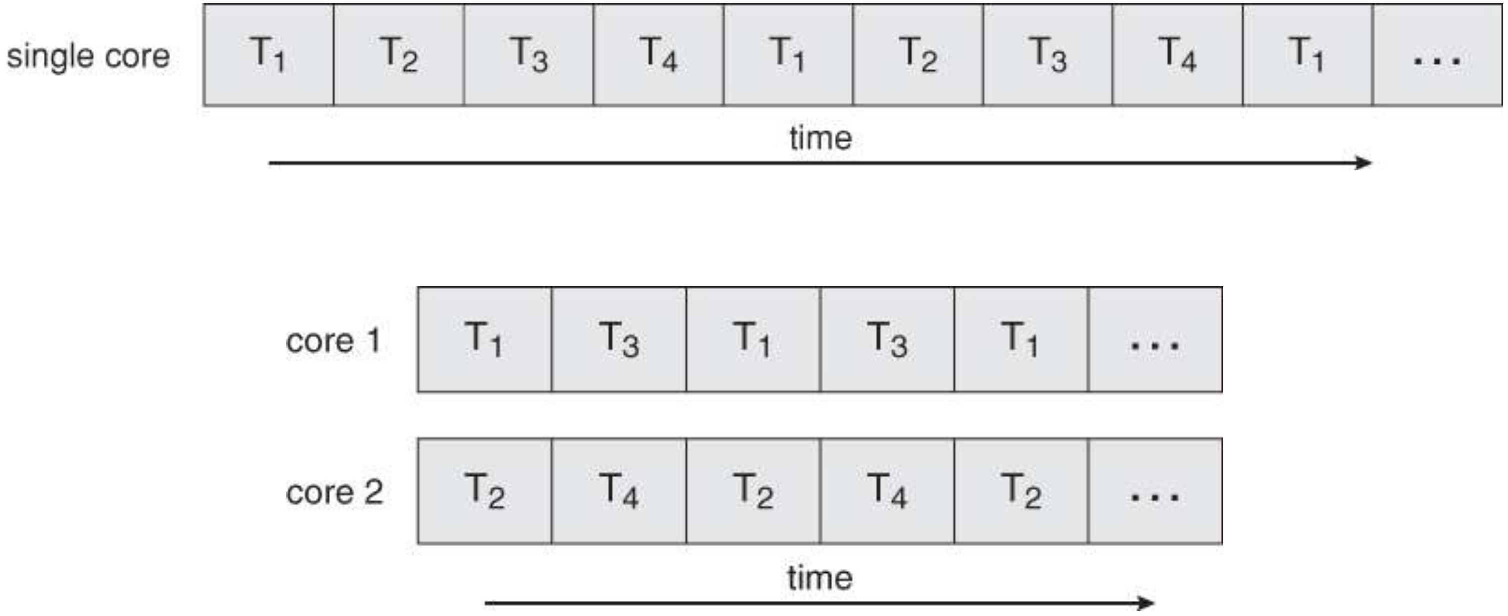
Single- vs. Multi-core Programming



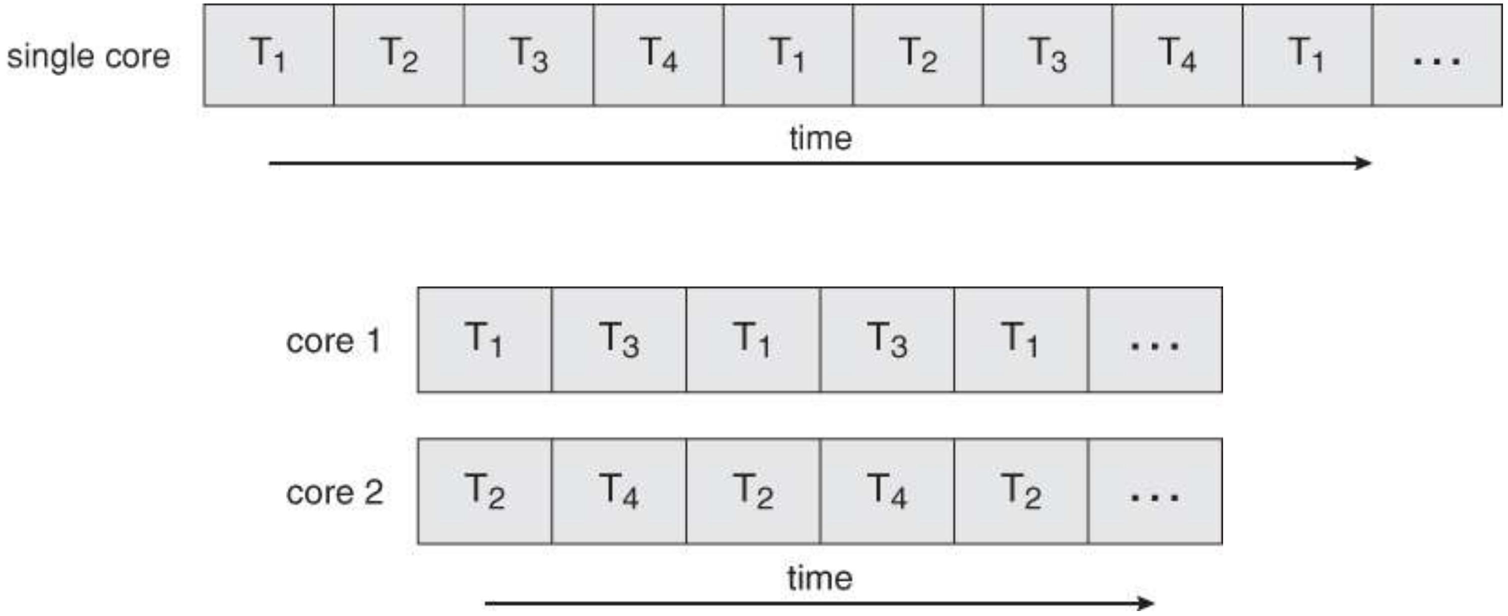
Hyper-threading

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

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
- In practice, no program is ever divided up solely by one or the other of these, but instead by some sort of hybrid combination

Classifying OSs

 address space

 thread

Classifying OSs

single thread

address space

thread

multiple threads

Classifying OSs

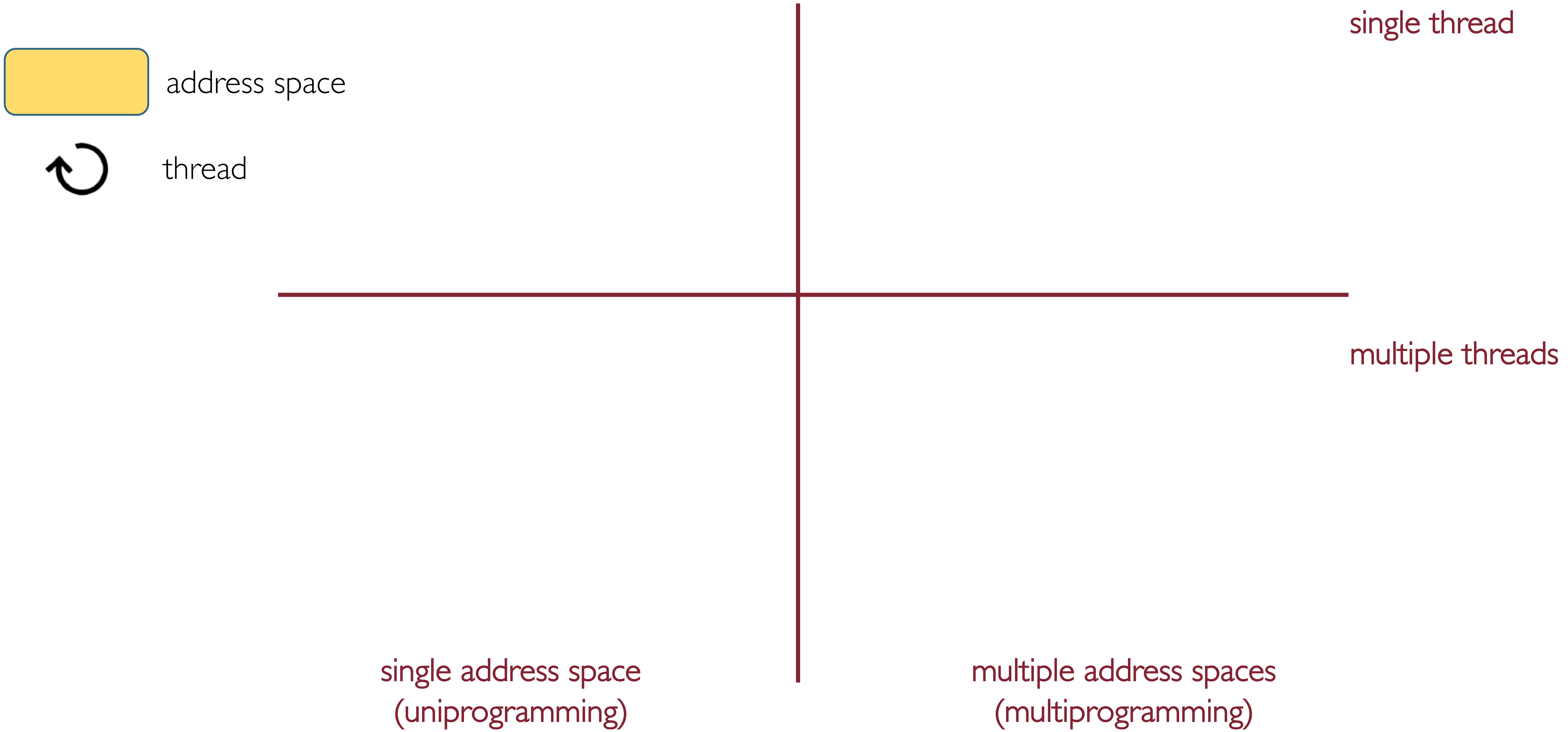
address space

thread

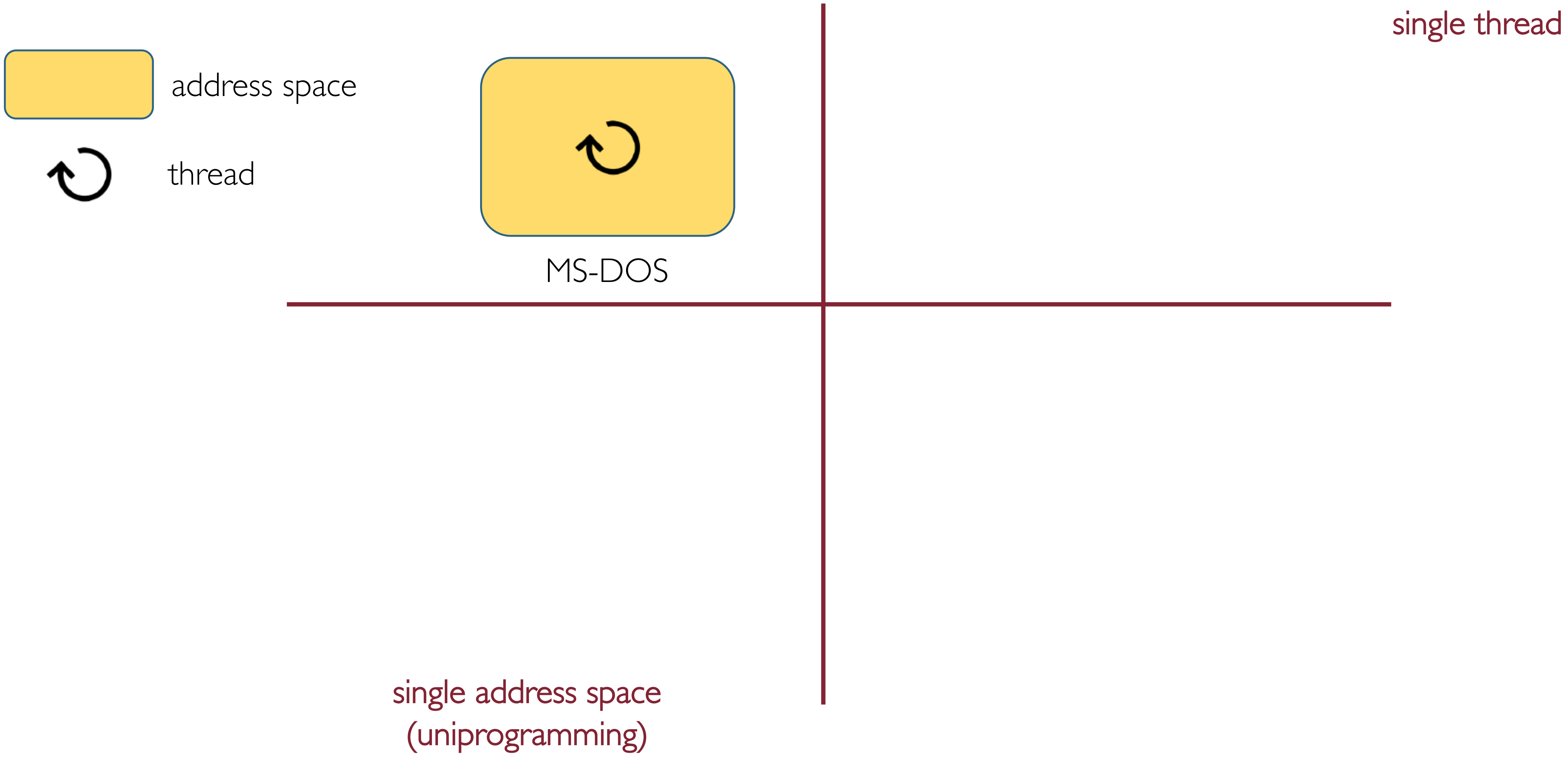
single address space
(uniprogramming)

multiple address spaces
(multiprogramming)

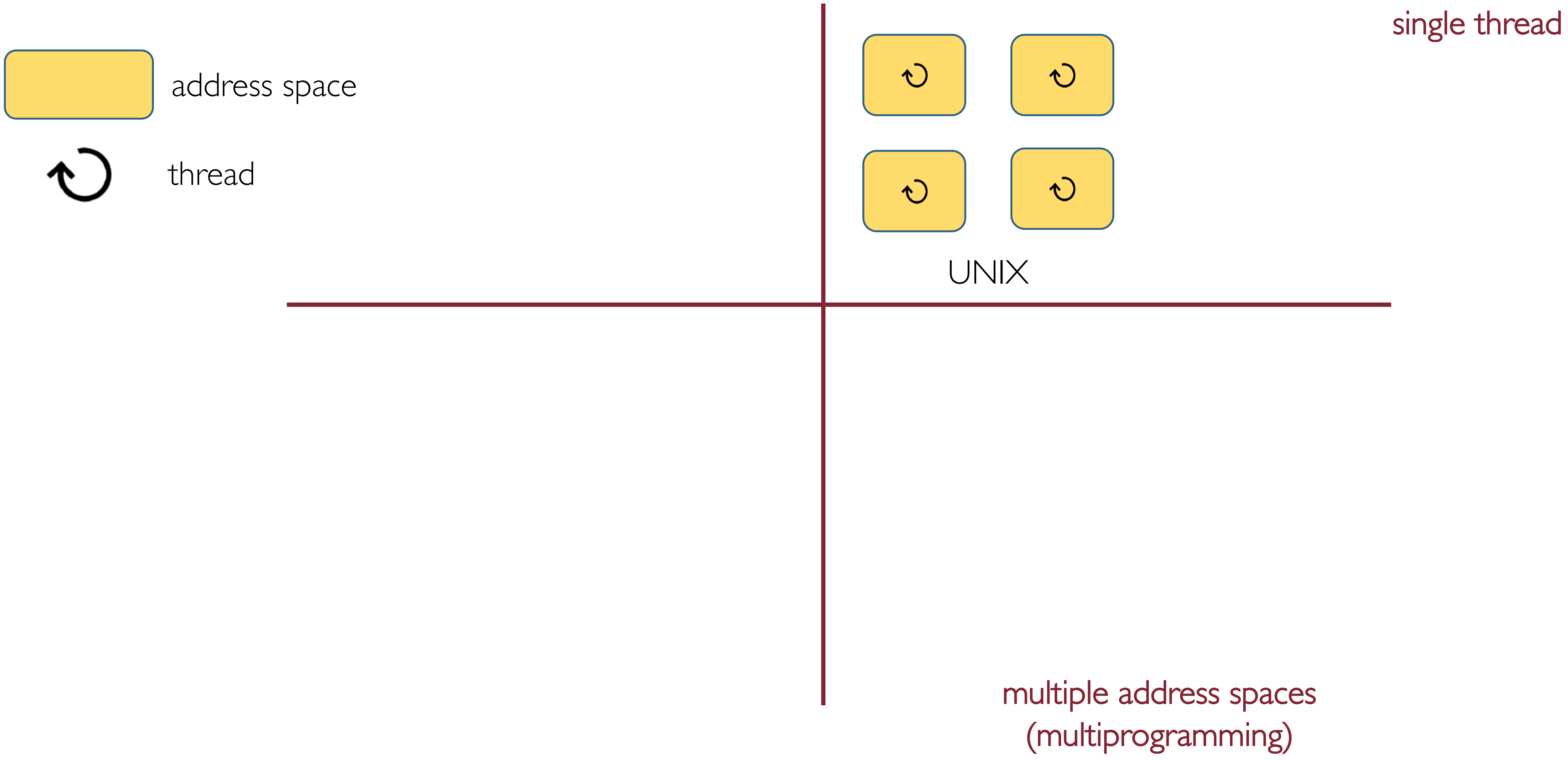
Classifying OSs



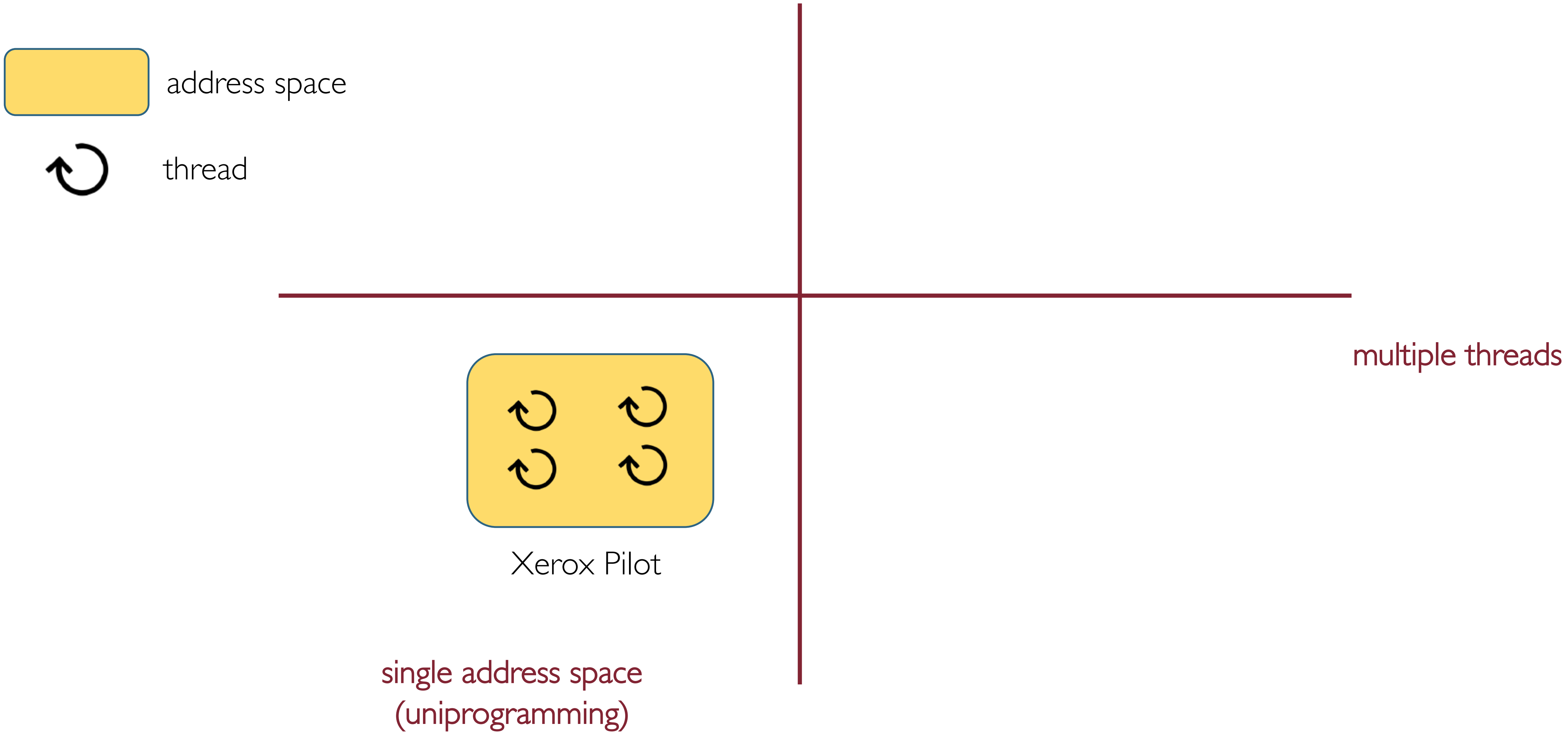
Classifying OSs



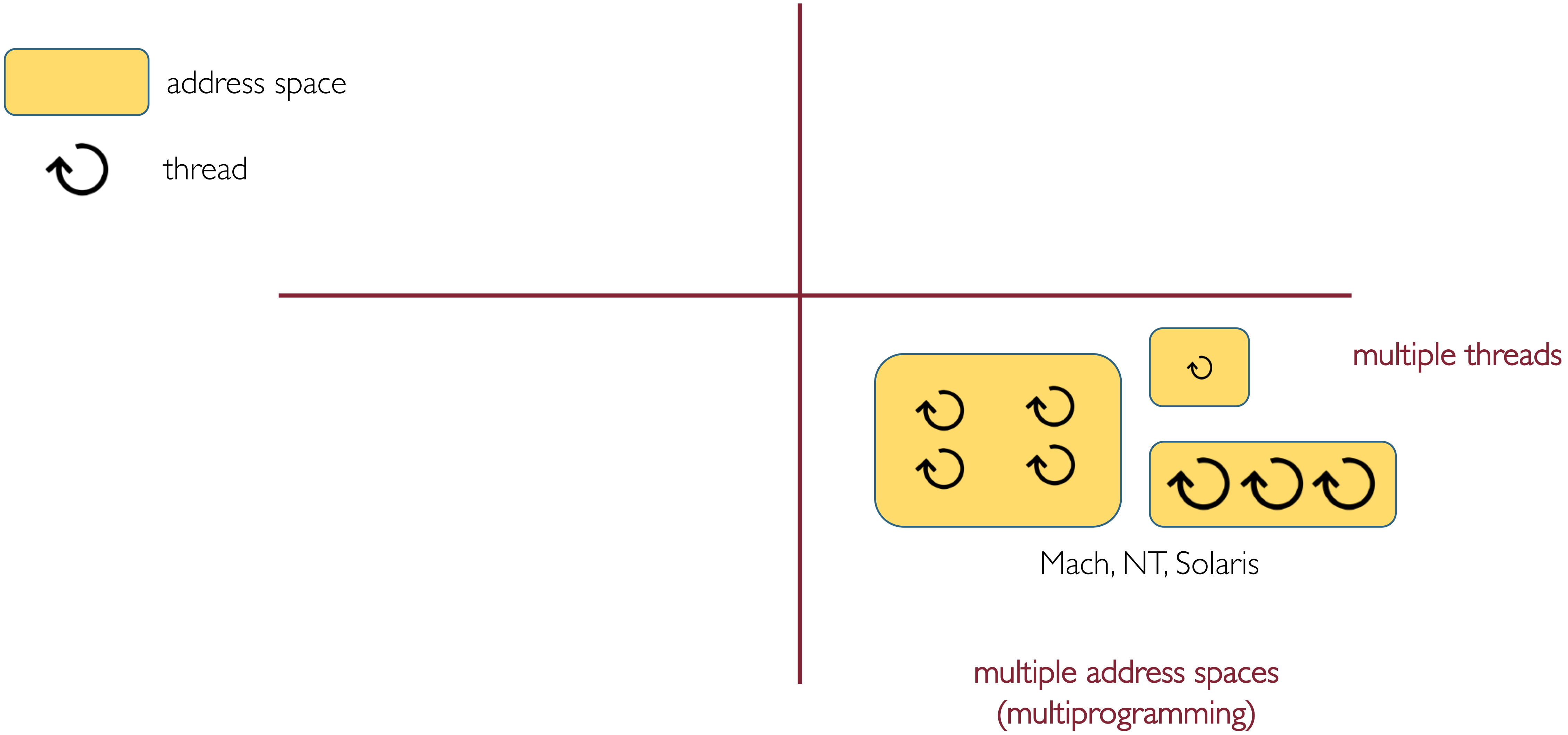
Classifying OSs



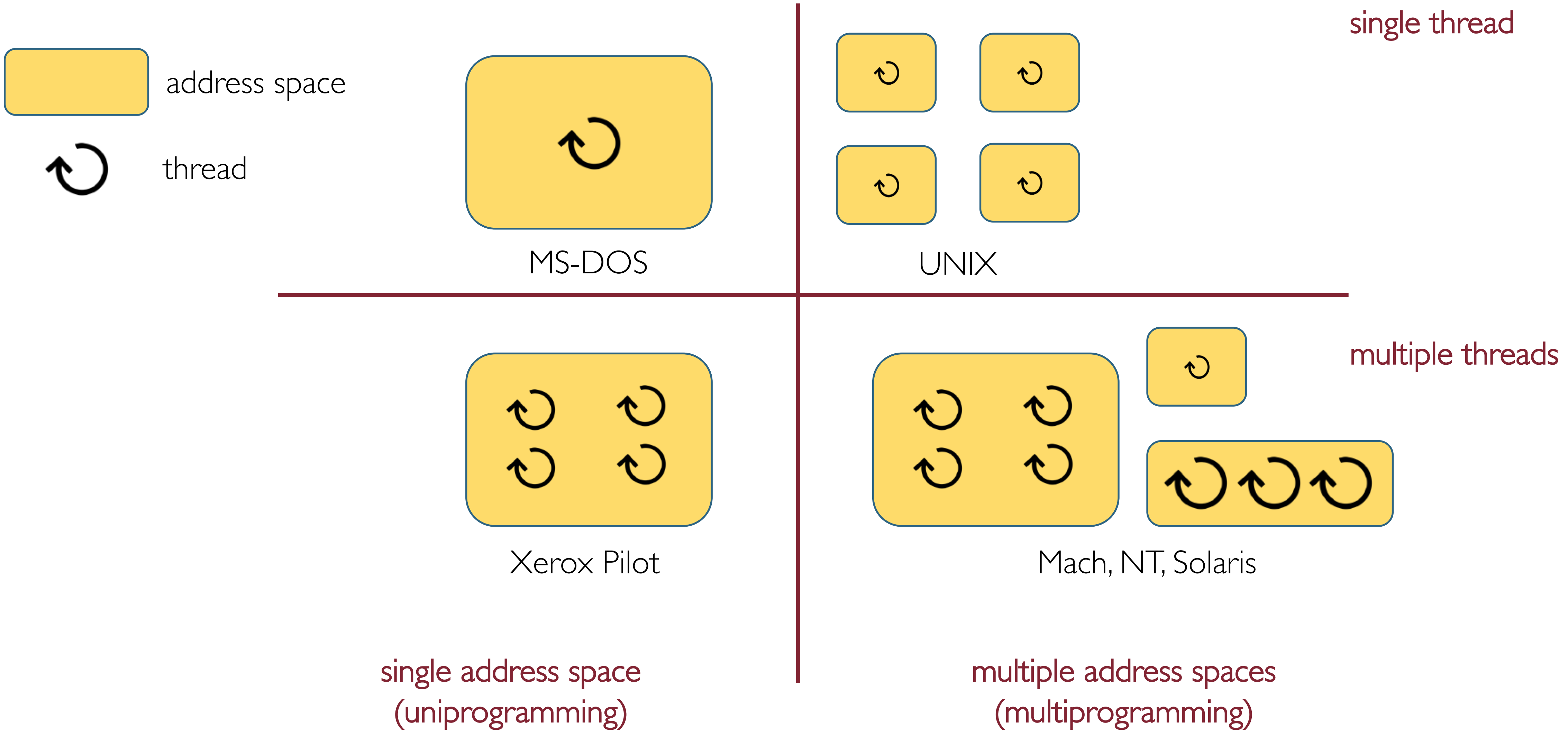
Classifying OSs



Classifying OSs



Classifying OSs



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

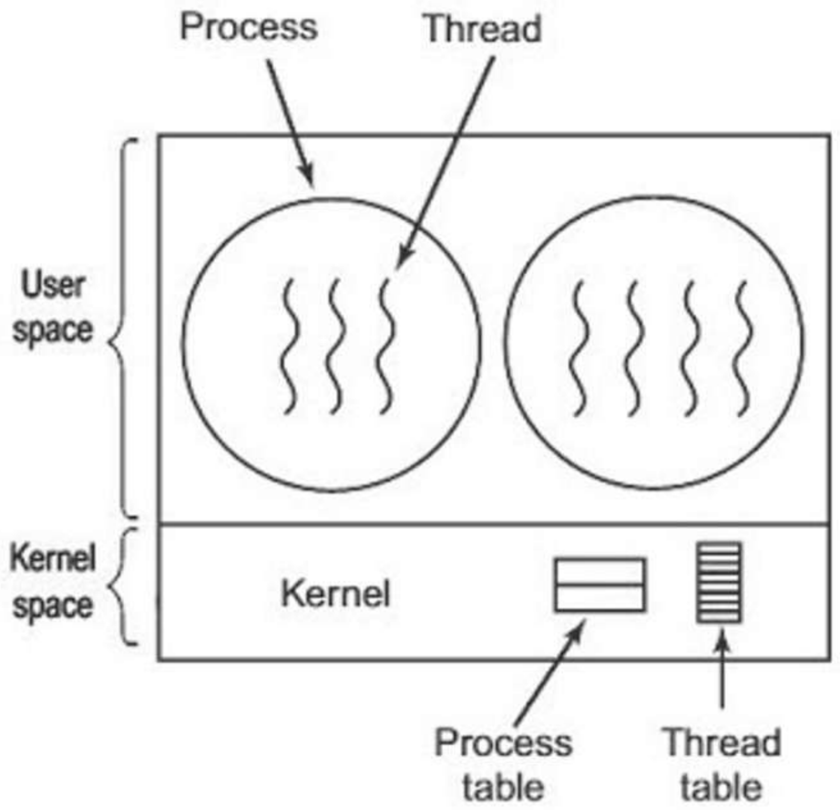
Kernel Threads

- 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

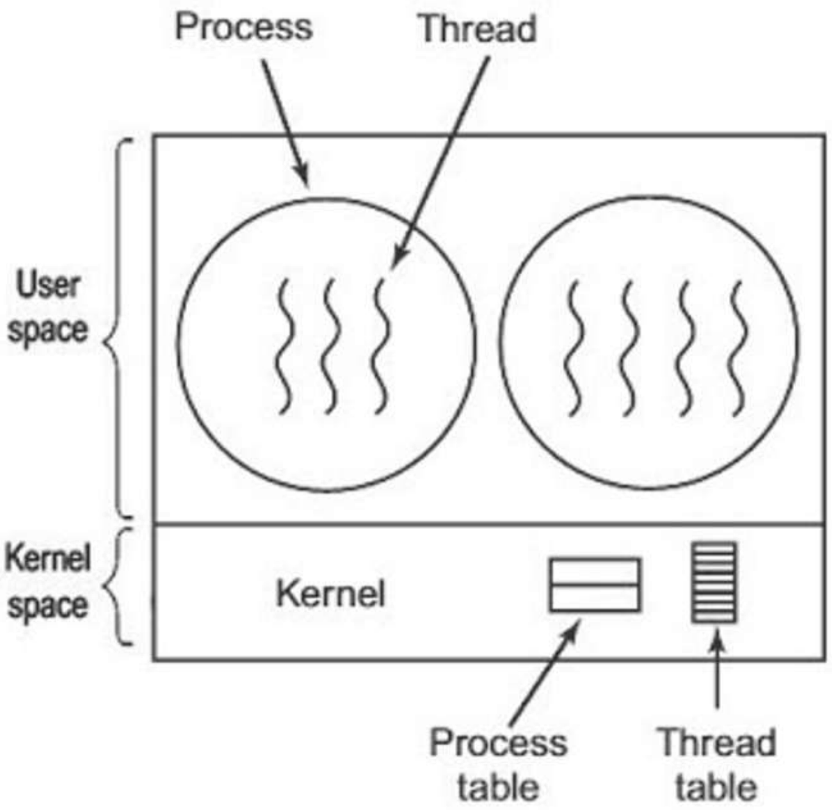
- The smallest unit of execution that can be scheduled by the OS
- The OS is responsible for supporting and managing all threads
- One Process Control Block (PCB) for each process, one Thread Control Block (TCB) for each thread
- The OS usually provides system calls to create and manage threads from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

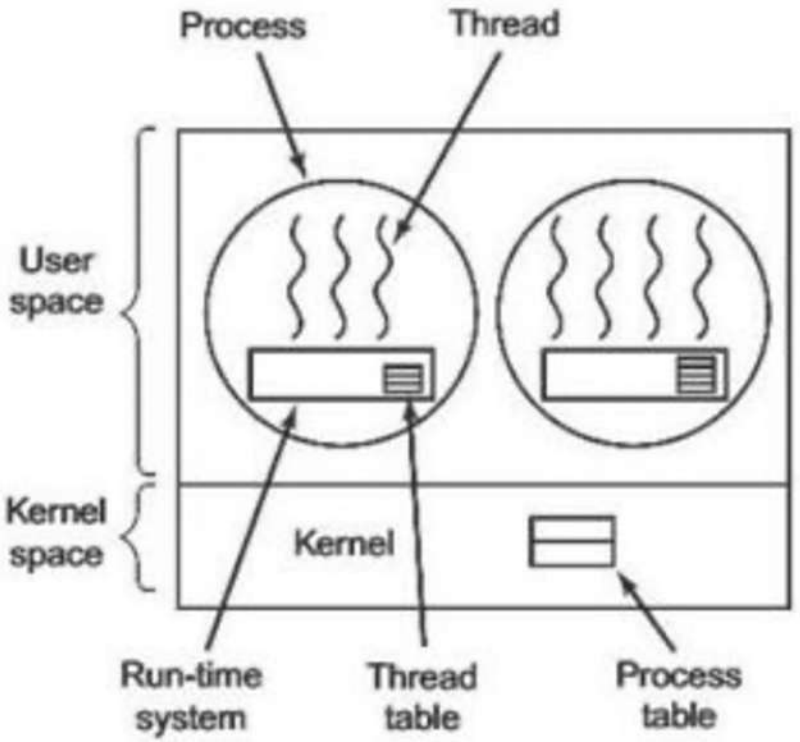
User Threads

- 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

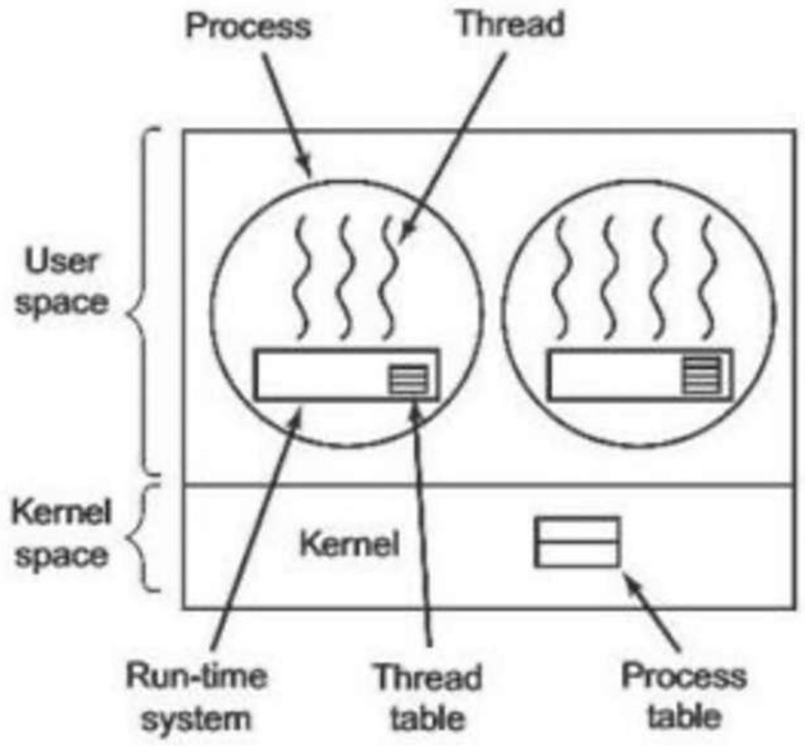
- Managed entirely by the run-time system (user-level thread library)
- The OS kernel knows nothing about user-level threads
- The OS kernel manages user-level threads as if they where single-threaded processes
- Ideally, thread operations should be as fast as a function call

User Threads: PROs



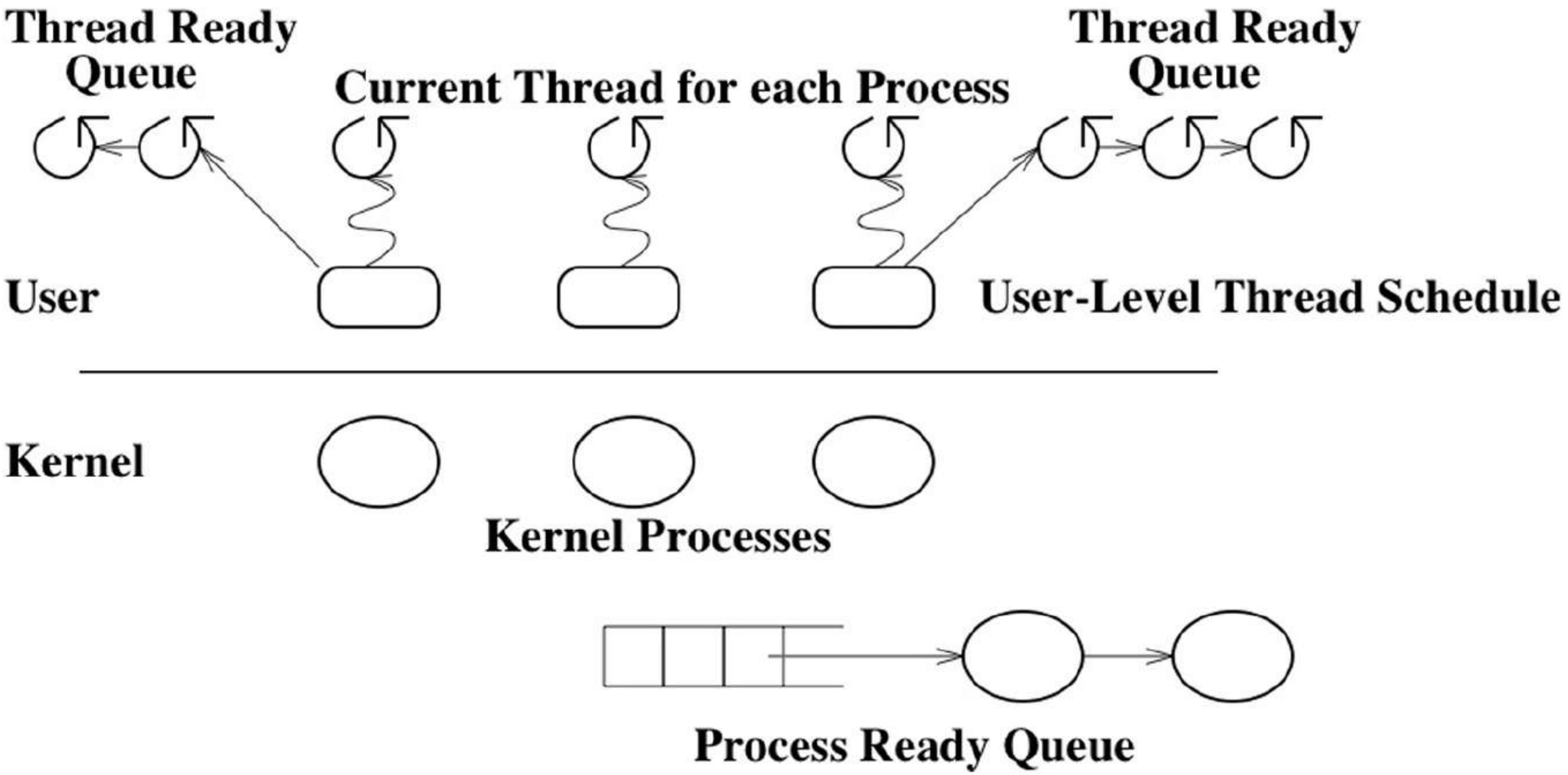
- PROs
  - Really fast and lightweight
  - Scheduling policies are more flexible
  - Can be implemented in OSs that do not support threading
  - No system calls involved, just user-space function calls
  - No actual context switch

User Threads: CONs

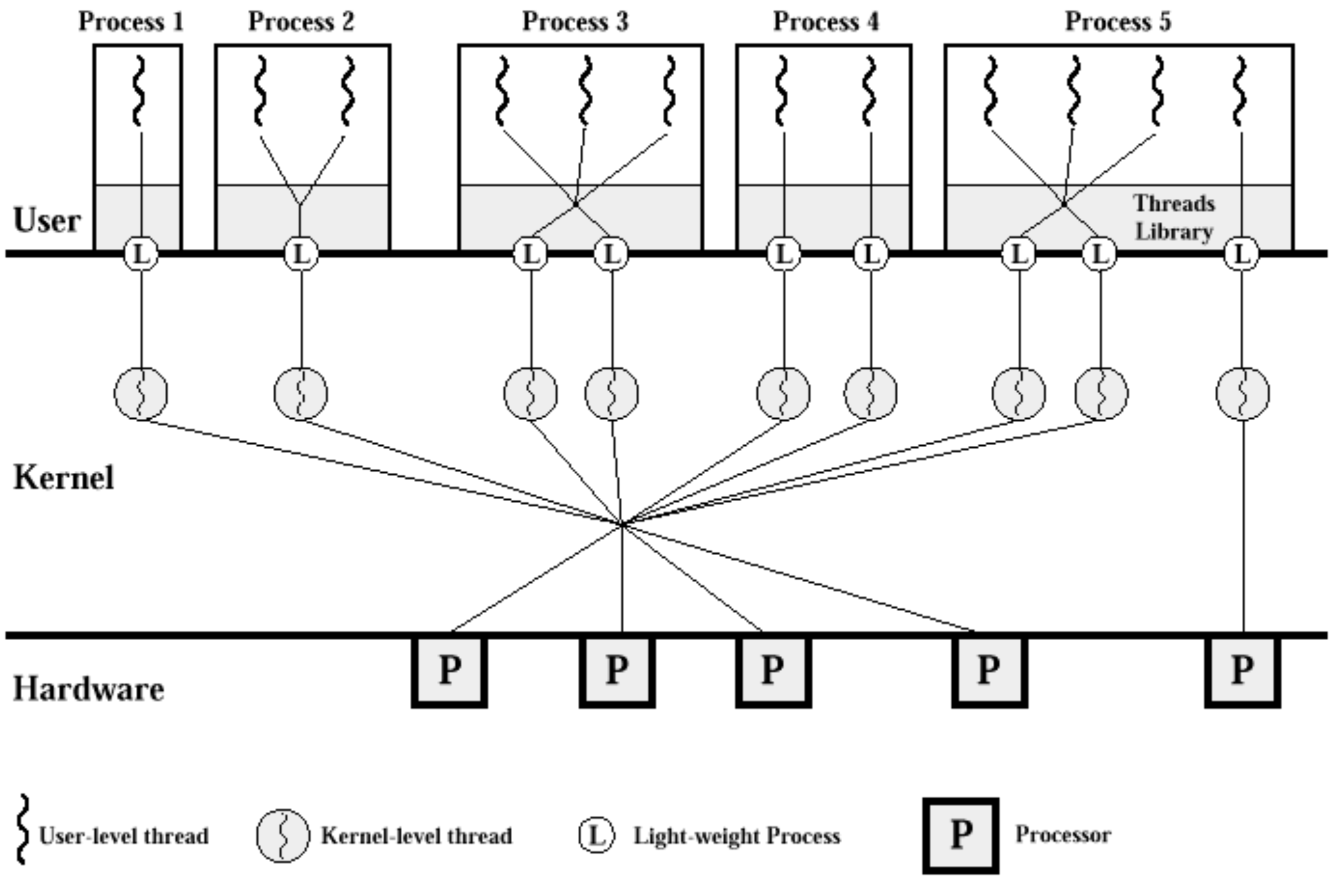


- CONs
  - No true concurrency of multi-threaded processes
  - Poor scheduling decisions
  - Lack of coordination between kernel and threads
    - A process with 100 threads competes for a time slice with a process with just 1 thread
  - Requires non-blocking system calls, otherwise all threads within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

User Threads



Hybrid Management: Lightweight Processes



Multi-threading Models

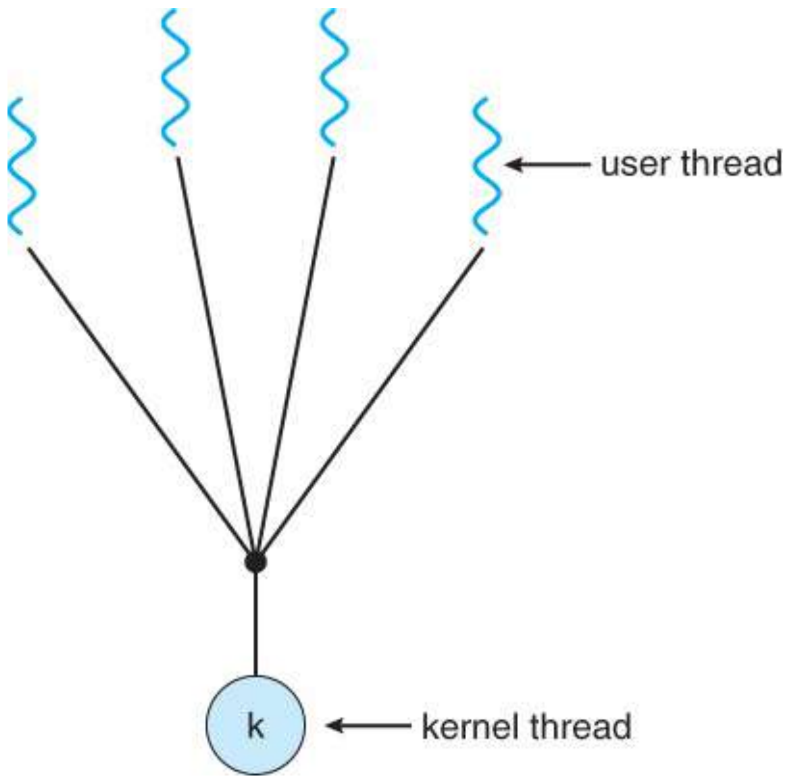
- In a specific implementation, user threads must be mapped to kernel threads in one of the following ways:
  - Many-to-One
  - One-to-One
  - Many-to-Many
  - Two-level

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

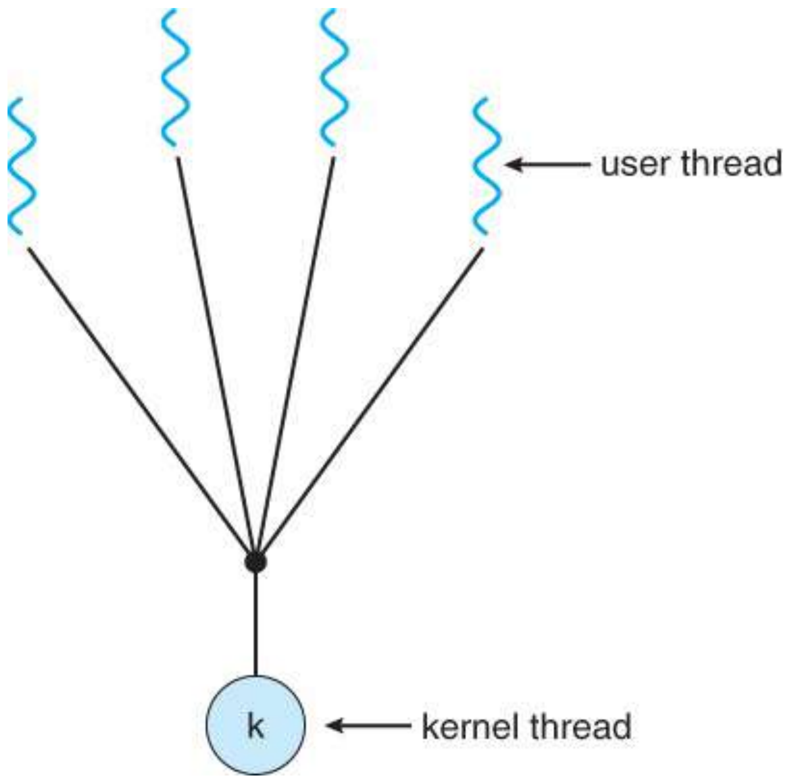
<u>Remember:</u>
A kernel thread is the unit of execution that is scheduled by the OS to run on the CPU (similar to single-threaded process)

Many-to-One Model



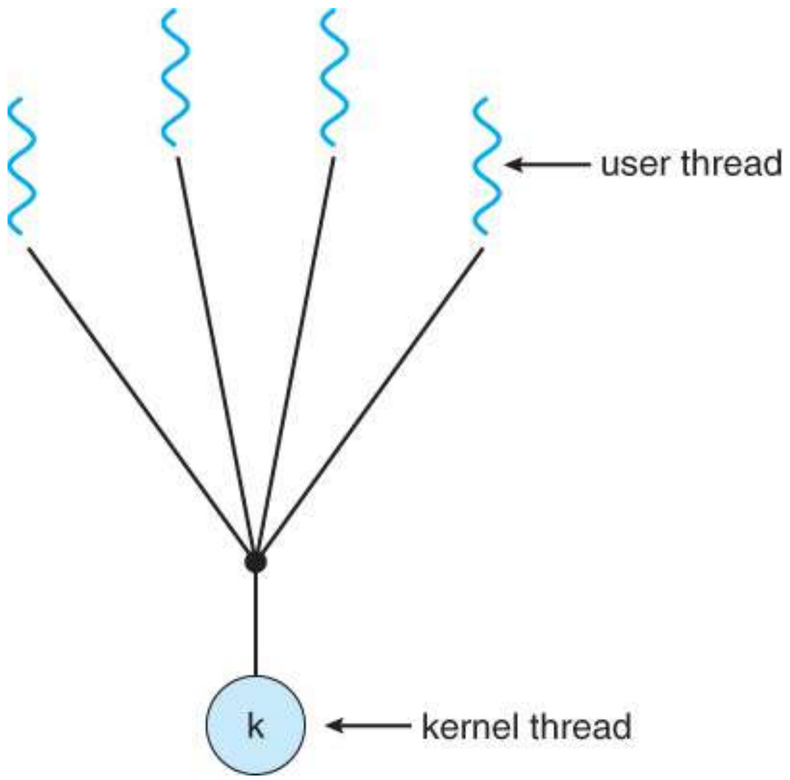
- Many user threads are all mapped onto a single kernel thread
- The process can only run one user thread at a time because there is only one kernel thread associated with it
- As single kernel thread can operate on a single CPU, multi-user-thread processes cannot be split across multiple CPUs
- If a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

Many-to-One Model



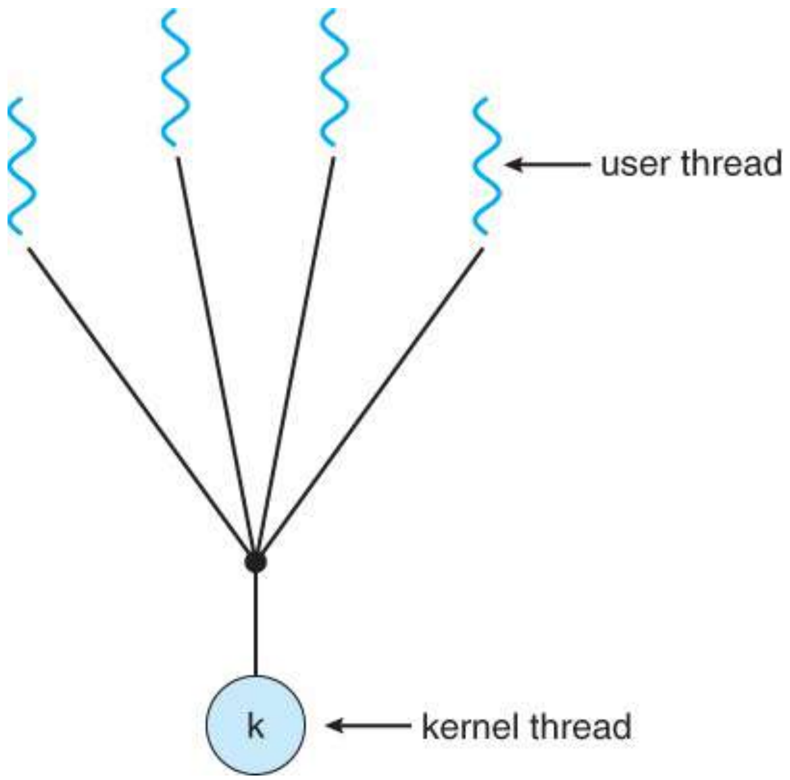
- Many user threads are all mapped onto a single kernel thread
- The process can only run one user thread at a time because there is only one kernel thread associated with it
- As single kernel thread can operate on a single CPU, multi-user-thread processes cannot be split across multiple CPUs
- If a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

Many-to-One Model



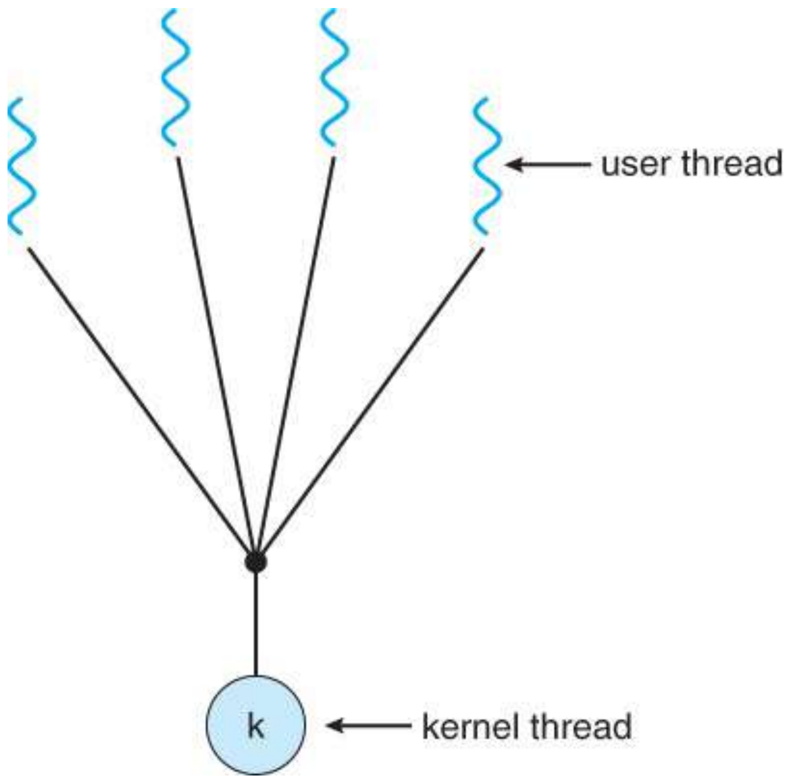
- Many user threads are all mapped onto a single kernel thread
- The process can only run one user thread at a time because there is only one kernel thread associated with it
- As single kernel thread can operate on a single CPU, multi-user-thread processes cannot be split across multiple CPUs
- If a 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

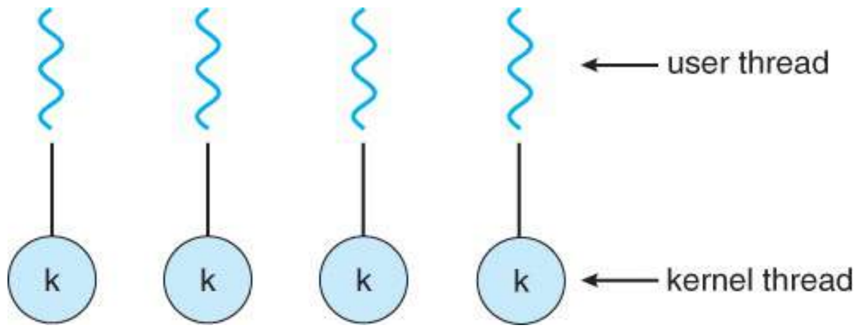
Many-to-One Model



- 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

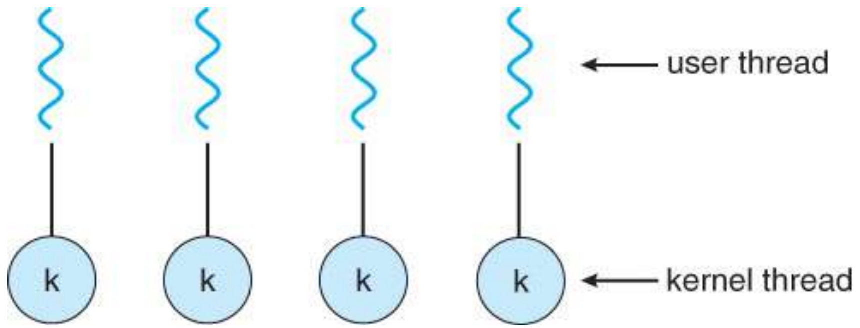
pure user-level

One-to-One Model



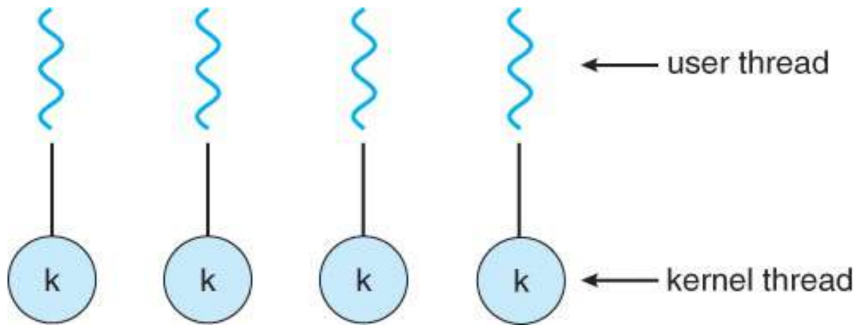
- A separate kernel thread to handle each user thread
- Overcomes the limitations of blocking system calls and splitting of processes across multiple CPUs
- The overhead of managing the one-to-one model is more significant and may slow down the system
- Most implementations of this model place a limit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

One-to-One Model



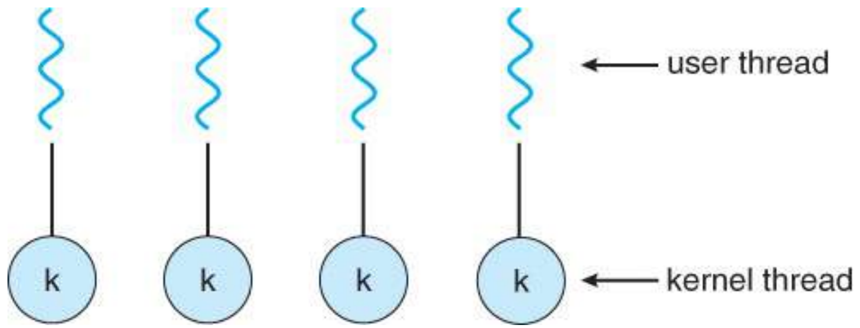
- A separate kernel thread to handle each user thread
- Overcomes the limitations of blocking system calls and splitting of processes across multiple CPUs
- The overhead of managing the one-to-one model is more significant and may slow down the system
- Most implementations of this model place a limit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

One-to-One Model



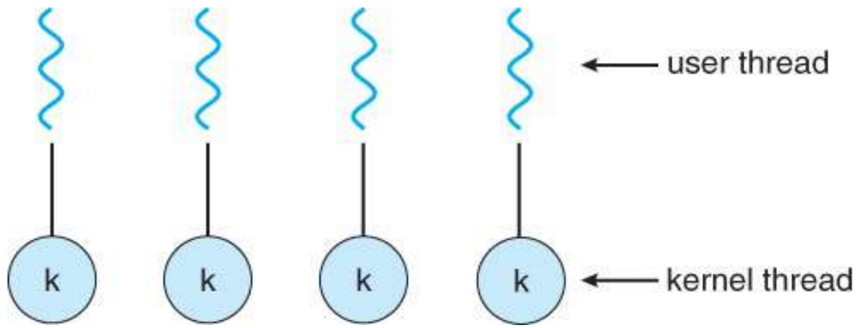
- A separate kernel thread to handle each user thread
- Overcomes the limitations of blocking system calls and splitting of processes across multiple CPUs
- The overhead of managing the one-to-one model is more significant and may slow down the system
- Most implementations of this model place a limit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

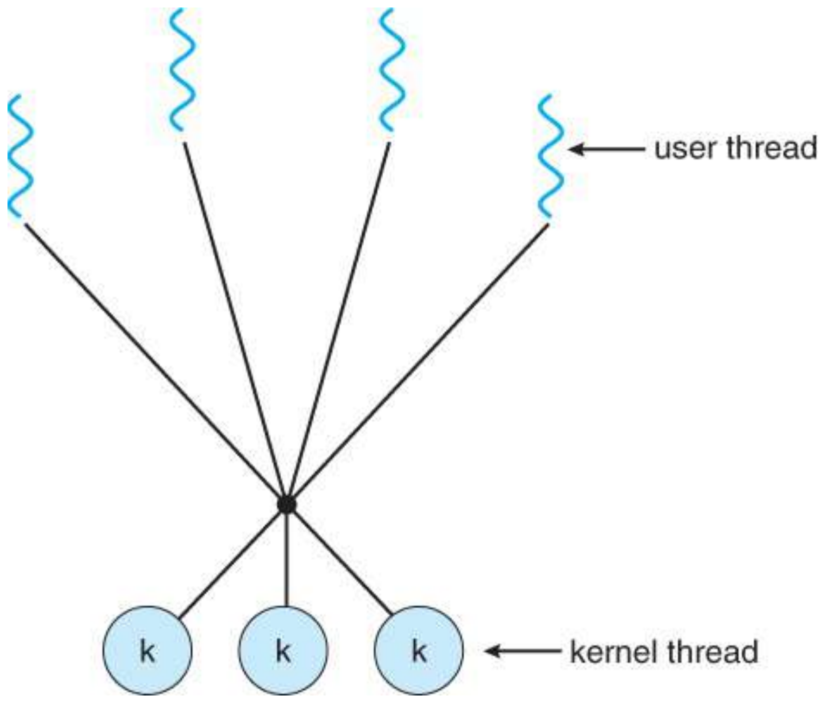
One-to-One Model



- 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

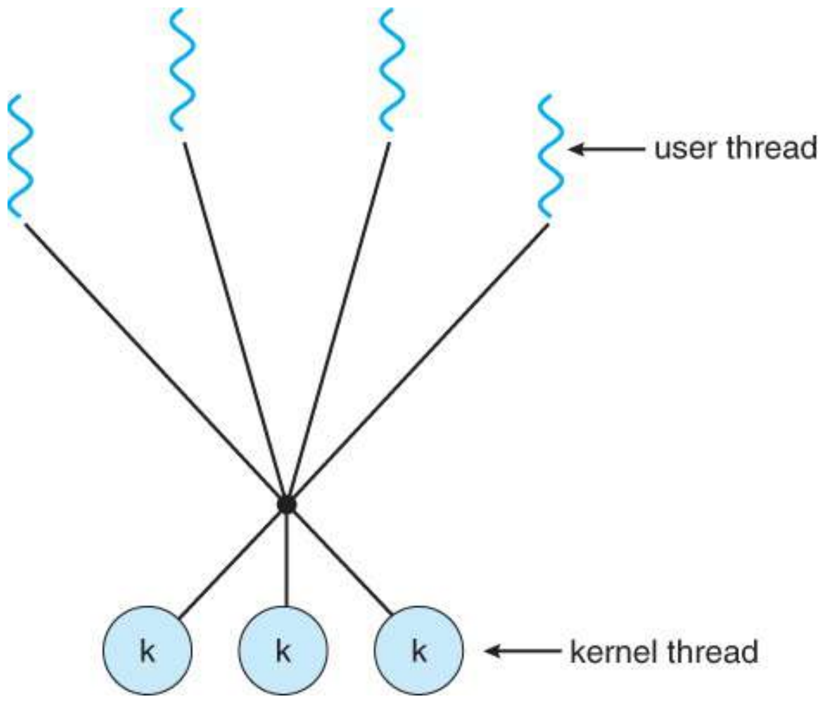
pure kernel-level

Many-to-Many Model



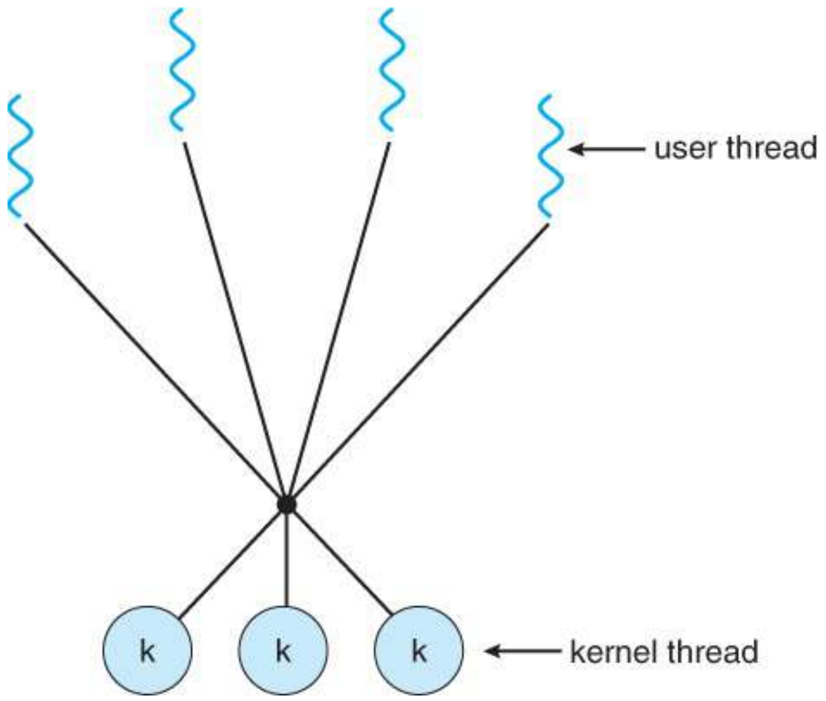
- Multiplexes any number of user threads onto an equal or smaller number of kernel threads
- Users have no restrictions on the number of threads created
- Processes can be split across multiple processors
- Blocking kernel system calls do not block the entire process

Many-to-Many Model



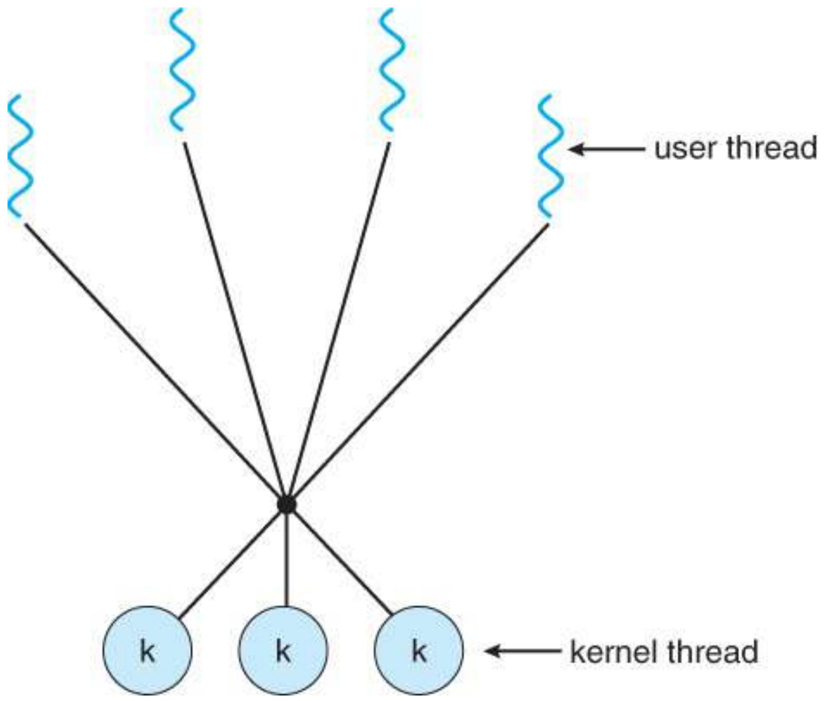
- Multiplexes any number of user threads onto an equal or smaller number of kernel threads
- Users have no restrictions on the number of threads created
- Processes can be split across multiple processors
- Blocking kernel system calls do not block the entire process

Many-to-Many Model



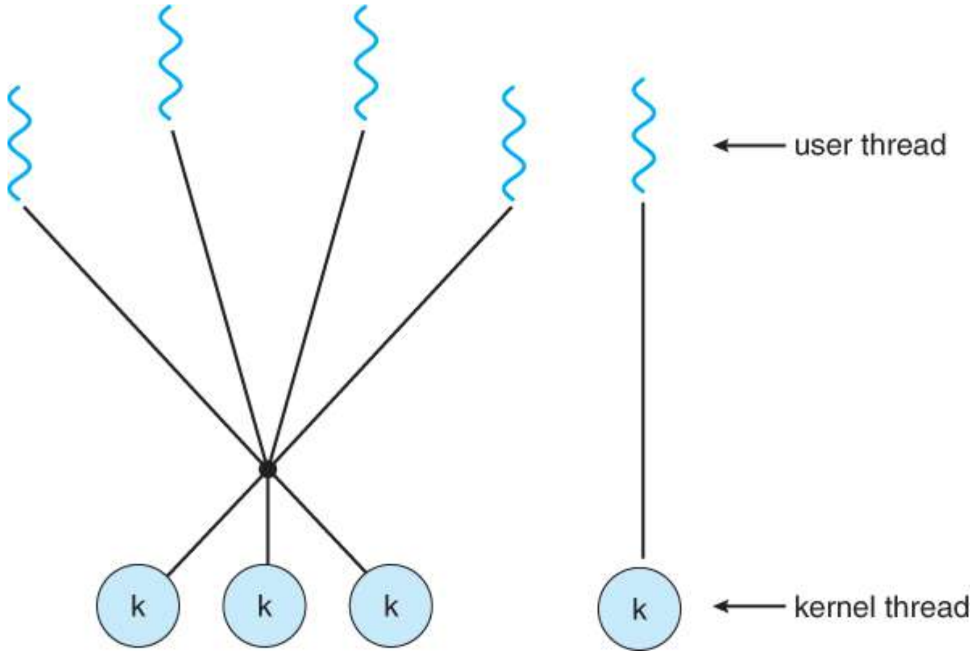
- Multiplexes any number of user threads onto an equal or smaller number of kernel threads
- Users have no restrictions on the number of threads created
- Processes can be split across multiple processors
- Blocking kernel system calls do not block the entire process

Many-to-Many Model



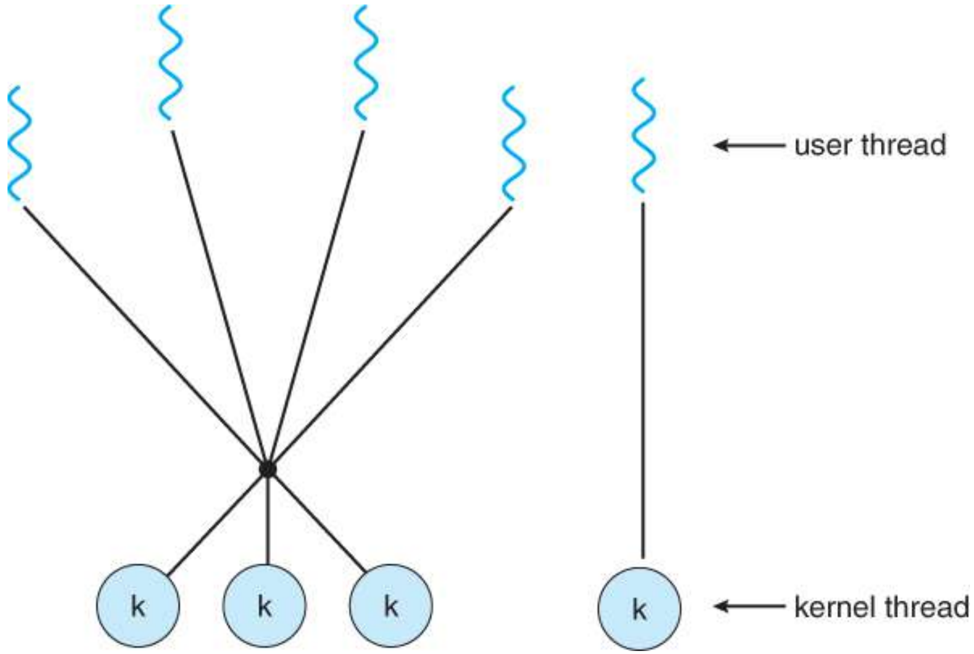
- Multiplexes any number of user threads onto an equal or smaller number of kernel threads
- Users have no restrictions on the number of threads created
- Processes can be split across multiple processors
- Blocking kernel system calls do not block the entire process

Two-Level Model



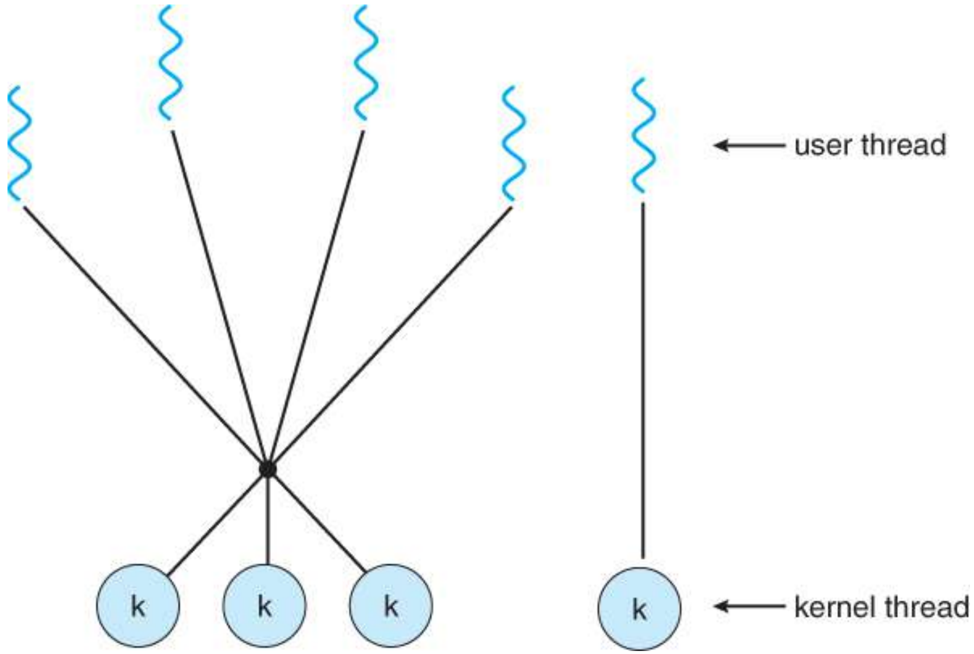
- A variant of the many-to-many model
- Mixes many-to-many with one-to-one
- Increases the flexibility of scheduling policies

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

Pthreads: Sum Example

Compute the sum of the first *N* integers on a separate thread

Pthreads: Sum Example

Compute the sum of the first *N* integers on a separate thread

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }

  /* get the default attributes */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

Figure 4.6 Multithreaded C program using the Pthreads API.

Pthreads: Sum Example

Compute the sum of the first *N* integers on a separate thread

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }

  /* get the default attributes */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

Figure 4.6 Multithreaded C program using the Pthreads API.

Pthreads: Sum Example

Compute the sum of the first *N* integers on a separate thread

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }

  /* get the default attributes */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

Figure 4.6 Multithreaded C program using the Pthreads API.

Pthreads: Sum Example

Compute the sum of the first *N* integers on a separate thread

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }

  /* get the default attributes */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

Figure 4.6 Multithreaded C program using the Pthreads API.

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 (although 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

```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

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 up

Thread Pools: Idea

- A specific number of threads are created when the process starts up
- 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 up
- 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 up
- 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 up
- 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 simply waits for one

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
  - For 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
- 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
- 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 as an interface between user thread and the kernel thread (many-to-many or two-tier)
- This virtual processor is known as a Lightweight Process (LWP)
- There is a one-to-one correspondence between LWPs and kernel threads
- The number of kernel threads available may change dynamically
- The application (user-level thread library) maps user threads onto available LWPs
- Kernel threads are scheduled onto the real processor(s) by the OS

Thread Scheduling: Activation

- The kernel communicates to the user-level thread library when certain events occur (e.g., a thread is blocking) via an upcall
- The upcall is handled in the thread library by an upcall handler
- The upcall also provides a new LWP for the upcall handler to run on, which it can then use to reschedule the user thread that is about to become blocked
- The OS will also issue upcalls when a thread unblocks, so the thread library can make appropriate adjustments
- If the kernel thread blocks then the LWP blocks, which blocks the user thread

Summary

- A thread is a single execution stream within a process

Summary

- A thread is a single execution stream within a process
- Thread vs. Process:

Summary

- A thread is a single execution stream within a process
- Thread vs. Process:
  - common vs. separate address spaces → quicker communication

Summary

- 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

Summary

- 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
- User- vs. Kernel-level threads

Summary

- 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
- User- vs. Kernel-level threads
- Scheduling algorithms operates (almost) transparently with threads