Operating Systems
[4. Threads & Concurrency]

Chung-Wei Lin
cwlin@csie.ntu.edu.tw
CSIE Department
National Taiwan University

From Operating System Concepts (10th Edition)

Objectives

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designing multithreaded applications
- Design multithreaded applications using the Pthreads, Java, and Windows threading APIs
- 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

Outline

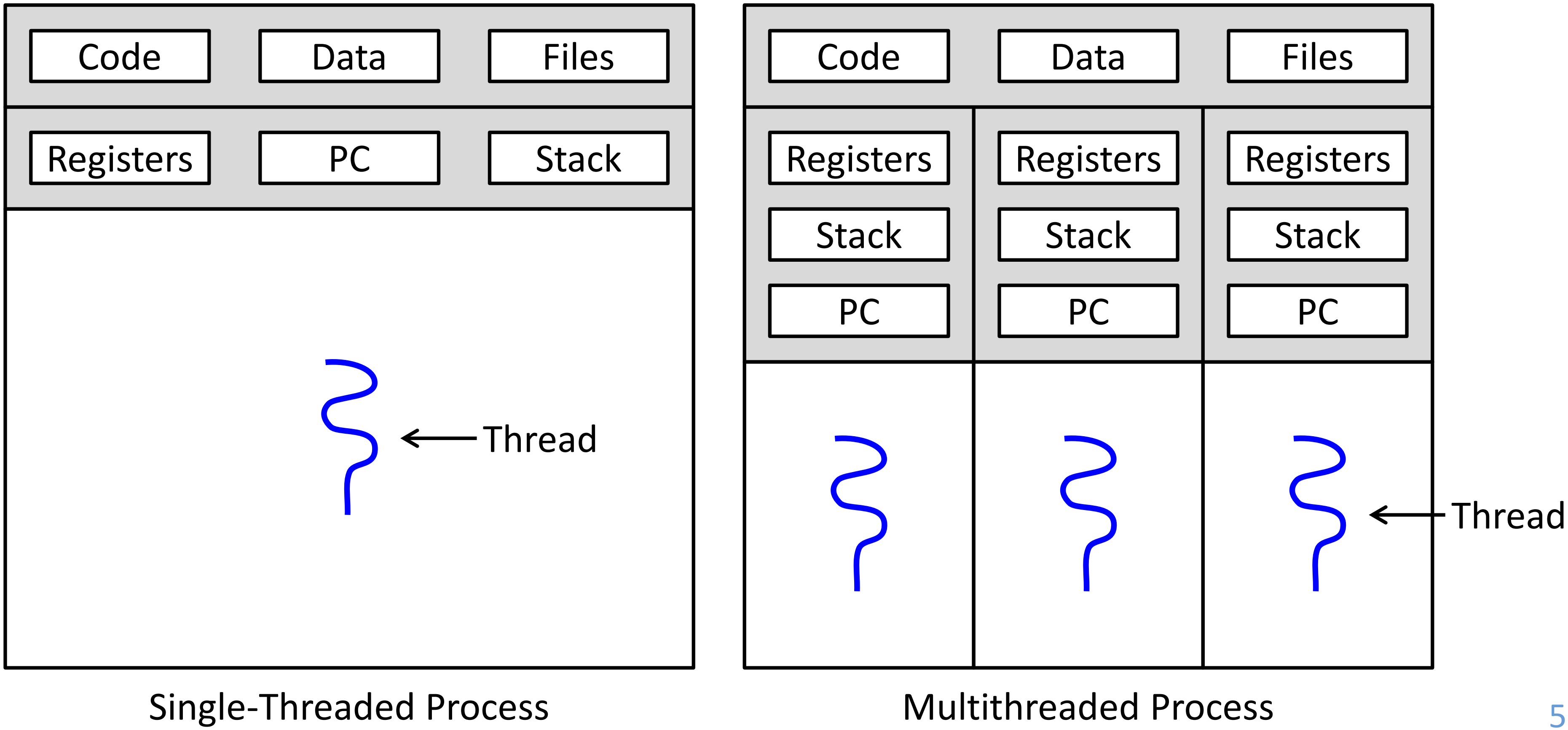
- **Overview**
  - Motivation
  - Benefits
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Overview (1/2)

- A thread is a basic unit of CPU utilization
  - A thread comprises
    - A thread ID
    - A program counter (PC)
    - A register set
    - A stack
  - A thread shares the following items with other threads belonging to the same process
    - Code section
    - Data section
    - Other operating-system resources, such as open files and signals
- Thread [Wikipedia]
  - The smallest sequence of programmed instructions that can be managed independently by a scheduler

Overview (2/2)

- 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



Motivation

- Most software applications that run on modern computers and mobile devices are multithreaded
  - Example: web browser
    - A thread displays images or text
    - A thread retrieves data from the network
  - Example: word processor
    - A thread displays graphics
    - A thread responds to keystrokes from the user
    - A thread performs spelling and grammar checking
  - Example: CPU-intensive tasks in parallel across the multiple cores
  - Example: web server which has several clients concurrently accessing it
- Process creation is time consuming and resource intensive
  - Using one process containing multiple threads is usually more efficient
- Most OS kernels are also typically multithreaded

Benefits

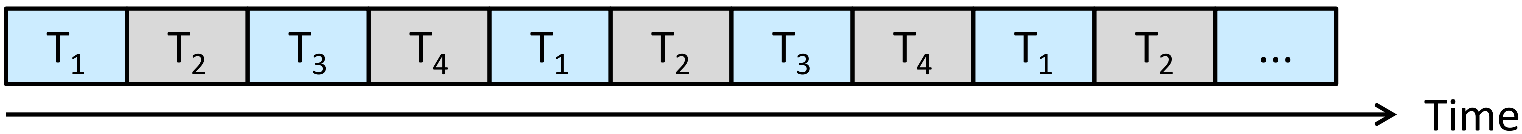
- **Responsiveness**
  - Multithreading allows a program to continue running even if part of it is blocked or is performing a lengthy operation
    - Especially useful in designing user interfaces
- **Resource sharing**
  - Processes can share resources only through techniques which must be explicitly arranged by the programmer
- **Economy**
  - It is more economical to create and context-switch threads
- **Scalability**
  - Threads may be running in parallel on different processing cores

Outline

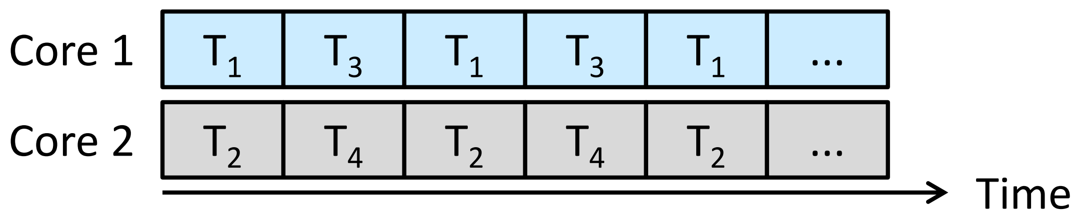
- Overview
- **Multicore Programming**
  - Programming Challenges
  - Types of Parallelism
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Multicore Programming

- A system with a single computing core
  - Concurrency means that threads run interleavingly over time



- A system with multiple cores
  - Concurrency means that some threads can run in parallel



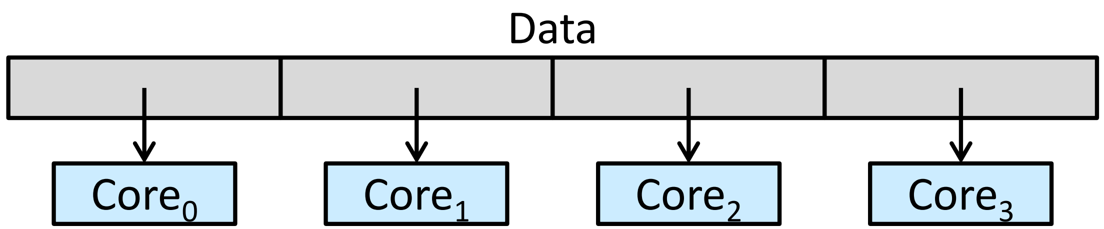
- Concurrency and parallelism
  - A concurrent system supports more than one task by allowing all the tasks to make progress
  - A parallel system can perform more than one task simultaneously
  - It is possible to have concurrency without parallelism

Programming Challenges

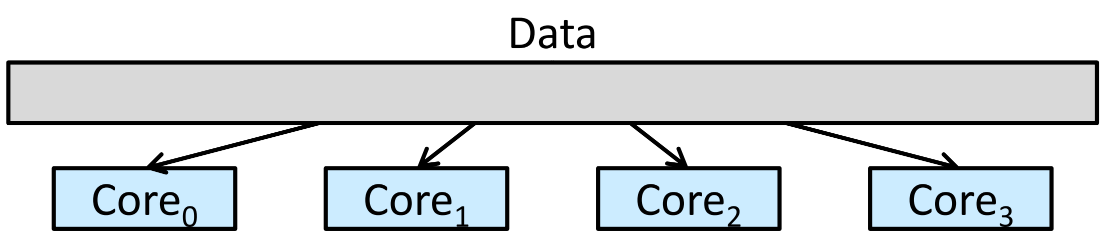
- The trend toward multicore systems continues to place pressure on system designers and application programmers
  - Identifying tasks
    - Find areas that can be divided into separate and concurrent tasks
  - Balance
    - Ensure that tasks perform equal work of equal value (or appropriate work)
  - Data splitting
    - Divide the data accessed and manipulated by tasks to run on separate cores
  - Data dependency
    - Examine the data accessed by tasks for dependencies between the tasks
  - Testing and debugging
    - Test and debug as there are many different execution paths are possible

Types of Parallelism

- Data parallelism
  - Distribute subsets of the same data across multiple computing cores and perform the same operation on each core



- Task parallelism
  - Distribute not data but tasks (threads) across multiple computing cores
    - Each thread is performing a unique operation



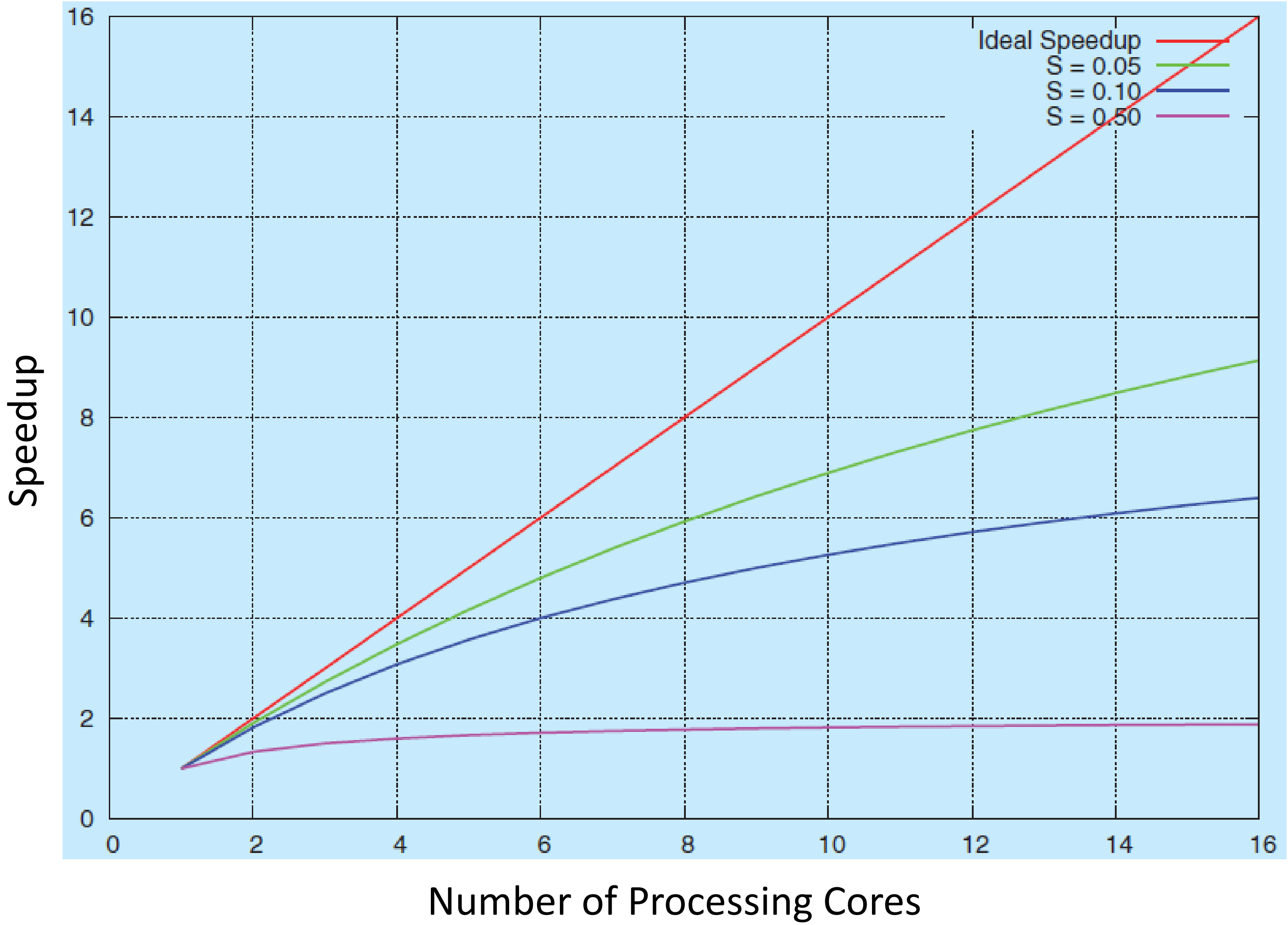
- They are not mutually exclusive

Amdahl's Law (1/2)

- Identify potential performance gains from adding additional computing cores to an application that has
  - Serial (nonparallel) components
  - Parallel components
- Speedup $\leq 1 / \left(S + \frac{1 - S}{N} \right)$
  - S: the portion of the application that must be performed serially
  - N: the number of processing cores
- Example
  - An application that is 75 percent parallel and 25 percent serial
  - If we run this application on a system with two processing cores, we can get a speedup of 1.6 times
- Serial portion of an application has disproportionate effect on performance gained by adding additional cores

Amdahl's Law (2/2)

- Amdahl's Law in several different scenarios



Outline

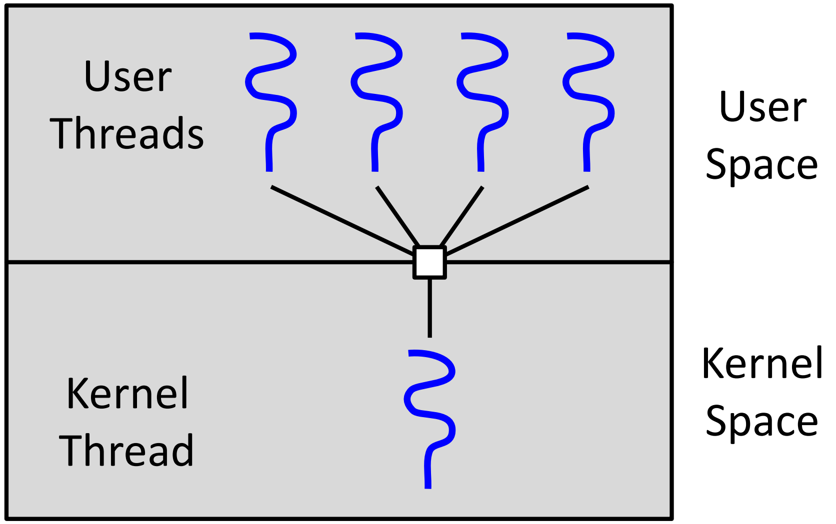
- Overview
- Multicore Programming
- **Multithreading Models**
  - Many-to-One Model
  - One-to-One Model
  - Many-to-Many Model
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

User Threads and Kernel Threads

- Support for threads may be provided either
  - At the user level, for **user threads**, or
  - By the kernel, for **kernel threads**
- User threads
  - Supported above the kernel and managed without kernel support
    - Primary thread libraries: POSIX Pthreads, Windows threads, Java threads
- Kernel threads
  - Supported and managed directly by the operating system
    - Virtually all contemporary operating systems support kernel threads: Windows, Linux, macOS, iOS, Android
- Ultimately, a relationship must exist between user threads and kernel threads

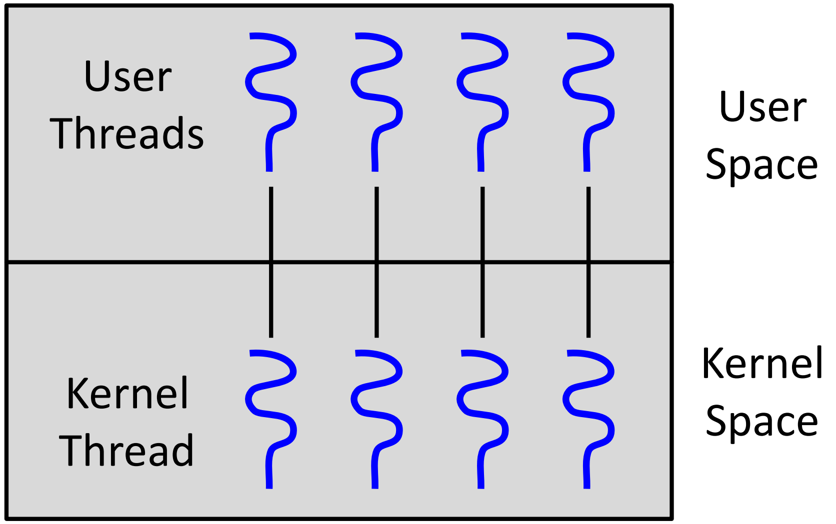
Many-to-One Model

- Map many user-level threads to one kernel thread
  - Thread management is done by the thread library in user space
    - Therefore, it is efficient
  - The entire process will block if a thread makes a blocking system call
  - Only one thread can access the kernel at a time
    - Multiple threads are unable to run in parallel on multicore systems
  - Very few systems continue to use the model



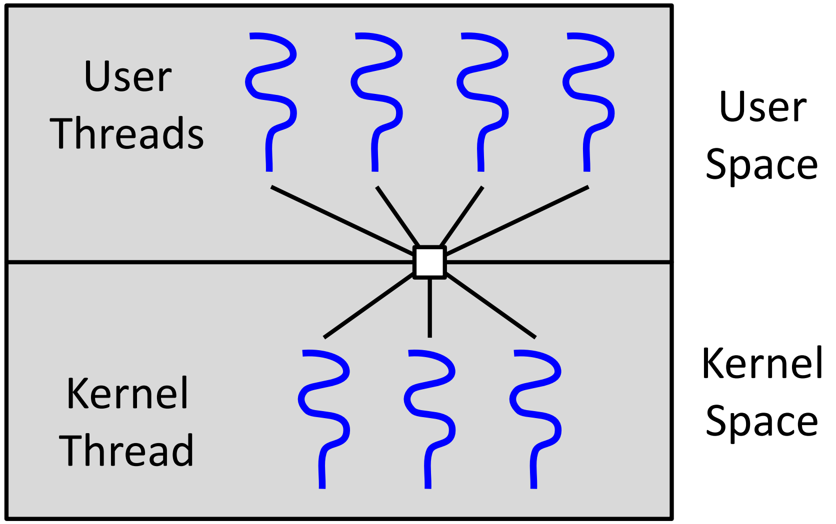
One-to-One Model

- Map each user thread to a kernel thread
  - Other threads can run when a thread makes a blocking system call
  - Multiple threads can run in parallel on multiprocessors
  - Creating a user thread requires creating the corresponding kernel thread
    - A large number of kernel threads may burden the performance of a system
  - Linux, along with the family of Windows, implement this



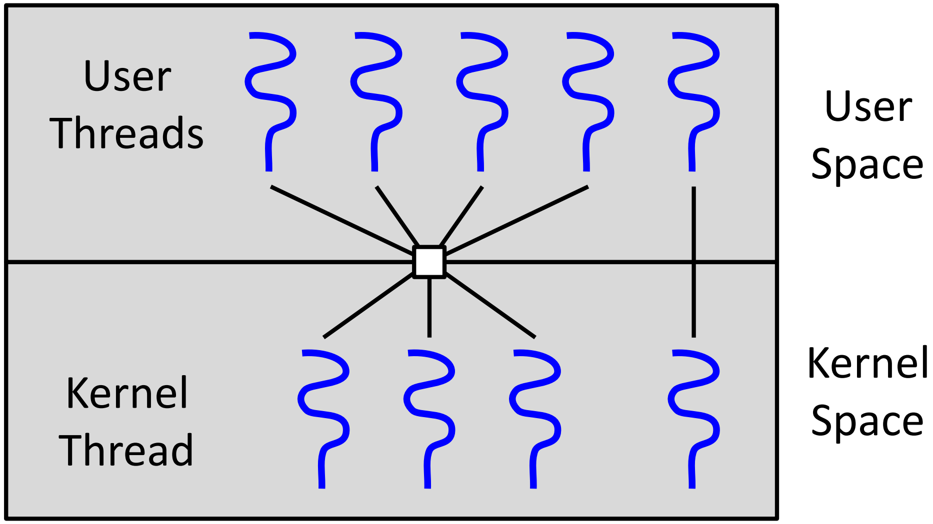
Many-to-Many Model (1/2)

- Multiplex many user-level threads to a smaller or equal number of kernel threads
  - The number of kernel threads may be specific to either a particular application or a particular machine
  - Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor
    - Other threads can run when a thread makes a blocking system call



Many-to-Many Model (2/2)

- Two-level model, one variation on the many-to-many model
  - Multiplex many user level threads to a smaller or equal number of kernel threads
  - Also allow a user-level thread to be bound to a kernel thread



- The many-to-many model is the most flexible here
  - It is difficult to implement
- There is an increasing number of processing cores appearing on most systems
  - Limiting the number of kernel threads has become less important
  - As a result, most operating systems now use the one-to-one model

Threading Models [Wikipedia]

- Thread [Wikipedia]
  - The smallest sequence of programmed instructions that can be managed independently by a scheduler
- Many-to-one model (user-level threading)
  - All application-level threads map to one kernel-level scheduled entity
- One-to-one model (kernel-level threading)
  - Threads created by the user in a 1:1 correspondence with schedulable entities in the kernel
- Many-to-many model (hybrid threading)
  - Map M application threads onto N kernel entities (or virtual processors)

Outline

- Overview
- Multicore Programming
- Multithreading Models
- **Thread Libraries**
  - Pthreads
  - Windows Threads
  - Java Threads
- Implicit Threading
- Threading Issues
- Operating System Examples

Thread Libraries (1/2)

- A **thread library** provides the programmer with an API for creating and managing threads
- Approach 1
  - Provide a library entirely in user space with no kernel support
    - Code and data structures for the library exist in user space
    - Invoking a function in the library results in a local function call in user space, not a system call
- Approach 2
  - Implement a kernel-level library supported directly by the operating system
    - Code and data structures for the library exist in kernel space
    - Invoking a function in the API for the library typically results in a system call to the kernel

Thread Libraries (2/2)

- Running example: summation from 1 to N
  - Input N = 5
  - Output sum = 15
- **Asynchronous threading**
  - Once the parent creates a child thread, the parent resumes its execution
    - The parent and child execute concurrently and independently of one another
- **Synchronous threading**
  - The parent thread creates one or more children and then must wait for all of its children to terminate before it resumes
    - The threads created by the parent perform work concurrently, but the parent cannot continue until this work has been completed
- All of the following examples use synchronous threading

Outline

- Overview
- Multicore Programming
- Multithreading Models
- **Thread Libraries**
  - **Pthreads**
  - Windows Threads
  - Java Threads
- Implicit Threading
- Threading Issues
- Operating System Examples

Pthreads

- A POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization
  - This is a **specification** for thread behavior, not an implementation
    - Numerous systems implement the Pthreads specification
    - Most are UNIX-type systems, including Linux and macOS
  - Threads may be provided as either a user-level or a kernel-level library

Pthreads API

```
#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);
}
/* 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);
}
```

Pthreads API

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

- **`pthread_attr_init()`** sets the attributes
  - Each thread has a set of attributes, including stack size and scheduling information
  - We use the default attributes provided

```
    /* 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);
}
```

```
/* 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);
}
```

Pthreads API

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

- **`pthread_create()`** creates a separate thread with
  - The thread identifier
  - The attributes for the thread
  - The name of the function where the new thread will begin execution
  - The integer parameter

```
    /* 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 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);
}
```

Pthreads API

```
#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 */
```

- The parent thread waits for the summation (child) thread to terminate by calling the **`pthread_join()`** function

```
    /* wait for the thread to exit */
    pthread_join(tid, NULL);
    printf("sum = %d\n",sum);
}
/* 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);
}
```

Pthreads API

```
#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 */
```

- The summation thread terminates when it calls the function **`pthread_exit()`**

```
    /* wait for the thread to exit */
    pthread_join(tid, NULL);
    printf("sum = %d\n",sum);
}
/* 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);
}
```

Comparison with Process Creation

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    pid t pid;
    /* fork a child process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```

Comparison with Function Call

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

int sum;
void runner(void *param);

int main(int argc, char *argv[]) {
   runner(argv[1]);
   printf("sum = %d\n",sum);
}

void runner(void *param) {
   int i, upper = atoi(param);
   sum = 0;
   for (i = 1; i <= upper; i++)
      sum += i;
}
```

Waiting on Several Threads

- Pthread code for joining 10 threads

```
#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);
```

Outline

- Overview
- Multicore Programming
- Multithreading Models
- **Thread Libraries**
  - Pthreads
  - **Windows Threads**
  - Java Threads
- Implicit Threading
- Threading Issues
- Operating System Examples

Windows Threads

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param) {
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 1; i <= Upper; i++)
        Sum += i;
    return 0;
}
int main(int argc, char *argv[]) {
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    Param = atoi(argv[1]);
    /* create the thread */
    ThreadHandle =
      CreateThread(NULL, 0, Summation, &Param, 0, &ThreadId);
    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle,INFINITE);
    /* close the thread handle */
    CloseHandle(ThreadHandle);
    printf("sum = %d\n",Sum);
}
```

Waiting on Several Threads

- **`WaitForMultipleObjects(N, THandles, TRUE, INFINITE);`**
  - The number of objects to wait for
  - A pointer to the array of objects
  - A flag indicating whether all objects have been signaled
  - A timeout duration (or **`INFINITE`**)

Outline

- Overview
- Multicore Programming
- Multithreading Models
- **Thread Libraries**
  - Pthreads
  - Windows Threads
  - **Java Threads**
- Implicit Threading
- Threading Issues
- Operating System Examples

Java Threads (1/3)

- Java threads are available on any system that provides a Java virtual machine (JVM) including Windows, Linux, and macOS
  - The Java thread API is available for Android applications as well
  - The Java thread API is generally implemented using a thread library available on the host system
- Two techniques for explicitly creating threads
  - Create a new class derived from the **`Thread`** class and override its **`run()`** method
    - The (default) **`run()`** method is called if the thread was constructed using a separate **`Runnable`** object; otherwise, this method does nothing and returns
    - The **`run()`** method can be called using the **`start()`** method
  - Define a class that implements the **`Runnable`** interface
    - **`Runnable`** is an interface used to execute code on a concurrent thread
    - More common

https://www.javatpoint.com/java-thread-run-method
https://www.javatpoint.com/runnable-interface-in-java

Java Threads (2/3)

- Define a class that implements the **`Runnable`** interface

```
class Task implements Runnable {
    public void run() {
        System.out.println("I am a thread.");
    }
}
```

  - The code in the **`run()`** method is what executes in a separate thread

- Create a thread

```
Thread worker = new Thread(new Task());
worker.start();
```

  - Invoking the **`start()`** method does two things
    - Allocate memory and initialize a new thread in the JVM
    - Call the **`run()`** method, making the thread eligible to run by the JVM
  - We call the **`start()`** method, not the **`run()`** method

Java Threads (3/3)

- Wait for a thread

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```

Java `Executor` Framework

- Beginning with Version 1.5 and its API, Java provide greater control over thread creation and communication
  - Available in the **`java.util.concurrent`** package
- Classes implementing the **`Executor`** interface must define the **`execute()`** method
  - The **`execute()`** method is passed a **`Runnable`** object
  - For Java developers, this means using the **`Executor`** rather than creating a separate **`Thread`** object and invoking its **`start()`** method

Outline

- ❑ Overview
- ❑ Multicore Programming
- ❑ Multithreading Models
- ❑ Thread Libraries
- ❑ **Implicit Threading**
  - ➢ Thread Pools
  - ➢ Fork Join
  - ➢ OpenMP
  - ➢ Grand Central Dispatch
  - ➢ Intel Thread Building Blocks
- ❑ Threading Issues
- ❑ Operating System Examples

Implicit Threading

- With the continued growth of multicore processing, designing applications containing many threads is difficult
- **Implicit threading**
  - Transfer the creation and management of threading from application developers to compilers and run-time libraries
- Alternative approaches to designing applications that can take advantage of multicore processors through implicit threading
  - These strategies generally require application developers to identify **tasks** (not threads) that can run in parallel
    - A task is usually written as a function
    - The run-time library maps it to a separate thread, typically using the many-to-many model

Outline

- ❑ Overview
- ❑ Multicore Programming
- ❑ Multithreading Models
- ❑ Thread Libraries
- ❑ **Implicit Threading**
  - ➢ **Thread Pools**
  - ➢ Fork Join
  - ➢ OpenMP
  - ➢ Grand Central Dispatch
  - ➢ Intel Thread Building Blocks
- ❑ Threading Issues
- ❑ Operating System Examples

Thread Pools

- Create a number of threads and place them into a pool
  - They sit and wait for work
- Example: a server
  - When a server receives a request, it submits the request to the thread pool and resumes waiting for additional requests
    - The server does not create a thread
  - If there is an available thread in the pool, it is awakened, and the request is serviced immediately
  - If the pool contains no available thread, the task is queued until one becomes free
  - Once a thread completes its service, it returns to the pool and awaits more work
- Demo [Prof. Shih]

Thread Pools Advantages

- Servicing a request with an existing thread is often faster than waiting to create a thread
- A thread pool limits the number of threads that exist at any one point
  - Important on systems that cannot support a large number of concurrent threads
- A thread pool allows us to use different strategies for running a task
  - Separating the task to be performed from the mechanics of creating the task
  - For example, the task could be scheduled to execute after a time delay or to execute periodically

Windows Thread Pool API

- A example function that is to run as a separate thread

```
DWORD WINAPI PoolFunction(PVOID Param) {
/* this function runs as a separate thread */
}
```

- An example of invoking a function

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

  - A pointer to the function that is to run as a separate thread
  - The parameter passed to the function
  - The flags indicating how the thread pool is to create and manage execution of the thread

Java Thread Pools

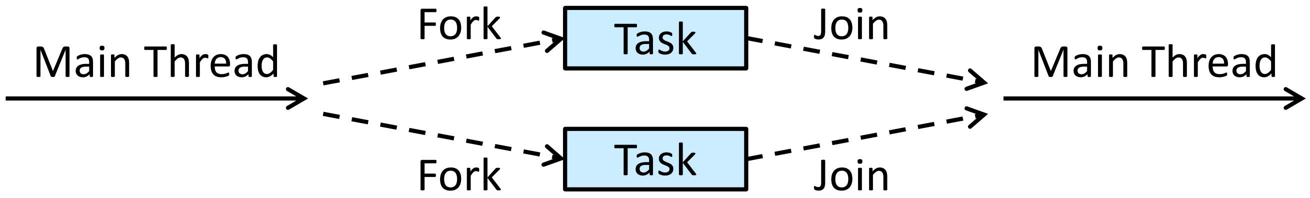
- Single thread executor creates a pool of size 1
  - `newSingleThreadExecutor()`
- Fixed thread executor creates a thread pool with a specified number of threads
  - `newFixedThreadPool(int size)`
- Cached thread executor creates an unbounded thread pool, reusing threads in many instances
  - `newCachedThreadPool()`

Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- **Implicit Threading**
  - Thread Pools
  - **Fork Join**
  - OpenMP
  - Grand Central Dispatch
  - Intel Thread Building Blocks
- Threading Issues
- Operating System Examples

Fork Join

- The **fork-join** model
  - Covered above
    - The main parent thread creates (forks) one or more child threads and then waits for the children to terminate and join with it
    - This synchronous model is often characterized as explicit thread creation
  - It is also an excellent candidate for implicit threading
    - **Threads** are not constructed directly during the fork stage
    - Parallel **tasks** are designated, instead

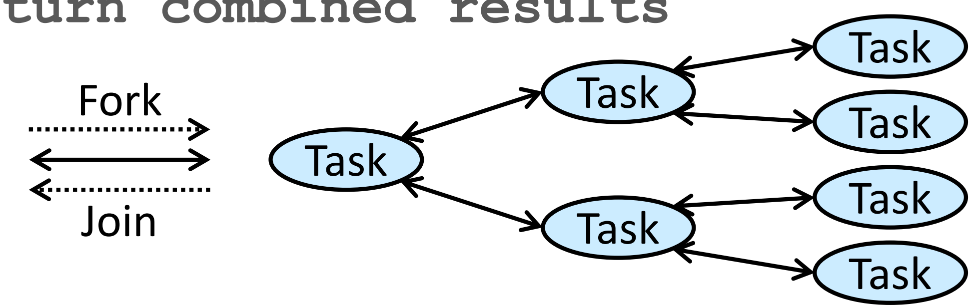


  - A library managing the number of created threads is also responsible for assigning tasks to threads
    - In some ways, this is a synchronous version of thread pools in which a library determines the actual number of threads to create

Fork Join in Java

- A fork-join library in Version 1.7 of the API that is designed to be used with recursive divide-and-conquer algorithms

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)
      result1 = join(subtask1)
      result2 = join(subtask2)
      return combined results
```



Fork Join in Java: Example (1/3)

```
public class SumTask extends RecursiveTask<Integer> {
    static final int THRESHOLD = 1000;

    private int begin;
    private int end;
    private int[] array;

    public SumTask(int begin, int end, int[] array) {
        this.begin = begin;
        this.end = end;
        this.array = array;
    }

    /* Next Slide */
}
```

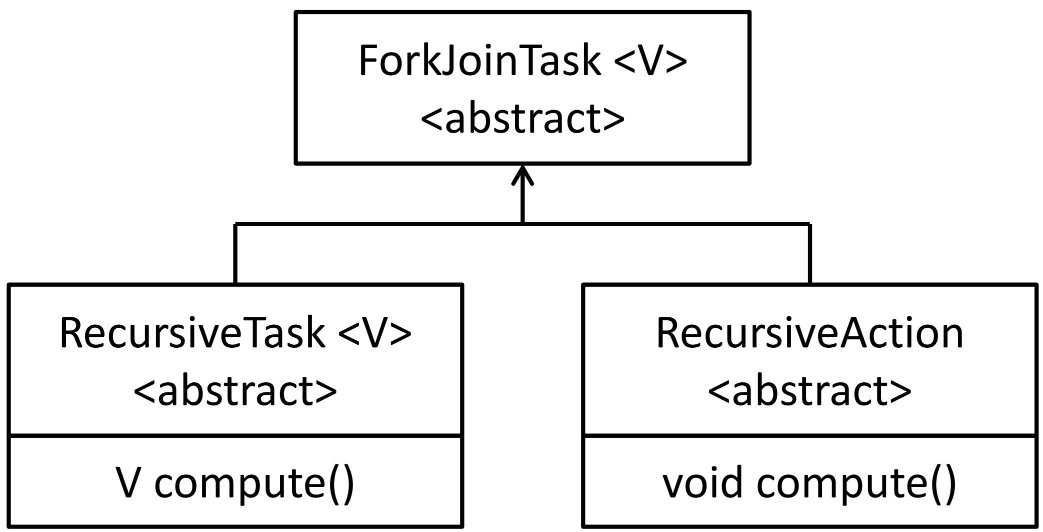
Fork Join in Java: Example (2/3)

```
public class SumTask extends RecursiveTask<Integer> {
    /* Previous Slides */

    protected Integer compute() {
        if (end - begin < THRESHOLD) {
            int sum = 0;
            for (int i = begin; i <= end; i++)
                sum += array[i];
            return sum;
        }
        else {
            int mid = (begin + end) / 2;
            SumTask leftTask = new SumTask(begin, mid, array);
            SumTask rightTask = new SumTask(mid + 1, end, array);
            leftTask.fork();
            rightTask.fork();
            return rightTask.join() + leftTask.join();
        }
    }
}
```

Fork Join in Java: Example (3/3)

- **`ForkJoinTask`** is an abstract base class
  - **`RecursiveTask`** and **`RecursiveAction`** classes extend it
  - **`RecursiveTask`** returns a result
    - Via the return value from the **`compute()`** method
  - **`RecursiveAction`** does not return a result



Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- **Implicit Threading**
  - Thread Pools
  - Fork Join
  - **OpenMP**
  - Grand Central Dispatch
  - Intel Thread Building Blocks
- Threading Issues
- Operating System Examples

OpenMP

- A set of compiler directives and an API for C, C++, or FORTRAN
  - Support parallel programming in shared memory environments
- Developers insert compiler directives at **parallel regions**

```
#pragma omp parallel
{
    printf("I am a parallel region.");
}
#pragma omp parallel for
for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

  - Instruct the OpenMP run-time library to execute the region in parallel
    - OpenMP creates as many threads as there are processing cores in the system
- Demo [Prof. Shih]

Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- **Implicit Threading**
  - Thread Pools
  - Fork Join
  - OpenMP
  - **Grand Central Dispatch**
  - Intel Thread Building Blocks
- Threading Issues
- Operating System Examples

Grand Central Dispatch (GCD) (1/2)

- A technology developed by Apple for its macOS and iOS
  - Include a run-time library, an API, and language extensions
  - Allow developers to identify sections of code (tasks) to run in parallel
  - Manage most of the details of threading (like OpenMP)
- GCD schedules tasks for run-time execution by placing them on a dispatch queue
  - Serial (private) dispatch queue
    - Each process has its own serial queue
    - Once a task has been removed from the queue, it must complete execution before another task is removed
  - Concurrent (global) dispatch queue
    - Several tasks may be removed at a time
    - Several system-wide concurrent queues are divided into four primary quality-of-service classes: user-interactive, user-initiated, utility, and background

Grand Central Dispatch (GCD) (2/2)

- Two different ways to express tasks submitted to dispatch queues
  - A language extension, **block**, for C, C++, and Objective-C languages

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

  - A closure, similar to a block, for the Swift programming language

    ```
    dispatch_async(queue,{ print("A closure.") })
    ```

Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- **Implicit Threading**
  - Thread Pools
  - Fork Join
  - OpenMP
  - Grand Central Dispatch
  - **Intel Thread Building Blocks**
- Threading Issues
- Operating System Examples

Intel Threading Building Blocks (TBB)

- A template library supporting parallel applications in C++
- A serial for loop

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

- The for loop using the TBB **`parallel_for`** template

```
parallel for (size t(0), n, [=](size t i) {apply(v[i]);});
```

Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- **Threading Issues**
  - **The `fork()` and `exec()` System Calls**
  - Signal Handling
  - Thread Cancellation
  - Thread-Local Storage
  - Scheduler Activations
- Operating System Examples

Recap: Process Creation

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() {
    pid t pid;
    /* fork a child process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```

`fork()` and `exec()` System Calls

- The semantics of the **`fork()`** and **`exec()`** system calls change in a multithreaded program
  - Some UNIX systems have chosen to have two versions of **`fork()`**
    - One duplicates all threads
    - One duplicates only the thread that invoked the **`fork()`** system call
  - The **`exec()`** system call typically works in the same way
    - Replace the entire process including all threads
- If **`exec()`** is called immediately after forking, then duplicating all threads is unnecessary

Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- **Threading Issues**
  - The **`fork()`** and **`exec()`** System Calls
  - **Signal Handling**
  - Thread Cancellation
  - Thread-Local Storage
  - Scheduler Activations
- Operating System Examples

Signals

- A **signal** is used in UNIX systems to notify a process
  - All signals follow the same pattern
    - A signal is generated by the occurrence of a particular event
    - The signal is delivered to a process
    - Once delivered, the signal must be handled?!
- Synchronous signal
  - Synchronous signals are delivered to the same process that performed the operation that caused the signal
  - Examples: illegal memory access and division by 0
- Asynchronous signal
  - The signal is generated by an event external to a running process
  - Examples: terminating a process with Ctrl+C and a timer expired

Signal Handling

- Two possible handlers
  - Every signal has a **default signal handler** that the kernel runs when handling that signal
  - This default action can be overridden by a **user-defined signal handler**
- Signals are handled in different ways
  - Examples: terminating the program, ignoring the signal
- For single-threaded programs
  - Signals are always delivered to a process
- For multithreaded programs
  - 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

Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- <u>**Threading Issues**</u>
  - The **`fork()`** and **`exec()`** System Calls
  - Signal Handling
  - <u>**Thread Cancellation**</u>
  - Thread-Local Storage
  - Scheduler Activations
- Operating System Examples

Thread Cancellation

- **Thread cancellation** involves terminating a thread before it has completed
  - A thread that is to be canceled is often referred to as the **target thread**
- Two scenarios of cancellation of a target thread
  - Asynchronous cancellation
    - One thread immediately terminates the target thread
  - Deferred cancellation
    - The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion
  - Canceling a thread asynchronously may not free a necessary system-wide resource

Pthread Cancellation (1/2)

- Thread cancellation is initiated using **`pthread_cancel()`**
  - Only a request to cancel the target thread
  - Actual cancellation depending on how the target thread is set up to
    - Three cancellation modes: disabled, deferred, asynchronous
- When the target thread is finally canceled, the call to **`pthread_join()`** in the canceling thread returns

```
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);
```

Pthread Cancellation (2/2)

- Default cancellation mode: deferred cancellation
  - However, cancellation occurs only when a thread reaches a **cancellation point** which can be set by `pthread_testcancel()`

```
while (1) {
    /* do some work for awhile */
    ...
    /* is there a cancellation request? */
    pthread_testcancel();
}
```

- A **cleanup handler** can be invoked if a thread is canceled
  - Release any resource that the thread may have acquired
- On Linux systems, thread cancellation using the Pthreads API is handled through signals

Thread Cancellation in Java

- A policy similar to deferred cancellation in Pthreads
  - The **`interrupt()`** method set the interruption status of the target thread to true

```
Thread worker;
. . .
/* set the interruption status of the thread */
worker.interrupt()
```

  - A thread can check its interruption status by invoking the **`isInterrupted()`** method

```
while(!Thread.currentThread().isInterrupted()){
    . . .
}
```

Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- **Threading Issues**
  - The **`fork()`** and **`exec()`** System Calls
  - Signal Handling
  - Thread Cancellation
  - **Thread-Local Storage**
  - Scheduler Activations
- Operating System Examples

Thread-Local Storage

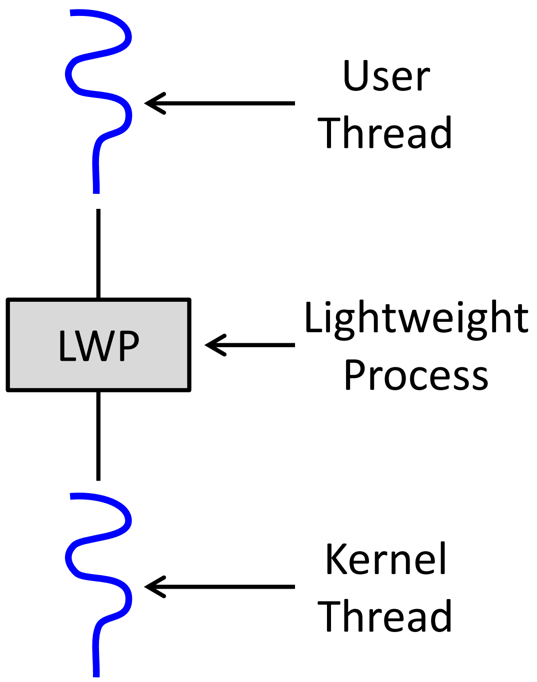
- **Thread-local storage** (TLS) allows each thread to have its own copy of certain data
  - Different from local variables
    - Local variables are visible only during a single function invocation
    - TLS data are visible across function invocations
  - Useful when the developer has no control over the thread creation process
    - Example: when using an implicit technique such as a thread pool
  - Similar to `static` data
    - A static variable inside a function keeps its value between invocations
    - However, TLS is unique to each thread

Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- **Threading Issues**
  - The **`fork()`** and **`exec()`** System Calls
  - Signal Handling
  - Thread Cancellation
  - Thread-Local Storage
  - **Scheduler Activations**
- Operating System Examples

Lightweight Process (LWP)

- Many-to-many and two-level models require communication to dynamically adjust the number of kernel threads
- An intermediate data structure, an LWP, is placed between the user and kernel threads by many systems
  - An LWP appears to be a virtual processor to the user-thread library
    - An application can schedule a user thread to run on it
  - Each LWP is attached to a kernel thread
    - An operating system schedules kernel threads to run on physical processors



Scheduler Activations

- One scheme for communication between the user-thread library and the kernel
  - The kernel provides an application with a set of LWPs
  - The application can schedule user threads onto an available LWP (virtual processor)
- The kernel must inform an application about certain events
  - This procedure is known as an **upcall**
  - Upcalls are handled by the thread library with an upcall handler
  - Upcall handlers must run on a virtual processor

Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- **Operating System Examples**
  - Windows Threads
  - Linux Threads

Windows Threads (1/2)

- A Windows application runs as a separate process
- Each process may contain one or more threads
  - Windows API for creating threads has been covered
  - Windows uses the one-to-one mapping model
- The general components of a thread
  - A thread ID uniquely identifying the thread
  - A register set representing the status of the processor
  - A program counter
  - User and kernel stacks, employed when the thread is running in user and kernel modes, respectively
  - A private storage area used by run-time and dynamic link libraries
- The register set, stacks, and private storage area are known as the **context** of the thread

Windows Threads (2/2)

- The primary data structures of a thread
  - **ETHREAD**: executive thread block (in kernel space)
    - A pointer to the process to which the thread belongs
    - The address of the routine in which the thread starts control
    - A pointer to the corresponding **KTHREAD**
  - **KTHREAD**: kernel thread block (in kernel space)
    - Scheduling and synchronization information for the thread
    - The kernel stack used when the thread is running in kernel mode
    - A pointer to the **TEB**
  - **TEB**: thread environment block (in user space)
    - The thread identifier
    - A user-mode stack
    - An array for thread-local storage

Linux Threads

- Linux provides the ability to create threads using **`clone()`**
  - In fact, Linux uses the term **task**, rather than process or thread, when referring to a flow of control within a program
- **`clone()`** is passed a set of flags that determine how much sharing is to take place between the parent and child tasks
  - **`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
  - If none of these flags is set, no sharing takes place, resulting in functionality similar to that provided by **`fork()`**
- A unique kernel data structure (**`struct task_struct`**) exists for each task in the system

Objectives

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designing multithreaded applications
- Design multithreaded applications using the Pthreads, Java, and Windows threading APIs
- 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

Q&A