CS211 Lecture: Concurrency, Threads; UML Activity Diagrams

last revised October 5, 2009

Objectives

1. To introduce the notion of concurrency
2. To introduce the Java threads facility
3. To introduce UML Activity Diagrams
4. To introduce the use of synchronized
5. To introduce the use of wait() and notify()

Materials:

1. Projectable of internal structure of a typical CPU
2. AWTThreadDemo1.java.- demo and code to project
3. Racer1.java - demo and handout of code
4. AWTThreadDemo2.java - demo and code to project
5. Handout of activity diagram for racers
6. Racer2.java - demo and code to project
7. Racer3.java - demo and code to project
8. Racer4.java - demo and handout of code
9. Racer5.java - demo and code to project
10. ATM Example system statechart for class ATM; code for classes ATM, EnvelopeAcceptor

I. Introduction

A. When we mentally visualize the computation being done by a program, we tend to do so in terms of a series of steps, with the program doing exactly one thing at any given time.

B. This mental model is consistent with the physical structure of most computers.

1. Most computers have a single CPU.

2. A CPU has a memory plus special register called the *program counter* (PC). At any given moment in time, the PC specifies a location in memory, which contains the next instruction to be executed.

3. The CPU repeatedly executes the following *fetch-execute* cycle:

   Fetch an instruction from the memory location specified by the PC
   Update the PC to point to the next instruction
   Execute the instruction just fetched

   PROJECT: Internal structure of a typical CPU

4. At any given time, the state of a computation can be specified by knowing the following values:

   a) The contents of memory
   b) The contents of the PC and other registers

C. Though computers physically do exactly one thing at a time, they often give the illusion of doing several different things simultaneously: We call this appearance of doing several things at once *concurrency*.

   1. *EXAMPLE:* On most computers, it is possible to have several different applications open at the same time. Further, it is possible for two different applications to appear to be doing computation simultaneously - e.g. you may have all of the following going on at the same time:

      a) A CD player program playing music
      b) A web-browser downloading a large file
      c) A game that you are playing while waiting for the download to finish.

   2. How is this possible, given that physically a computer is doing only one thing at any given instant of time? Note that the PC always points to the next instruction to be executed, which is always part of some one program. So how can it appear to be executing several different programs at once?

      ASK

   3. The answer lies in the fact that the computer processes instructions very fast (100's of millions or even billions per second on fast CPU's). A computer can give the illusion of doing several things at once by rapidly switching from task to task, giving a small slice of time to each before moving on to the next.

      a) If you were able to slow the system way down, you would actually observe that at any given instant it is executing just one program.

      b) How does switching from program to program take place? Two possibilities:

         (1) A program may *yield* the use of the CPU to another program when it is unable to proceed further until some external event occurs - e.g.

(a) A CD player can yield the CPU while it is waiting for the CD to to read the next chunk of data (head movement, rotational latency, data transfer).

(b) A web browser downloading a page can yield the CPU while it is waiting for the next packet of data to arrive

(c) A game program (at least one that doesn't do ongoing animation) can yield the CPU when it is waiting for the human player to select a next move.

(2) External devices may be allowed to *interrupt* the currently running program when some operation has completed, causing it to yield the CPU to some other program - e.g.

(a) A disk controller may issue an interrupt when a unit of data has been read or written. The CPU may be yielded to the program that was waiting for the data.

(b) A network controller may issue an interrupt when a packet of data arrives. The CPU may be yielded to the program that was waiting for the packet.

(c) The computer may contain an internal clock that issues interrupts periodically. The CPU may be yielded to some other program that is also ready to use the CPU, in round-robin fashion.

(3) Older systems did not support forcing a program to yield the CPU when their time quantum was up. (This is technically called pre-emption) On such computers, a "freeze" was typically a manifestation of a program that had gone into an infinite loop and was not yielding to other programs. This is still possible on more modern systems, if a program can somehow disable interrupts, though it is much less common.

4. We have introduced concurrency in terms of the CPU dividing its time between several distinct programs. It is also possible to have concurrency within a single program, where several distinct *threads* of execution are going on apparently simultaneously.

   EXAMPLE: a server that is simultaneously servicing several clients may use a distinct thread for each. From the standpoint of the software design, it looks like each client has a program dedicated to serving it.

D. Note, then, that concurrency is possible at different levels:

1. You can have distinct programs, each running in its own *process*. Each process has its own memory space, which ideally should be protected from other processes.

   a) A consequence of having distinct memory spaces is that processes cannot share information directly with one another - though they can share information indirectly through the file system. (A few operating systems make it possible for processes to share regions of memory as well, but that's not an issue for us in this discussion.)

   b) Thus, in particular, distinct processes cannot share objects.

   c) This is called *multitasking*.

2. You can have multiple threads within a single process, all sharing the same memory space. The distinct threads can share the same objects. This called *multithreading*.

3. A third possibility, which we will not pursue here, is to have more than one CPU in a single computer, in which case you can have true concurrency, not simulated concurrency. This is called *multiprocessing*.

   Many newer computers use "dual core" CPU's. On such a system, it is possible for the system to actually be doing two different things at the same time. However, the concepts we will discuss here don't change - the same issues arise, and in any case most systems give the appearance of doing even more than two things at once. Further, true multiprocessing involves even more conceptual separation than is the case with multicore CPU's.

4. Of course, different forms of concurrency can be combined - e.g. if you have multiple processes, each process can also have multiple threads. If you have multiple CPU's, each CPU can, in turn, have multiple processes and/or multiple threads.

II. Threads in Java and Similar Languages

A. One important feature of Java is that it incorporates support for threads as part of the JVM and the standard library.

Other new languages incorporate similar mechanisms, and much the same effect can be achieved in any language if the underlying operating system provides support for threads. We will disucss the Java approach here, but the basic concepts are transferable to other models.

B. It is fairly easy to create Java programs that use multiple threads.

1. In fact, you've already done so without being aware of it, since even the simplest Java program uses multiple threads.

a) Every Java program has a main thread that executes the main() method of an application or the `start()` method of an applet.

b) Every Java program has one or more implementation-supplied background threads that handle various "behind the scenes" tasks - e.g. garbage collection.

c) Java programs that use the awt have one or more implementation-supplied threads that perform various awt tasks, as well.

EXAMPLE: `AWTThreadDemo1.java`

(1) Run. Note how clicking buttons changes direction of counting.

(2) Examine code - show `count()` and `actionPerformed()` methods. Note that two methods perform the major tasks: `count()` increments/decrements and displays the count value, and actionPerformed responds to clicks on up and down by setting the increment value to +1 or -1 as the case may be.

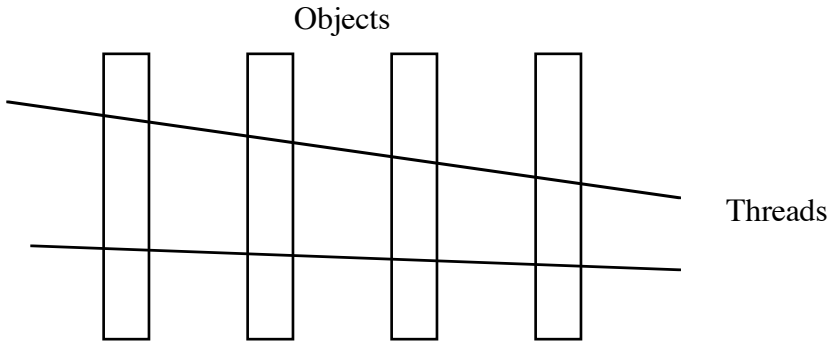
(3) How does execution switch from counting to modifying the increment when a button is clicked? These two methods are actually executed by two different threads: `count()` is executed by the main thread, and `actionPerformed()` by the awt event thread. The awt event thread is a part of the Java implementation that waits for a gesture that would cause an event to occur, and then calls the handler for that event.

(4) Comment out the time waster loop in the source code, recompile and run. Note that:

(a) It now counts much faster

(b) Only a small fraction of the values computed are actually displayed. This is because another thread - the awt painting thread - is actually responsible for updating the display. It is triggered each time the label contents is changed - however, it takes long enough to redraw the label once that the count is bumped up/down many times during the same period.

2. Threads in Java are orthogonal to objects: a given thread may access many objects, and a given object may have its methods performed by many threads.



E.g. in the example just given:

a) The main thread accesses the `AWTThreadDemo1` object, the frame, and the various components that are part of it - including, in particular, the label that displays the count value.

b) The event thread accesses the `AWTThreadDemo1` object.

c) The painting thread is given a task to do when the event thread modifies the text of the label that displays the count value, and then accesses the label to update its visual display on the screen.

C. While all Java programs make implicit use of threads, it is also possible to *explicitly* use threads in a Java program.

Why would one want to do so?

ASK

1. Some applications lend themselves to using multiple threads - the logic of the application is cleaner this way.

EXAMPLE: Racer1 program

a) DEMO

b) HANDOUT, DISCUSS code for Racer1.java

2. A server program may use a separate thread for servicing each client. This tends to produce cleaner, more modular code.

3. A web browser that is downloading a large movie may start playing it before it is completely downloaded, finishing the download while earlier portions are playing. This is typically done by using two separate threads: a "producer" thread that carries out the download, and a "consumer" thread that plays the file. (Of course, the speeds must be such that the consumer doesn't catch up with the producer or the movie has to stop playing.)

4. In a program that uses the awt (including Swing), any computation that is performed by an event handler is, in fact, done by the awt event thread. A consequence of this is that, while one awt event is being handled, no other awt events can be responded to.

   a) For this reason, it is good practice to minimize the amount of computation done by event handlers or methods they call directly.

   b) If handling an event requires a great deal of processing, or entails the risk of going into an infinite loop, it is better to delegate this to a separate thread.

   EXAMPLE: AWTThreadDemo2.java

   (1) DEMO - Note how it is unresponsive to stop button until count reaches 100

   (2) SHOW CODE - Note that the problem is that both `count()` and stop() are executed by the awt thread - therefore stop() cannot be executed while the thread is busy doing `count()`

III. Using Threads in Java

A. While we have said that Java threads are conceptually orthogonal to objects, the `java.lang` package includes a class `java.lang.Thread`, and an object of this class is needed for each thread in a given program. The `Thread` object serves as a mechanism for creating and accessing the thread. However, the thread itself is a flow of control, not an ordinary object - i.e. the thread is conceptually distinct from the `Thread` object that provides access to it.

1. To create a new thread, one must first create an object of class `java.lang.Thread` or a subclass.

2. Then, the associated thread must be started. This is accomplished by activating the `start()` method of the `Thread` object.

3. The code that is to be executed by the new thread must be specified in one of two ways.

   a) Create a class that implements the `Runnable` interface (which requires a run() method), and pass an instance of this class to the constructor of class `java.lang.Thread`. Create and start the new thread. The code that the new thread executes is the `run()` method of this `Runnable` object.

   EXAMPLE: Racer1.java

   (1) This class defines a main method that creates four instances (racer objects).

   Each racer object is a GUI component that has a color and keeps track of a position (tbat goes from 0 to 100) - see instance variables on the bottom of page 2.

   Each racer object can draw itself as a partially filled in boix on the screen. (See `paint()` method on page 2.)

   Each racer object is added to the GUI just after it is created.

   (2) The main program also creates four threads (one for each racer) in the main program immediately following creation of the racers. Each thread is associated with a racer that is specified when it is created (parameter to constructor). Thus, each racer is actually represented by two objects: a Racer object and a Thread object.

(3) Each thread is started by invoking its `start()` method, just after the threads are created.

(4) Each thread executes the `run()` method of the racer object is is associated with (page 2). This `run()` method sleeps for a random amount of time, increments its position, and then redraws itself. The `run()` method exits when the position reaches 100, at which point the corresponding thread terminates (this happens automatically).

(5) While the racers are running, there are actually five threads in operation - the four racer threads, plus the main thread that created them. The main thread executes the `join()` method of each thread in turn (bottom of page 1).

This method causes the main thread to wait until the racer thread has completed - at which point the two are joined into one thread. Because the main thread waits, in turn, for each racer, it does not get out of the loop until all four racers have terminated, at which point it prints the message `"All racers are done"`.

b) Java allows an alternate method for doing the same thing - one can subclass `java.lang.Thread` by a subclass that has its own `run()` method.. This is somewhat simpler than creating a `Runnable` object and then using that to create a thread (half as many objects involved) - but can't be used in this case because Java does not allow multiple inheritance, so creating a subclass of `java.lang.Thread` means that the object containing the `run()` method cannot subclass any other class. Creating a separate implementation of the `Runnable` interface allows the object containing the `run()` method to subclass some class as well (e.g. class Racer1, which subclasses `JPanel`) (However, we will use this simpler approach in the lab you will do on threads.)

4. Actually, the Java language recognizes two distinct categories of threads:

a) User threads execute user code. (E.g. the main program is executed by a user thread, and threads that it creates are typically user threads.) The five threads in our racer example are all user threads.

b) Daemon threads typically execute system code. (E.g. the garbage collection thread etc. are daemon threads.)

c) The basic distinction is this: when all the user threads for a given program have terminated, the program itself terminates. Daemon threads can still be in existence; they are terminated when the last user thread terminates.

d) Note that the awt threads are actually set up as user threads. This is because it is quite common for the main program of a GUI application to simply set up the GUI and then terminate. If the awt thread(s) were daemon threads, the program would terminate at that point, before the user could do anything!

B. Each thread has a number of properties - though only sophisticated programs typically make use of these.

1. A name

2. A priority: At any given time, there is one current thread - the one that is currently using the CPU.

a) There may be other runnable threads - threads the could use the CPU.

b) The relative priorities of the runnable threads control how CPU time is allocated between them.

c) However, the details of how the priority is used in scheduling threads is implementation-dependent; therefore, the ability to control execution using priorities is limited for software designed to run on multiple platforms.

3. A `ThreadGroup`. Threads belong to groups - we won't discuss this further.

C. In addition to allowing for the creation and starting of threads, the class `java.lang.Thread` also provides some mechanisms to allow for controlling the execution of threads. We will only look at a few of these.

1. It is important to note that these methods are of two general kinds.

a) Some of these methods are instance methods, and must be applied to a particular `Thread` object. They affect the thread that the

`Thread` object controls, and thus allow one thread (the one that calls the method) to affect another (the thread controlled by the `Thread` object whose method is called.)

b) Others are class (static) methods, and affect the thread that calls them.

2. `public static native void sleep(long millis)`
   `throws InterruptedException`

   causes the current thread to sleep for a specified period of time

3. `public void interrupt()`

   allows one thread to send a signal (an interrupt) to another thread. If the destination thread is sleeping or waiting, it gets an immediate InterruptedException that awakens it.

4. `public static boolean interrupted()`
   `public boolean isInterrupted()`

   allows a thread to find out if it has been interrupted. The first form of the method tests the current thread; the second tests the thread controlled by the `Thread` object to which it is applied.

5. `public final void join(long millis, int nanos)`
   `throws InterruptedException`

   `public final void join(long millis)`
   `throws InterruptedException`

   `public final void join()`
   `throws InterruptedException`

   cause the current thread to wait for the specified thread to terminate, before continuing execution. The first two methods put an upper bound on the waiting time, after which the thread that was waiting continues whether or not the other thread has terminated. (Not needed in most cases - but sometimes necessary as in the Racer example - we'll see more of a need for this in the next version.)

6. `public static Thread currentThread()`

   returns the Thread object representing the current thread.

IV. UML Activity Diagrams

A. The chapter in the book that you read for today discussed a type of UML Diagram called an Activity Diagram. Such a diagram can be constructed for a simple program, but it is particularly useful for showing processes in which there is concurrent processing.

EXAMPLE: HANDOUT: Activity Diagram for Racer Problem

B. Features:

1. Rounded rectangles represent activities.

2. Arrows represent flow from one activity to the next. Each activity is assumed to start as soon as its predecessor completes.

3. Where there is concurrency, the diagram shows "forking" of one thread into two or more, and joining of two or more threads into one. (Forks and joins should match.)

4. The diagram uses "swim lanes" to show the various parts of the system that are performing a task concurrently.

C. *ANOTHER EXAMPLE:* Figure 8.10 on page 209 of the book. In this case the diagram is showing concurrency between three humans and the computer.

V. The Critical Section Problem

A. As soon as we allow for the possibility of a program containing two or more distinct threads, we raise the question of how they can *coordinate* their activities and avoid interfering with one another.

EXAMPLE:

1. Consider the racer program again. As it stands, each racer keeps running until it completes, so we have no way of knowing who won except by careful observation.

2. Suppose, instead, we add a `StringBuffer` that allows the winning thread to report its name.

   a) We will pass this as a parameter to the constructor of each racer, and keep a reference to it as an instance variable in each racer.

   b) Note that all four racers *share* the *same* `StringBuffer`.

   c) We set the initial contents of the `StringBuffer` to empty.

   d) When a thread finishes, it checks to see if the `StringBuffer` is empty. If it is, it writes its name into the `StringBuffer`. (We have to check first, else threads that finish later will overwrite the name of the winner). At the end of the race, the main program writes the results. The code to do this can be added to the end of the `run()` method of the racers.

   SHOW CODE - `run()` method in Racer2.java

   DEMO

3. Is this code correct?

   ASK

   Surprisingly, the answer is no! Although it will work correctly most of the time, it can sometimes produce the wrong result. Consider the following scenario: suppose Red finishes with Green close behind it. Suppose, further, that due to the way the threads are scheduled, the following series of events occurs:

   a) Red finishes, and checks to see if the `StringBuffer` is empty - it is.

b) Green finishes, and checks to see if the `StringBuffer` is empty - it still is.

c) Red writes its name into the `StringBuffer`

d) Green writes its name into the `StringBuffer`.

e) Although Red won, Green is reported as the winner!

f) It may be argued that this scenario depends on the race being very close, and even then is improbable. Try telling that to runners in the Olympics! The fact that a scenario like this is rare does not mean its impossible, and the insidious thing is that finding such an error during testing, or making it repeat itself during debugging, is virtually impossible. Thus, the only way to produce correct concurrent software is to make sure such a scenario *cannot* occur.

4. To see that this is really a problem, we will run a version of the program that has been modified to insert some extra delay into the finishing code, between the time that the thread checks the contents of the `StringBuffer` (and sees that it's empty) and the time that the thread writes new content into it.

   DEMO: Racer3

   SHOW CODE at end of `run()` method - note that the logic is the same, but that delay loops and println's have been added.

B. The problem we have just encountered is an example of a common issue in concurrent systems called the *critical section problem*.

1. A critical section exists with regard to some piece of shared data if there is a section of code such that we must not allow some other thread to access the object while one thread is executing this code. In this case, the critical section for the `StringBuffer` is the code that extends from the if to the write - once one thread has started to execute the if, no other thread can be allowed access to the `StringBuffer`'s contents until the first thread has finished its write, or incorrect results can occur.

2. In Java, critical sections are protected by locking what is called the *monitor* of an object. (The term comes from theoretical work on critical sections done in early 1970's). This is accomplished by using the keyword `synchronized`.

HANDOUT, DISCUSS code for Racer4.java - a correct solution to this problem.

a) In brief, once one thread reaches the `synchronized` statement, it *locks* the `StringBuffer` object, excluding all other threads from proceeding past the `synchronized` statement until the first thread has finished writing it - at which point, future threads will see the `StringBuffer` as non-empty, and will not try to declare themselves the winners. (Any subsequent thread that reaches the `synchronized` statement while the first thread has the object locked will be forced to wait until the first thread releases the lock. Moreover, only one thread at a time will be allowed to lock the object - subsequent objects will have to wait their turn, one at a time.)

b) DEMO this program.

DEMO Racer5 - this program with additional delay and printing code used in Racer3. Note that it now works correctly.

c) In this case, the `synchronized` statement explicitly specified the object to lock. Often, it turns out that when a method is invoked, the object that needs to be locked is the "this" object of the method. In such cases, the method itself can be declared `synchronized`, as follows:

```
someMethod()
{
   synchronized (this)
   {
        --- method body
   }
}
```

is equivalent to

```
synchronized someMethod()
{
   --- method body
}
```

The latter form is preferable, both because it is shorter, and because declaring the method to be `synchronized` makes it clearer to the reader what is happening. In fact, this form of the `synchronized` statement is probably the most common.

d) Note that synchronization allows a thread to lock a particular *object* - not a particular body of code. Thus :

(1) If two threads encounter the same `synchronized` statement but referring to distinct objects, they can both proceed at the same time.

(2) If two threads encounter two different `synchronized` statements referring to the same object, only one can proceed at a time.

EXAMPLE: One place this may be used is in software accessing bank accounts. Suppose we had two methods belonging to class BankCustomer, defined as follows. (Assume that money amounts are represented in cents)

```
/** Return total of balances in all of a customer's
accounts. */
synchronized long totalBalance()
{
     long total = 0;
     Iterator it = accounts.iterator();
     while (it.hasNext())
          total += ((Account)
               it.next()).balance();
     return total;
}
synchronized long transfer(long amount,
     AccountType from, AccountType to)
     throws InsufficientBalanceException
{
     ((Account) accounts.get(from)).
          withdraw(amount);
     ((Account) accounts.get(to)).
          deposit(amount);
}
```

It is important that these methods be synchronized. This ensures that if one thread starts one of these methods and a second thread tries to start the other (or the same method, for that matter), the second thread will have to wait until the first thread is finished.

Absent this, we could report a balance that is too low if we got the balance of the from account after it was reduced by the transfer amount, and the balance of the to account before it was increased by the transfer amount.

(3) Note further that a given thread may lock the same object more than once - i.e. one synchronized method can call another method that synchronizes on the same object. This is permitted, and does not cause deadlock.

e) A weakness in the Java solution to the critical section problem is that while one thread has locked an object, other threads can still access it through code that is not specified as `synchronized`.

(1) Java is defined this way because locking an object involves a fair amount of overhead, so we don't want to do it unless we have too.

(2) However, this leaves open the possibility that a programmer might forget to specify that a given section of code is synchronized when it should be, negating the protection afforded by declaring some other section of code for the same object to be synchronized.

(Sort of like the possibility that one roommate might lock the door of the room and the other roommate might forget to lock it, leaving the first roommates' stuff vulnerable.)

C. This solution to the critical section problem is not without its risks, however. If synchronization is not done carefully, it is possible to create a situation known as *deadlock*. This happens if something like the following occurs:

Thread 1 locks some object A
Thread 2 locks some object B
Thread 1 now tries to lock B while it still holds the lock on A
Thread 2 now tries to lock A while it still holds the lock on B

D. Critical sections and deadlock will be discussed in much greater detail in the operating systems section of the Computer Systems course. Tune in then for further details!

VI. Explicit wait() and notify()

A. Sometimes, it is necessary for some thread to wait until some other thread has done something. In this case, it is possible to make use of the methods `wait()`, `notify()`, and `notifyAll()` that are defined in class `Object`, and therefore available for all objects.

1. A thread that holds a lock on some object can execute the `wait()` method of that object. When this occurs:

   a) The thread's lock on the object is released, so other threads can access it.

   b) The thread that executed the `wait()` is rendered unable to proceed - it is said to be *blocked* on that particular object.

2. Some other thread (which must now hold the lock on this object) may subsequently execute the locked object's `notify()` method.

   a) When this occurs, one thread that was blocked on the object is unblocked. (If several threads are blocked on the same object, there is no guarantee as to which is unblocked.)

   b) The thread that was unblocked may proceed after re–obtaining the lock on the object.

3. It is also possible to use the `notifyAll()` method of an object to unblock *all* threads that are blocked on that object - though they will, of course, have to proceed one at a time since a thread that was blocked must re-obtain the lock on the object before it can proceed.

4. The `wait()` method can optionally specify a timeout value. If it is not notified within this time period, the thread is unblocked anyway.

B. One place where the wait/notify mechanism is useful is if a thread can only proceed if some condition is true concerning a synchronized object. This often occurs in producer-consumer problems.

EXAMPLE: Consider the problem we discussed earlier where one thread is downloading a large movie file and another thread is playing it. Clearly, the second thread cannot be allowed to get ahead of the first - i.e. it can't play a frame that has not yet been downloaded. This might be handled as follows. Assume we have an object that represents the movie being downloaded, with methods as follows:

```
/** Add a newly downloaded frame to end of movie */
synchronized void putFrame(Frame f)
{
   -- store f appropriately
   notify()
}

/** Get the next frame to be shown */
synchronized Frame getFrame()
{
   while there is no frame available
        wait();
   return the next frame
}
```

This simple mechanism allows the consumer thread to wait just long enough for the next frame to arrive, if necessary.

C. Another place where `wait()` and `notify()` is useful is in cases where we want to avoid having the awt event thread do extensive computation. In this case, we use a separate thread to do the computation, which waits until the awt thread notifies it that an appropriate event has occurred.

There are several examples of this in the ATM Example system.

1. EXAMPLE: SHOW Statechart for class ATM, then

SHOW CODE for class ATM

a) Note that ATM implements `Runnable`, and has a `run()` method. When the simulation starts up, a `Thread` is created to execute this (i.e. there is a special thread for actually running the simulation.)

b) The `run()` method uses `wait()`/`notify()` in two places:

(1) When in the OFF_STATE, the thread waits. It will be notified by a call of `switchOn()` by the awt thread.

(2) When in the IDLE_STATE, it waits. It will be notified either by a call to `cardInserted()` or by a call to `switchOff()` from the awt thread

2. Threads with wait/notify are also used for the simulation of the Keyboard and the EnvelopeAcceptor.

   a) In the former case, the main thread waits until the user clicks a button simulating one of the keys on the keyboard.

   b) In the latter case, the main thread waits until the user clicks the button to insert the envelope - or until a timeout occurs.

      SHOW CODE for `acceptEnvelope()` method in class `SimEnvelopeAcceptor`.