Operating Systems

Chapter 4
Threads

Authors:
Student 1
Student 2
Supervisor: Teacher
Term: HT19
Course code: 1DV512

Contents

1 Introduction

The smallest individual components of CPU utilization are threads. Threads consist of a thread ID, program counter, register set & a stack. With the other threads that belong to the same process, a thread also shares a code section, data section & other operating-system resources. Like signals & open files. A process is either single-threaded or multithreaded. Multithreaded processes, unlike single-threaded processes, can perform multiple tasks at a time. Today most operating-systems are multithreaded thereat all the threads have specific tasks that they operate in the kernel. Multithreaded programming has the following advantages:

Increased responsiveness: A program can continue to run even if some part of it is blocked or takes a lot of time to run. Therefore, an interactive application will not have the user wait for an operation to finish before it is responsive again, which is the case for a single-threaded application.

Sharing of resources: The memory is less affected because the resources of a multithreaded process is automatically shared by the threads it consists of. In extension this also makes multithreaded programming more **economical** than single-threaded programming.

Scalability: A multiprocessor environment makes multithreading even more beneficial. When available, threads can run in parallel on different cores unlike a single-threaded process that are unable to run on more than one.

2 Multicore Programming

Multicore or multiprocessor systems are systems where each core is a separate processor in the operating system. Multithreaded programming makes better use of these multicore systems & improves concurrency. Concurrency in a multicore system means that a separate thread can be assigned to each core whereas the threads run in parallel, while in a single core system only one thread at a time is being executed. However there is a difference between parallelism & concurrency, parallel systems simultaneously perform more than one task while concurrent systems allows all tasks to make progress. Therefore parallelism is not required to achieve concurrency. Switching quickly between processes in the system is concurrency but provides the illusion that the system runs in parallel.

The potential gain of implementing an additional core can be calculated by Amdahl's law (Equation 1) where N is the number of processing cores in a system & S the portion of the application that has to be performed serially.

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

2.1 Programming challenges

Making use of multicore system is a challenge for programmers. The five most common difficulties are:

Identifying Tasks: Finding & dividing areas in applications into separate, concurrent tasks.

Balance: Making sure that the tasks running in parallel perform equal work of equal value.

Data splitting: Divide the data accessed & manipulated by the separated tasks to run on different cores.

Data Dependency: Finding dependencies between tasks & adjusting them by making them synchronized.

Testing & debugging: It is much more difficult to test a multicore program since there are many different execution paths possible.

The design of software in the future may need a whole new approach because of these difficulties.

2.2 Types of parallelism

Generally there are two types of parallelism; data parallelism & task parallelism.

Data parallelism: Data is divided into subsets & divided across multiple cores. The same operation is performed on the data but on different cores.

Task parallelism: Different tasks are distributed across multiple cores. The same data may be operated on at the same time by different threads.

Most applications use both data & task parallelism.

3 Multithreading Models

Multithreading models are used to establish the relationship between user threads & kernel threads. The difference between user threads & kernel threads are the way they are supported by the operating system.

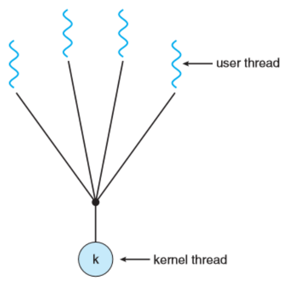
User threads: Supported above the kernel & managed without kernel support.

Kernel threads: Supported & managed directly by the operating system.

The three common models for this relationship that will be presented are many-to-one, one-to-one many-to-many.

3.1 Many-to-one model

One kernel thread has many user-level threads mapped to it. Because of its incapacity to make use of multiple processor cores the many-to-one model (Figure 1) is rather outdated.



[1]

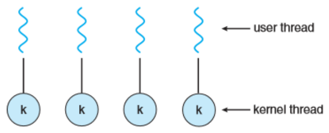
Figure 1: many-to-one

Disadvantage:

- If a thread makes a blocking system call the entire process will be blocked.
- Multiple threads are unable to run in parallel on multicore system since only one thread at a time can access the kernel.

3.2 One-to-One Model

Each user-level thread is distributed to a kernel thread by the one-to-one model (Figure 2). This allows a thread to run even when another thread is making a blocking system call unlike in the many-to-one model.



[1]

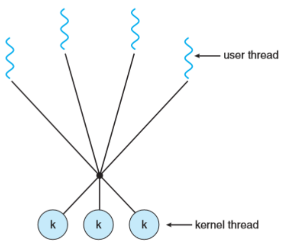
Figure 2: one-to-one

Disadvantage:

- Implementation will most likely restrict the number of threads supported by the system.

3.3 Many-to-Many Model

Multiple user-level threads are multiplexed to an equal or smaller number of kernel threads. Therefore the many-to-many model (Figure 3) manages to avoid the disadvantages of the many-to-one mode & one-to-one model.



[1]

Figure 3: many-to-many

Another model that is a variation of the many-to-many model is the two-level model. In addition to having many user-level threads multiplexed to an equal or smaller number of kernel threads, a user-level thread is allowed to be bound to a kernel thread.

4 Thread Libraries

A thread library provides the programmer with an API for creating & managing threads. As a programmer, there are two different ways to implement a library thread. They can be implemented in either:

- User Space
- Kernel-Level

An implementation made in user space does not have any kernel support & therefore, any function invocation will lead to a local function call rather than a system call.[1]

If you instead implement the thread library in kernel-level supported by the OS, code & data structures for the library it will be in the kernel space. Function invocations will then result in a system call to the kernel. So after implementing a thread library, a thread can be created & here there are two general strategies for creating multiple threads:

- Asynchronous
- Synchronous.

In asynchronous threading, the parent thread will create a child thread & then continue it's execution, in other words, the parent & the child will execute concurrently. They both execute independently from one another & usually don't share any info or data at all with each other.[1]

4.1 Pthreads

Pthreads is an API for creating & Synchronizing Threads. implementation of Pthreads specifications is up to the Operating Systems designers, which may do however they wish within the set limits. Most of the System implementation specifications are UNIX based, which includes Linux, Mac OS & Solaris. However, as Windows doesn't support Pthreads natively, there are Third-Party options to achieve similar functionality.[1]

4.2 Windows Threads

As described in the Pthreads sub-section above, Windows doesn't natively support Pthreads, however, the implementation for creating threads in Windows using the Windows Library is similar to the Pthreads technique.[1]

Threads are created in the Windows API using the CreateThread() function, similar to Pthreads this function takes a set of attributes. These attributes include: Security Information Size of the Stack Flag to indicate Suspension State of Threads. In situations where it is required to wait for multiple threads to complete, the *WaitForMultipleObjects()* function is used.[1] The *WaitForMultipleObjects()* function takes the parameters:

- Number of Objects waited for.
- Pointer to an Array of Objects.
- Flag Indicating if signal to all objects has been received.
- Timeout Duration (or Infinite).

It could look like this:

```
    WaitForMultipleObjects(N, ThreadArray, True, Infinite);
```

4.3 Java Threads

The Java APIs management & creation of threads is extensive & all Java Applications (No matter the size) including a main(String[] args) method runs on a single thread in the Java Virtual Machine (JVM). Since Java is a cross-platform language, all operating systems can take advantage of the Java API rich set of features.[1]

One out of two ways of creating threads in Java is to create a new class that is derived from the Thread class & overriding its run() method. The second option, which is more commonly used is to implement the Runnable interface:

```
public interface Runnable {
    public abstract void run();
}
```

Whenever a class implements the Runnable interface it must also include the run() method, & the code in this run() method is what runs on a separate thread.[1]

Creating a new thread object does not specifically imply that a new thread has been created. This is instead done within the start() method. When calling the start() method for a new object it does two things:

- Allocates memory & initializes a new thread in the JVM.
- Calls the run() method, making the thread eligible to be run by the JVM.

Data sharing between threads is easily done since both Windows & Pthreads can access globally declared data. In Java, this data is shared by passing references to the objects shared by the appropriate threads.[1]
A way this referencing is performed is through getters & setters like:

```
getSize() { return value; }
setSize(int value) { this.value = value; }
```

where getSize() returns the size value & setSize() takes a parameter value & sets the size.

5 Implicit Threading

5.1 Thread Pools

If you have a multi-threaded web server that sends & receives a request, it needs to create a separate thread to serve each new request received. While this is beneficial compared to creating a new process it has its disadvantages;[1] This concerns:

- The amount of **time** it takes **to create a Thread**.
  The usefulness of creating a Thread which will be **discarded once complete**.
- Allowing the system to run **simultaneous threads without any bounds**.
  Unlimited Threads would eventually exhaust the systems resources (CPU, Time or Memory).

Possible solutions to this could be using a **Thread Pool**. The idea behind using Thread Pools is that it would create a certain number of threads at start-up & place them into a "pool" (or collection) where they will wait for something to do during run-time.[1] So whenever the server receives a request it tells the thread to do its work, if one is available at this time. Then instead of being terminated as by default it would, after its work, return to the pool & wait again for a new request.[1]

Advantages to Thread Pooling:

- Servicing is faster than creating new threads.
- No unlimited threads running simultaneously.
- Enabling more diverse operations such as delays, timers & periodical executions.

Certain more sophisticated thread-pool architectures can change & adjust the number of threads in the pool dynamically depending on the usage patterns. The direct benefit of a dynamically changing thread-pool architecture is that it originally has a smaller pool & therefore it's less demanding on the memory usage when the system isn't under stress. One example of this is Apple's Gr& Central Dispatch. However, we will not discuss this in detail.[1]

The Windows API also has several related functions to thread pools which is similar to creating a thread with the Thread Create() function. In this case, a function is defined as a separate thread.[1]

Figure 4: Example of such a function

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
     * This function runs as a separate thread.
     */
}
```

Thereafter a pointer to PoolFunction() is passed to one of the thread pool functions in the API, & thread from the select pool can then execute this function.[1]

Example of this is the QueueUserWorkItem() which has three parameters:

- Pointer to the function that is supposed to be run: LPTHREAD START ROUTINE
- Parameter to be passed to the above function: PVOID
- Flag to indicate how the creation of threads are made & management of the execution of said threads: ULONG

An example of this would be:
QueueUserWorkItem(&PoolFunction, NULL, 0);

5.2 OpenMP

OpenMP is a set of compiler directives & APIs written in the language C, C++ or FORTRAN which is used to identify parallel regions as blocks of code that can run in parallel to each other. Developers can insert compiler directives to their applications to decide which blocks of code should run in parallel to each other during run-time.[1]

Figure 5: Example of this written in C

```
1     #include <omp.h>
2     #include <stdio.h>
3     int main(int argc, char *argv[]) {
4         /* sequential code */
5         #pragma omp parallel {
6             printf("I am a parallel region.");
7         }
8     /* sequential code */
9     return 0;
10    }
```

Once the OpenMP reaches line 5, "*#pragma omp parallel*" it creates an equal number of threads to the number of cores in the CPU of the system. Thus a dual-core CPU would generate two threads. Henceforth all the threads will then execute all the desired blocks of code in parallel to each other. Once a thread is finished with its task it is then terminated.

Additionally, to parallel executions during run-time, OpenMP allows developers to, for example, set the number of threads manually. They can also identify if threads are sharing information/data.[1]

5.3 Grand Central Dispatch

Grand Central Dispatch (GCD) is a technology developed for Mac OS X and iOS, & is an extention of the C language (known as **Blocks**). It allows developers to identify blocks of code which can be run in parallel to eachother. Much like OpenMP, GCD managed the majority of details containing threading. GCD schedules the blocks of code to be executed during run-time placing them on a dispatch queue.

Blocks of code placed on a process, which has its own unique serial queue (known as **Main Queue**), & is removed in First-in-First-out (FiFo). Once a block of code has been removed from the serial queue it must the complete execution before moving on to the next block of code.

Blocks of code placed on a concurrent queue are also removed in FiFo order. However, multiple blocks of code may be removed simultaneously. Allowing blocks of code to be run in parallel.[1]

6 Threading Issues

When designing a Multi-threaded program there are certain issues you have to consider. These include, but are not limited to:

- System Calls.
- Signal Handling.
- Thread Cancellation.
- Thread-Local Storage.
- Scheduler Activations.

6.1 System Calls

System Calls are actions such as exec() & fork(). Whereas fork() has slightly different functions depending on the application & operating systems it is designed to either duplicate all threads or the thread it was called upon. exec() however is designed to immediately overwrite all current processes on a thread. Thus creating a possible issue when first calling fork() & then exec() immediately after.[1]

6.2 Signal Handling

Signal Handling is used to notify a process of a particular event & it can either be of a synchronous or asynchronous type. In Single-Threaded Program signals are always delivered to a process. However, in Multi-Threaded Programs, the signal would either be sent to the Thread which the signal applies, to every thread in the process, to certain threads in the process, or to an assigned thread specifically designed to receive all signals for the process.

Every signal has a default signal handler which is run by the kernel whenever handling that specific signal. This signal handler can, however, be overridden by a user-defined signal handler. Certain signals are ignored completely (For example resizing a window) & others terminate the program (For example illegal memory access). However, signals always follow a certain pattern of being generated by the occurrence of a particular event then being delivered to a process & once delivered it must always be handled.[1]

Synchronous signals include illegal memory access & divisions by 0; which is sent to the same process that performed the operation which caused the signal to occur. The method for delivering a signal depends on the type of the generated signal. On the other hand, asynchronous signals are sent to a different process than the one which caused the signal to occur & is usually generated externally from this process. For example, a signal that is to terminate a process should be sent to all threads. This means the signal must find a thread that is not blocking that specific signal & send it to the first available thread to be handled.

Windows handles signals differently from UNIX based operating systems. It is roughly the same procedure as sending signals, though it differs slightly. Instead of having signals it allows for emulating them using asynchronous procedure calls (APC). whenever a notification is sent of a particular event the APC enables a user thread to specify a function that is to be called, much like signals, however it is always delivered to a particular thread rather than a process, as signals would.[1]

6.3 Thread Cancellation

Thread Cancellation involves terminating a thread before it has completed. A single thread that is to be canceled is referred to as the 'Target Thread' & cancellation of a 'Target Thread' may either occur by asynchronous cancellation; that is one Thread which immediately terminates the 'Target Thread', & Deferred Cancellation; where the 'Target Thread' periodically checks whether it should terminate itself properly.

In contrast Deferred Cancellation one thread is tasked with indicating if a ‘Target Thread’ is to be canceled. However, cancellation can only occur after a flag has been checked to determine whether it can be canceled. In these cases, the ‘Target Thread’ can be safely canceled, due to these threads being able to check these flags whenever.[1]

> "*[...] if multiple threads are concurrently searching through a database & one thread returns the result, the remaining threads might be canceled. Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further.*" [1]

Certain issues which may present itself during cancellation situations such as resources being allocated to an already canceled thread or a thread which is simultaneously being canceled while updating data with other threads. This could cause the system to not properly reclaim all necessary resources which are essential to the system running smoothly. This is especially apparent with asynchronous cancellation.[1]

> "*Actual cancellation depends on how the target thread is set up to handle the request. Pthreads supports three cancellation modes. Each mode is defined as a state & a type, as illustrated in the table below. A thread may set its cancellation state & type using an API.*

Figure 6: Cancellation Modes

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

> *As the table illustrates, Pthreads allows threads to disable or enable cancellation. Obviously, a thread cannot be canceled if cancellation is disabled. However, cancellation requests remain pending, so the thread can later enable cancellation & respond to the request.*" [1]

6.4 Thread-Local Storage

Data sharing among threads & one of the benefits with Multi-Threaded Programming. However, sometimes a thread might need a copy of some data & this is called Thread-Local Storage. Thread-Local Storage can be used to associate each thread to its unique id & it is unique to all threads.[1]

6.5 Scheduler Activations

Scheduler Activations is the communication between the kernel & the thread library. Coordinating between the kernel & the thread library dynamically; could be essential to optimize performance.[1]

The lightweight Process is the implementation of a many-to-many or two-level model in a system. To the user-thread library, the lightweight Process appears as a Virtual-Processor on which the application can schedule a user-thread to run.[1]

On a Single-Thread one, Lightweight Process is enough. However, an application running with Multi-Threads might require multiple Lightweight Processes to operate efficiently. For example, imagine you request five different “Read-File” operations simultaneously, then you would require five Lightweight Processes. If however you only have four Lightweight Processes the last “Read-File” would wait for another Lightweight Process is available.[1]

Participation

Presentation:

Student 3

- *Multicore programming*
- *Multithreading models*

Student 4

- *Threading issues*

Student 5

- *Implicit threading*

Student 6

- *Thread libraries*

Report:

Student 1

- *Introduction*
- *Multicore programming*
- *Multithreading models*

Student 2

- *Thread libraries*
- *Implicit threading*
- *Threading issues*

References

[1] P. G. G. Silberschatz, Abraham. Baer Galvin, *Operating system concepts*, 9th ed. Jon Wiley Sons INC, 2013.