CPS221 Lecture: Threads

last revised 8/28/14

Objectives

1. To introduce threads in the context of processes
2. To introduce UML Activity Diagrams

Materials:

1. Diagram showing state of memory for a process
2. Diagram showing code being executed by a thread and corresponding stack
3. Diagram showing thread states
4. Diagram showing single-threaded vs multi-threaded process
5. UnthreadedRacers and ThreadedRacers to demonstrate + code to project
6. AWThreadDemo.java to demonstrate
7. AWTBlockDemo.java
8. Handout of code for ThreadedRacers
9. UML Activity Diagram for Racer problem

I. Processes and Threads

A. As you know, abstraction is a fundamental design tool in many areas of computer science. In the realm of operating systems, the fundamental abstraction is the notion of a PROCESS.

1. A process is a PROGRAM IN EXECUTION.

2. At any given time, the status of a process includes

a) Information about resources that may be allocated to the process (e.g. open files)

b) One or more threads of execution

c) A region of memory holding its code and data. The memory belonging to a process is typically divided into four regions

(1) Code (called text in the Unix world)

(2) Fixed data (static variables).

(3) Heap: objects created dynamically as the program is running (e.g. by an operation such as new)

(4) Stack - contains a frame for each routine currently executing that holds parameters, local variables and return addresses.

PROJECT

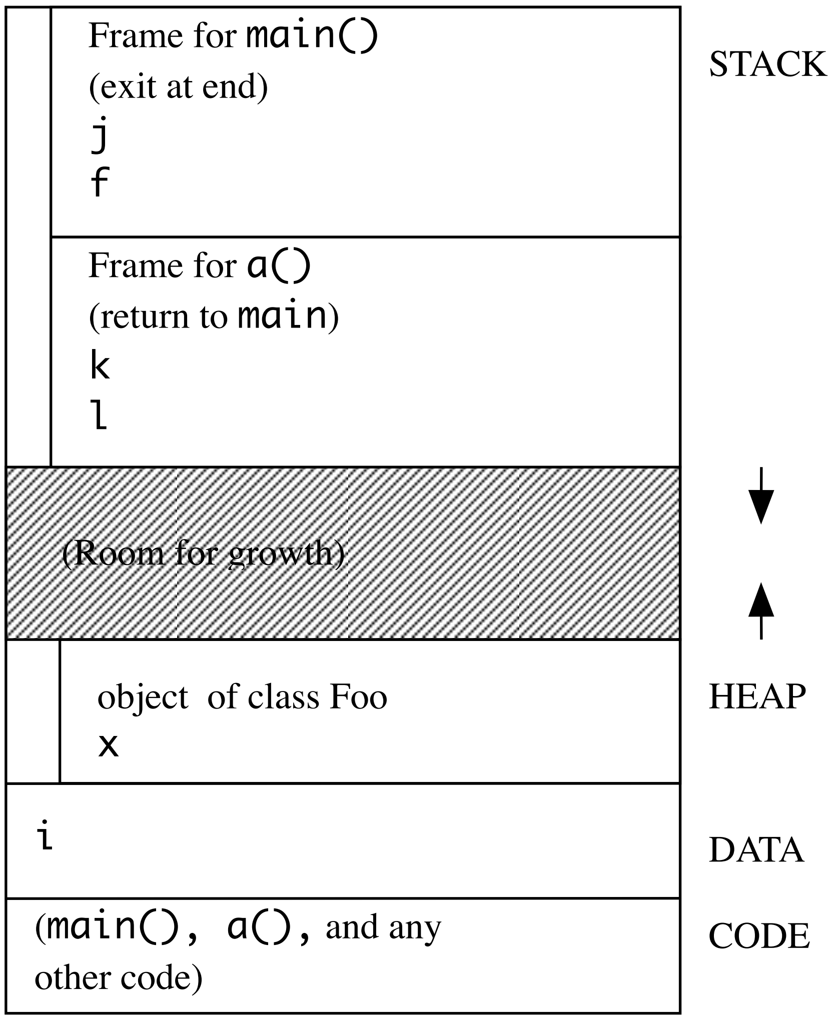
State of memory when executing the following code at point shown

```
int i = 3;

a() {
main() {
  int j = 7;
  Foo f = new Foo()
  f.a(3);
}



class Foo
{
  double x;
  void a(int k) {
    int l;
───────►  ...
  }
}
```



(5) The code and data regions are of fixed size, but the heap and stack can change size as the program runs.

(a) The heap grows whenever an object is created by an operation such as new. Depending on the programming language in use, the space allocated for the object can be freed up for use by another object by explicit deletion or by garbage collection. But since it is not possible to predict when this might occur (if it ever does), such objects are said to have indefinite lifetime.

(b) The stack grows when a routine is called, and shrinks when the routine returns. Routine call and return obeys a last-in-first-out order.

(c) To allow the heap and stack to grow and shrink independently, they are allocated as two separate regions, growing in opposite directions as shown by the arrows.

3. Frequently, a process has only a single thread of execution

a) Historically, this was always the case.

b) Today, for a variety of reasons, this is not necessarily the case. Modern operating systems incorporate support for multiple threads within a single process. For this reason, the book we are using distinguishes between threads of execution (discussed in chapter 2) and processes (discussed in chapter 7). We will largely follow that distinction.

B. More about Threads of Execution (commonly abbreviated to just Threads)

1. Until now, the programs you have written have typically had a single thread of execution. We will see shortly how to make use of a multiple threads within a program; but for now, we'll focus on an individual thread.

2. A thread is defined by three things:

   a) The address in memory of the instruction it is executing - typically contained in a special register in the CPU called the program counter (PC).

   b) The values that are present in other CPU registers.

   c) An execution history, representing the procedure calls that have brought it to its current point. This history is commonly called the thread's **stack**, and it contains one **stack frame** for each procedure that is still active, which holds its parameters, local variables, and other information:

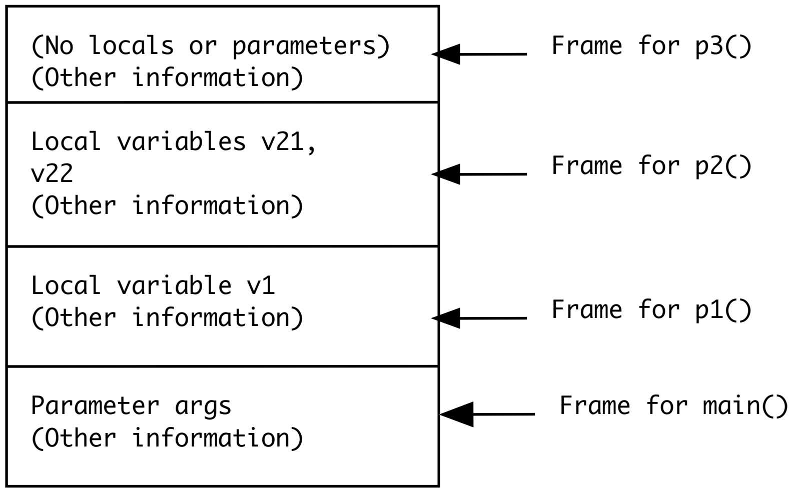
   Example: Suppose we have the following: PROJECT

```
class SomeClass {
     void p0() {
          ...
     }
     void p1() {
          int v1;
          ...
          p2();
          ...
     }
     void p2() {
          int v21, v22;
          ...
          p3();
          ...
     }
     void p3() {
          ...
           /* Here */
          ...
     }
```

```
        public static void main(String [] args) {
            p0();
            p1();
    }
```

When execution is at the point marked Here, the stack would look like this:

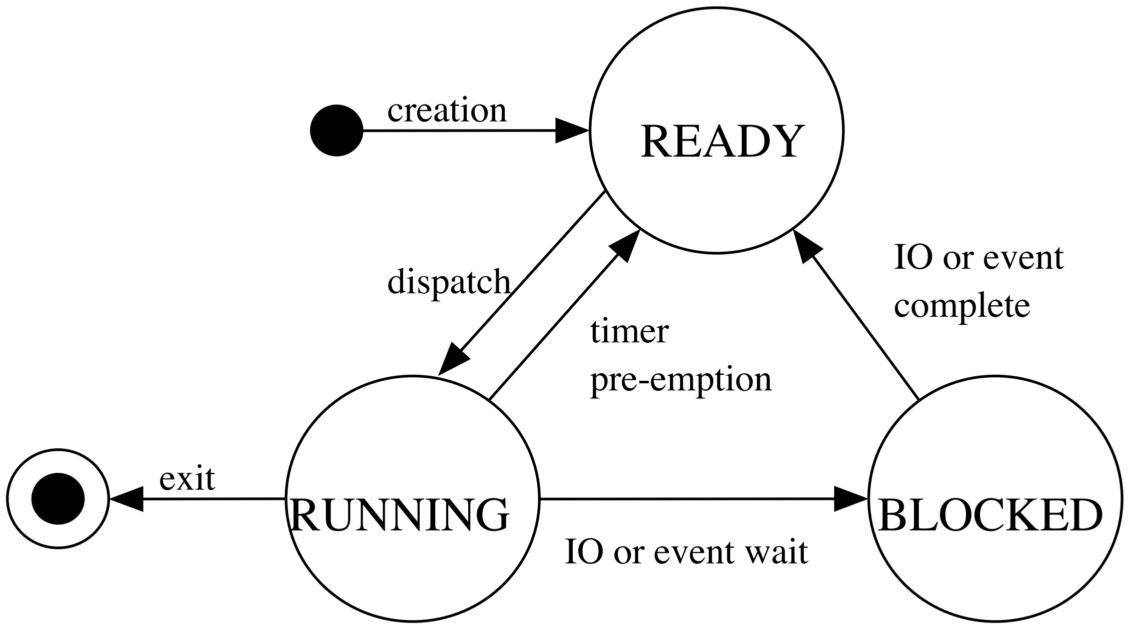
PROJECT



(Notice there is no frame for p0(), since it completes execution before p1() is called.)

3. One other important piece of information maintained for a thread is its state - is it currently able to continue execution, or must further execution wait until some event occurs (e.g. the completion of an IO operation it has requested) Over the course of its lifetime, a thread transitions between various states.

   PROJECT



a) On a one-CPU system with a single core, there is exactly one running thread at any given time. On a multi-core system (or one with multiple CPU's), there can be as many running threads as there are cores. (If there are not enough threads eligible to run, the operating system includes a "do-nothing" null thread that can be run until some other thread becomes ready.)

b) (Discuss other transitions)

c) The diagram shows that a thread terminates as a result of executing an exit() operation. Actually, many systems also allow a thread to be terminated from the other states by some other thread.

C. To manage the various threads, the operating system maintains a set of data structures called THREAD CONTROL BLOCKS (TCB's) - one per process. (On a system that only supports one thread per process, this may be combined with the information the system maintains about the process in the process control block, or PCB).

1. Each TCB contains information about its thread, including:

    a) Identification information (what process it is part of, and some sort of identification within its process)

    b) The state of the thread (running, ready, or blocked).

    c) The values of the CPU registers if the thread is not currently running. (When the thread is running, these values are actually in the hardware registers and change constantly)

2. One major use of the TCB's is in conjunction with a CONTEXT SWITCH operation.

    a) A context switch occurs whenever the a new thread is given use of the CPU in place of the one that is currently running. This can happen for one of two reasons:

        (1) The running thread has performed some operation - e.g. issued a request for IO - that requires it to wait until the operation is completed.

        (2) The running thread is pre-empted by some some other thread, either because of priority or because of timer preemption.

    b) During a context switch, the first thing that must happen is that the register values associated with the currently running thread must be stored into its TCB, to be saved until the thread next gets a chance to run.

    c) Then, the stored register values found in the TCB of the thread about to run are copied into the CPU registers.

d) A context switch can be quite computationally expensive - particularly if the new thread is part of a different process than the running thread.

3. At any given time, each TCB is typically in one of several operating system queues.

a) There is a ready queue (ready list) which holds the TCB's of all currently ready threads.

b) There is a device queue associated with each shared device (such as a disk). It is not uncommon for a thread to request an operation on such a device while it is busy servicing a request from some other thread. In this case, the PCB for the requesting thread is placed in a queue, waiting its turn to use the device.

c) A component of the operating system - called the scheduler - is responsible for managing these queues in order to make the most effective possible use of the resources. (We will discuss CPU scheduling later).

II. Multiple Threads in a Single Process

A. As we noted earlier, historically there was a 1:1 correspondence between processes (programs in execution) and threads of execution. Modern operating systems provide support for having multiple threads in a single process.

1. Some contexts speak of two kinds of processes - traditional, or "heavy weight" processes, and "light weight" processes - also known as threads. However, for clarity we will stick with using "process" for traditional "heavy weight" processes, and will call "light weight" processes threads.

2. If a single process contains several threads, then all the threads share the same memory allocation.

   a) Thus, code, global data, and heap memory is common to all threads.

   b) However, within the stack region each thread has its own stack.

      PROJECT Diagram of single-threaded vs multithreaded process

3. The use of multiple threads is driven by considerations of modularity

   a) It is often possible to express the solution to a problem in terms of a set of fairly simple interacting threads, rather than in terms of a much more complex single thread. This makes it easier to produce correct software.

      Example: Demo UnthreadedRacers

      PROJECT Code

      PROJECT Code for ThreadedRacers (program does exactly the same thing, but is much simpler)

   b) In the case of servers serving multiple clients (e.g. a web server), much simpler software often results from using a separate thread for each client, rather than having to keep track of which client is being served in a single thread.

4. Using multiple threads within a single process - as opposed to using multiple processes with one thread each - has two major advantages:

a) A context switch between threads it the same process tends to be faster than a context switch between processes, since only register context needs to be saved and restored - not memory management context. Thus one can have the modularity advantages of separate threads with less overhead.

b) Since all the threads share the same global data and heap, information sharing between the threads is simpler.

B. Multiple threads within a single process can be implemented in two different ways:

1. By the use of library routines that run in user mode, independent of the operating system. (I.e. the operating system just manages a single thread for the process.) Another name for this is the many to one model - many threads in a single process map to a single thread as far as the kernel is concerned.

2. By incorporating support for multiple threads within a single process as part of the operating system. We now have distinct system services for:

a) Creating a new process (initially consisting of a single thread).

b) Creating a new thread within an existing process.

Another name for this is the one to one threading model - each thread in the process is know to the OS kernel.

3. The former approach is simpler, and minimizes the overhead involved in a context switch between threads (since no system call is involved). The latter, however, allows one thread in a process to block itself on an IO operation while allowing the other threads to continue executing.

4. Moreover, on a multicore system that supports threads in the operating system, it is typically possible for two or more threads in a single process to be running concurrently on separate cores.

III. The Threading Facilities of Java.

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 by using a threading library such as the pthreads facility in Unix-like systems or the Windows threading facilities. We will discuss 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: `AWTThreadDemo.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.

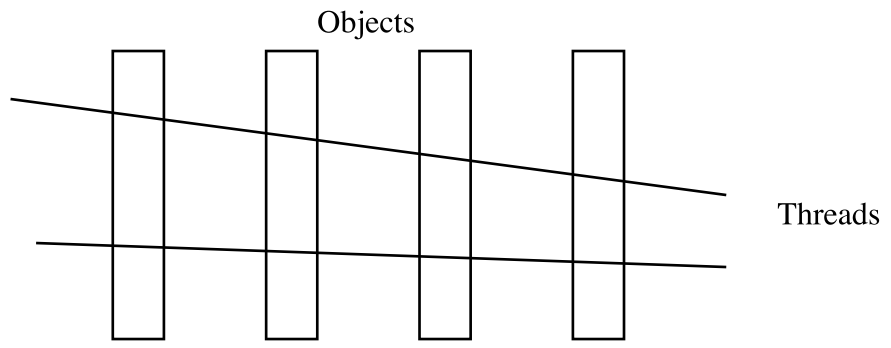
(3) How does execution switch from counting to modifying the increment when a button is clicked? These two methods are actually executed by two different threads: `count()` is executed by the main thread, and `actionPerformed()` by the awt event thread. The awt event thread is a part of the Java implementation 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 `AWTThreadDemo` 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 `AWTThreadDemo` 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: The Racer program we looked at earlier

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*:* AWTBlockDemo.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()`

D. 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.

   We will illustrate this by walking through the rest of the code from the threaded version of the Racer example

   HANDOUT Code for ThreadedRacer

   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: Note in inner class Racer

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

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

         (3) Each racer object is added to the GUI when it is created.

(4) 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. (See page 1 again)

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

(6) 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).

Note how each object need only keep track of the position of one racer

(7) While the racers are running, there are therefore actually <u>five</u> threads in operation - the four racer threads, plus the main thread that created them.

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. in this case `JPanel`) (However, we will use this simpler approach in the lab you will do on threads.)

4. Java also provides an operation known as join() which allows one thread to wait for another to complete. When this happens, the first thread continues. (In effect, the two threads are joined into one).

   a) This is used in the Racer program. 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.

   b) 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"`.

5. 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!

IV. **UML Activity Diagrams**

Recall that UML provides a nice mechanism for depicting multithreaded programs: the Activity Diagram.

A. *EXAMPLE: HANDOUT*: Activity Diagram for Racer Problem

B. Rounded rectangles represent activities.

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

D. 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.)

E. The diagram uses “swim lanes” to show the various parts of the system that are performing a task concurrently.