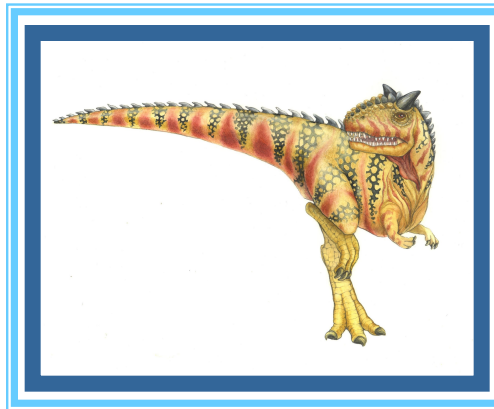


Chapter 4: Threads & Concurrency





Outline

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





Objectives

- 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
- Design multithreaded applications using the Pthreads, Java, and Windows threading APIs





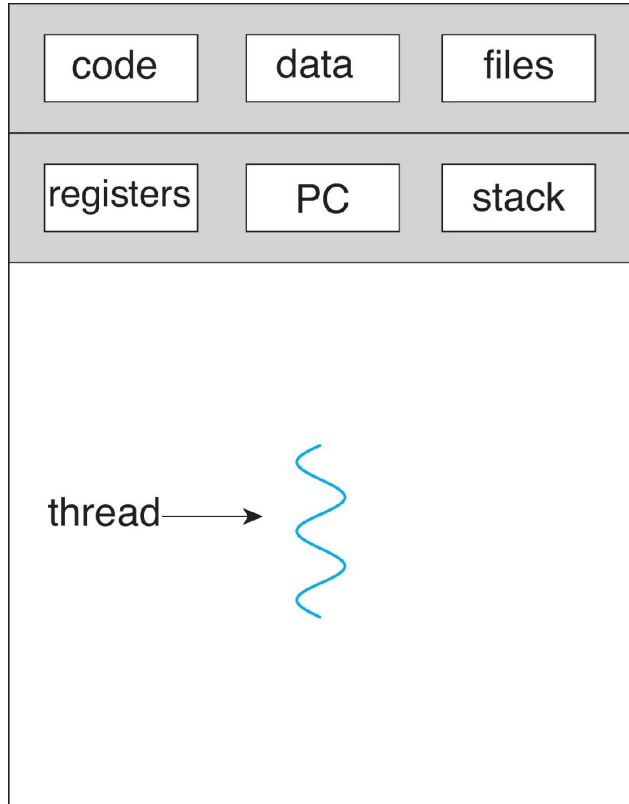
Motivation

- 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

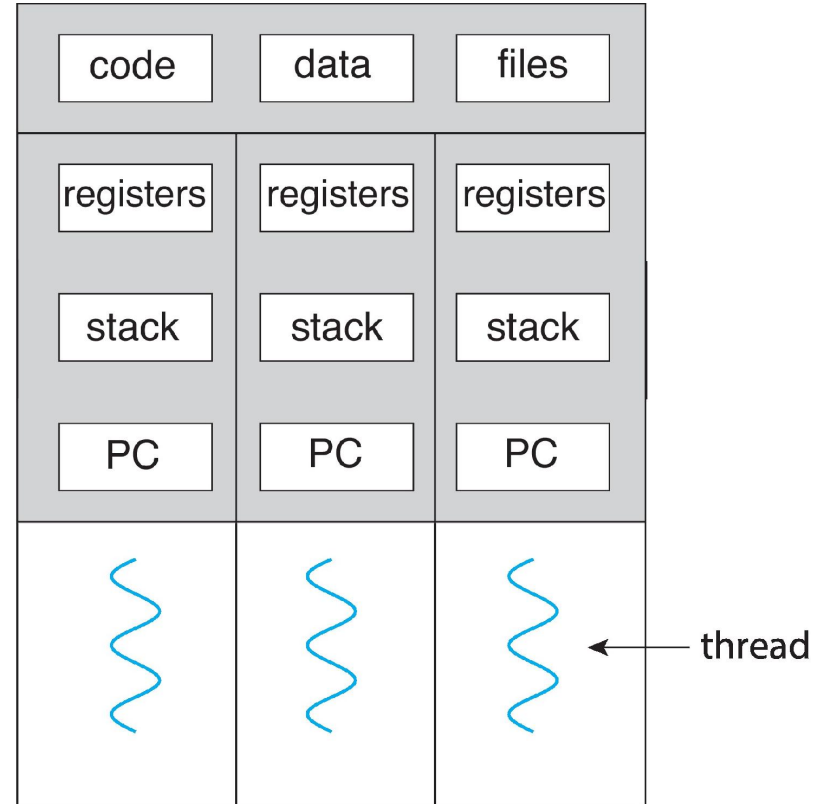




Single and Multithreaded Processes



single-threaded process

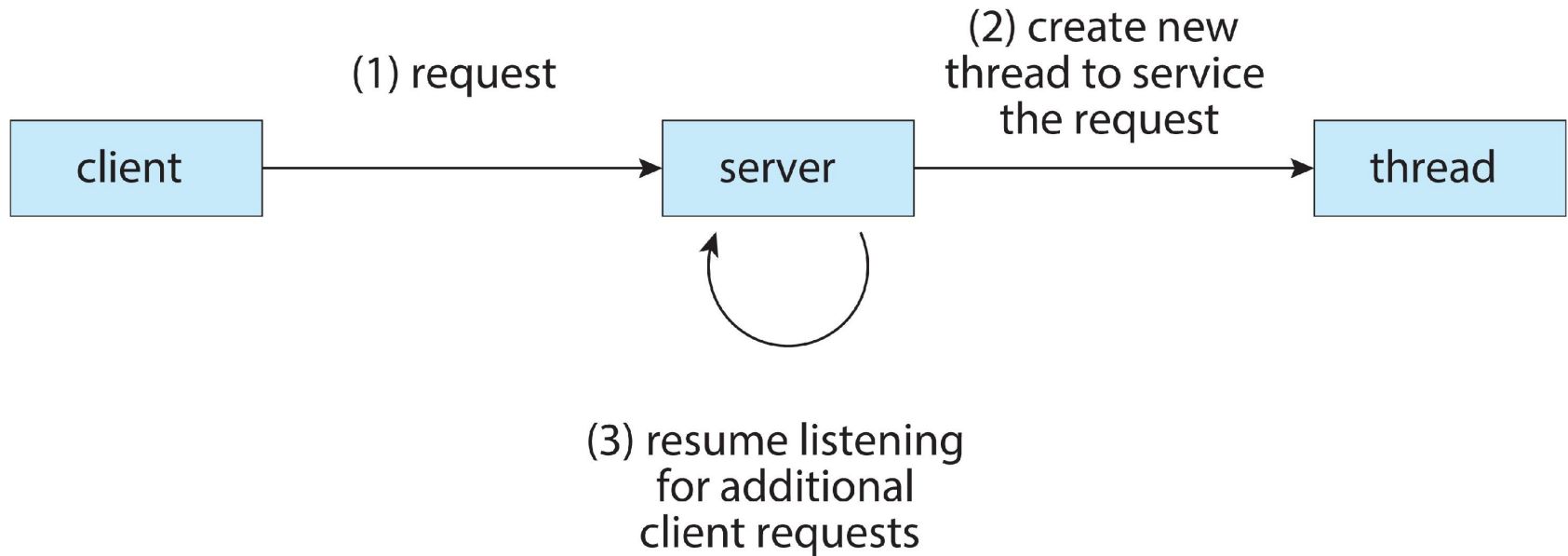


multithreaded process





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

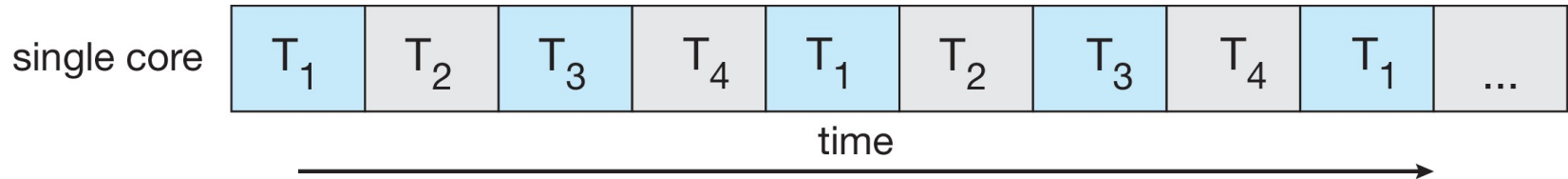
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
 - Single processor / core, scheduler providing concurrency



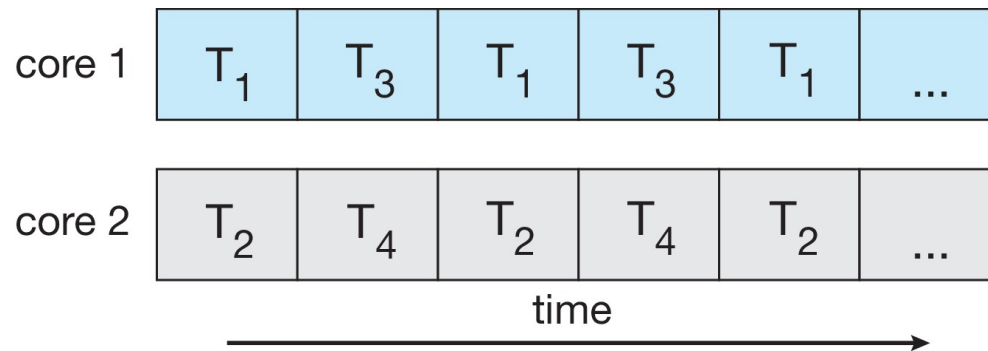


Concurrency vs. Parallelism

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



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





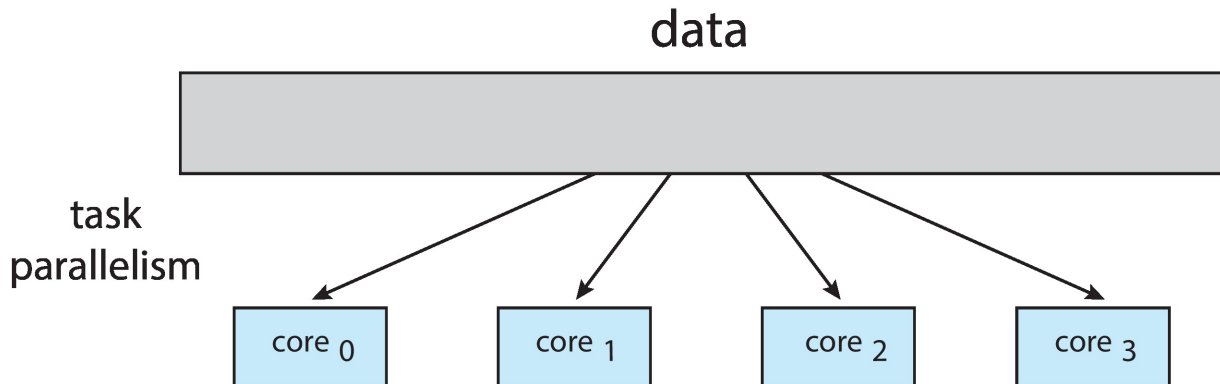
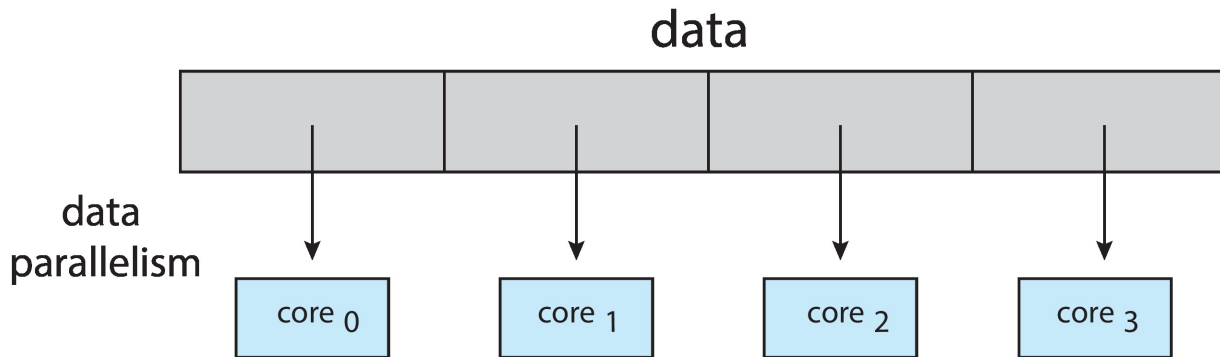
Multicore Programming

- 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





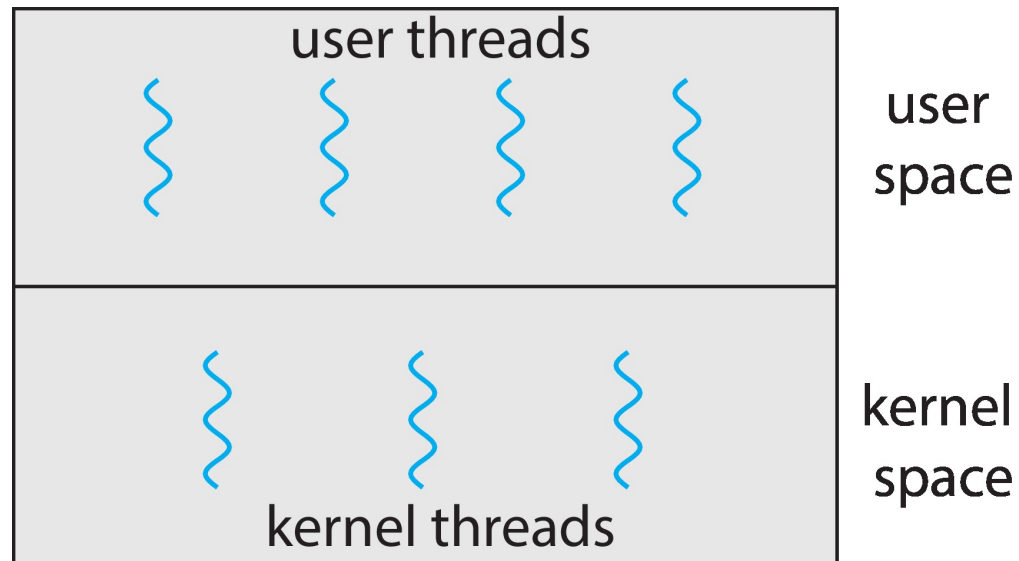
User Threads and Kernel Threads

- **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





Multithreading Models

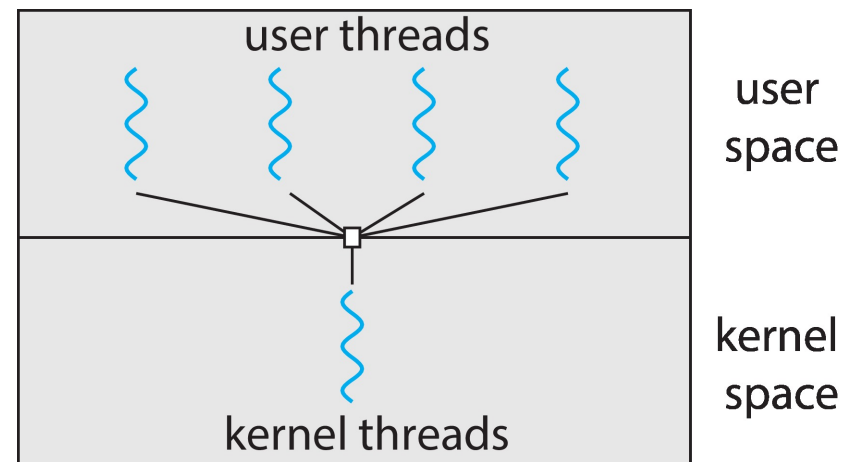
- Many-to-One
- One-to-One
- Many-to-Many





Many-to-One

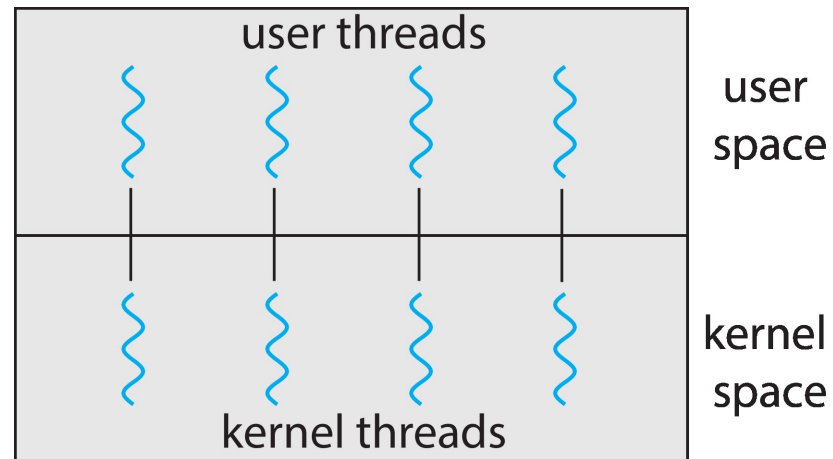
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
 - **Solaris Green Threads**
 - **GNU Portable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





One-to-One

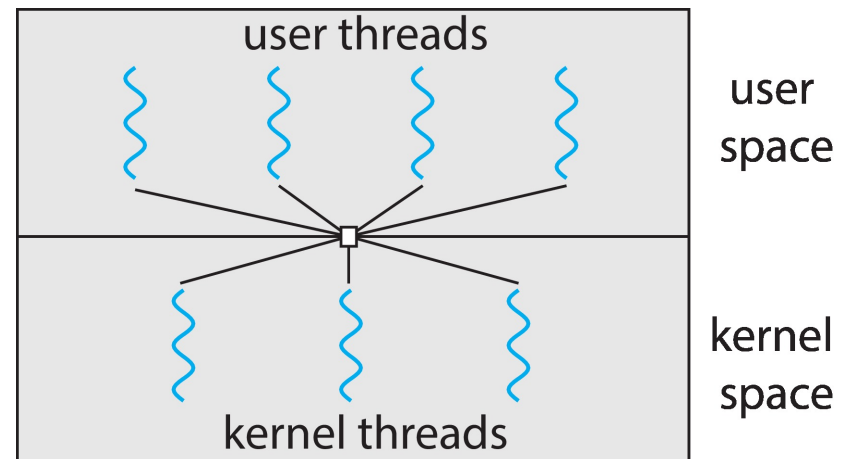
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows
 - Linux





Many-to-Many Model

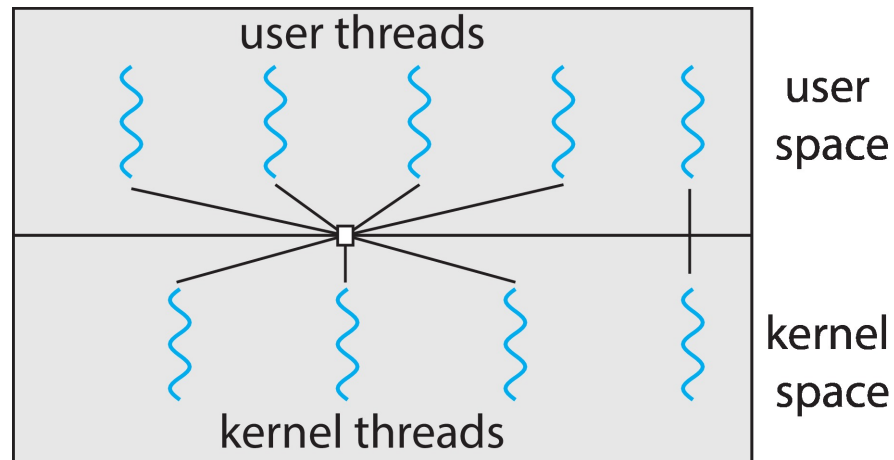
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Otherwise not very common





Two-level Model

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





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





Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Five methods explored
 - Thread Pools
 - Fork-Join
 - OpenMP
 - Grand Central Dispatch (GCD)
 - Intel Threading Building Blocks (ITBB)





Thread Pools

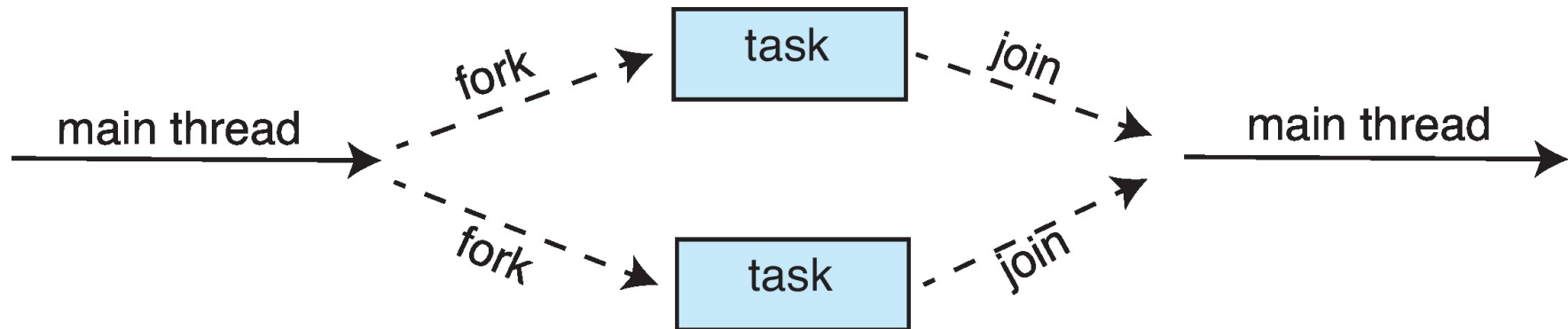
- Create a number of threads in a **pool** where they await work
- Advantages:
 - Usually **slightly faster to service** a request with an existing thread than create a new thread
 - Allows the **number of threads** in the application(s) to be bound to the size of the pool
 - **Separating task** to be performed from mechanics of creating task allows different strategies for running task
 - 4 i.e., Tasks could be scheduled to run periodically
- **Windows API** supports thread pools





Fork-Join Parallelism

- Multiple threads (tasks) are **forked**, and then **joined**.





OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

#pragma omp parallel

Create as many threads as there are cores

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

int main(int argc, char *argv[])
{
    /* sequential code */

    #pragma omp parallel
    {
        printf("I am a parallel region.");
    }

    /* sequential code */

    return 0;
}
```





Run the Loop in Parallel

- Run the for loop in parallel

```
#pragma omp parallel for  
for (i = 0; i < N; i++) {  
    c[i] = a[i] + b[i];  
}
```





Grand Central Dispatch

- Apple technology for macOS and iOS operating systems
- Extensions to C, C++ and Objective-C languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in “`^ { }`” :

```
^ { printf("I am a block"); }
```

- Blocks placed in dispatch queue
 - Assigned to available thread in thread pool when removed from queue





Grand Central Dispatch

- Two types of dispatch queues:
 - **serial** – blocks removed in FIFO order, queue is per process, called **main queue**
 - 4 Programmers can create additional serial queues within program
 - **concurrent** – removed in FIFO order but several may be removed at a time





Intel Threading Building Blocks (TBB)

- Template library for designing parallel C++ programs
- A serial version of a simple for loop

```
for (int i = 0; i < n; i++) {  
    apply(v[i]);  
}
```

- The same for loop written using TBB with **parallel_for** statement:

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```





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





Semantics of `fork()` and `exec()`

- Does **`fork()`** duplicate only the calling thread or all threads?
 - Some UNIXes have two versions of `fork`
- **`exec()`** usually works as normal – replace the running process including all threads





Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
 1. Signal is generated by particular event
 2. Signal is delivered to a process
 3. Signal is handled by one of two signal handlers:
 1. default
 2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
 - **User-defined signal handler** can override default
 - For single-threaded, signal delivered to process





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





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**
- On Linux systems, thread cancellation is handled through signals





Thread-Local Storage

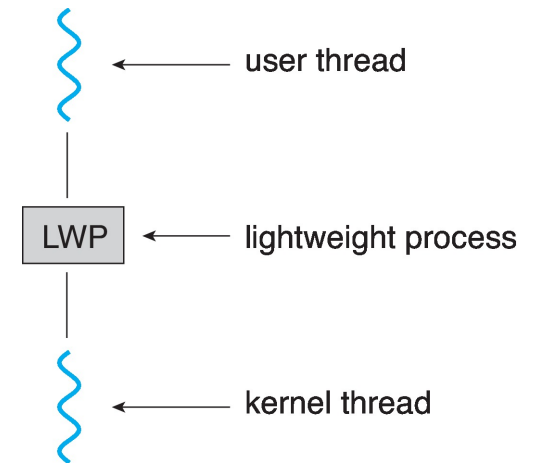
- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
 - Local variables visible only during single function invocation
 - TLS visible across 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



End of Chapter 4

