CS-2006
Operating System
Lecture 6

Mahzaib Younas

Lecturer Department of Computer Science

FAST NUCES CFD

Outlines

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Threading Issues

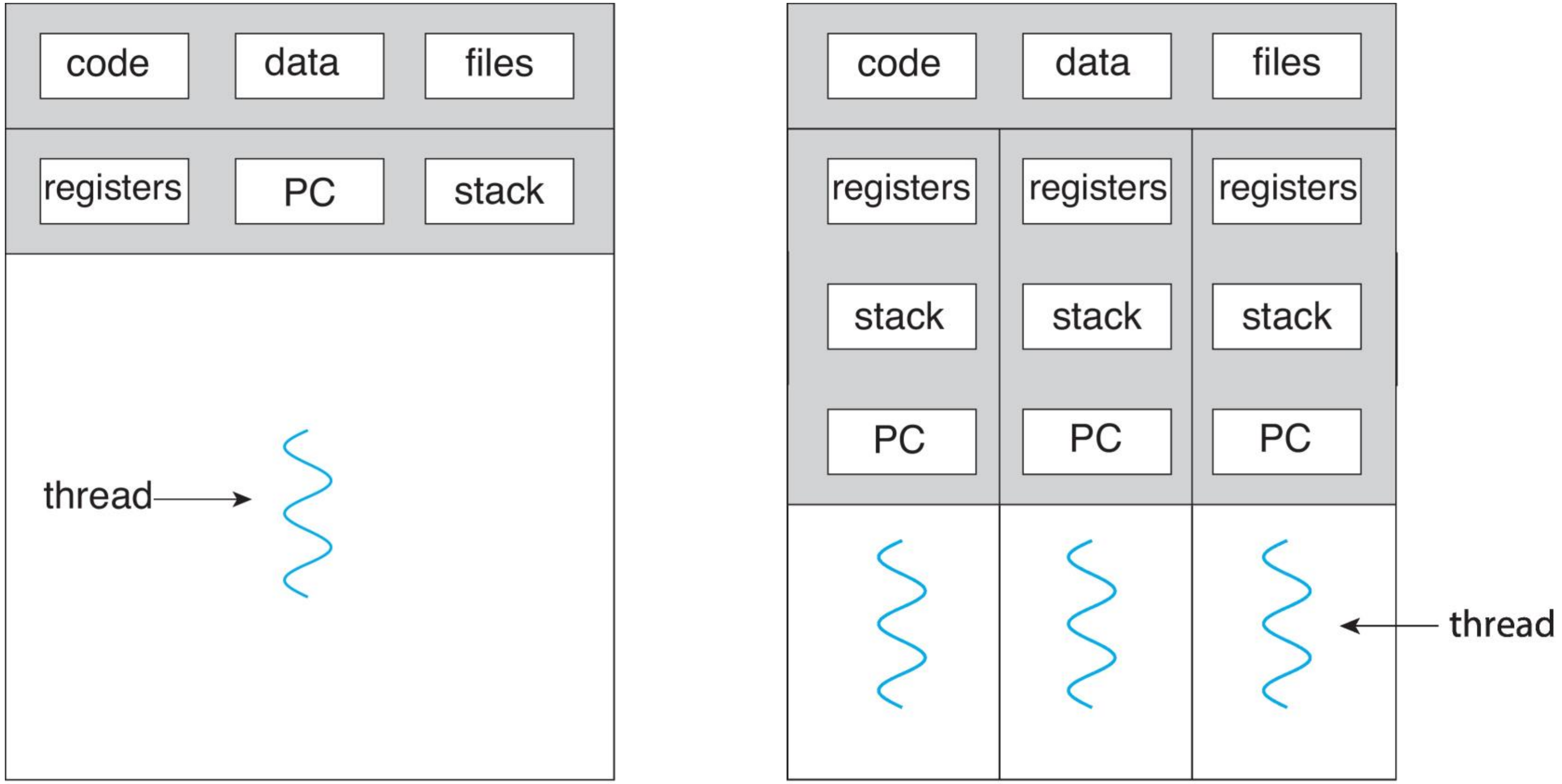
Overview:

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

Threading

- A thread is a basic unit of CPU utilization; it comprises
  - thread ID
  - program counter (PC)
  - register set
  - stack.
- It shares with other threads belonging to the same process its
  - code section
  - data section
  - operating-system resources
- For Example: open files and signals.
- A **traditional process has a single thread of control**. If a process has **multiple threads of control**, it **can perform more than one task at a time**.

Threading



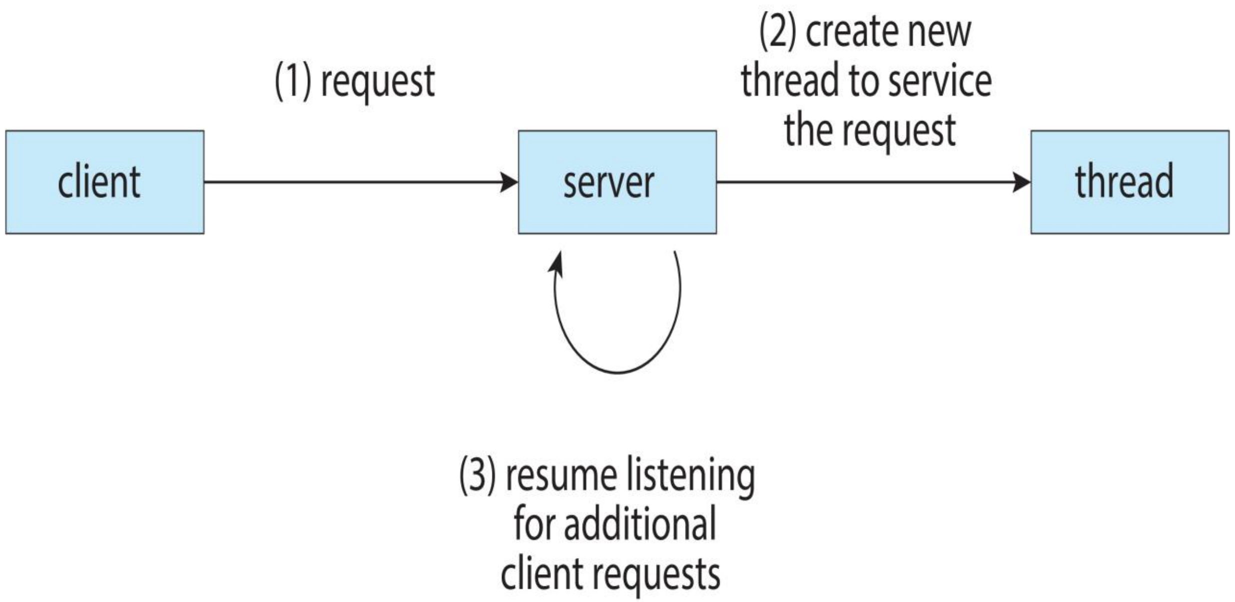
single-threaded process

multithreaded process

Multi Threading Server Architecture

Example

If the **web-server process is multithreaded** the server will create a **separate thread that listens for client requests**. When a r**equest is made**, rather than creating another process, the **server creates a new thread to service the request and resumes listening for additional requests**



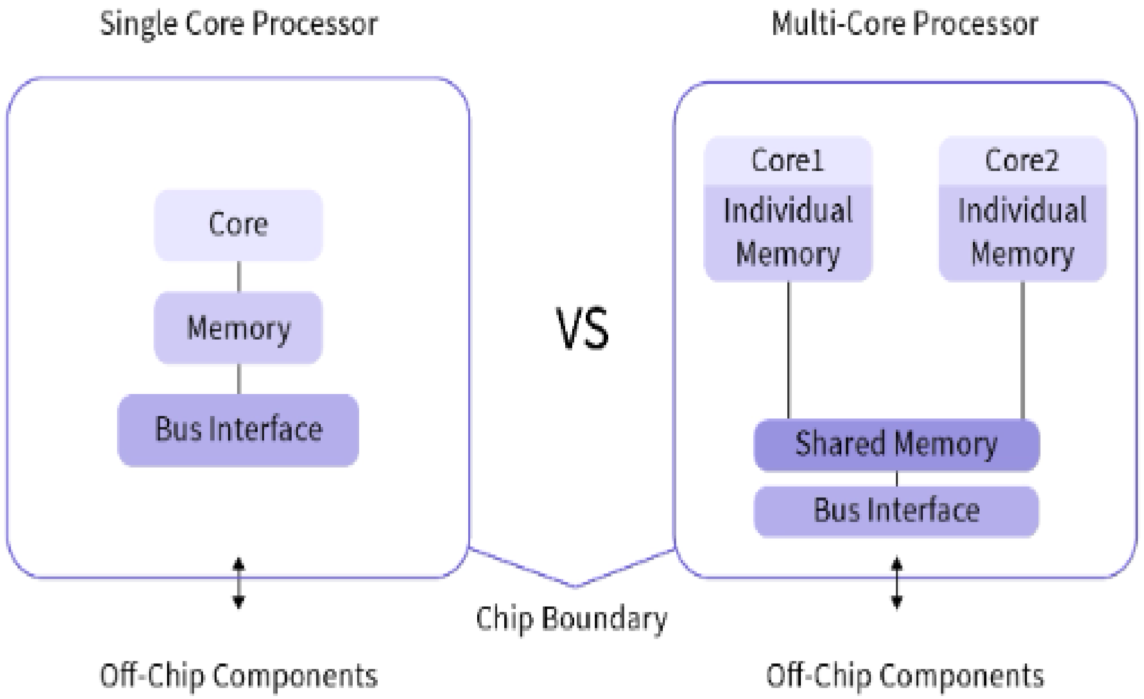
Benefits of Multi Threading

- **Responsiveness**
  - may allow **continued execution** if part of process is blocked, especially important for user interfaces.
  - Mouse Click Response
- **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**
  - A single-threaded process can run on only one processor, regardless how many are available

Multi Core Programming

A multicore processor system is a single processor with multiple execution cores in one chip.

- Multicore or multiprocessor systems puts pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**



Multicore Programming Challenges

- **Identifying Task / Dividing Activities**

  Examine the application to find area that can be divided into separate, concurrent task. Ideally, tasks are independent of one another and this can run in parallel cores.

- **Balance**
  - Identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value.
- **Data Splitting**
  - Applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores

Multicore Programming Challenges

- **Data Dependency**

  The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency

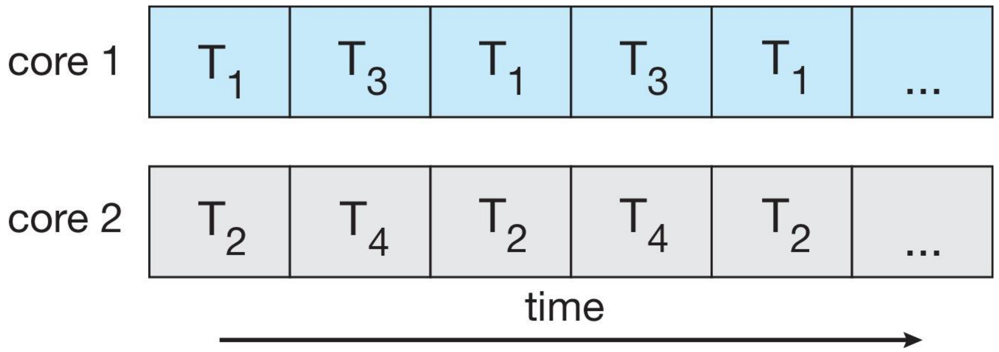
- **Testing and debugging**

  A program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult than testing and debugging single-threaded applications

Multi Core Programming (Cont..)

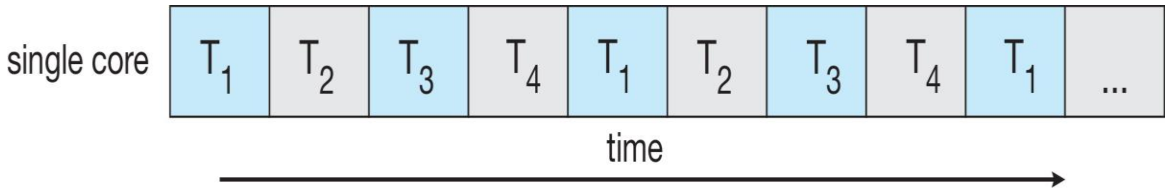
Parallelism

- implies a system can perform more than one task simultaneously
- Execute on multi core system



Concurrency

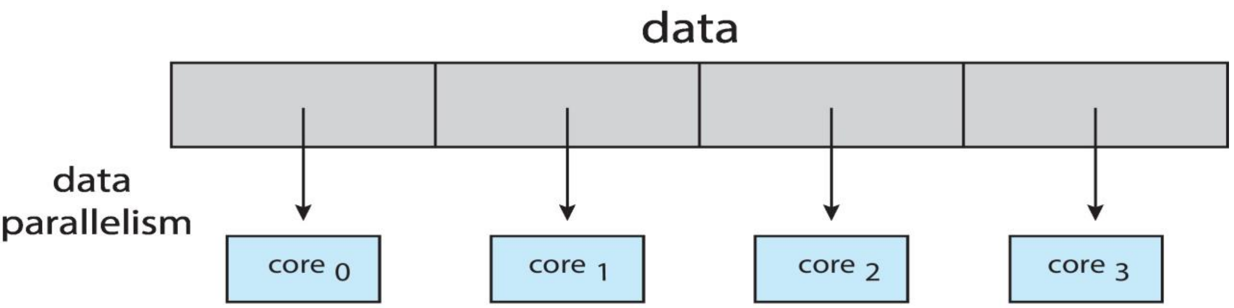
- supports more than one task making progress Single processor / core, scheduler providing concurrency
- Execute on single core System



Multi Core Programming (Cont…)

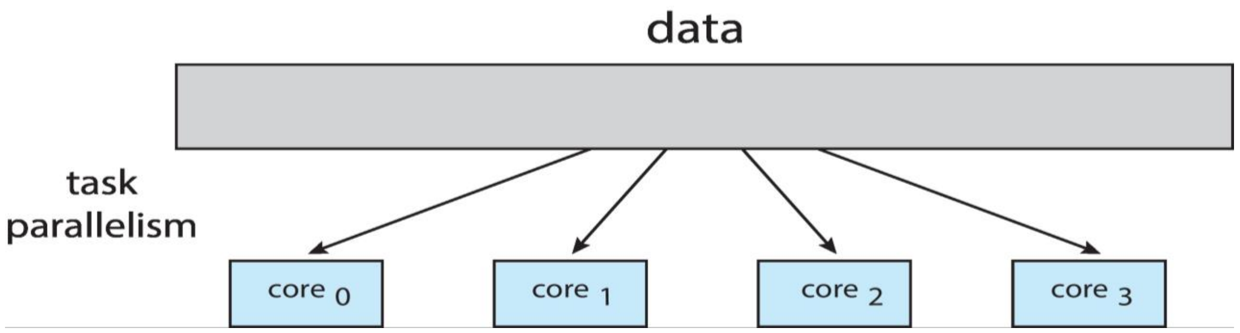
Data Parallelism

distributes **subsets of the same data across multiple cores**, same operation on each



Task Parallelism

distributing threads across cores, each thread **performing unique operation**



Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both **serial and parallel components**
- *S* is serial portion
- *N* processing cores

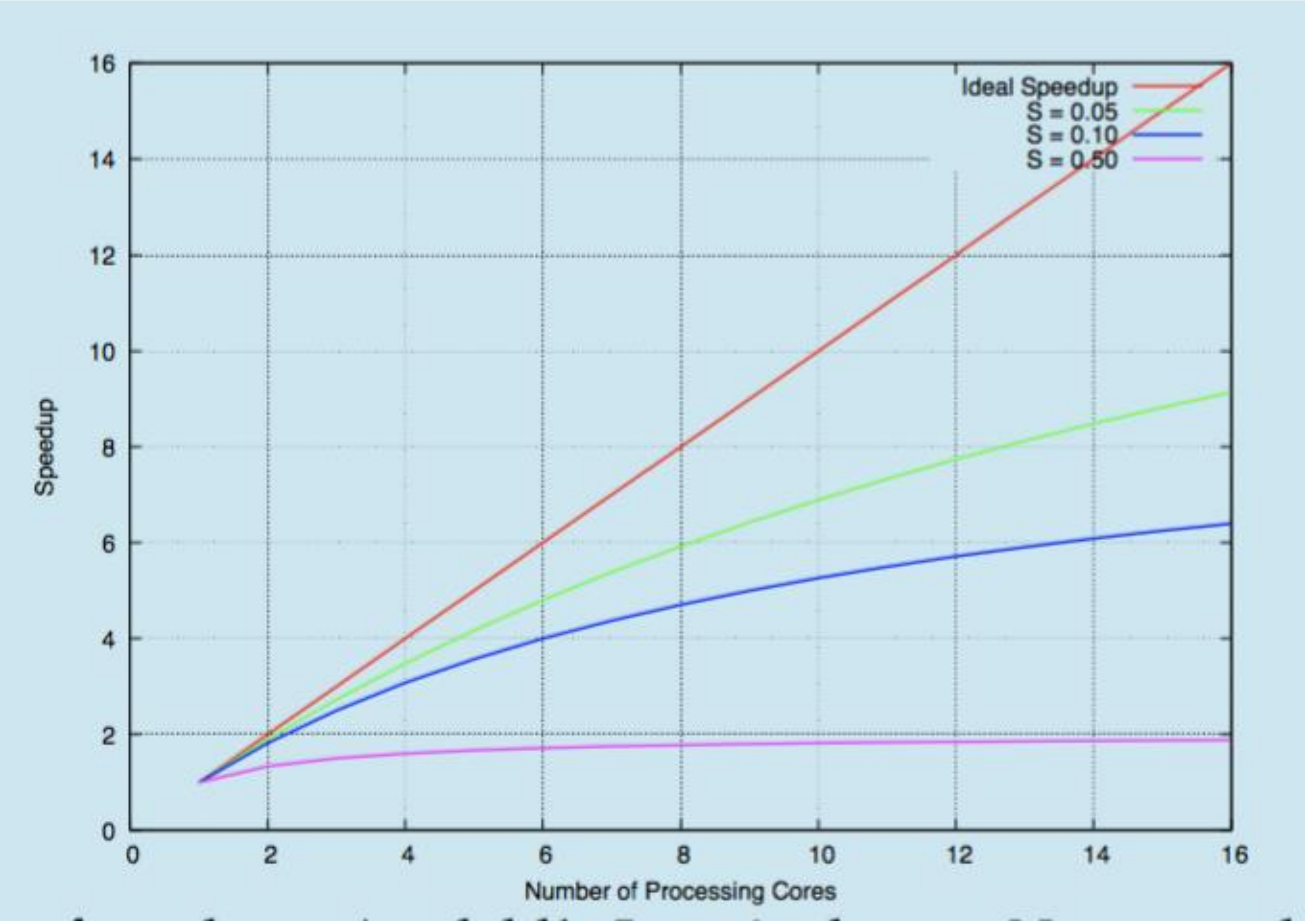
$$speedup \le \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, **if application is 75% parallel** / **25% serial**, moving from 1 to 2 cores results in **speedup of 1.6 times**
- As ***N* approaches infinity**, speedup **approaches 1 / *S***

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?

Amdahl Law’s (Cont…)

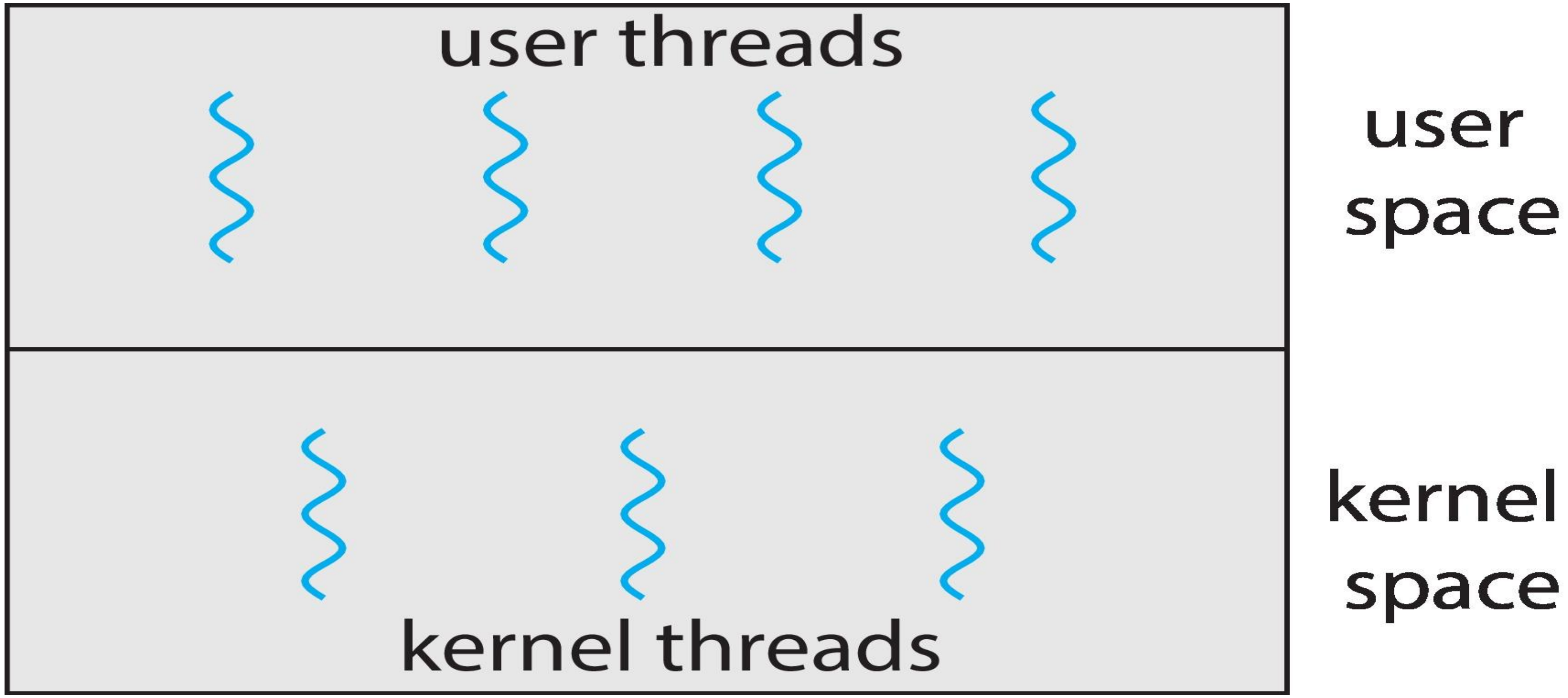


User Threads and Kernel Threads

- **User threads**
  - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel Thread:**

  Supported by the Kernel
  - Examples – virtually all general-purpose operating systems, including:
    - Windows
    - Linux
    - Mac OS X
    - iOS
    - Android

User Threads and Kernel Threads (cont…)



Threading Models

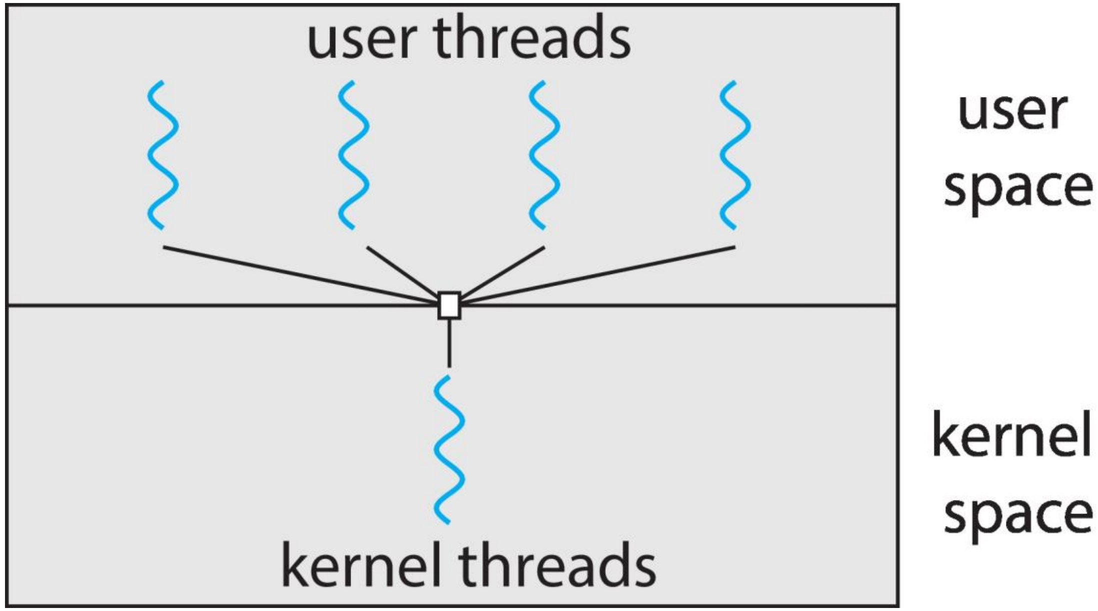
A relationship **must exist between user threads and kernel threads**

- Three common ways of establishing such a relationship:
  - many-to-one model
  - one-to one model
  - many-to-many model

Many to One Model:

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

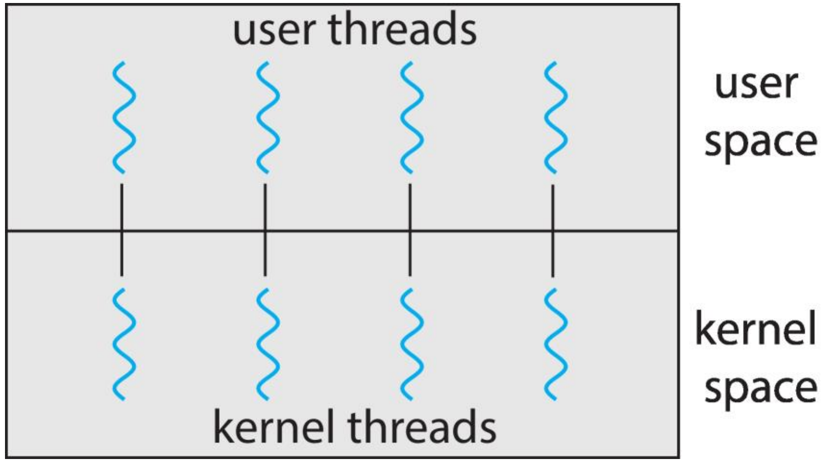
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



One to One Model

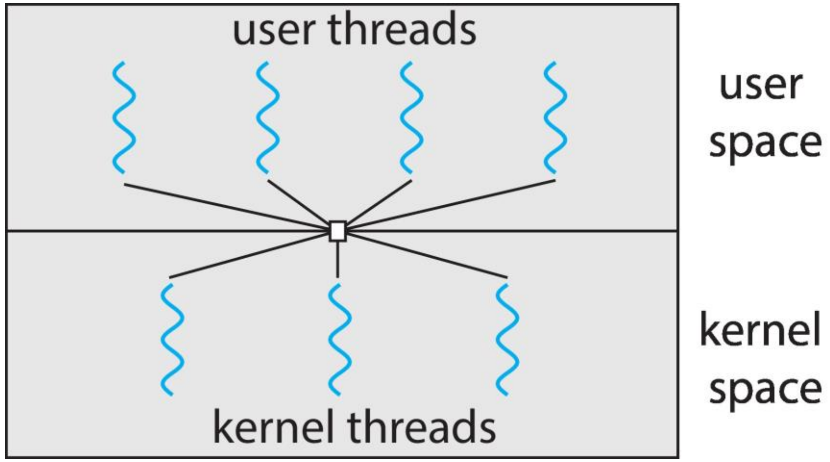
Each user-level thread maps to kernel thread

- 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
- Examples
  - Windows
  - Linux



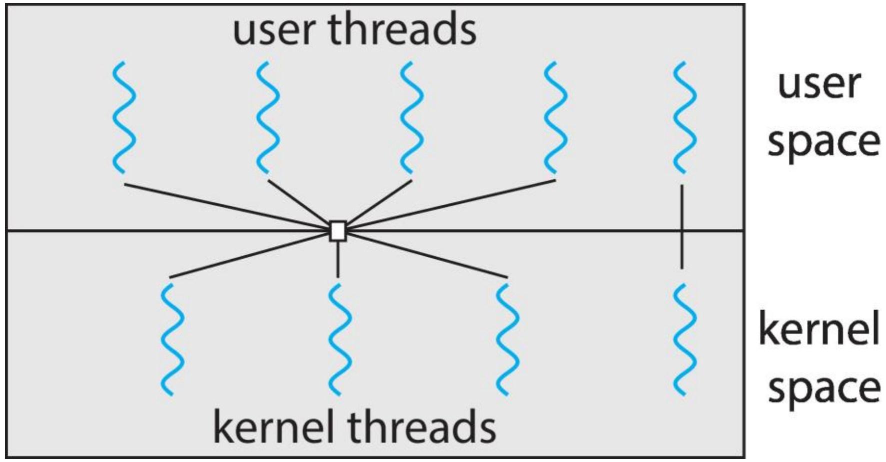
Many to Many Model

- Allows **<u>many user level threads to be mapped to many kernel threads</u>**
- Allows the **<u>operating system to create a sufficient number of kernel threads</u>**
- Windows with the *ThreadFiber* package
- Otherwise not very common



Two Mode Model

- Similar to Many to Many
- except that it allows a user thread to be **bound** to kernel thread



Thread Libraries

- Thread library provides **programmer with API for creating and managing thread**s
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Types of Libraries

- Three main libraries are used
  - POSIX Pthreads
    - The threads extension of the POSIX standard, **may be provided as either a user-level or a kernel-level library**
  - Windows
    - Windows thread library is a **kernel-level library** available on Windows systems.
  - Java
    - Java thread API allows threads to be **created and managed directly** in Java programs.

Pthreads

- A POSIX **standard (IEEE 1003.1c)** API for thread creation and **synchronization**
- Specification, not implementation
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Linux & Mac OS X)

How to create a Thread in C

- **<u>pthread_create</u>**
  - It is used to create a new thread.

```
int pthread_create(pthread_t * thread,
                   const pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void *arg);
```

- Parameters Detail
  - Thread

    Pointer to an unsigned integer value that return the **thread ID** of the thread that is created.
  - Attr

    Pointer to a structure that is used to define the **<u>thread attributes</u>** like state, scheduling policy
  - Start-Routine

    Pointer to subroutine that is **<u>executed</u>** by thread. It return types must be void.
  - Args

    Pointer to void that contain the arguments to the function which defined earlier

Pthreads Code Example:

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

#include <stdlib.h>

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

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

   /* set the default attributes of the thread */
   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);
}
```

Pthread Example

```
/* The thread will execute 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);
}
```

Function of Threads

Function Name	Purpose
Pthread_Create	Used to Create the thread
Pthread_exit	Used to terminate the thread
Pthread_join	Used to wait for termination of thread
Pthread_self	Used to get the id of current running thread
Pthread_cancel	Used to send the cancellation request

Practice Code

Write a program in C by using threading, in which threads prints 0-4 while main process prints 20-24.

Code

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

//Prototyping of thread
Void *thread_print( void *arg)
int i, j;
```

```
int main ( )
{
Pthread_t a; //Thread declaration
Pthread_create(&a, NULL, thread_print, NULL);
Pthread_join(a, NULL);
Printf("print 4 no of given range");
For(j=20;j<25;j++)
{
Printf("%d\n", j);
}
```

```
Void *thread_print( void *arg)
Printf("inside thread");
For(i=1; i<5; i++)
{
Printf("%d\n", i);
}
```

Practice Code

Write a program that creates threads based on the input given by user. Each thread should execute function print () and display its thread ID. The output should be like:

Hello I am thread 1 my ID is 123

Hello I am thread 2 my ID is 234....

The main thread should wait for the child threads to terminate and then call exit.

Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print(void *threadid)
{
pthread_t tid = pthread_self();
int id = *((int *)threadid);
printf("Hello -> I AM THREAD %d AND MY ID IS = %d\n", id, tid);
pthread_exit(NULL);
}
```

```c
int main()
{
int num;
printf("ENTER THE NUMBER OF THREADS TO CREATE = ");
scanf("%d", &num);

pthread_t threads[num];
int threadids[num];
for (int i = 0; i < num; i++)
{
threadids[i] = i + 1;
pthread_create(&threads[i], NULL, print, &threadids[i]);
}
for (int i = 0; i < num; i++)
{
pthread_join(threads[i], NULL);
 }
printf("ALL THREADS HAVE BEEN EXECUTED SUCCESSFULLY...\n");
 exit(0);
 }
```

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - 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** for the process

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
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

  . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

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

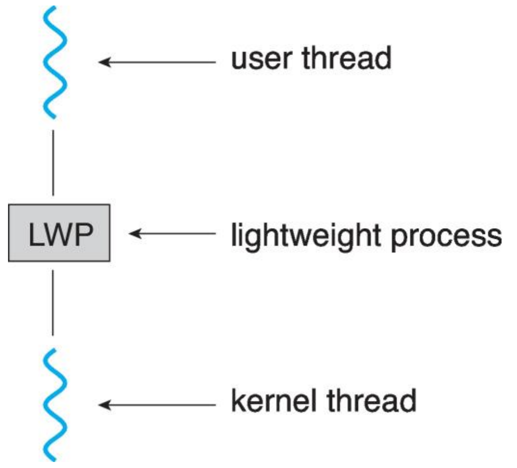
- If thread has **cancellation disabled, cancellation remains pending until** thread enables it
- Default type is **deferred**
  - Cancellation only occurs when thread reaches **cancellation point**
    - i.e., **pthread_testcancel()**
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

Thread-Local Storage

- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- 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

Scheduler Activations

- 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 – lightweight 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



Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **clone()** system call
- clone() allows a child task to share the address space of the parent task (process)
  - Flags control behavior

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- struct task_struct points to process data structures (shared or unique)