Lecture 11: Threads

Traditional operating systems were designed to handle processes with a single thread of control. Modern operating systems are designed to handle processes with multiple threads of control. For this reason, a discussion of threads is of paramount importance. The lecture proceeds under the following captions:

- Overview of Threads
- Multithreading Models
- Implementation Issues
- Summary and Concluding Remarks

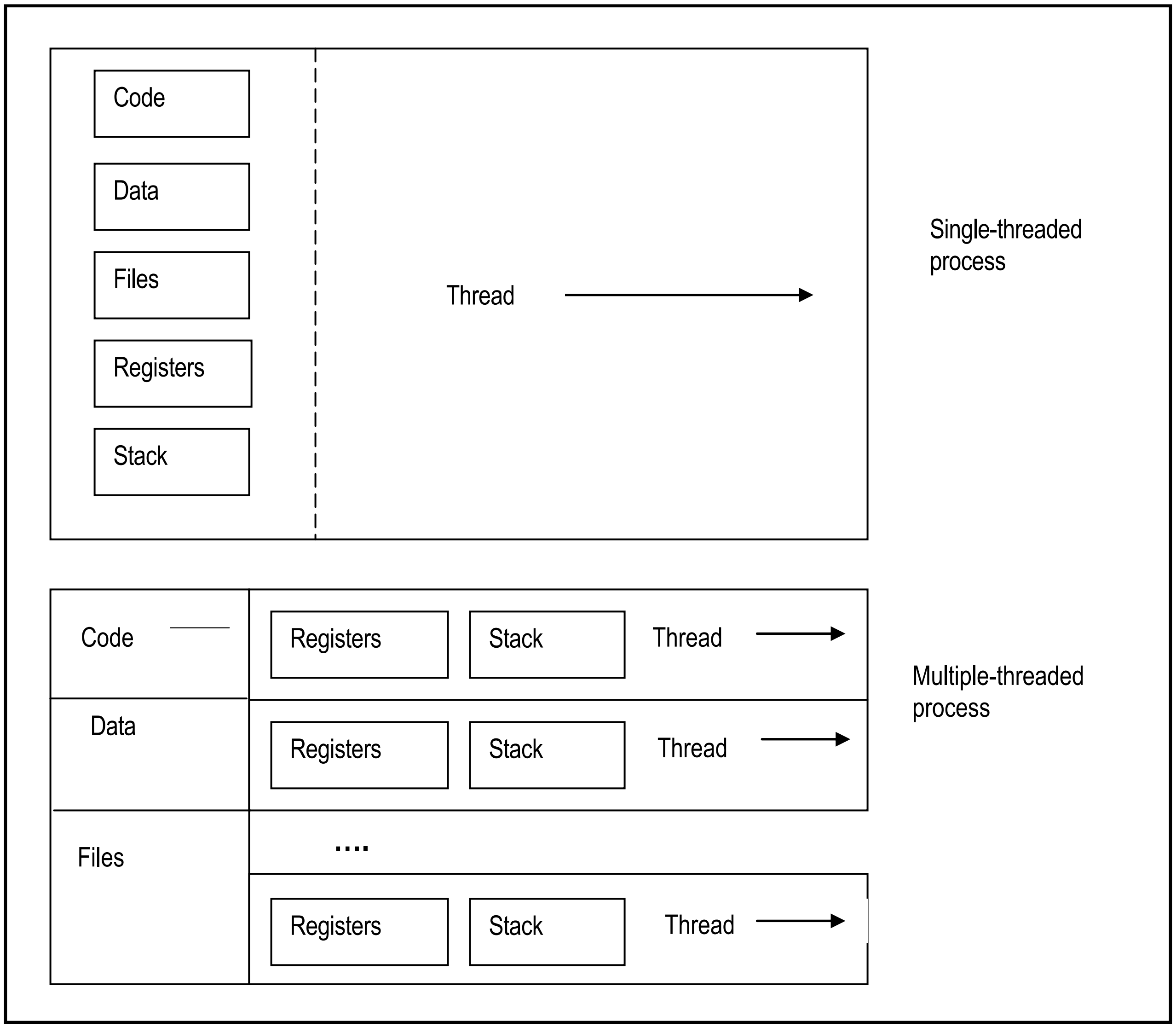
11.1 Overview of Threads

A *thread*, also called a *lightweight process* (LWP), is a basic unit of CPU utilization for a process. The thread consists of the following components:

- A thread ID
- A program counter
- A register set
- A stack

A traditional (heavy weight) process has a single thread of control. Figure 11.1 illustrates the difference between a single thread process and a multiple thread process.

Figure 11.1 Simple-Thread and Multiple-Thread Process



11.1.1 Rationale and Benefits

The rationale for multi-threaded system is obvious, as exemplified in the following examples:
- A Web server responding to hundreds of client requests for web pages, images, text, sound, etc.
- A network server responding to client requests from various nodes on the network.
- A database server responding to multiple client requests from applications running on a corporate network.

Multiple-threaded systems bring a number of significant benefits to the operating system arena, as summarized below:
- **Responsiveness:** A program can continue to run even if part of it is blocked, or locked in a lengthy operation. The end result is more responsiveness to the user.
- **Resource Sharing:** Threads share memory of resources of the process to which they belong. This results in improved efficiency of operation.
- **Economy:** Since threads share memory and other resources of their parent process, it is more economical to context switch among threads. Also, in most systems, creating and managing a process is far more difficult than creating and managing a thread (for example, in Solaris 2, creating a process is 30 times slower than creating a thread, while context switching a process is 5 times slower than context switching a thread).
- **Utilization of Multiple Processor Architectures:** The benefits of multiple threading are greatly enhanced in a multiprocessor architecture, where multiple threads can be running on different processors.

11.1.2 Thread Support

Support for threads may be provided at the user level for user threads, or the kernel level for kernel threads.

User Threads: These threads are implemented by a thread library at the user level (above the kernel). The thread library provides support for the creation, scheduling and management of threads. All this is done oblivious to the kernel and without its intervention. Generally speaking, user threads are faster (to create and manage) than kernel threads. However, their operation is to some extent, constrained by the design of the kernel (if the kernel is single threaded, then any user thread performing a blocking system call will cause the entire process to block, even if other threads are theoretically available to run). Example of user thread libraries includes POSIX Pthreads, Mack C-threads, and Solaris-2 UI-threads.

Kernel Threads: These threads are directly supported by the operating system's kernel. The kernel performs thread creation, scheduling, and management. Generally speaking, kernel threads are slower (to create and manage) than user threads. However, if there is a thread performing a blocking call, the kernel can schedule another thread in the application, for execution. Also, in a multiprocessor environment, the kernel can schedule multiple threads on different processors. Most contemporary operating systems support kernel threads.

11.2 Multithreading Models

Four common multithreading models are many-to-one, one-to-many, one-to-one, and many-to-many. Figure 11.2 provides graphic illustrations of these models.

Many-to-One: In many-to-one (M:1) threading, several user threads are mapped to a simple kernel thread. The advantage and drawback of this approach were mentioned in the previous section (easy to create but constrained by the kernel). These threads are common in Solaris.

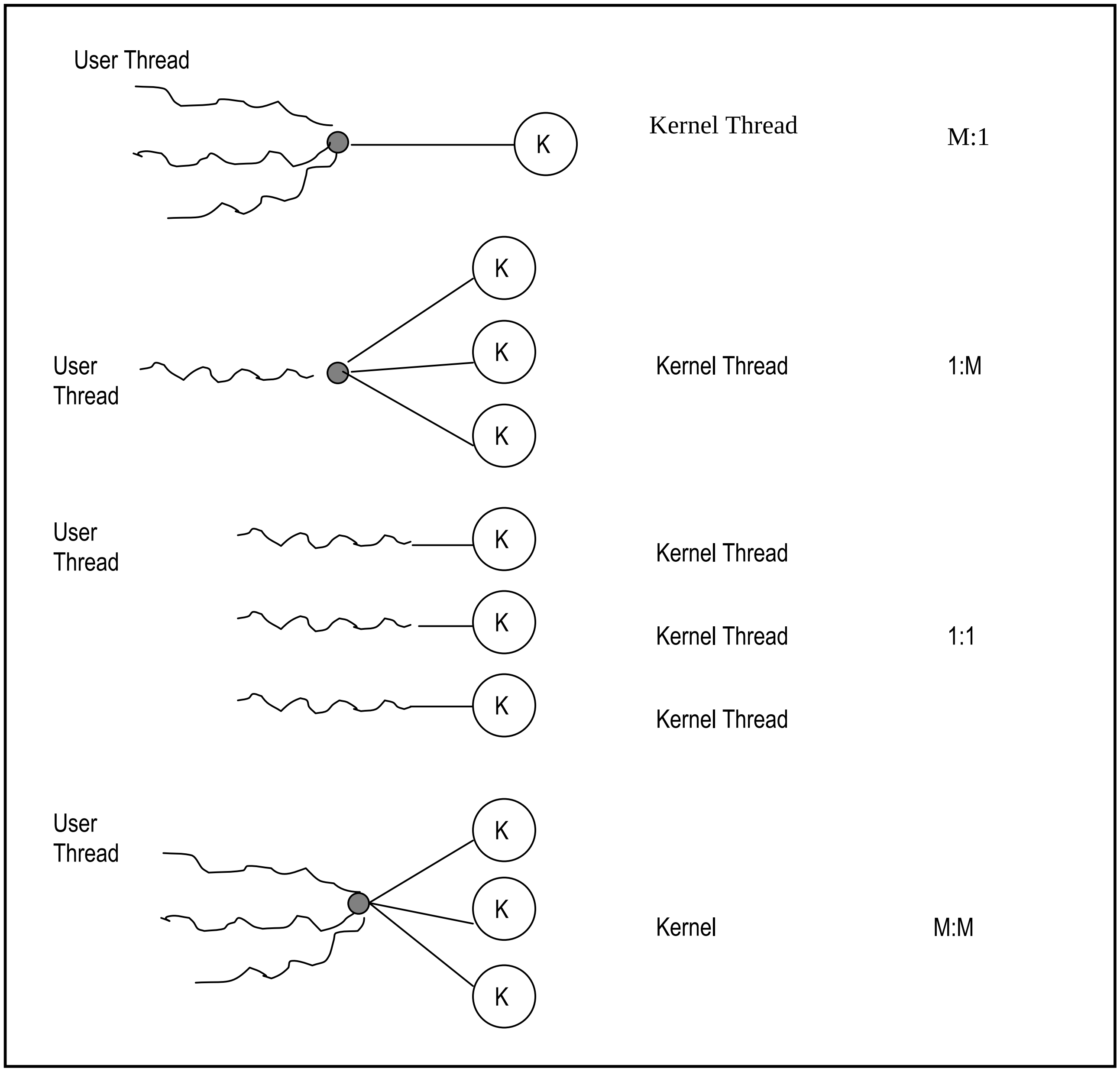
One-to-One: In one-to-one (1:1) threading, each user thread is mapped to a kernel thread. It provides more concurrent processing than the many-to-one model. Neither is it constrained by a single thread making a blocking call, as other threads can be scheduled to execute. Also, it is ideal for parallel processing on multiple processors.

The approach suffers from one major drawback: creating each user thread necessitates the creation of a kernel thread – an expensive activity. Because of this constraint, most implementations of the model restrict the number of threads supported by the system. Examples of 1:1 threads are found in Windows and Linux.

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 advantage of the approach is that the application developer is allowed to develop as many user threads a desired. Another advantage is that it is not constrained by a single thread making a blocking call, since the kernel can schedule another thread for execution. However true concurrency is not achieved since the kernel can schedule only one thread at a time. Examples of systems implementing the M:M threading model are Solaris-2, IRIX, HP-UX, and Tru-64 Unix.

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. The main advantage of this approach is load balancing particularly for complex calculations: Examples of systems that employ this approach are the Clouds operating system and the Emerald System.

Figure 11.2 Multithreading Models



11.3 Implementation Issues

Depending on the system, certain other implementation issues relating to the creation, cancellation and signal management of threads will need to be resolved.

Thread Creation: To illustrate, consider the fork system call on the Unix operating system **fork** is used to create a new process or thread. If one thread in a process P calls **fork**, does the new process, P2 duplicate all threads of P, or just the thread responsible for its creation? Resolution of this will depend on the application requirements.

Canceling Threads: Thread cancellation is the act of terminating a thread before it completes. For example, if concurrent multiple threads are searching a database for a data set and one thread returns a result, it is desirable to terminate the other threads. Another example is user accessing a Web site. A user may make the request to load a given page, and then decide to exit (before the requested page is loaded). The Web server will need to cancel the thread(s), which might have been active on the user's former request.

A thread to be canceled is often referred to as the *target thread*. Cancellation may occur in one of two possible ways.

- **Asynchronous Cancellation:** One thread immediately terminates the target thread.
- **Deferred Cancellation:** The target thread can be designed to periodically check if it should terminate, when it receives this message, it terminates it self.

Signal Handling: A thread must be able to respond to signals it receives. This is typically handled via a signal handler or event handler. The OO paradigm to software construction is particularly suited for event handling.

11.4 Thread Libraries

A thread library provides the programmer with the facilities for managing threads. Three common libraries are POSIX Pthreads, Win32 threads, and Java threads. For more information, see the respective product documentation.

By way of illustration, figure 11.3 shows the UML diagram for Java's **Thread** class and figure 11.4 shows a section of Java code for creating a simple multi-thread application. For example on Java threads, see [Oracle, 2011].

Figure 11.3: UML Diagram for Java's Thread Class

```
Thread
-----------------------------------------------------------------
static int MAX_PRIORITY // Maximum priority the thread can have
static int MIN_PRIORITY // Minimum priority the thread can have
static int NORM_PRIORITY // Default priority the thread has
-----------------------------------------------------------------
// Constructors
Thread() // Creates a new thread; used in the situation where inheritance is used for thread construction.
Thread (Runnable thisO) // Used when a class implements Runnable interface; thisO is an instance of the custom class
Thread (ThreadGroup thisG, Runnable thisO) // Places the new thread in thread group thisG; thisO is as above
Thread (Runnable thisO, String thisName) // Gives name thisName to the thread; thisO is as above
Thread (String tName) // Gives name thisName to the thread
Thread (ThreadGroup thisG, String thisName) // Places the new thread in thread group thisG; Gives name thisName to the thread
Thread (ThreadGroup thisG, Runnable thisO, String thisName)
    // Places the new thread in thread group thisG; Gives name thisName to the thread; ; thisO is as above
Thread (ThreadGroup thisG, Runnable thisO, String thisName, long stackSize)
    // Places the new thread in thread group thisG; Gives name thisName to the thread; ; thisO is as above; assigns a stack size

// Other Methods (this is not a comprehensive list)
static int activeCount()  // Returns an estimate of the number of active threads in the current thread's group and its subgroups.
static void dumpStack() // Prints a stack trace of the current thread to the standard error stream.
long getId() // Returns the identifier of this thread.
String getName() // Returns the name of this thread.
Int getPriority() // Returns the priority of this thread.
void interrupt() // Interrupts the thread.
boolean isAlive() // Checks if the thread is alive.
boolean isDaemon() // Checks if the thread is a daemon thread
boolean isInterrupted() // Checks if the thread is interrupted
void join() // Waits for the thread to die
void join(long waitTime) // Waits for a specified time (in milliseconds) for the thread to die
void setDaemon (boolean onOrOff) // Sets the daemon flag
void setName (String newName) // Gives a new name to the thread
void setPriority (int newP) // Assigns a new priority to the thread
void sleep (long sleepTime) // Causes the thread to sleep for a specified time in milliseconds
void start () // Causes the thread to start execution
```

11.5 One Operating System Examples

Linux: Through the **fork()** and **clone()** system calls, process threads can be created. All processes and threads are simply referred to as *tasks*. The **clone()** call may be invoked with the following flags:

- 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

11.6 Summary and Concluding Remarks

Here is a summary of this brief lecture:

- A thread is a basic unit of CPU utilization for a process. Older operating systems are single-threaded, while modern operating systems tend to be multi-threaded.
- Threads are useful in enhancing system performance.
- User threads are faster but less flexible than kernel threads.
- Four possible thread models are M:1, 1:1, M:M, and 1:M.
- Programmatically, thread creation and cancellation must give due considerations to the possible scenarios involved, and take appropriate actions.
- All the modern general purpose operating systems support some form of threading.

More could be said about threads; however, from this point on, the discussion would become very programmatic. That is left for more advanced courses in programming or operating system design and development. But from the foregoing discussion, you should have a good appreciation of threads in the arena of modern operating systems.

11.5 References and/or Recommended Readings

[Silberschatz 2012] Silberschatz, Abraham, Peter B. Galvin, & Greg Gagne. 2012. *Operating Systems Concepts*, 9th Ed. Update. New York: John Wiley & Sons. See Chapter 4.

[Stallings 2005] Stallings, William. 2005. *Operating Systems* 5th ed. Upper Saddle River, New Jersey: Prentice Hall. See Chapters 4 & 5.