Chapter 04
Threads & Concurrency

A fundamental unit of CPU utilization

Outline

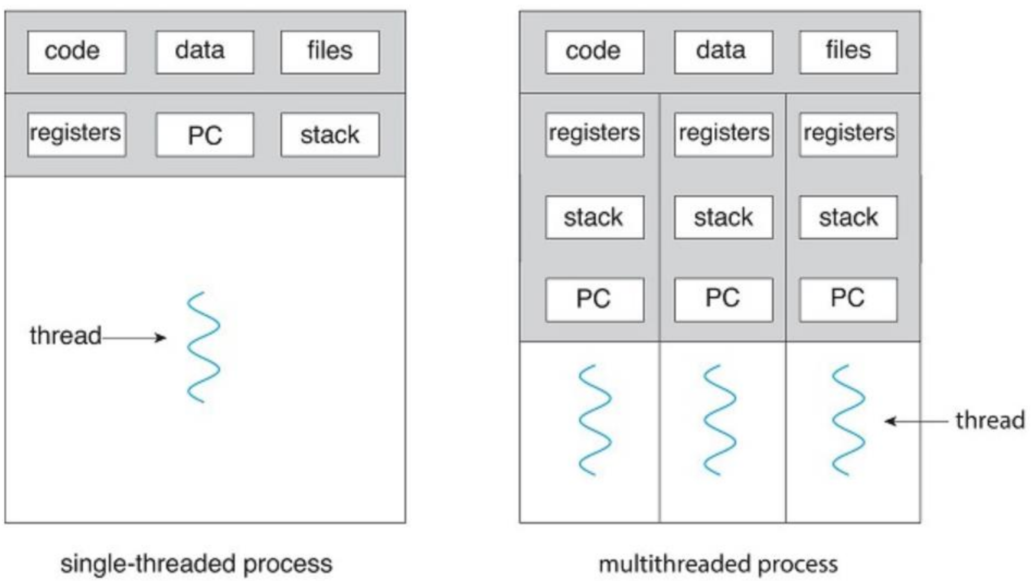
- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues

Objectives

- Identify the basic components of a thread, and contrast threads and processes.
- Describe the major benefits and significant challenges of designing multithreaded processes.
- Illustrate different approaches to implicit threading
- Describe threading issues

Overview

- Thread
  - Basic unit of CPU utilization
  - Include a **thread ID**, a **program counter**, a **register set**, and a **stack**
  - Share code section, data section, and other resources such as open files and signals
  - Program with multiple threads can perform multiple tasks

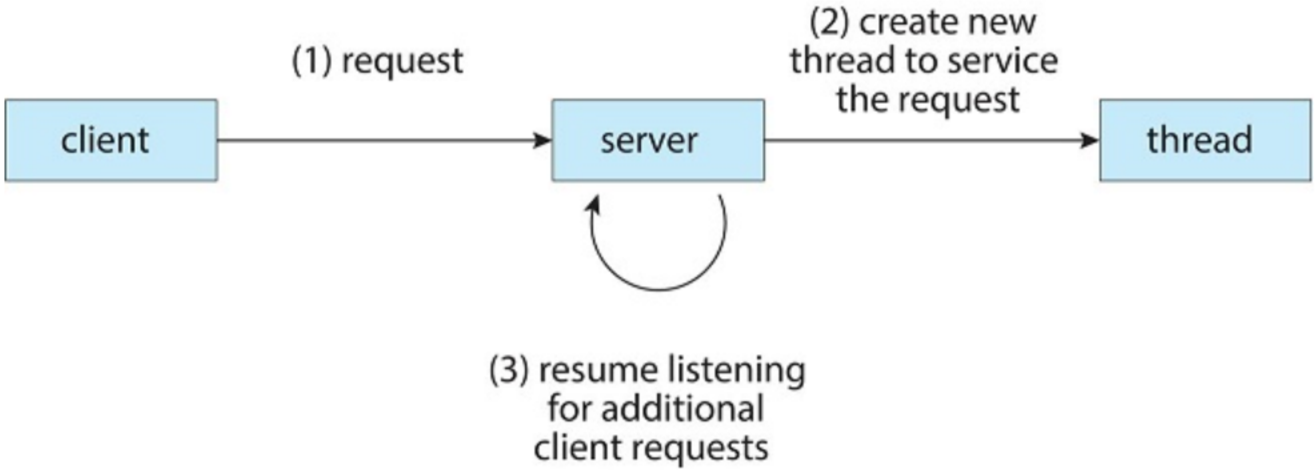


Motivation

- Most modern applications are multithreaded
- Threads run within application
  - Thumbnails
  - Web browser
  - Word processor
- Leverage processing capabilities on multicore systems
- Consider a web server
  - Provide web pages, images, …
  - Busy for multiple clients
  - Handle by traditional single-threaded process
    - One client at a time

Motivation

- Handle by multiple processes
  - Each process serves a request
  - Process is heavy-weight: time consuming and resource intensive
- Handle by on process with multiple threads
  - Each thread serves a request
  - Thread is light-weight



Motivation

- Kernels are generally multithreaded
  - Linux's kernel threads for varying tasks (shown by **`ps -ef`**)
  - **`kthreadd`** (with pid = 2) as parent of all kernel threads

- High performance computing applications via threads for running in parallel
  - Data mining
  - Graphics
  - Artificial intelligence

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 multiprocessor architectures

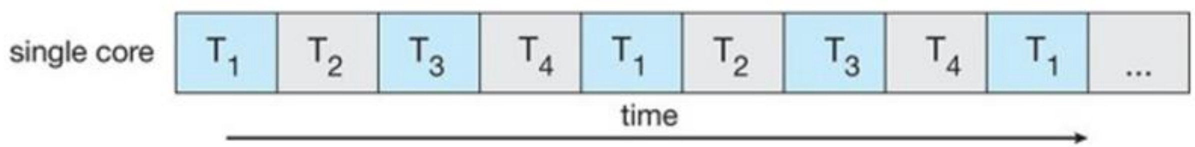
Exercises

- Describe the actions taken by a kernel to context-switch between kernel-level threads.

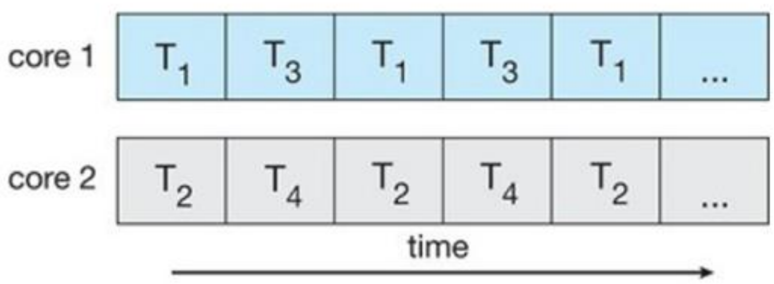
- What resources are used when a thread is created? How do they differ from those used when a process is created?

Multicore Programming

- Multicore or multiprocessor systems bring parallelism and improve concurrency.
  - ***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



Concurrent execution on a single-core system.



Parallel execution on a multicore system.

Exercise

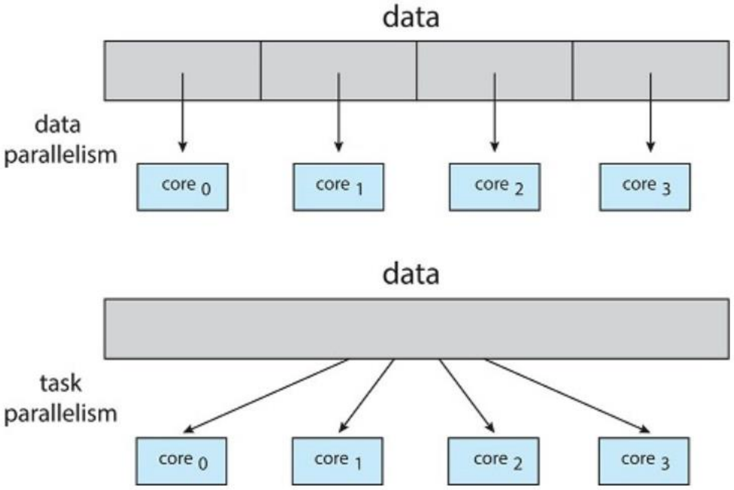
- Is it possible to have concurrency but not parallelism? Explain.

Programming Challenges

- The programming challenges in multicore programming include:
  - **Identifying tasks**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

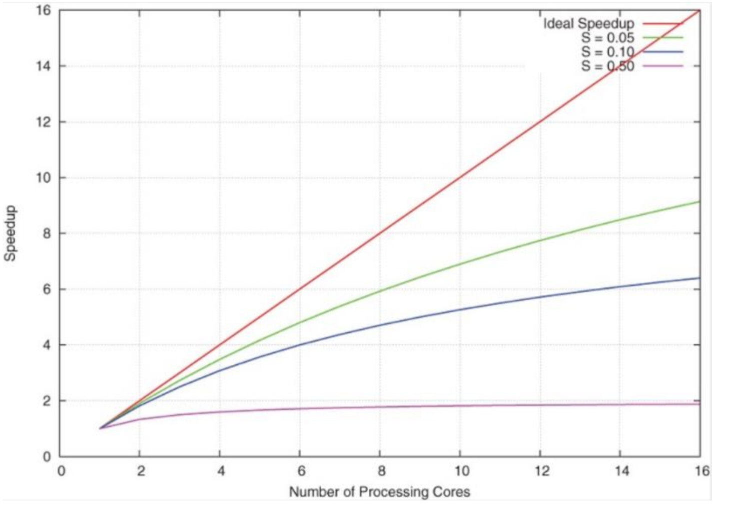
Types of parallelism

- 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



- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Amdahl's Law



- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- Given *S* as serial portion and *N* processing cores, we have:

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

- As *N* approaches infinity, speedup approaches 1 / *S*
- For instance, if application is 75% parallel / 25% serial
  - Moving from 1 to 2 cores results in speedup of 1.6 times
  - Moving to infinity cores results in speedup of 4 times

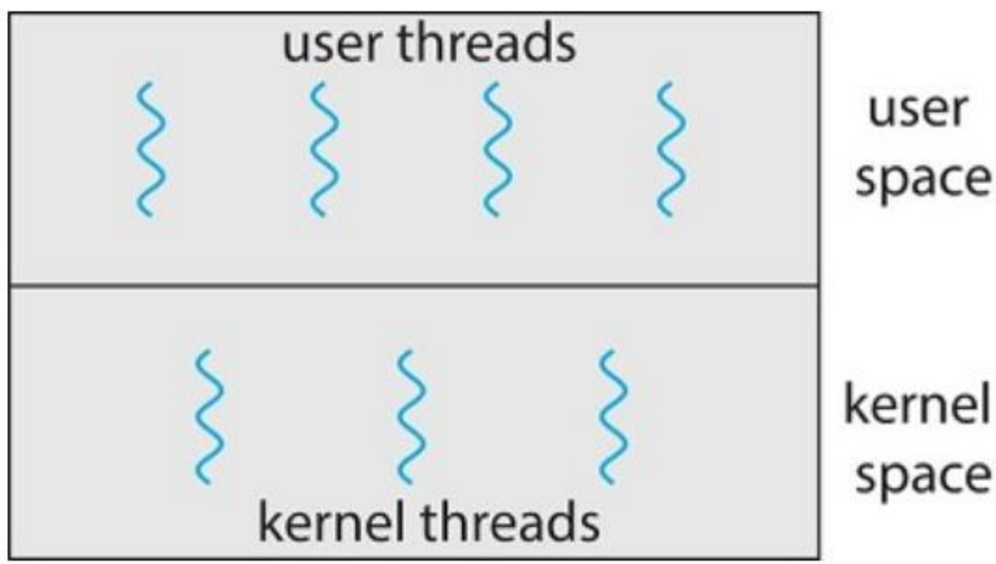
Serial portion of an application has disproportionate effect on performance gained by adding additional cores

Multithreading Models

- 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 include virtually all general purpose operating systems:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

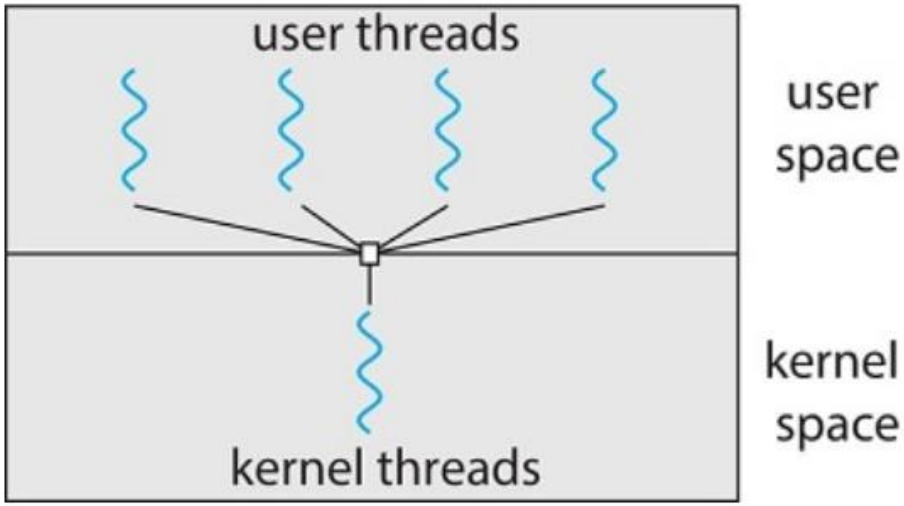
Multithreading Models

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



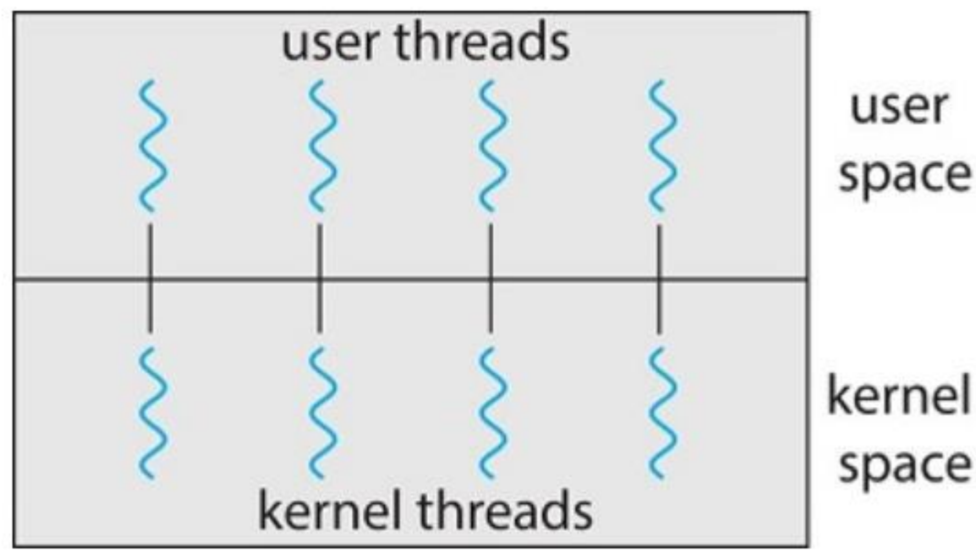
Many-to-One

- 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



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

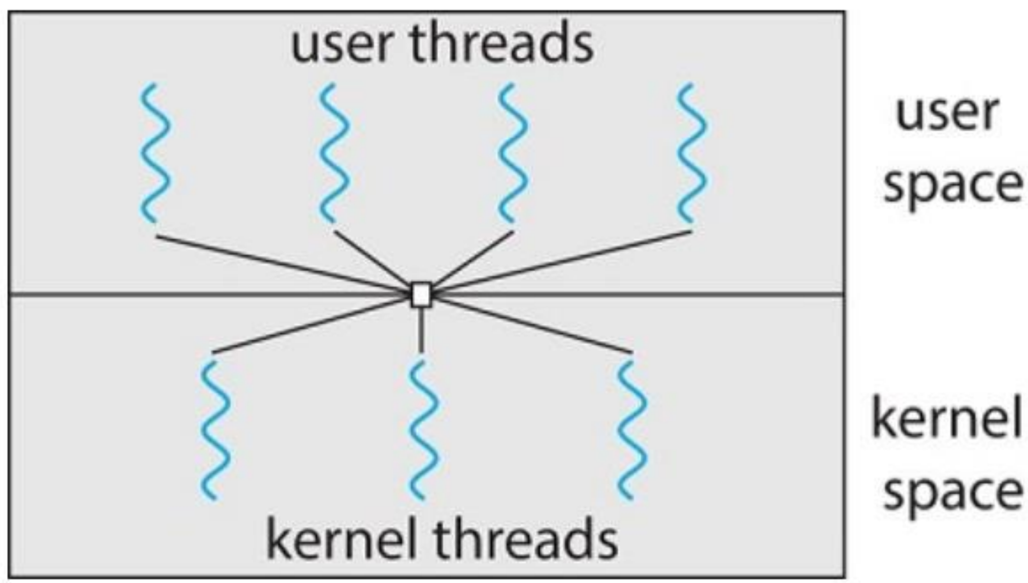


Exercise

- A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application is running on this system. All input is performed at program start-up, when a single file must be opened. Similarly, all output is performed just before the program terminates, when the program results must be written to a single file. Between start-up and termination, the program is entirely CPU-bound. Your task is to improve the performance of this application by multithreading it. The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).
  - How many threads will you create to perform the input and output? Explain.
  - How many threads will you create for the CPU-intensive portion of the application? Explain.

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

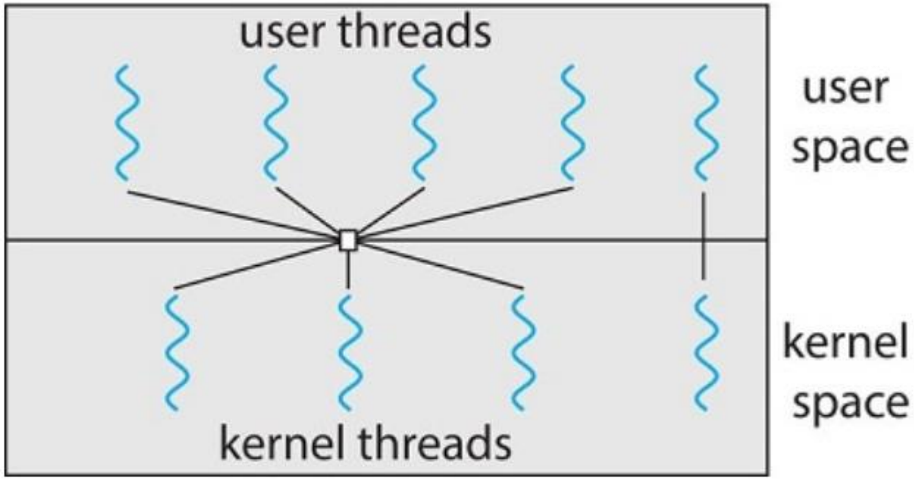


Exercise

- Consider a multicore system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processing cores in the system. Discuss the performance implications of the following scenarios.
  - The number of kernel threads allocated to the program is less than the number of processing cores.
  - The number of kernel threads allocated to the program is equal to the number of processing cores.
  - The number of kernel threads allocated to the program is greater than the number of processing cores but less than the number of user-level threads.

Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Comparisons among Models

- Many-to-one
  - The number of user threads does not imply parallelism
- One-to-one
  - Limited number of threads can be used.
- Many-to-many
  - Flexible
  - Model suffers from neither of these shortcomings.
- Most operating systems use one-to-one model
  - Many-to-many is difficult to implement.
  - The increasing number of cores lowers down the importance of the number of kernel threads.

Exercises

- Provide a programming example in which multithreading does not provide better performance than a single-threaded solution.

- Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.

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
- Three main thread libraries
  - POSIX Pthreads
    - Either user level or kernel level
  - Windows
    - Kernel level
  - Java
    - Managed in Java program.
    - Implemented using a thread library available on the host system since JVM is running on top of a host operating system.

Thread Libraries

- General strategies for creating multiple threads
  - Asynchronous threading
    - Parent resumes its execution after generating child thread
    - Run parent and child concurrently and independently
    - Little data sharing
  - Synchronous threading
    - Parent waits for all of its children threads to terminate
    - Children threads run concurrently
    - Significant data sharing

Exercise

- Consider the following code segment:

```
pid_t pid;
pid = fork();
if (pid == 0) { /* child process */
    fork();
    thread_create( . . .);
}
fork();
```

  - How many unique processes are created?
  - How many unique threads are created?

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. This is known as implicit threading.
  - Identify tasks (function) rather than threads which can run in parallel.
  - Map task to a separate thread via M-M model.
  - Developers only need to identify parallel tasks.
  - Libraries determine thread creation and management.
- Two methods explored
  - Thread Pools
  - OpenMP

Thread Pools

- Recall the multithreaded web server with two issues:
  - The amount of time required to create the thread
  - No bound on the number of active threads
- A solution is to use a thread pool.
  - Create a number of threads in a pool where they await work
  - A request is served by awaking a thread in pool.
  - Requests are queued if no thread is available.
  - A thread returns to the pool when task has completed.
  - Work well for asynchronous execution

Thread Pools

- Advantages of using thread pools
  - 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
    - i.e. tasks could be scheduled to run periodically
- The number of threads in the pool
  - Can be set heuristically such as according to #CPUs, size of main memory, #concurrent requests
  - Can be set dynamically according to usage pattern
    - E.g. Apple's Grand Central Dispatch

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

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

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

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

    /* sequential code */

    return 0;
}
```

OpenMP

- Parallelizing loops
  - ```
    #pragma omp parallel for
    for(int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
    ```
  - Create as many threads as there are cores
  - Run for loop in parallel by using `#pragma omp parallel for`
    ```

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
- Depends on application
  - fork() then exec(): Duplicate calling thread is okay.
  - fork() without exec(): Duplicate all thread is needed.

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
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled after delivered
- Synchronous signals
  - Delivered to the same process that performed the operation that caused the signal
  - E.g. illegal memory access and division by 0
- Asynchronous signals
  - Generated by an external event
  - Sent to another process
  - E.g. terminating a process with specific keystrokes, timer expire

Signal Handling

- A signal may be ***handled*** by one of two possible handlers:
  - default
  - user-defined
- Every signal has default handler that kernel runs when handling signal
  - User-defined signal handler can override default

Signal Handling

- 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

- The method for delivering a signal depends on the type of signal generated.
  - Synchronous signal to the thread causing the signal.
  - Asynchronous signal depends.

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
- Difficulty with cancellation
  - Allocated resources
  - In the midst of updating shared data
  - Troublesome with asynchronous cancellation
    - May not free all resources
  - Safely canceled by deferred cancellation

Thread Cancellation

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

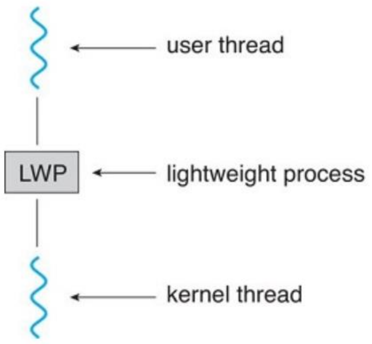
Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches cancellation point
  - 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.
  - Most thread libraries and compilers provide support for TLS.

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
- 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

Exercise

- Assume that an operating system maps user-level threads to the kernel using the many-to-many model and that the mapping is done through LWPs. Furthermore, the system allows developers to create real-time threads for use in real-time systems. Is it necessary to bind a real-time thread to an LWP? Explain.