CSC 4320/6320: Operating Systems

Georgia State University

Chapter 04: Threads and Concurrency

Spring 2025

Instructor: Dr. Md Mahfuzur Rahman
Department of Computer Science, GSU

Disclaimer

The slides to be used in this course have been created, modified, and adapted from multiple sources:

- *The slides are copyright Silberschatz, Galvin and Gagne, 2018. The slides are authorized for personal use, and for use in conjunction with a course for which Operating System Concepts is the prescribed text. Instructors are free to modify the slides to their taste, as long as the modified slides acknowledge the source and the fact that they have been modified.*

Outline

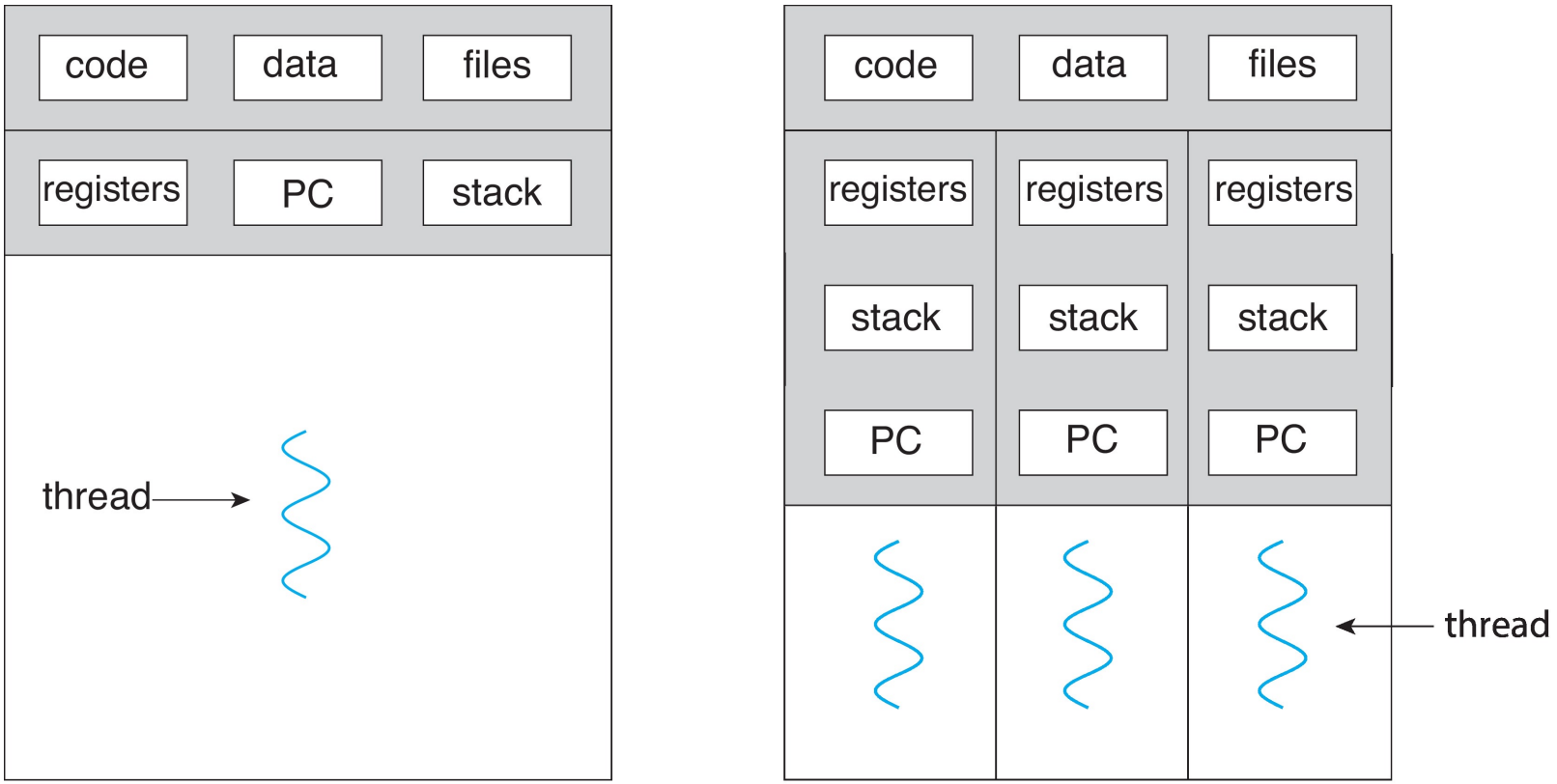
- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Objectives

Georgia State University

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designing multithreaded applications
- Illustrate different approaches to implicit threading including thread pools, fork-join, and Grand Central Dispatch
- Describe how the Windows and Linux operating systems represent threads
- Designing multithreaded applications using the Pthreads, Java, and Windows threading APIs

Single and Multithreaded Processes



single-threaded process

multithreaded process

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

Questions

Q1. A thread is composed of a thread ID, program counter, register set, and heap.

True
False ✓

Q2. Each thread has its own register set and stack.

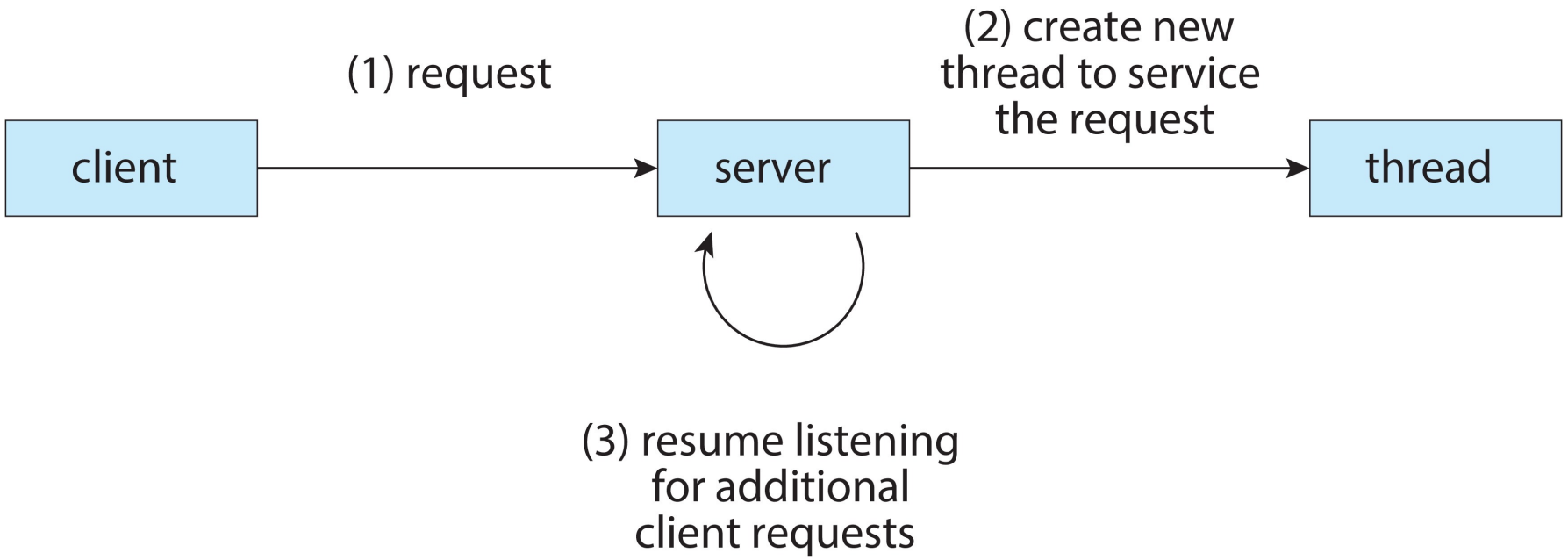
True ✓
False

Motivation

Georgia State University

- 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

Multithreaded Server Architecture



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

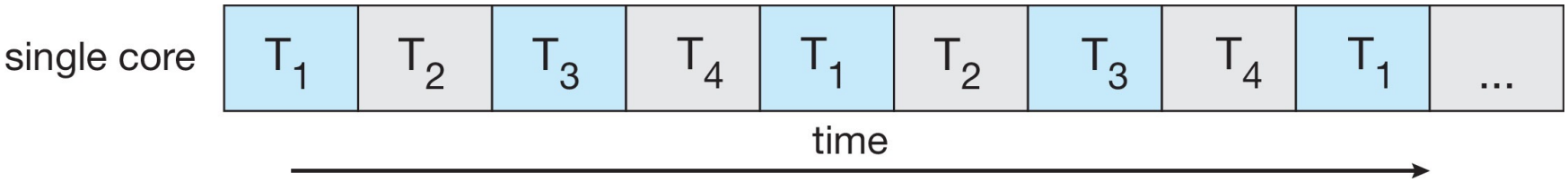
Multicore Programming

Georgia State University

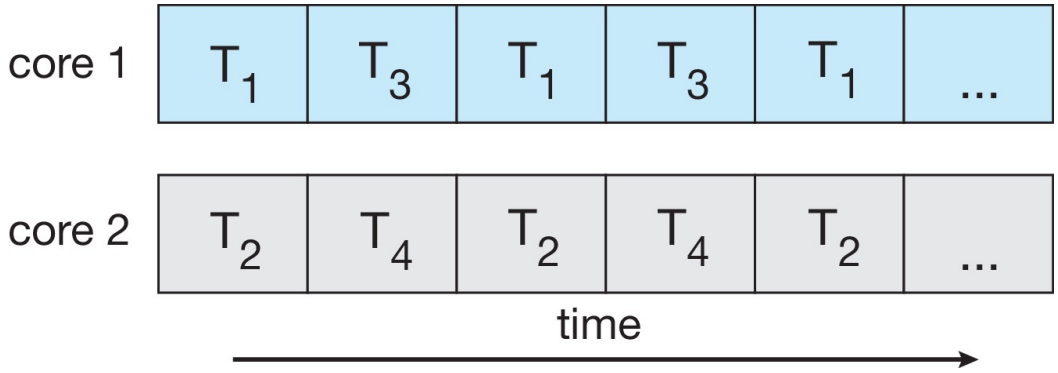
- **Multicore** or **multiprocessor** systems puts pressure on programmers, challenges include:
  - **Dividing activities** (which activities can run in parallel?)
  - **Balance** (are the activities balanced?)
  - **Data splitting** (data must be divided to run in multicores)
  - **Data dependency** (must be examined and handled)
  - **Testing and debugging** (can be harder; knowledge of parallel programming might be helpful)
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**

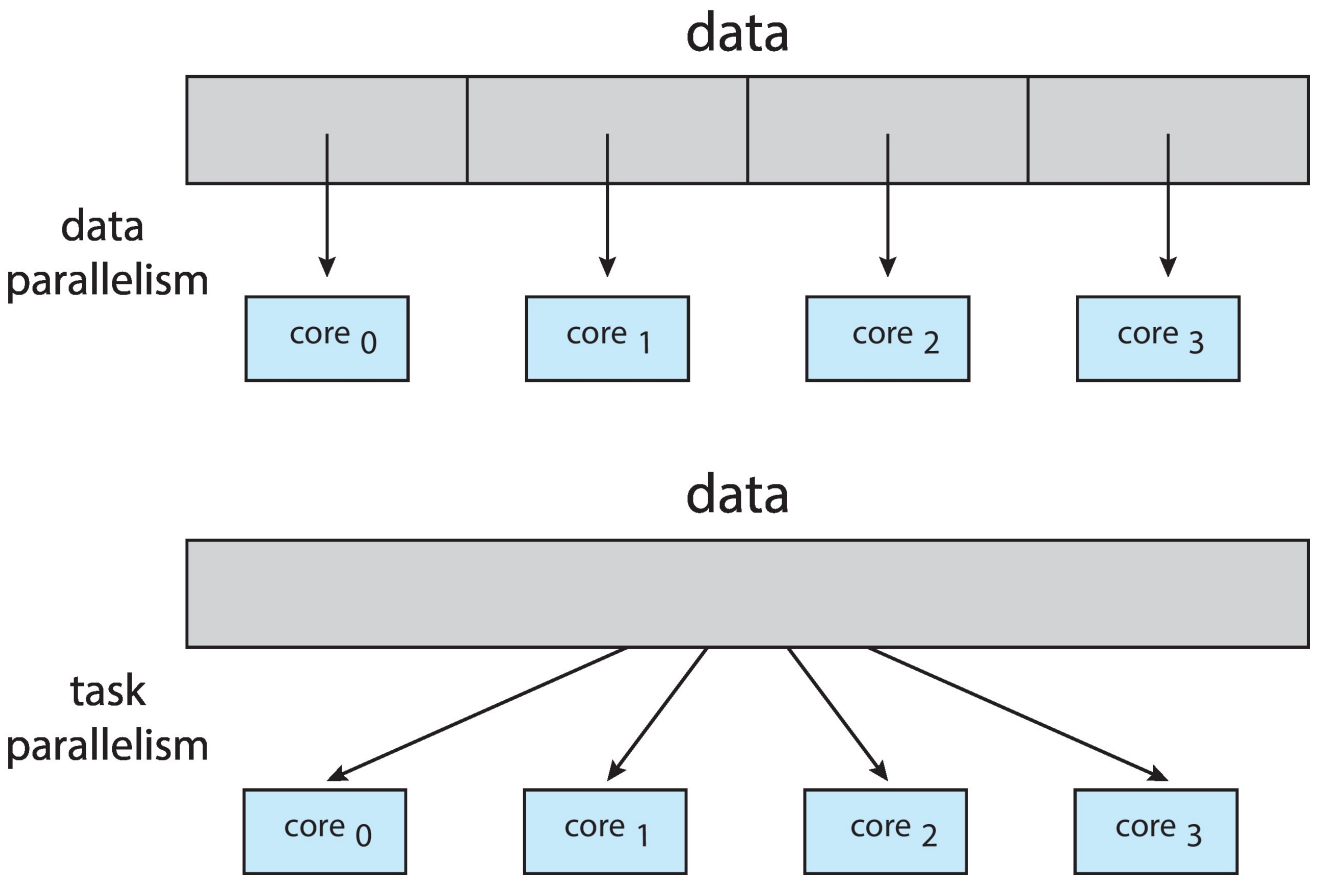


Multicore Programming

Georgia State University

- 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



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

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

- 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.* For example, if 50 percent of an application is performed serially, the maximum speedup is 2.0 times

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

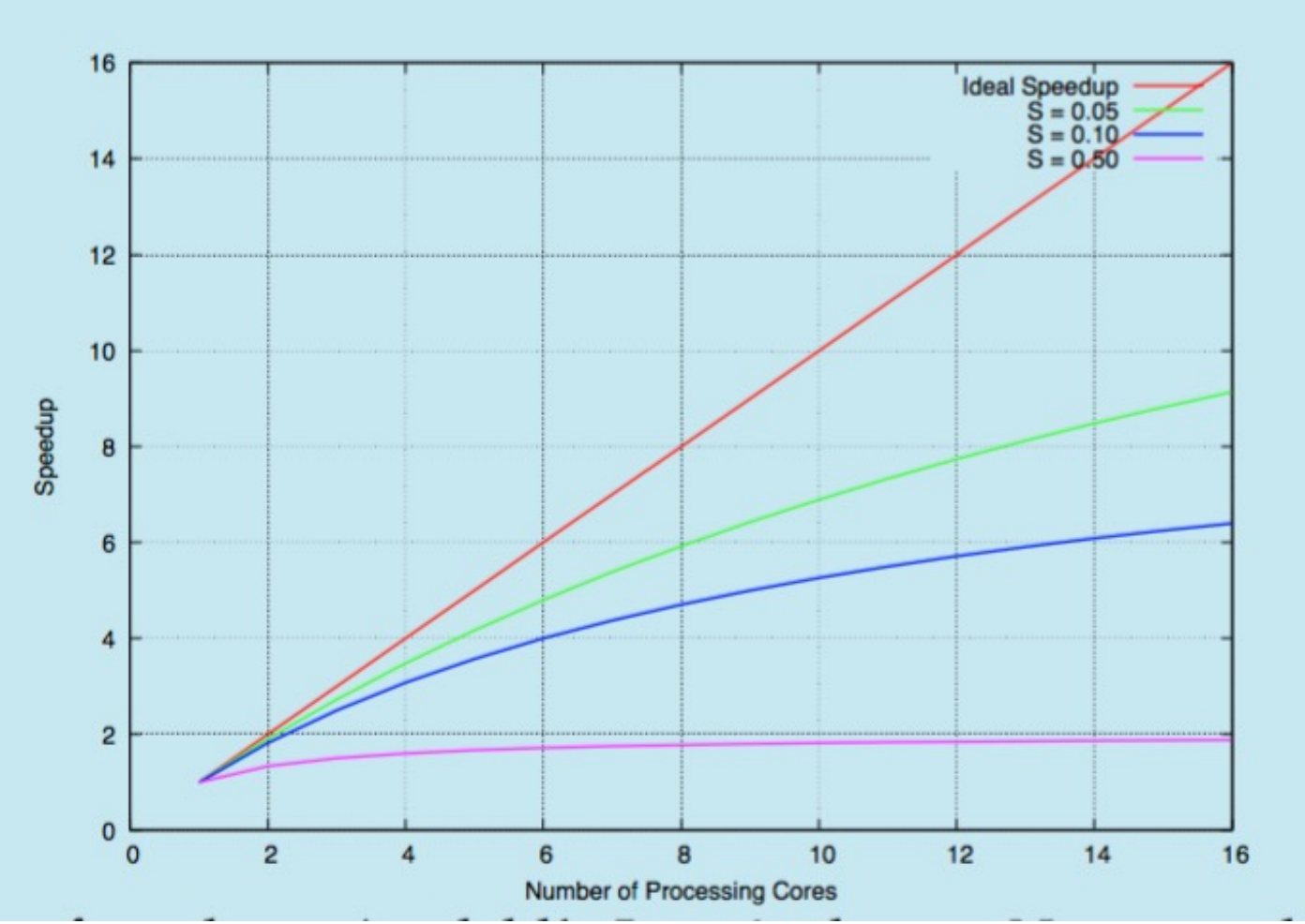
- But does the law take into account contemporary multicore systems?

Calculation

- Let's calculate the speedup
- Speedup = $S(2) = \frac{1}{0.25+\frac{0.75}{2}} = \frac{1}{\frac{1.25}{2}} = \frac{2}{1.25} = 1.6$

- Exercise:
  Suppose a program has 70% of its code that can be parallelized (P=0.7) and the remaining 30% is inherently sequential (1−P=0.3). Calculate the speedup using 4 processors (N=4).

Amdahl's Law


Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16

Questions

Q. _____ involves distributing tasks across multiple computing cores.

A) Concurrency
B) Task parallelism ✓
C) Data parallelism
D) Parallelism

Q. It is possible to have concurrency without parallelism.
True ✓
False

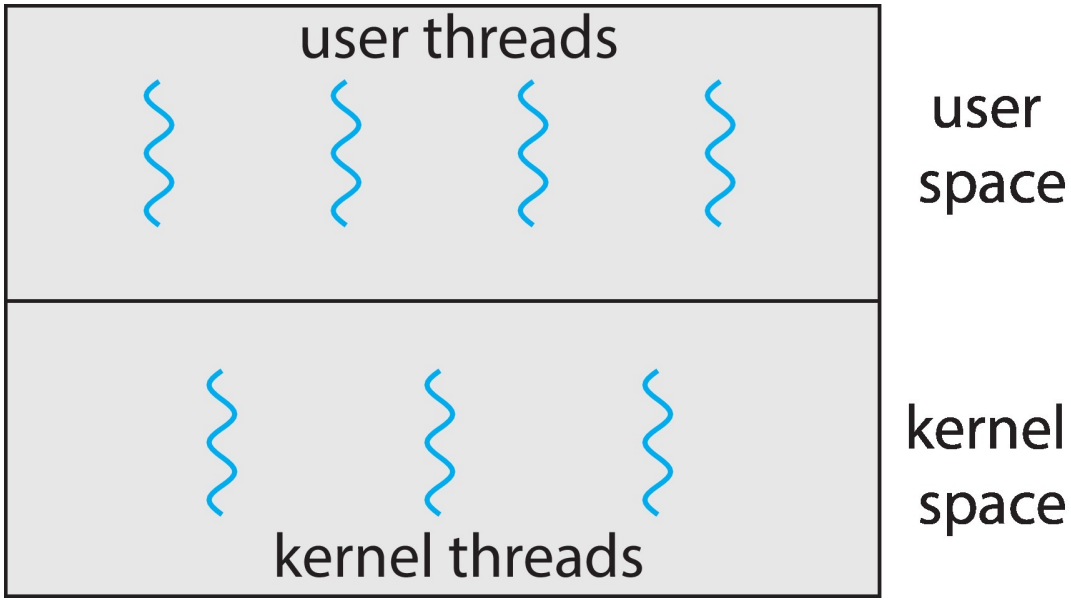
Q. According to Amdahl's Law, what is the speedup gain for an application that is 60 percent serial code. Calculate speed gain for N=4 processors

A) 1.82
B) .7
C) .55
D) 1.43 ✓

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

User and Kernel Threads



User threads are managed without kernel support
kernel threads are supported and managed directly by the OS

Multithreading Models

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

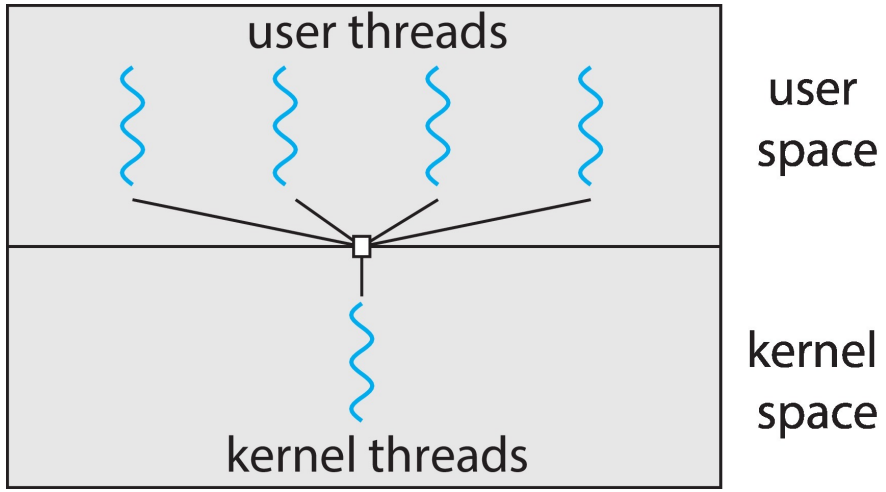
Three common ways of establishing a relationship between user threads and kernel threads

```
Pthreads in Linux/Unix uses 1:1 model
int main() {
    for(int i = 0; i < 10; i++)
        pthread_create(xxx....);
    return 0;
}
```

Many-to-One

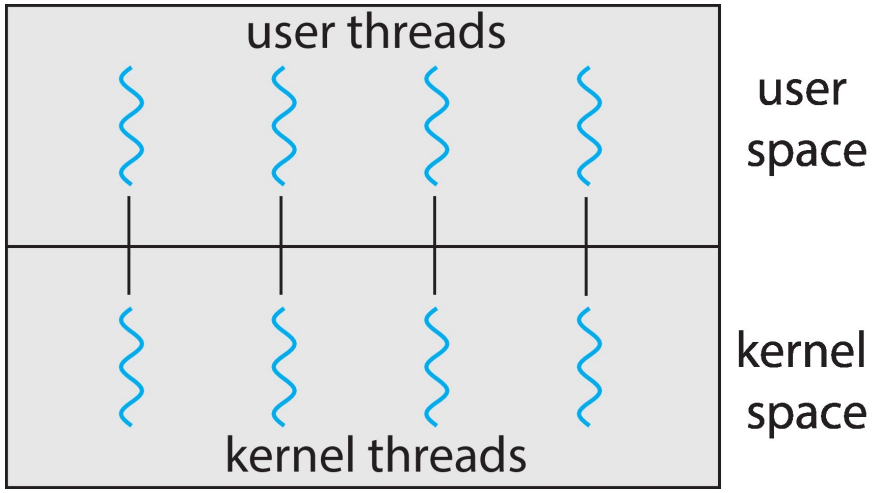
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model because of its inability to take advantage of multiple processing cores
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads (pth.h)**

The kernel can schedule only one kernel thread at a time.



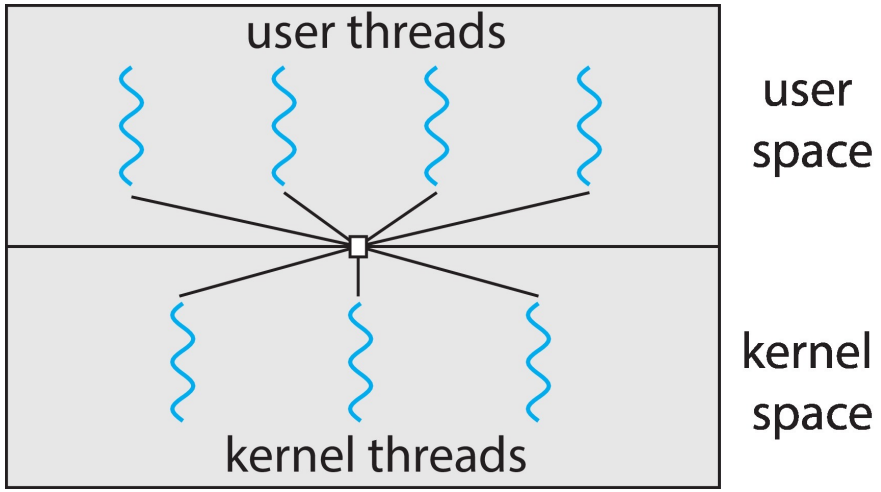
One-to-One

- 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
- Examples
  - Windows
  - Linux



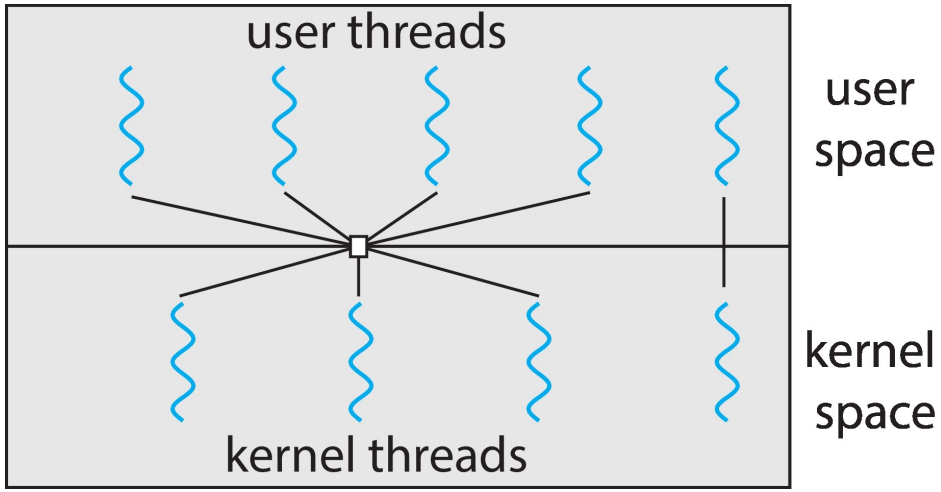
Many-to-Many Model

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



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Many to small and one to one (both combinations)



one can optionally bind a particular user thread to a kernel thread

Questions

The _____ model allows a user-level thread to be bound to one kernel thread.

A) many-to-many
B) two-level ✓
C) one-to-one
D) many-to-one

Thread Libraries

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

It is possible to create a thread library without any kernel-level support.

True ✓
False

Asynchronous vs. Synchronous

Asynchronous threading: once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently and independently.

Synchronous threading: the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes. Children threads perform work concurrently, but the parent cannot continue.

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation*** *(Operating-system designers may implement the specification)*
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Linux & Mac OS X)

Pthreads Example

Task: construct a multithreaded program that calculates the summation up to a non-negative integer in a separate thread.

$$sum = \sum_{i=1}^{N} i$$

```c
#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);
}
```

Pthreads Example (Cont.)

```
/* 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);
}
```

`void*` is a generic pointer, the thread function can return **any data type** by casting it to `void*`. A concrete example of how to use this generic pointer (`void *`) to return an integer is shown in the next slide.

How to use void* to return any data


```c
#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]);

   /* handle return value */
  void *return_value;
  /* wait for the thread to exit */
  pthread_join(tid, &return_value); /*use NULL (if not interested)*/

  int *result = (int *)return_value;
  printf("The returned value:%d\n", *result); /*a dummy ret value*/

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

  free(result);
  return 0;
}
```

```c
/* 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;

  /* do the following to return a local variable */
  int* result = malloc(sizeof(int));/*heap will be alive*/
  *result = 42;  /* e.g., store an integer value

  /* We can use both ways to return values */
  // return (void*)result; /*one way to return */
  pthread_exit((void *)result); /*another way to return */

}
```

Pthreads Code for Joining 10 Threads

A simple method for waiting on several threads using the `pthread_join()` function is to enclose the operation within a simple for loop

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
  pthread_join(workers[i], NULL);
```