

Operating Systems

Prepared by Dr. Alaa Eldeen Ahmed

IT263 / CYB510
Threads

Lecture 3



Threads (Objectives)

- Motivation of using threads
- Multithreading Models
- Threading Issues

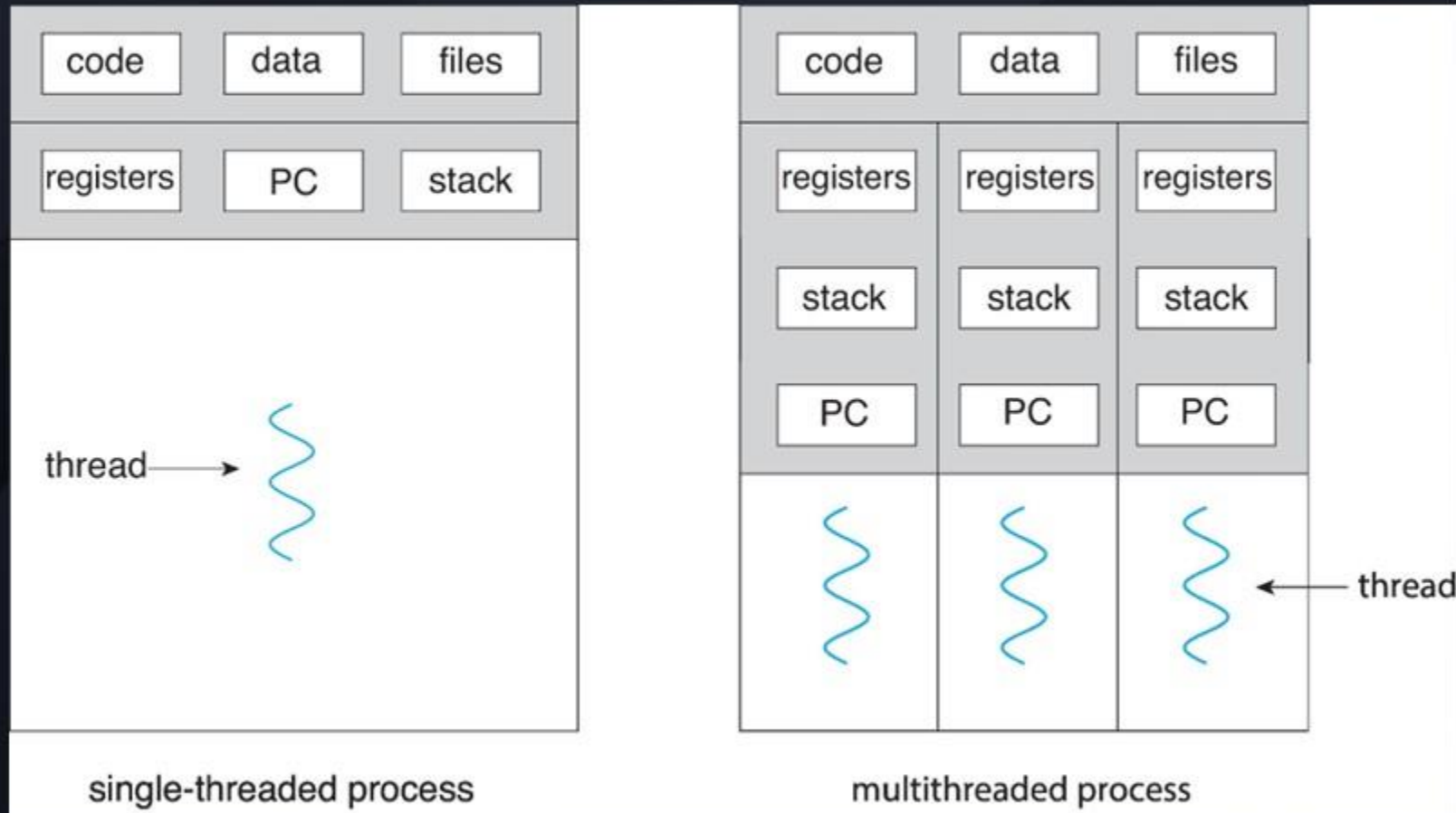
- Programs, processes, and threads are all terms that related to software execution.
- In last lecture we explained the meaning of Program and Processes.
- In this lecture we go in deep explaining threads.
- A thread is the unit of execution within a process

- The general structure of a process assumes that the process has an address space with a **single sequential execution sequence**.
- Thread is a mechanism to provide multiple execution control to the processes
- All modern operating systems, provide features enabling a process to contain multiple threads of control.

Single and Multithreaded Processes

- A thread is a basic unit of CPU utilization;
- it comprises a program counter, a register set, and a stack.
- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

Single and Multithreaded Processes



Single and Multithreaded Processes

- Process creation is heavy-weight while thread creation is light-weight
- In multi-threaded environments, each thread has its own Thread ID. All the threads from the same process have the same PID.
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

- Most software applications that run on modern computers are multithreaded.
- An application typically is implemented as a separate process with several threads of control.
- for example, a word processor may have:
 - 1- a thread for displaying graphics
 - 2- another thread for responding to keystrokes from the user
 - 3- a third thread for performing spelling and grammar checking in the background.

Single and Multithreaded Processes

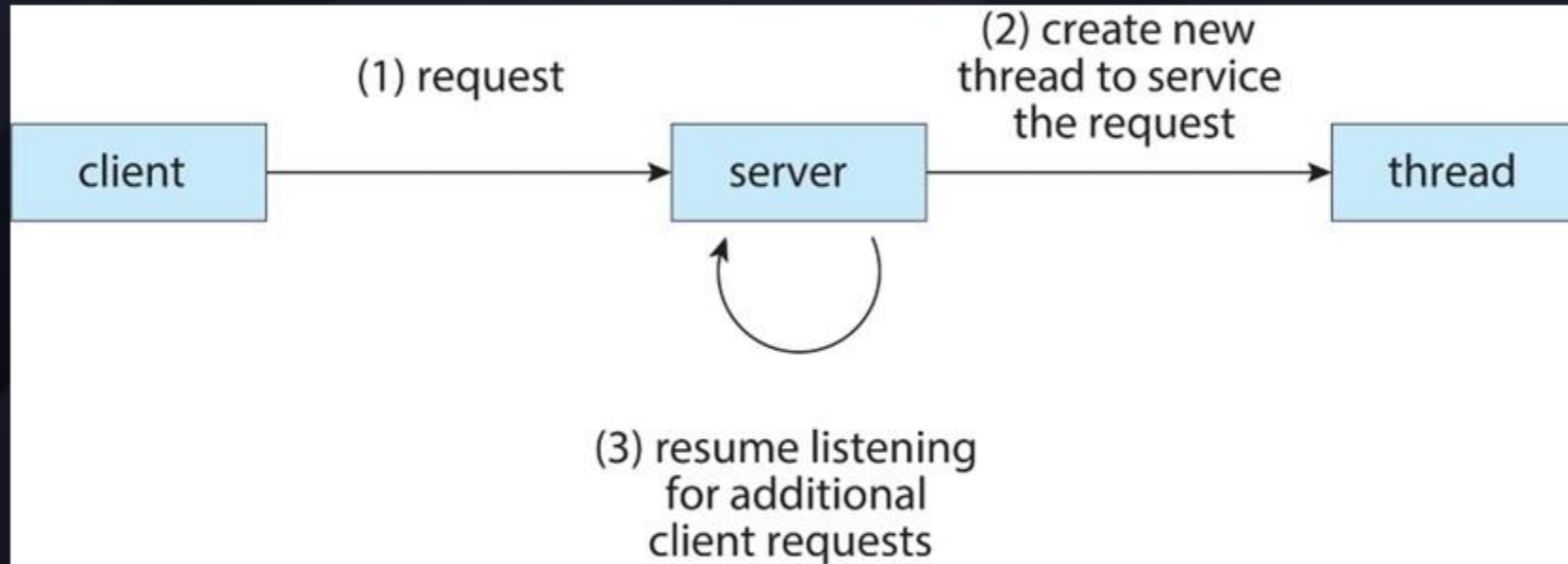
- In certain situations, a single application may be required to perform several similar tasks.
- **For example**, a web server accepts client requests for web pages, images, sound, and so forth.
- A busy web server may have several (perhaps thousands of) clients concurrently accessing it.
- Traditional web server with a single-threaded process, would be able to service only one client at a time, and other clients wait a very long time for its request to be serviced.

- One solution is to have the server run as a single process that accepts requests.
- When the server receives a request, it creates a separate process to serve the request.
- This solution need to create a new process per request.
- Process creation is time consuming and resource intensive,

Single and Multithreaded Processes

- In this example, the new process will perform the same tasks as the existing process, so why incur all that overhead?
- It is more efficient to use one web-server process that contains multiple threads.
- The server process create a separate thread to listen for client requests.
- The server process creates a new thread to service the request and resume listening for additional requests.

Single and Multithreaded Processes

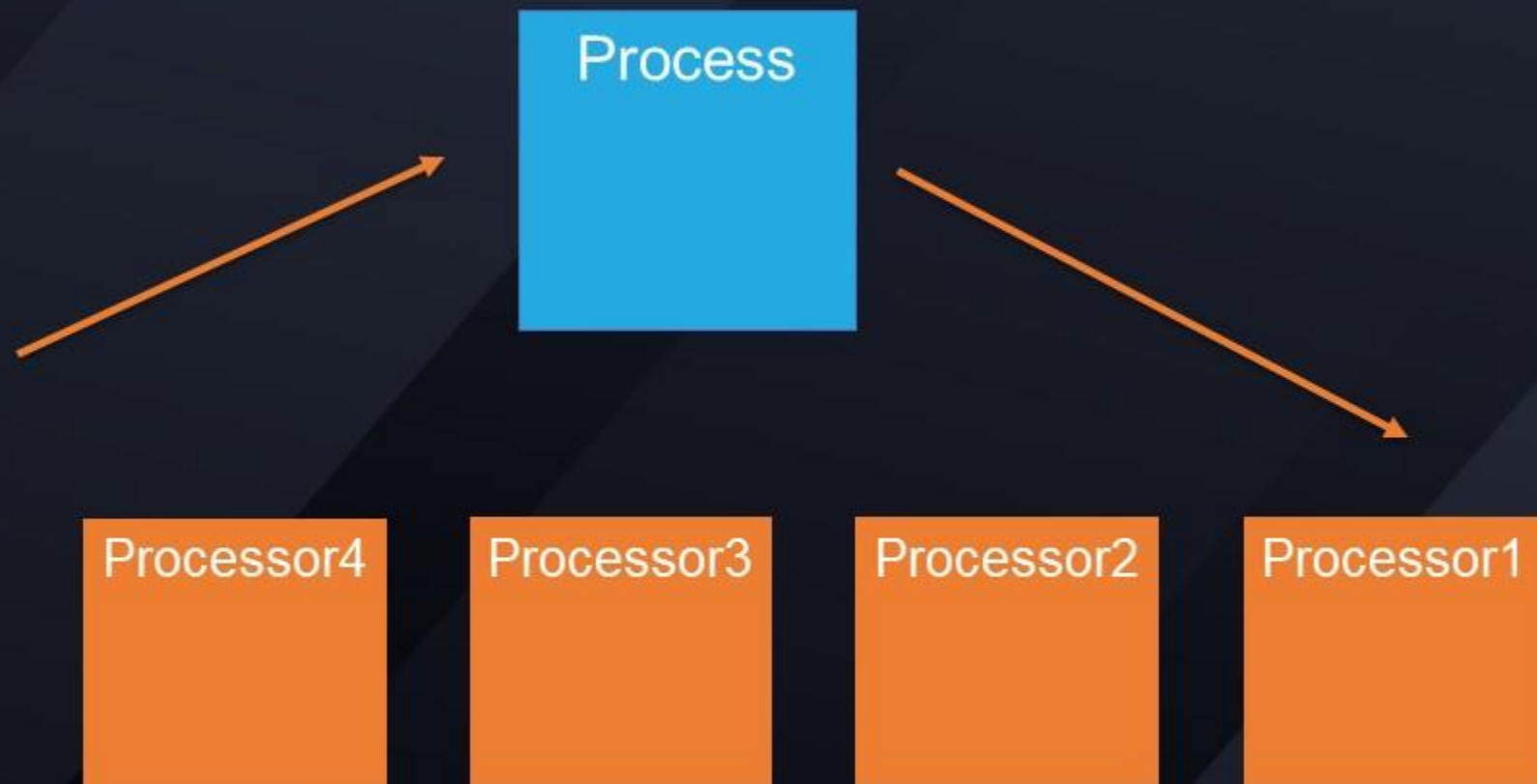


Lecture 3

Single and Multithreaded Processes

```
long add()  
{  
    long sum = 0;  
    for (i = 0 ; i<100000000, i++)  
        sum = sum + i;  
    return sum;  
}
```

```
int main ()  
{  
    long sum ;  
    sum = add();  
    print ("sum = ", sum);  
}
```



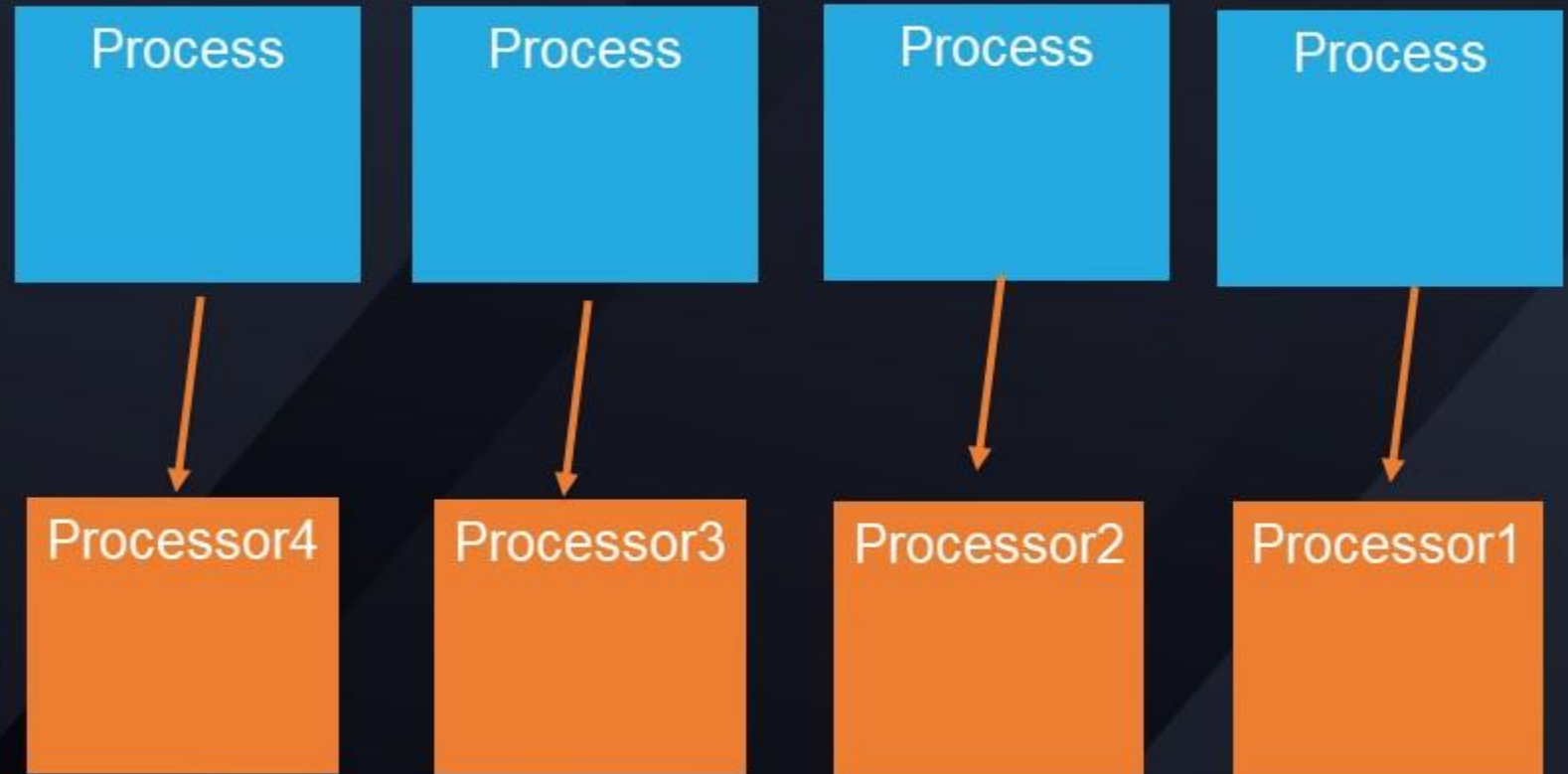
One solution to improve the performance is to:

1- $100000000/4 = 25000000$

2- one solution is to create 4 isolated processes where each process will do $\frac{1}{4}$ of the all task

3- Interposes communication is needed to sum up the result (communicate processes together)

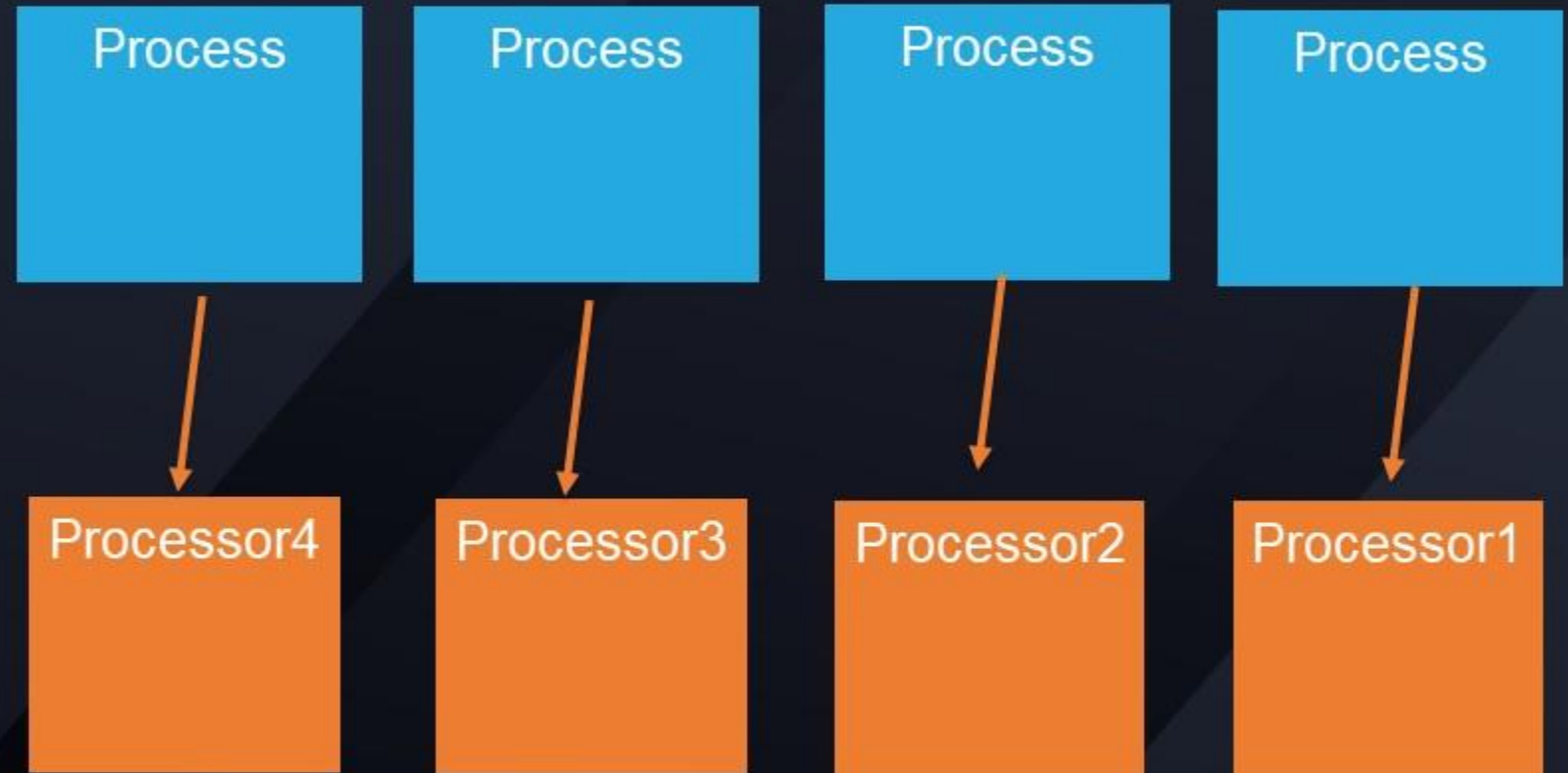
4- more system calls are needed



5- in memory will have 4 processes. Each process is represented with a PCB (Stack, heap, counter, files, code, etc....)

6- There will be an efficient use of processors.

7- The solution may be improved approximately by 4 times faster.

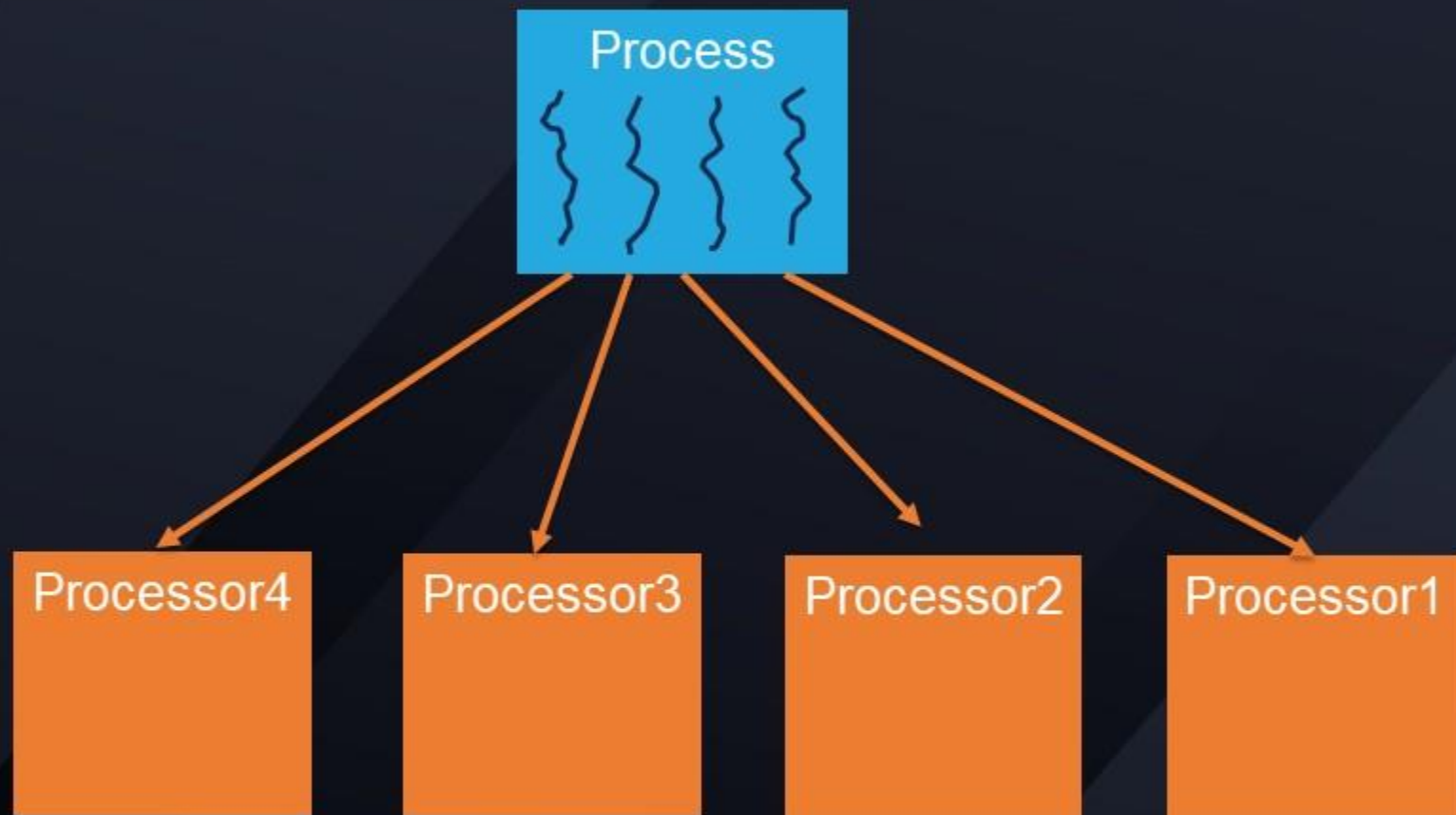


1- $100000000/4 = 25000000$

2- another solution is to create only one processes with 4 threads; each thread will do $\frac{1}{4}$ of the task

4- No Interposes communication is needed to sum up the result since all threads belong to the same process.

3- less system calls are needed

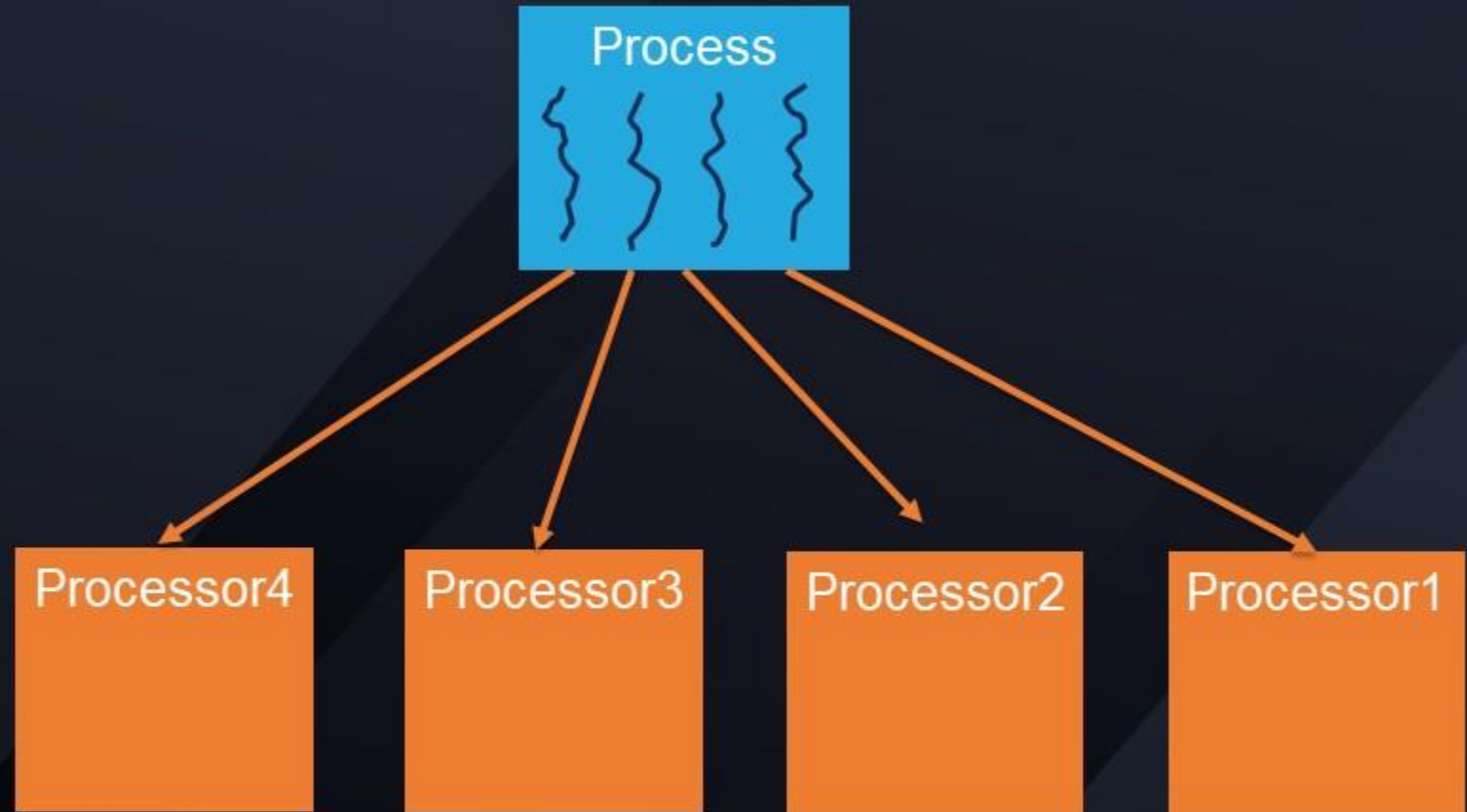


Single and Multithreaded Processes

5- in memory will have one processes wit the same (files, code, data) and 4 threads which represented by only (Stack, heap, counter)

6- will be an efficient use of processors.

7- The solution may be improved approximately by 4 times faster.



Benefits

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multicore architectures

Multicore or multiprocessor systems putting pressure on programmers, challenges include:

- **Dividing activities**
- **Balance**
- **Data splitting**
- **Data dependency**
- **Testing and debugging**

Concurrency vs. Parallelism

□ Concurrent execution on single-core system:

Concurrency occurs when two or more Processes start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll be running at the same instant.,

□ Parallelism on a multi-core system.

Most probably is applied with Single processor / core scheduler to provide concurrency

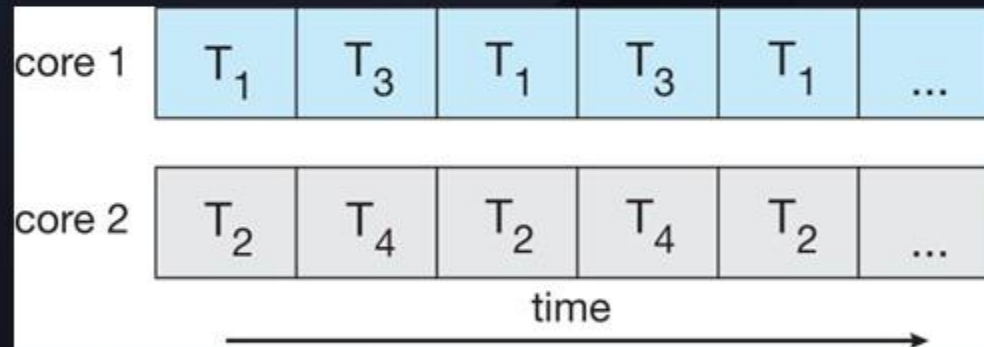


Concurrency vs. Parallelism

- Concurrent execution on single-core system:

Parallelism implies a system can perform more than one task simultaneously

- Parallelism on a multi-core system:

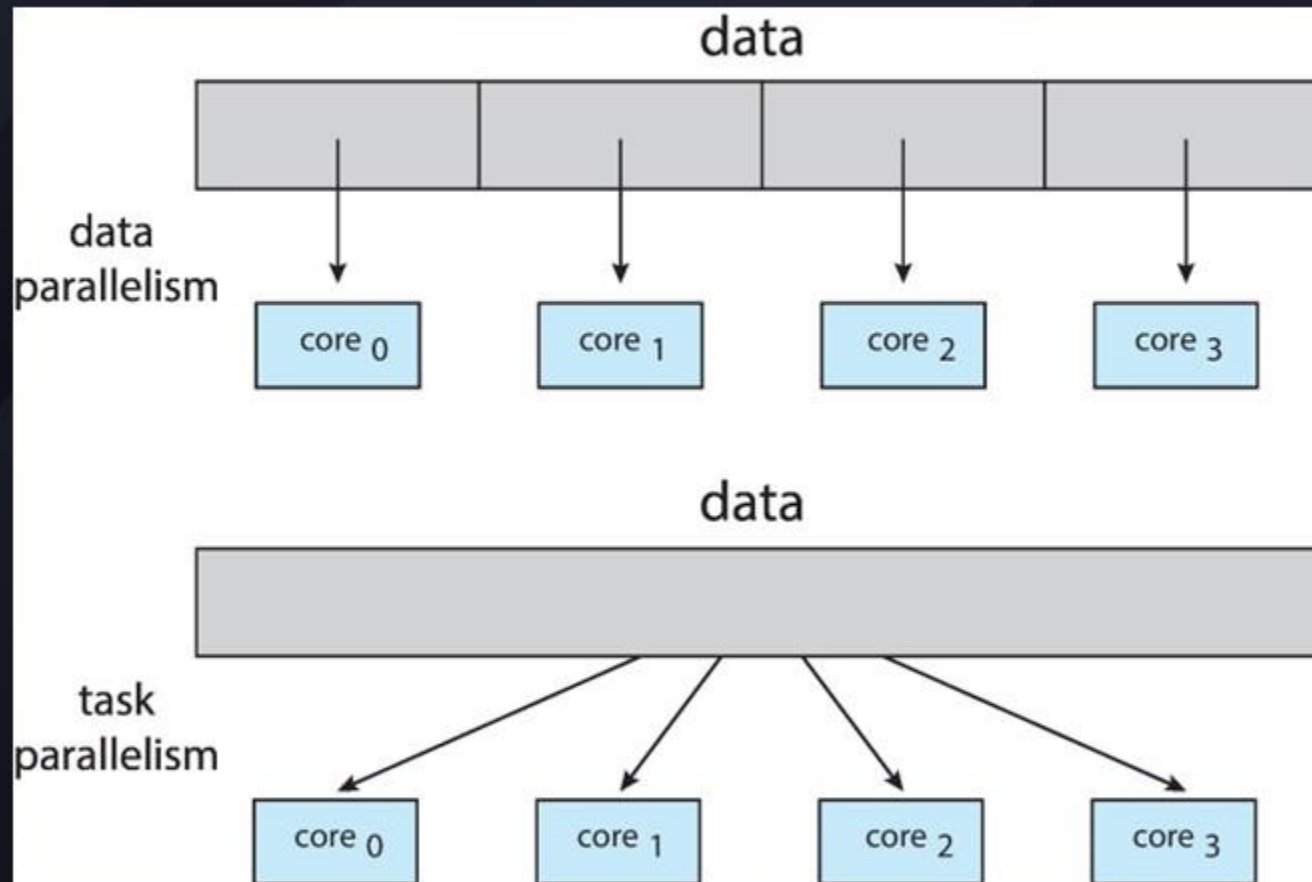


Types of parallelism

Data parallelism – distributes subsets of the same data across multiple cores, same operation on each

Task parallelism – distributing threads across cores, each thread performing unique operation

Data and Task Parallelism



User threads - management done by user-level threads library

Three primary thread libraries:

- POSIX **Pthreads**

- Windows threads

- Java threads

Kernel threads - Supported by the Kernel

Examples – virtually all general purpose operating systems, including:

- Windows

- Linux

- Mac OS X

- iOS

- Threads managed by a threads library
- Kernel is unaware of presence of threads

Advantages:

- Efficient: creation/deletion/switches don't need system calls
- Flexibility in scheduling: library can use different scheduling algorithms, can be application dependent
- User-level threads can be run on any operating system.
- There are no kernel mode privileges required for thread switching in user-level threads.

Disadvantages

- Need to avoid blocking system calls [all threads block]
- Threads compete for one another
- Does not take advantage of multiprocessors [no real parallelism]

Kernel Threads

- Kernel-level threads are managed directly by the operating system kernel.
- Kernel aware of presence of threads

Advantages:

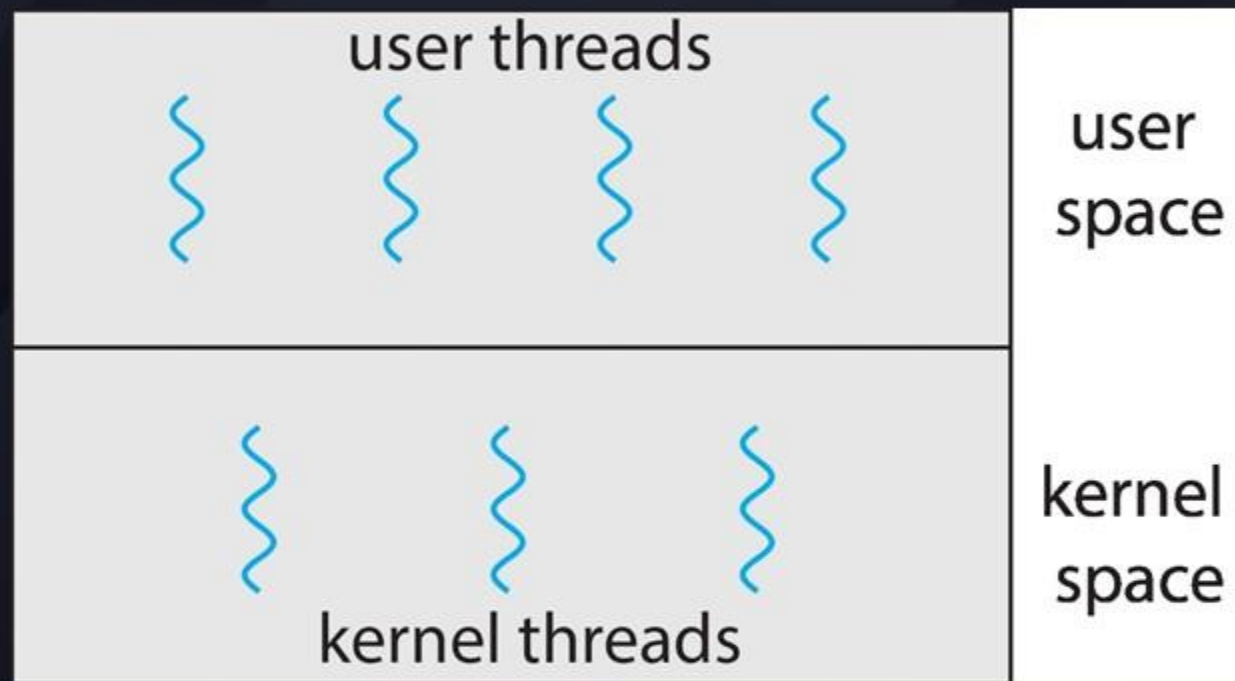
- Multiple threads of the same process can be scheduled on different processors in kernel-level threads.
- The kernel functions can also be multithreaded.
- If kernel-level thread is blocked, another thread of the same process can be scheduled by the kernel.

Disadvantage:

- They are slower to create and manage as compared to user-level threads.
- Its Context switching requires more resources and time.
- They are highly dependent on the operating system kernel and may not be portable across different systems.

- Many-to-One
- One-to-One
- Many-to-Many

User and Kernel Threads

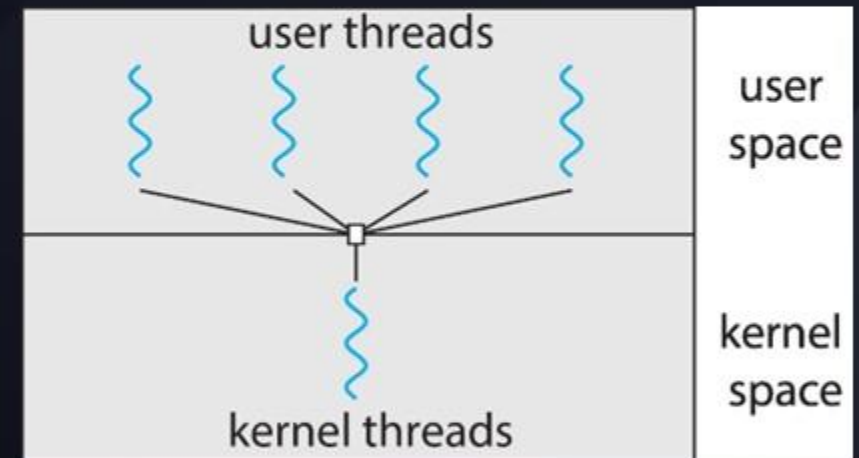


Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model

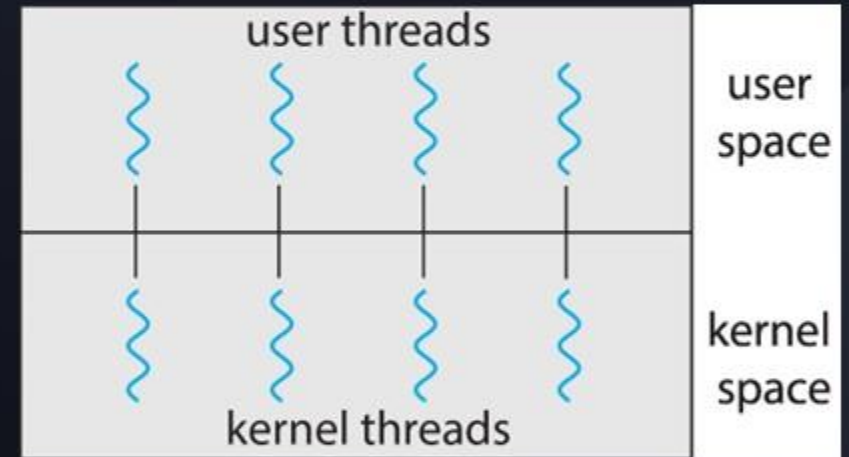
Examples:

Solaris Green Threads
GNU Portable Threads



One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
so, The kernel may become a bottleneck for managing large numbers of threads.



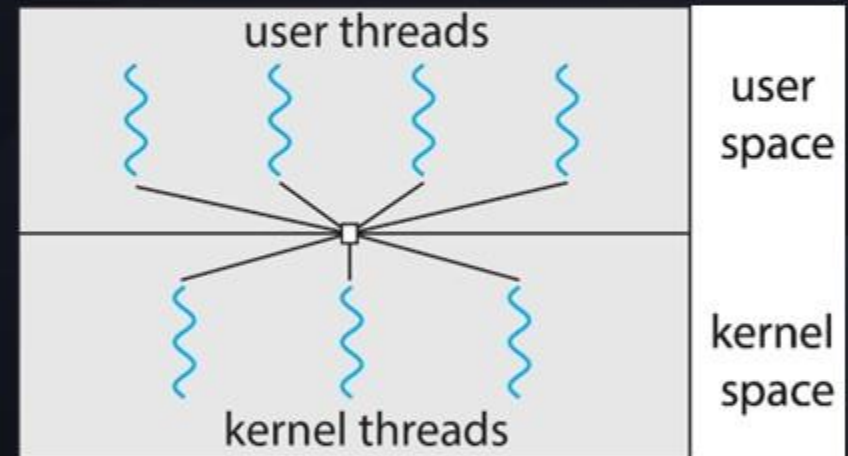
Examples

Windows

Linux

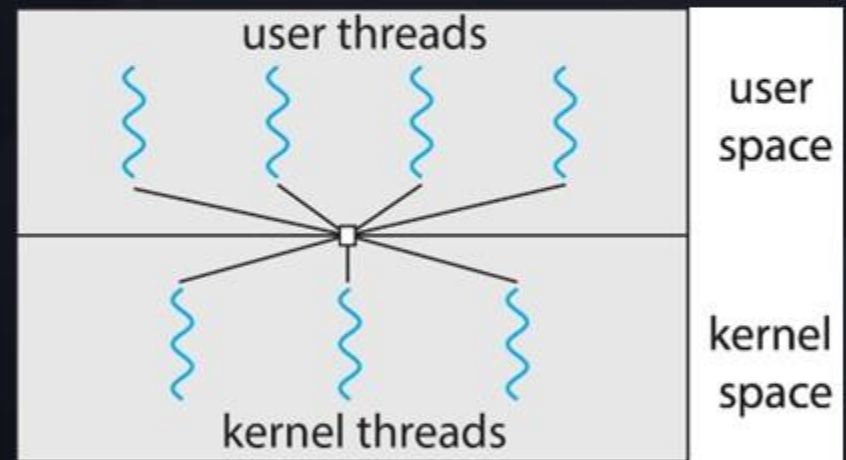
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create enough kernel threads (which is less than as a number compared to user level threads)
- It is a compromise between the other two models.



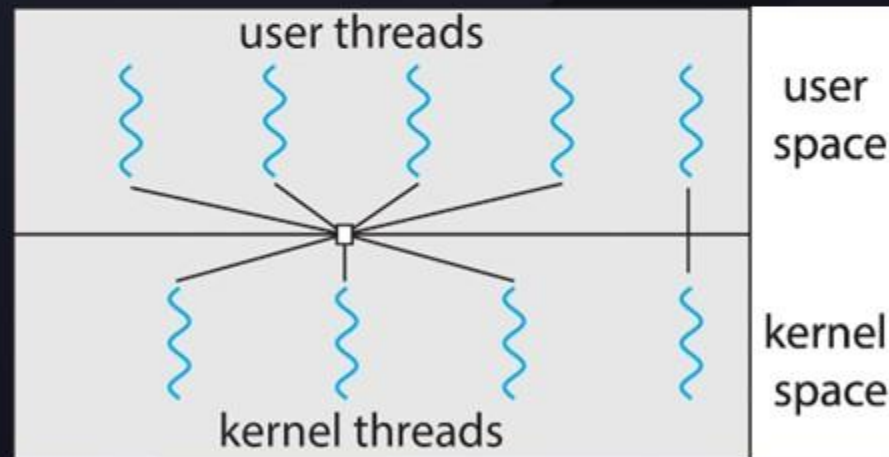
Many-to-Many Model

- The number of kernel threads created depends upon a particular application.
- if any thread makes a blocking system call, the kernel can schedule another thread for execution.
- Windows with the *ThreadFiber* package



Two-level Model

Similar to M:M, except that it allows a user thread to be **bound** to kernel thread



Threading Issues

- Semantics of `fork()` and `exec()` system calls
- Thread cancellation of target thread
- Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of fork() and exec()

Does `fork()` duplicate only the calling thread or all threads?

- Some UNIXes have two versions of fork
 - 1- duplicates all threads
 - 2- duplicates only the thread that invoked the `fork()` system call.
- `exec()` usually works as normal – replace the running process including all threads

Does `fork()` duplicate only the calling thread or all threads?

- Which of the two versions of `fork()` to use depends on the application.
- 1) If `exec()` is called immediately after forking, then duplicating all threads is unnecessary, as the program called by `exec()` will replace the process. so, duplicating only the calling thread is appropriate.
- 2) If, The separate process does not call `exec()` after forking, the separate process should duplicate all threads.

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**

Two general approaches:

- **Asynchronous cancellation** terminates the target thread immediately
- **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Thread Cancellation (Cont.)

Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

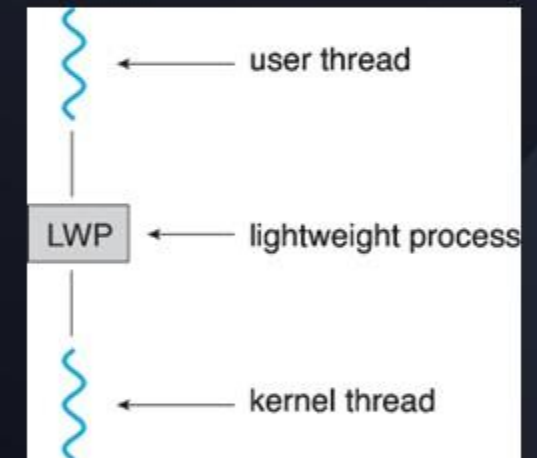
Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- The Default type is “deferred” where Cancellation only occurs when thread reaches **cancellation point**

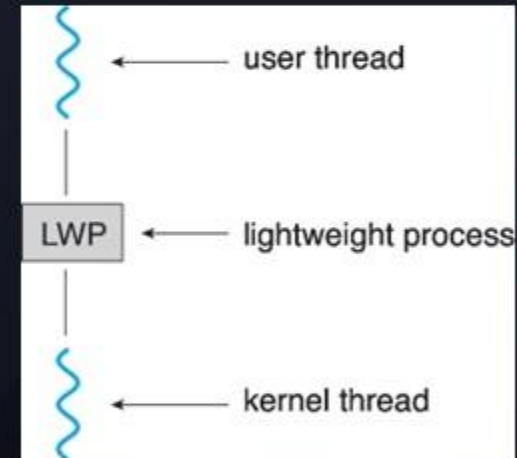
Thread-Local Storage

- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Different from local variables Local variables visible only during single function invocation
- **TLS** visible across function invocations Similar to **static** data
- **TLS** is unique to each thread

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **light weight process (LWP)**
- Appears to be a virtual processor on which process can schedule user thread to run



- Each LWP attached to kernel thread
- How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads





Thank You