Justin Ciocoi

Oct. 29, 2023

CSCI 375 Textbook Notes

Chapter 4: Threads and Concurrency

4.1: Overview

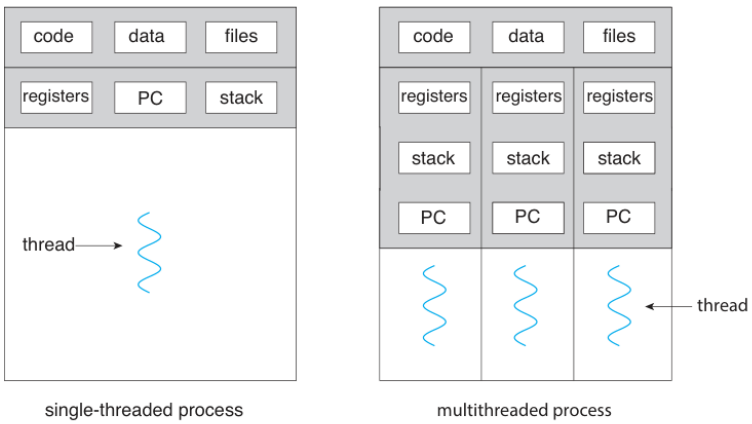
- A *thread* is a basic unit of CPU utilization, which is comprised of a *thread ID, program counter, register set*, and a *stack*
- A traditional process has a single thread of control, but if a process has multiple threads, it can perform more than one task at a time
- **4.1.1: Motivation**
  - Most applications that run on modern computers and mobile devices are multithreaded
  - Typically, an application is implemented as a separate process with several threads of control
  - 

    Figure 4.1 Single-threaded and multithreaded processes.

  - Additionally, applications can be designed in order to leverage processing capabilities on multicore systems, allowing them to perform several CPU-intensive tasks in parallel across multiple computing cores
  - Most operating system kernels are multithreaded, so during system boot, several kernel-level threads are created

- **4.1.2: Benefits**
    - The benefits of multithreaded programming can be broken down into four major categories
        - **Responsiveness**, since multiple operations may be ongoing and the user will not have to wait for each process to finish in serial
        - **Resource sharing**, since threads share, by default, the data of the process to which they belong
        - **Economy**, since there is less overhead required when dealing with multithreaded applications
        - **Scalability**, since a multithreaded program will benefit greatly as the number of computing cores increases

4.2: Multicore Programming

- As computers have evolved from single-CPU to multi-CPU systems, they have also evolved to place multiple CPU computing cores on a single processing chip, where each core appears as a separate CPU to the operating system
- We refer to such systems as *multicore* systems, and multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency
- **4.2.1: Programming Challenges**
    - The trend towards multicore systems continues to place pressure on system designers and application programmers to make better use of the multiple available computing cores
    - In general, five areas present challenges in programming for multicore systems
        - **Identifying tasks**, which involves examining applications to find areas that can be divided into separate, concurrent tasks, which are ideally independent of each other and can run in parallel on individual CPU cores
        - **Balance**, such that processes that are being run in parallel perform approximately equal work of equal value
        - **Data splitting**, where the data accessed and manipulated by tasks must be divided to run on separate cores

        - **Data dependency**, where data that is accessed by two tasks must be examined for any potential dependencies between the two tasks, and proper synchronization must be implemented to avoid data race conditions
        - **Testing and debugging**, which is inherently more difficult in multicore programming due to different possible execution paths across multiple CPU cores
    - **Amdahl's Law**
        - Amdahl's Law is a formula that describes the potential performance gains from adding additional computing cores to an application that has both parallel and serial components
        - If S is the percentage of computation that must be performed in serial on a system with N processing cores, the formula for Amdahl's Law is as follows

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

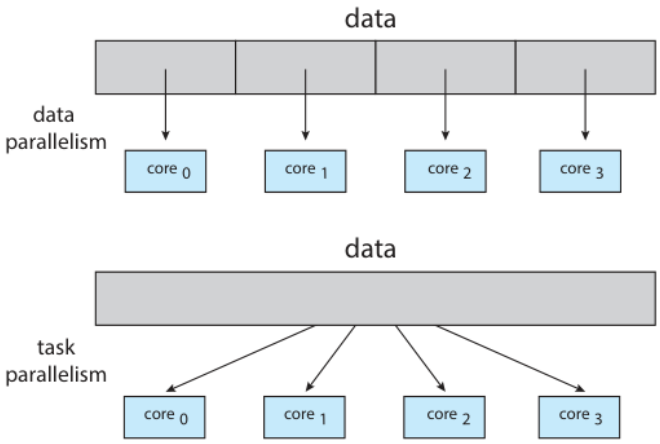
- **4.2.2: Types of Parallelism**
    - In general, there are two types of parallelism: *data parallelism* and *task parallelism*
    - *Data parallelism* focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core
    - *Task parallelism* focuses on distributing not data, but tasks, or threads, across multiple computing cores where each thread is performing a unique operation
    - 

Figure 4.5 Data and task parallelism.

4.3: Multithreading Models

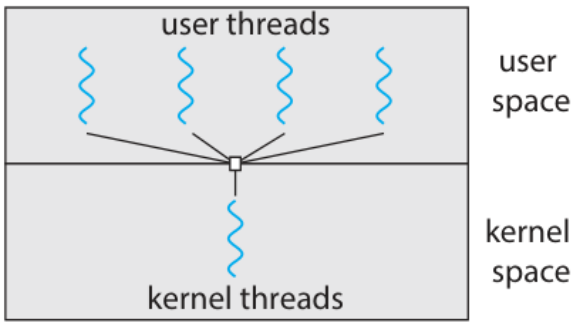
- So far, we have treated threads in a fairly generic sense, but support for threads may be provided either at the user level, for *user threads*, or at the kernel level, for *kernel threads*
- Virtually all contemporary operating systems support kernel threads
- In this section, we will differentiate between the different models that can be used to implement the relationship between user- and kernel-level threads
- **4.3.1: Many-to-One Model**
    - 

Figure 4.7 Many-to-one model.

    - This model maps many user-level threads to a single kernel-level thread
    - Here, thread management is done by the thread library in user space, so it is efficient
    - However, if a thread makes a blocking system call, the entire process will block
    - Additionally, since only one thread can access the kernel at a time in this scheme, multiple threads are unable to run in parallel on multicore systems
    - This scheme is used sparingly in the modern era due to its inability to take advantage of multiple processing cores, which has become standard on almost all computer systems
- **4.3.2: One-to-One Model**
    -

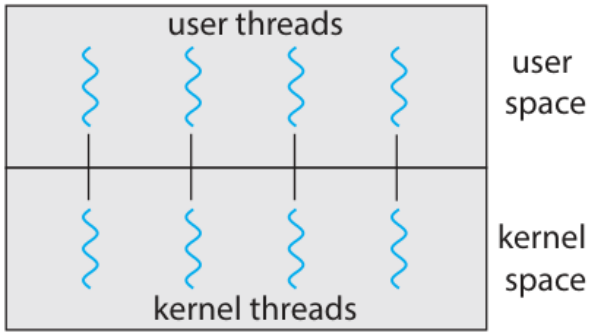


Figure 4.8 One-to-one model.

- The one-to-one model maps each user thread to a kernel thread
- This allows for more concurrency that the many-to-one model by allowing another thread to run when a thread makes a blocking system call
- It also allows multiple threads to run in parallel on multiprocessors
- The main drawback to this scheme is that the large number of kernel level threads may burden the performance of a system

- **4.3.3: Many-to-Many Model**
  -

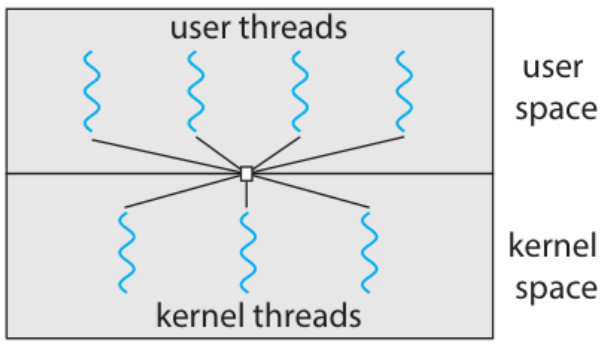


Figure 4.9 Many-to-many model.

- In the many-to-many model, user-level threads are multiplexed and passed to a smaller or equal number of kernel-level threads
- This model is free from the drawbacks from the previous two models since the developer can create as many user threads as necessary, and the kernel threads can run in parallel on a multiprocessor
- A variation on the many-to-many model, called the *two-level model*, allows this multiplexing and demultiplexing of threads, but also allows user level threads to be bound directly to kernel level threads

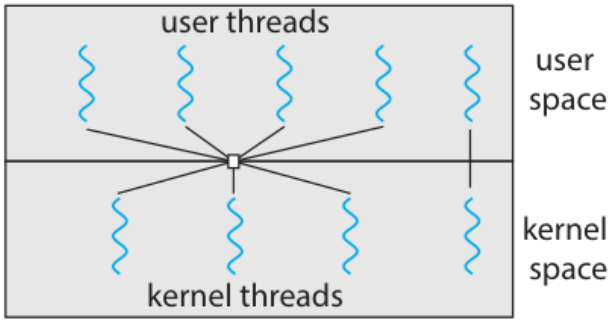


Figure 4.10 Two-level model.

4.4: Thread Libraries

- A *thread library* provides a programmer with an API for creating and managing threads
- There are two primary ways in which thread libraries are generally implemented
  - The first is to provide a library entirely in user space with no kernel support such that all code and data structures for the library exist in user space and invoking a library function results in a local function call rather than a system call
  - The second approach is to implement a kernel-level library supported directly by the operating system such that code and data structures for the library exist in kernel space and invoking a library function typically results in a system call
- The three main thread libraries that are in use today are *POSIX Pthreads, Windows*, and *Java*
- We will illustrate this in the POSIX API using the well known summation function

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

- Before we proceed, we must introduce the concepts of *synchronous* and *asynchronous* threading
  - In asynchronous threading, once the parent thread creates a child thread, the parent resumes its execution such that the parent and child execute concurrently and independently of one another
  - Synchronous threading occurs when the parent thread creates one or more child threads and then must wait for all of its child threads to terminate before it resumes
  -

- 4.4.1: Pthreads
    - *Pthreads* refers to the POSIX standard defining an API for thread creation and synchronization
    - Now, let us show a multithreaded C program using the POSIX Pthreads API
    - ```c
      #include <pthread.h>
      #include <stdio.h>
      #include <stdlib.h>

      //declare global sum variable shared by threads
      int sum;

      //declare function which threads will call
      void *runner(void *param);

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

          //set attributes of 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);
      }
      ```

```
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for(i=1; i<= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

4.5: Implicit Threading
```

- As multicore processing has grown over the years, applications containing hundreds, or even thousands, of threads are looming on the horizon
- In order to develop applications that effectively utilize such a framework, developers must invest more time and effort to program and debug than in a single-threaded application
- One way to address these difficulties and better support the design of concurrent and parallel applications is to transfer the management of threading from application developers to compilers and run-time libraries
- This strategy has been coined *implicit threading*, and it allows application designers and developers to more easily take advantage of multicore processing
- **4.5.1: Thread Pools**
    - A *thread pool* can be utilized such that a number of threads are created at startup and placed into a pool, where they idly sit and wait for work
    - For instance, when a server receives a request, rather than creating a new thread, it can submit the request to the thread pool and resumes waiting for additional requests
    - Once a thread completes its service, it will return to the pool to await more work
    - Thread pools offer the following benefits
        - Servicing a request with an already existing thread is usually faster than waiting to create a thread
        - A thread pool limits the total number of threads that exist at any one point, which prevents systems from becoming burdens due to too large a number of threads
        - Separating the actual task from the mechanics of creating ones allows for more flexibility when it comes to running tasks, such as allowing a task to execute after a time delay, or periodically
- **4.5.2: Fork Join**
    - The strategy of thread creation covered in section 4.4 is often called the *fork-join* model
    - In this method, the main parent thread creates (forks) one or more child threads, and then waits for the children to terminate and join with it, at which point it can retrieve and combine their results
    -

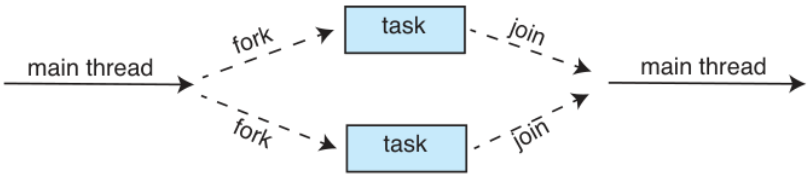


Figure 4.16 Fork-join parallelism.

- **4.5.3: OpenMP**
  - OpenMP is a set of compiler directives, as well as an API written for programs written in C, C++, or FORTRAN which provide support for parallel programming in shared-memory environments
  - OpenMP defines *parallel regions* as blocks of code that may run in parallel
  - Let us look at the following C program illustrating the use of OpenMP

```c
#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;
}
```

  - When OpenMP encounters the directive `#pragma omp parallel`, it creates as many threads as there are processing cores in the system
    - For a dual core system, two threads are created
    - For a quad core system, four threads are created
    - And so on
  - All the threads then simultaneously execute the parallel portion of the code and as each thread exits the parallel portion, it is terminated

4.6: Threading Issues

- **4.6.1: The `fork()` and `exec()` System Calls**
  - If one thread in a program calls `fork()`, does the new process duplicate all threads, or is the new process single-threaded
  - Some UNIX systems have two versions of `fork()`, one that duplicates all threads and one that duplicates only the thread that invoked the `fork()` system call
  - Which version of the `fork()` system call to use is decided on a case by case basis as a function of the application being used
- **4.6.2: Signal Handling**
  - A *signal* is used in UNIX systems to notify a process that a particular event has occurred
  - A signal can be received either synchronously or asynchronously, however 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
  - A signal can be *handled* by one of two possible handlers
    - A default signal handler
    - A user-defined signal handler
  - Different signals are handled in different ways
    - Some may be ignored
    - Others might result in the immediate termination of an application, such as illegal memory access
  - For a system utilizing multithreaded applications, how does the system know where a signal should be delivered
  - In general, the following options exist
    - 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
- **4.6.3: Thread Cancellation**
    - *Thread cancellation* involves terminating a thread before it has completed
    - A thread that is to be cancelled is often referred to as the *target thread*, and cancellation of a target thread may occur in two different scenarios
        - **Asynchronous cancellation**, where one thread immediately terminates the targeted thread
        - **Synchronous cancellation**, where the target thread periodically checks whether it should terminate, allowing itself an opportunity to terminate itself in an orderly fashion