Lecture 10: Concurrency Control

This lecture discusses how programming languages support concurrent programming. Topics to be covered include:

- Introduction
- Fundamental Concepts
- Semaphores
- Threading Models
- Java threads
- C# Threads
- C++ Threads
- Summary and Concluding Remarks

10.1 Introduction

Concurrency can be divided into the following areas:

- Instruction-level: Two or more machine instructions executing simultaneously
- Statement-level: Two or more (language-specific) statements executing simultaneously
- Unit-level: Two or more (language-specific) subprograms executing simultaneously
- Program-level: Two or more programs executing simultaneously

Instruction-level and program-level concurrency are best discussed in a course in computer architecture and/or operating systems design. Our focus will be statement-level and unit-level concurrency, since these areas relate to programming language issues.

There are two categories of concurrency:

- **Physical Concurrency:** Several program units from the same program execute simultaneously on one or more processors
- **Logical Concurrency:** A single processor is interleaved to operate in several threads

10.2 Fundamental Concepts

Here are a few fundamental concepts that need to be clarified (your course in Operating Systems Design would have clarified them but a quick review is useful):

A *task* (or *process*) is a subdivision of a program. A task differs from a subprogram in the following ways:

- A task may start implicitly or explicitly; a subprogram has to be called explicitly.
- A program may invoke a task and move on to other tasks; when a subprogram is invoked, control has to return to the calling statement.
- When a task completes, control may or may not return to the point of invocation; for a subprogram, control always returns to the invocation point.

Heavyweight tasks operate in their own address space, and have their own runtime stacks. *Lightweight tasks* share address space and runtime stacks.

Synchronization has to do with the order in which the tasks operate. Two strategies exist:

- **Cooperative Synchronization:** If task A invokes task B, it must wait until task B completes before it can continue.
- **Competition Synchronization:** Multiple tasks compete for a non-sharable resource (for instance a memory location). The first task to gain access to the resource holds it until it is finished with it. Other tasks must wait until the resource is released. Competition then continues.

10.3 Semaphores

A *semaphore* is a non-negative integer variable, used as a flag that controls access of a critical region. The idea of semaphores was introduced by Dijkstra in 1965.

Dijkstra defined two operations to manipulate the semaphore: The **P-operation** (from the Dutch word "proberan", which means, to test) allows a test; the **V-operation** (from the Dutch word "verhogen" which means, to increment) increments the semaphore.

If **s** is a semaphore variable, then the **V-operation** [increment] on **s** is given by

V(s): s:= s+1

This necessitates a *fetch*, an *increment* and a *store* sequence. The **V-operation** is indivisible — it must be performed in a single machine cycle to avoid deadlock.

The **P-operation** [test] on **s** is to test the value of **s**, and if it is not zero, to decrement it by 1. Thus

P(s): If s>0 then s:= s-1

This necessitates a *test*, *fetch*, *decrement* and *store* sequence. Again, the sequence must be indivisible in a machine cycle.

A process can only access a critical region if its semaphore is non-zero (i.e. s>0). When accessed, the **V-operation** [increment] takes effect for that critical region. When the process concludes use of a critical region, it issues a **V-operation** [increment] on the semaphore, thus s:= s+1

The end result is that the concept of mutual exclusion is enforced — no two processes can access a critical region at the same time. For this reason, the name given to the semaphore variable in the literature is *mutex*.

Mutual exclusion is automatic for sequential processes, but for parallel processes, it must be explicitly stated and maintained, hence semaphores.

10.4 Threading Models

Four common threading models are many-to-one, one-to-many, one-to-one, and many-to-many. Figure 10-1 provides graphic illustrations of these models. Full treatment is deferred to your course in Operating System Design. However, a cursory overview is provided here.

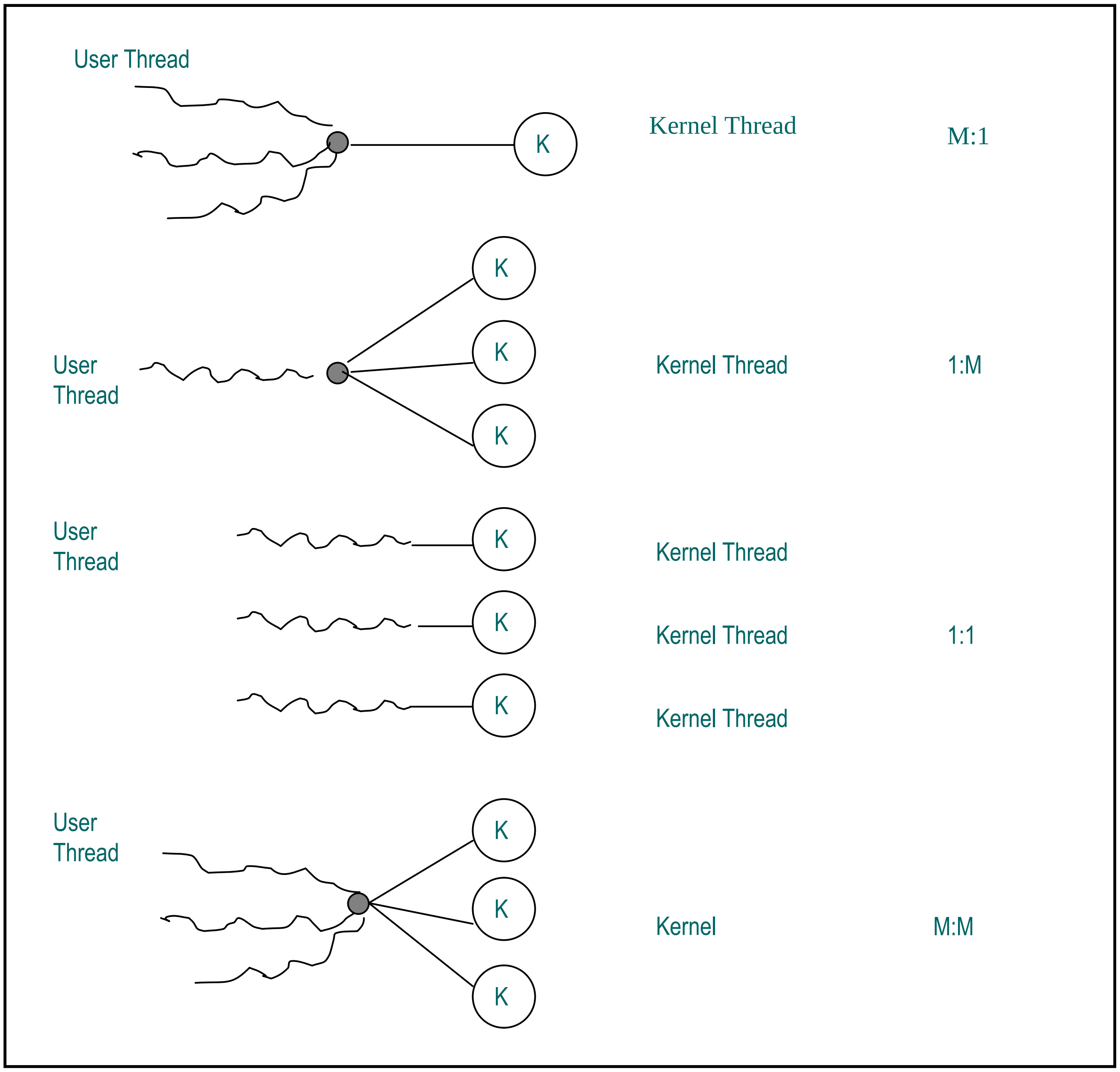
Many-to-One: In many-to-one (M:1) threading, several user threads are mapped to a simple kernel thread.

One-to-One: In one-to-one (1:1) threading, each user thread is mapped to a kernel thread. It provides more concurrent processing than the many-to-one model.

Many-to-Many: The many-to-many (M:M) model multiplexes several user threads to a smaller or equal number of kernel threads (the number of kernel threads may be application dependent or machine dependent).

One-to Many: In the one-to-many (1:M) model a single user thread may migrate to different processors, in order to be executed in the most efficient manner.

Figure 10-1 Threading Models



10.5 Java Threads

Java supports logical concurrency of lightweight tasks via threads in the following way:

- The **Thread** class is a base class consisting of two methods: The **run()** method describes the actions to be taken, and is normally overridden in subclasses. The **start()** method causes the thread to start execution. Other methods of interest include **interrupt()**, **sleep (long x)**, and **join()**, and **join(long x)**. Note, the **join(…)** method waits for the thread to die.
- To have methods that run concurrently, the programmer is required to create a class that inherits the **Thread** class. Alternately, create classes that implement the **Runnable** interface. Then override the **run()** method and specify the desired statements. Depending on the level of concurrent processing, you can specify several similar subclasses.
- Create a driver class that manipulates instances of the classes defined for concurrent processing.

Competition synchronization is achieved by specifying the **synchronized** keyword as a qualifier on each method that conforms to this regime. This means that once the method begins execution, that execution will complete before any other method begins execution.

Cooperative synchronization is achieved via appropriate invocation of the **wait(long x)**, **wait()**, **notify()**, and **notifyAll()** methods of the **Object** class.

10.6 C# Threads

C# handles threads in a manner that is similar to Java, but with some differences as summarized below:

- C# supports physical concurrency. Thus, any C# method can run its own thread by simply creating a **Thread** object.
- The **Thread** constructor must be sent an instantiation of a specially predefined class called **ThreadStart,** which in turn requires as its argument, the name of the method creating a thread.

Example:

```
public void MyMethod()
{…}
// …
Thread myThread = new Thread(new ThreadStart(MyMethod));
// …
```

Synchronization is achieved in any of three ways:

- The **Interlock** class allows you to manage a shared object or variable in a manner that is similar to the management of semaphores. In contains **an Increment (…)** method and a **Decrement(…)** method.
- The l**ock** statement is used to flag a critical section of code in a thread.
- The **Monitor** class is slightly more sophisticated class for more intricate concurrency situations.

10.7 C+ Threads

C++ does not have its own built-in threading mechanism; instead, it relies on and utilizes the multithreading features of the host operating system on which it runs. Fortunately, the worldwide C++ online community has written and provided many add-ons and tutorials on how to implement multi-threaded applications using C++. In the recommended readings at the end of the lecture, two such resources are provided.

10.8 Summary and Concluding Remarks

Here is a summary of what has been covered in this lecture:

- Concurrency can be divided into the following areas: instruction-level, statement-level, unit-level, and program-level.
- Two broad categories are physical concurrency and logical concurrency.
- Tasks are subdivisions of programs and may be heavyweight or lightweight.
- Synchronization relates to the order in which the tasks operate. Two prominent strategies are cooperative synchronization and competition synchronization.
- A semaphore is a non-negative integer variable, used as a flag that controls access of a critical region.
- Four common threading models are many-to-one (M:1), one-to-many (1:M), one-to-one (1:1), and many-to-many (M:M).
- Java supports logical concurrency of lightweight tasks via threads. C# threads are similar to Java's but with a few points of differences. C++ piggy-backs on the multithreading features of the host operating system on which it runs.

At first glance, you may be tempted to pounce on the failure of C++ to have its own multithreading mechanism and not piggy-back on the underlying operating system on which the language runs. On closer examination, you will realize that this is not a bad idea; it's part of a strategy that renders the C++ compiler one of the leanest and most efficient in its category.

As with each topic discussed in this course, you are encouraged to use the basic principles discussed as a set of guidelines for launching your own probe into other programming languages. The next lecture takes on the topic of exception handling.

10.9 Recommended Readings

[Codebase 2014] Codebase.eu. 2014. "Creating multi-threaded C++ code." Accessed on December 24, 2014. http://codebase.eu/tutorial/posix-threads-c/

[Pratt & Zelkowitz 2001] Pratt, Terrence W. and Marvin V. Zelkowits. 2001. *Programming Languages: Design and Implementation* 4th Edition. Upper Saddle River, NJ: Prentice Hall. See chapter 11.

[Sebesta 2012] Sebesta, Robert W. 2012. *Concepts of Programming Languages* 10th Edition. Colorado Springs, Colorado: Pearson. See chapter 13.

[Tutorialspoint 2014] Tutorialspoint. 2014. "C++ Multithreading." Accessed December 24, 2014. http://www.tutorialspoint.com/cplusplus/cpp_multithreading.htm