Chapter 5

Threads and Events

5.1 Introduction

This chapter introduces threads as they relate to events and event based programming. There are really three closely related topics covered. First, we will discuss how using threads can solve some of the problems we face with long running event handlers. Unfortunately, this solution introduces several new problems, and we will briefly discuss each of these. Finally, we will discuss threads as event based systems. That is, rather than using threads in our event based programs, we will consider a thread as an event based system.

5.2 Background

A *thread of execution* is the smallest unit of computation that can be scheduled to run by the operating system (OS) or virtual machine (VM). It is responsible for carrying out some part of a computation, possibly an entire program, or possibly only some small part of a program. Every program, even the simplest *Hello World!* program has a thread of execution that runs the program. More complex systems often have multiple threads running concurrently. The basic concept is illustrated in 5.1.

Threads are different than *processes*. A process may be thought of as an entire program in execution. By contrast, a thread is on a finer granularity. It may be a only very small portion of a program.

Programs in many modern languages divide memory up into three basic areas:

- The *code segment* contains the executable instructions for the program.
- The *heap* stores dynamically allocated objects. Any time the programmer says `new` in Java or `malloc` in C, memory is allocated from the heap.
- The *stack* stores local variables. The stack consists of *activation records*. Every time the program makes a method call, a new activation record is created and pushed onto the stack to store the methods variables. When a method returns, its activation record is popped off the stack.

A program's threads share the code segment and the heap. Each has its own stack. Thus, each thread has its own calling and return sequence, and its own local variables. For example, in Figure 5.1 Thread 1 has made calls to methods in Objects A, C, D and E, in that order. Its call stack will

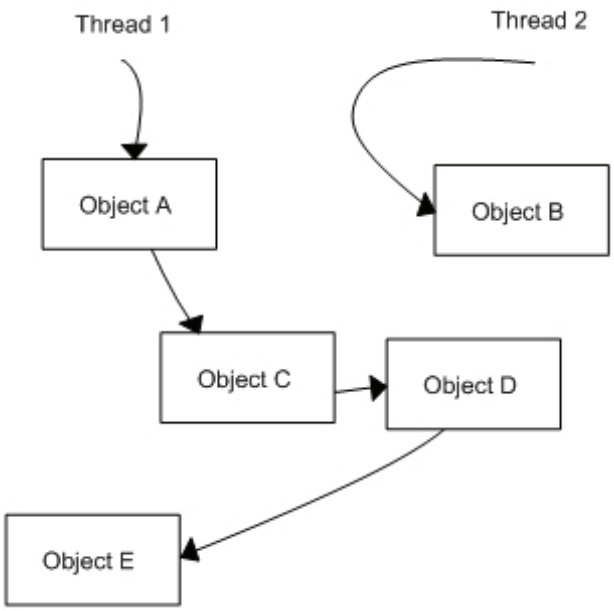

Figure 5.1: Multiple threads may be executing concurrently within the same application.

contain activation records for each of the calls. By contrast, Thread 2 has only made one call, to a method in Object B. Its call stack will only contain one activation record.

5.2.1 Threads and Concurrency

Conceptually, all running threads in a system are executing *concurrently*. We know that, in general, a computer doesn't have enough CPU power to execute all the threads at the same time, but we may think of them as running concurrently, while in reality, the operating system takes responsibility for swapping the threads in and out of the CPU, giving each a slice of processing time.

Many modern computers, including high end personal computers, contain multi-core processors. A core is the portion of the CPU that reads and executes program instructions. A single core processor reads and executes one instruction at a time. A multi-core processor can read and execute multiple instructions at a time, giving true concurrency. Each core reads and executes instructions independently of the others.

A multi-threaded program can run on a single core processor by swapping different threads into the processor, giving the illusion of true concurrency, see Figure 5.2.

A multi-threaded program running on a multi-core processor can achieve true parallelism, see Figure 5.3.

5.3 Why use Threads

Multi-threaded programs have several advantages over single threaded programs, including:

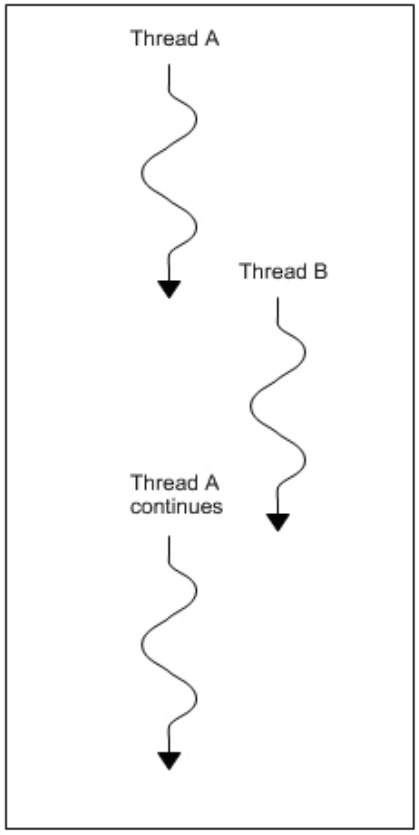

Figure 5.2: Threads alternate turns executing on a single core CPU.

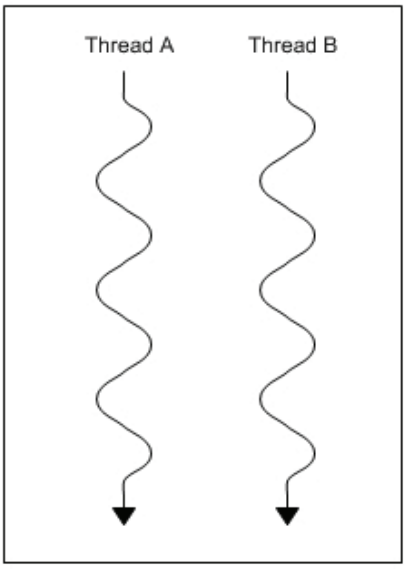

Figure 5.3: True concurrency can be achieved on multi-core CPUs.

Responsiveness

With a single thread of execution, the program behaves sequentially. It does one thing, then the next, then the next. If the user is waiting for the program, she must wait until the program gets around to responding. If the program is doing some heavy duty number crunching, accessing a remote database, there may be a significant delay. This is an important when dealing with GUIs or other interactive programs. If the delay is long enough, the user may think the program is hung and shut it down, while really it is just completing some processing.

Java, for example, uses a separate thread, named the *event dispatching thread* to handle GUI events. This lets the user continue to interact with the program, even when there is a delay in the event handling.

Java event dispatching thread only addresses part of the problem, however. Consider what happens if there are multiple handlers for the same event. Say, for example, you have a fire alarm system. A teacher, not wanting to go outside to smoke in the January cold, grabs a quick cigarette in the restroom and sets off the alarm[1]. Multiple things should happen simultaneously:

- The sprinkler system should spray water drenching the inveterate smoker and putting out his cigarette.
- The acoustic alarm should sound throughout the building, telling other teachers and students to exit the building a quickly as possible.
- The strobing visual alarm should be set off, giving hearing impaired students and teachers the same warning.
- The fire department should be notified.

[1]Possibly more realistically, a student wanting a few extra minutes to study for an exam intentionally pulls the fire alarm in the hallway.

We want all four responses to happen simultaneously. The system should not wait for the sprinkler system to complete before the acoustic alarm sounds. Nor should the acoustic alarm shut off before the strobing alarm begins. We definitely want the fire department to be notified as quickly as possible.

Threads are an ideal solution to making the system responsive, but, as we will see in the next section, in languages like Java we will need to manage them explicitly to take full advantage of their capabilities.

Efficiency

Ideally, there is the potential to speed up our computation by using multiple threads. If we can keep all the cores on the CPU actively executing our program, we can expect a speed up related to the number of cores. For example, if we have a quad-core processor, our program can run up to four times as fast.

There are a number of reasons why we never achieve this ideal case:

- First, when we write a program we may have no idea what kind of processor it will be run on. We might be developing it on our dual-core desktop, but the user may have a low end single core processor or a high end quad-core. This makes it impossible to guarantee to guarantee that we can take full advantage of all the cores.
- Second, even low end desktop computers run many programs simultaneously. Our program will be competing to use the CPU's cores, with no guarantee it can have all of them whenever it wants.
- Next, the OS or VM needs time to swap threads in and out of the CPU. If you have many active threads in your program, but are running on only a single core system, you may see a degradation in performance, rather than an improvement.
- Lastly, threads sometimes block. For example, if a thread is waiting for user input, it won't do any further computation until it gets the input. A thread may also be waiting for a resource being used by another thread. If we are not careful, this situation can lead to even more serious problems, like two threads waiting for each other, a problem known as *deadlock*.

Resource Sharing

An application's threads share the application's heap and all of them have access to the objects stored there. These objects may include data structures, open files and access to remote objects like databases. Having threads share these objects makes sense from the application's point of view, as something like a database of employee records may be needed for a number of different reasons by a number of different threads. Opening and closing databases can be time intensive, so sharing a database handle among the threads is an efficient use of resources.

5.4 A Multi-Threaded Event Handler

In the previous chapter we implemented a patient monitoring system to keep track of changes in a patient's blood pressure. Because we were implementing the event classes, not the client code, we didn't look at any of the event handlers. In this section, we return to that example, looking at how to implement the handlers. As you should recall, the handlers are executed by the event dispatching thread. In general, we can write handlers the same way we were doing it back in Chapter 2. That is, we implement the methods in the handler interface, taking care of the tasks required. However,

if there is the potential for a handler to block, or if a handler may run for a very long time, it is best to have the handler start a new thread to complete its tasks.

5.4.1 Simple Handler

The handler interface, `BloodPressureListener` contains three methods:
`public void systolicChange)(BloodPressureEvent e)`
`public void diastolicChange(BloodPressureEvent e)`, and
`public void warning(BloodPressureEvent e)`
one for each of the three possible causes of blood pressure events. We illustrate how to implement a handler just for the last one, *Warning* events.

```
66      // Processes warning events when systolic pressure is too high or too low.
67      public void warning (BloodPressureEvent e)
68      {
69           Patient patient = (Patient) e.getSource();
70           JOptionPane.showMessageDialog(
71                          null,
72                          "WARNING! Patient "  + patient.getName() +
73                          " has systolic pressure of: " + e.getValue());
74      }
```

Our handler is quite simple, it pops open a message box at the nurse's station telling them that the patient has a problem. There is an OK button on the dialog and the nurse clicks the button to close the dialog. The message box blocks the event dispatching thread, however, waiting for the nurse to click OK. This presents problems if there are other handlers for this event, e.g. notifying a doctor, or other, possibly more important events firing. They will be blocked until the nurse clicks OK.

5.4.2 Threaded Handler

The solution is to use another thread to process the warning event. The new thread should be started from within the handler. The event source should never be responsible for creating a new thread in which to execute the handler, since the source will not know if the handler even needs to be threaded, and unnecessarily creating a new thread each time a handler is called may itself result in system degradation.

The easiest way to have a handler's task execute in its own thread is to declare an inner class that implements the `Runnable` interface. There is one method in the `Runnable` interface, `run()`. Inside the `run()` method, place the handler's task code. The handler should then instantiate and start a new thread, using this inner class. The code snippet below shows how to do this for the `warning()` method. If this is the only handler for the event, you will not be able to discern and difference in how it runs for the previous section. If there are multiple handlers, however, this version will let those following this one begin running, without waiting for the message box to close.

```
66      // Processes warning events in a new thread
67      public void warning (BloodPressureEvent e) {
68          new Thread(new Warner(e)).start();
69      }
70
71      // We use a class implementing the Runnable interface
72      // to run the handler code in a new thread
73      private class Warner implements Runnable {
74          BloodPressureEvent e;
75          public Warner (BloodPressureEvent e) {
76              this.e = e;
77          }
78
79          // The thread's start method calls run
80          public void run() {
81              Patient patient = (Patient) e.getSource();
82              JOptionPane.showMessageDialog(
83                      null,
84                      "WARNING! Patient "  + patient.getName() +
85                      " has systolic pressure of: " + e.getValue());
86          }
87      }
```

5.5 Problems Arising from Multi-Threading

It would not be appropriate for us to end this discussion on multi-threaded event based programming without some mention of the problems that may arise. Multi-threaded programming remains a bug prone domain. You are advised to be careful when implementing multiple threads to avoid as many of these problems as possible. While a complete discussion of these issues is beyond the scope of this book, a good operating systems text will explore these topics in more detail.

5.5.1 Problems with Resource Sharing

Threads share the code and heap of the application. If multiple threads wish to access the same resource, problems can arise. Consider, for example, the problem of two threads, [T1] and [T2], wishing to increment the same integer variable, `count`, simultaneously. That is, both threads wish to execute

```
count++;
```

If `count` starts at 0, the correct new value is 2. On the assembly language level, a typical sequence of instructions to accomplish increment is:

```
66   load count a1          ; load count into register a1
67   incr a1                ; increment a1
68   stor a1 count          ; store register a1 back into count
```

Let the two threads running on separate cores intermix the instructions as follows:

```
[T1] loads count into its local register a1              // value of count = 0
[T2] loads count into its local register a1              // value of count = 0
[T1] increments its a1 register giving a result of 1     // value of count = 0
[T2] increments its a1 register giving a result of 1     // value of count = 0
[T1] stores its a1 back into count.                      // value of count = 1
[T2] stores its a1 back into count.                      // value of count remains = 1
```

`count` was only incremented once!

The basic solution to this problem is to provide a locking mechanism, so that a thread can claim exclusive use of a resource until it has finished its task. Unfortunately, locks introduce still more threading problems.

Deadlock

Deadlock occurs if two (or more) threads each possess a lock on a resource, and are waiting for the lock on the other's resource. This cycle of waits cannot be broken by the threads, as they are not aware that it exists.

Starvation

If one thread holds a lock and another thread wants the resource, the second thread waits until the first thread has relinquished the lock before proceeding. If multiple threads are waiting for the same resource, only one thread gets the lock next and is allowed to proceed. Starvation may arise if threads are given priorities and the lock is next given to the thread with the highest priority. It is possible for a low priority thread to never gain the resource because there is always a higher priority thread in front of it. The thread never makes any progress.

5.6 Threads as an Event Based System

In the first part of this chapter we saw that threads are a useful programming tool to improve our event based programs' performance and responsiveness. In this section, we explore threads from a very different perspective.

Threads may be thought of as a type of event based system! This shouldn't surprise you too much. by now that we can think of the threading of a program as event based, while the remainder of the program may not be. This was the same situation we saw with GUIs. The I/O for the GUI was event based, while the rest of the program followed more traditional approaches. The biggest difference between the two is where the events originate. With GUIs, the events originated with the user. With threads, most events originate with the thread scheduler, the OS or VM, or the program itself.

There are numerous implementations of threads, and they vary in some important details. Java has its won threads. MS Windows implements threads. Posix includes PThreads, an effort to standardize threads across various Unix platforms. In this chapter, we will keep the discussion general enough to apply to most of thread varieties, most of the time.

What do we mean when we say that threads are an example of an event based system? We have seen how events are useful when developing GUIs. They let us implement application classes and event handlers separately from the GUI components, e.g. the event sources. The separation of responsibilities implied by M–V–C is a powerful approach to developing GUI systems.

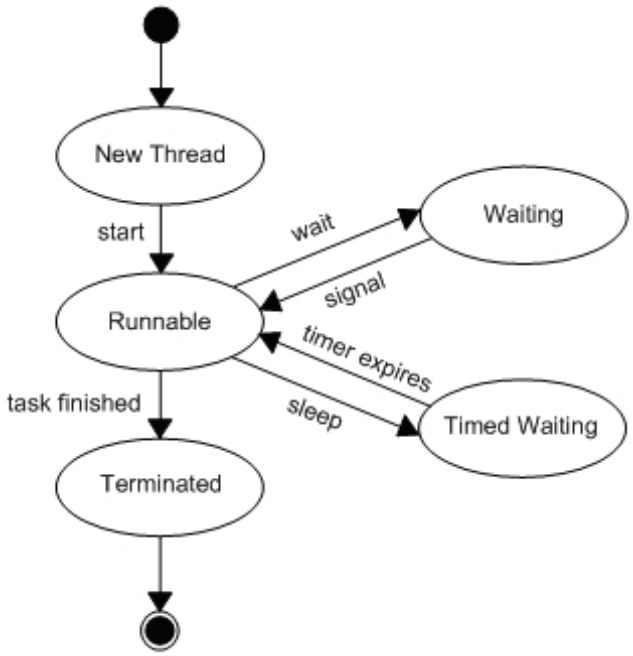

Figure 5.4: Threads have control state.

The same type of approach is useful whenever we need to decouple parts of a system. With threads, the decoupling is taking place between the application and the hardware on which it is running. As discussed earlier, threads need to be loaded into a CPU core to execute. At times, a thread also needs to be swapped out of a core, because it has exhausted its time allocation, or because it is waiting for a resource to become available. If other threads are waiting to execute, the current thread should give the CPU to one of the waiting threads.

Obviously, the OS or VM sits between the application and the hardware, and interacts with both. It takes responsibility for scheduling threads onto the CPU's cores. Hardware, OS and VM generate events that the threading system responds to.

In Chapter 1 we developed a conceptual model for event based systems. They had a number of different attributes. They were *state based*, contain *nondeterminism*, were *loosely coupled*, and had *decentralized control*. We discuss each as they relate to threads.

5.6.1 State Based

Figure 5.4 shows an overview of the various states of a thread.

A thread is created, in object-oriented programming typically through a call to a constructor. Creating the thread, however, does not automatically start is running. An additional call to a method named something like `start()` is generally required.

Once the thread is started, it may either be immediately assigned a processor core, or placed in a data structure, waiting for a core to become available. Similarly, while the thread is running, it may be asked to yield its core to another thread. This happens frequently in computing systems where there are many more processes and threads waiting to run than there are physical cores available. In our figure, a *Runnable* thread is any thread that is ready to run, whether currently running or waiting to run From the thread's point of view, it doesn't matter which, as it is ready to run in

either case.

While a thread is running, it may request a resource, e.g. open a file or connect to a database. This is done by calls to the operating system. These calls may take a significant amount of time for a couple of reasons. The resource may be slow by nature, e.g. reading from disk drives takes thousands of times longer than reading from memory; or the resource may be in use by a different thread, in which case our thread must wait for it to become available. In either case, the thread should enter a *Wait* state. Once the resource becomes available, the thread should be returned to the *Runnable* state. In Figure 5.4 the state in the upper right is for threads waiting for resources.

Sometimes it makes sense to wait for a specified amount of time. For example, in our stopwatch example in Chapter 3, we set timers to go off, updating the watches time. It may also make sense to have a combined state, where a thread returns to *Runnable* either when the requested resource becomes available or when a timer expires. A good example of this is a web browser. If a browser requests a page, it waits for the page to be loaded, or if no page is received after a given period of time it displays a default page with an error message. In Figure 5.4 *Timed Waiting* is the the lower right.

5.6.2 Nondeterminism

Nondeterminism means that it is impossible to determine exactly how a computation will proceed. The code a thread is executing tends to be deterministic. Our thread code is typically just ordinary Java, C++ or other code, the same type of programming we have seen since Computer Science 1.

Nondeterminism enters our programs only when there are multiple threads. Recall our incrementing `count` example from earlier in the chapter. We saw that there is the *potential* to get an incorrect result if two threads try to increment `count` concurrently. We might also get the correct result, however (see Exercise 3 at the end of this chapter).

The nondeterminism occurs because an executing thread can change state due to external events occurring at inopportune times, e.g. the OS or VM removes the thread from its core at a critical time during execution, giving the core to a different thread competing for the same resource.

5.6.3 Loose Coupling

Coupling refers to any way that objects interact within a computing system. Obviously, all parts of a system need to interact either directly or indirectly in order to accomplish a task, so coupling occurs. *Loose coupling* can occur in a couple of ways.

We can insert an additional object between two objects moving them further apart from each other. This is one of the roles that threads play when dealing with an application and an underlying OS or VM.

Decoupling Application from OS and VM

Threads provide a level of indirection between our application and the OS or VM. That is, we write our programs in a way that makes it looks like OS calls happen instantaneously, even though we know they don't. Reading from a file, or connecting to a remote database takes time, and typically our thread is removed from the CPU core to better utilize the core while our application waits. The application doesn't need to do anything to make the waiting work, however. The thread provides this service.

Thread Pooling

Another way that loose coupling can occur is to make the relationship between two objects dynamic, or changing over time. We have seen this repeatedly with the registration (and possible deregistration) of event handlers. Event handlers are registered with the event source at runtime. The action that a system takes when a button is clicked is determined by this registration. As we have seen, this is a looser from of coupling than direct method calls.

The same type of decoupling can happen with threads. The code a thread is to execute can be determined at runtime. We saw one way to do this in Java in Section 5.4, where a `Runnable` object was passed to the constructor of a `Thread`. This process can be more general than this, however. Threads can exist as objects without having application code assigned to them. When an application needs a thread, it can grab a free thread object, assign code to it, and start it running. This approach is known as *thread pooling*. Its primary advantage is that it saves on the overhead of creating (and later destroying) new thread objects every time one is needed by the application.

5.6.4 Decentralized Control

A thread is responsible for executing a section of code within our program. The thread has control over the execution, and in that sense, control is centralized.

This section isn't about who is controlling our program, however. It is about who is controlling our threads? Is there a centralized authority that decides when threads should change state? The answer is *No*. The thread is controlled from a variety of sources.

- The program is responsible creating and starting each new thread.
- The OS or VM maintains the list of *Runnable* threads and decides how to schedule onto CPU cores.
- A timer, under the control of the OS or VM, determines when a thread should relinquish its core.
- Other timers determine that a thread has waited long enough and return it to its *Runnable* state.
- OS or language libraries frequently determine that a thread should voluntarily relinquish its CPU core because it has made a request, say to open a file, that will take a long time to complete.
- Interrupt handlers tell the OS or VM that the request has been completed and that the thread should again be *Runnable*.

5.7 Summary

Threads play a central role in modern computing systems. Multi-core CPUs are becoming common place, making it possible for multi-threaded programs to be more responsive and efficient. Multi-threading poses challenges for programmers, as they need to understand locking mechanisms to prevent inappropriate resource sharing.

Threads may be thought of as a type of event based system. They are state based, contain nondeterminism, provide a way of decoupling an application from the underlying infrastructure, and are controlled in a decentralized fashion, all properties expected of event based systems.