Table of contents

1 Introduction

We (1) introduce you to the concept of "embedded systems", which can consist of microprocessors that are embedded in larger systems in order to control or sense physical devices; (2) introduce you to "distributed programming", where several different computers are executing simultaneously, with some form of communication and synchronization between them; and (3) tell you about *execution threads* in Java, which allow a Java program to consist of several program segments executing in parallel.

The program that we demo can be obtained from the course web page (look under handouts).

2 Microprocessors

Microprocessors are small computers, usually embedded in other systems. They are used to monitor and regulate the behavior of many common commercial products. E.g. dedicated microprocessor systems in cars regulate the fuel-air mixture for combustion and prevent skidding (using anti-lock brake systems). In the home, they appear in appliances such as cellular phones, microwave ovens and thermostats for heating and cooling systems. They are used in modems, laser printers, graphics boards, and disk drives. More exotic applications include medical instrumentation, guidance of "smart bombs", detection of collision and decay events in high energy particle accelerators.

Microprocessors appear in settings where *real-time computing* is necessary: responses to external signals or changes —which can come at unpredicatable times— must take place within a specific amount of time. Some microprocessors must also be able to generate electrical signals that can be used to control other devices.

The anti-lock brake system on a car is a good example of the challenging nature of real-time computing applications. A microprocessor must detect changes in the rotational rates of the wheels characteristic of a skid. If the angular velocity of the wheels increases too fast, the tire is slipping on the pavement and the car may be skidding; controlling the situation may be difficult for the driver. Detecting and correcting this situation involves monitoring the rate of change of the angular velocity (i.e. the angular acceleration), not just the angular velocity itself. Upon detecting this situation, the microprocessor must generate a sequence of output pulses to control the hydraulic brake system, allowing the wheels to intermittently rotate to prevent a skid, thus making the car easier to control. Clearly, quick response[1] to inputs from external transducers is essential in this application. Not only must the response be quick, it must be appropriate for all imaginable driving conditions. For example, it would unacceptable for the microprocessor to interfere with normal braking operations at slow speeds.

Most real-time programming is done in assembly language or a programming language like C (or C++) that allows the programmer to explicitly manipulate the contents of specific memory locations. This is because communication with a microprocessor is often done through fixed memory locations. The real-time microprocessor itself is often programmed in its native assembly language. The machine instructions are placed in the microprocessors memory, which is physically distinct from the memory system of the host computer, so that the instructions can be executed autonomously by the real-time processor system.

3 Real-time simulation

A real-time application may consist of a host computer that communicates with several microprocessors, each of which senses some physical property (like the temperature in a room, or the speed at which

[1] On the order of tens of milliseconds.

a wheel is spinning) or controls some device. The host and an individual microprocessor communicate through a "register": a location in the microprocessor's memory. This location is "mapped" onto a fixed location in the host processor's memory, so that when the host references that memory location, it is really referencing the microprocessor's register. This is why many real-time applications need to read and write specific memory locations.

The microprocessor usually has a "program status word", or *psw*, whose bits are used to indicate the status of the microprocessor. For example, one of the bits, say `C`, might be used as follows: When the host computer is ready to read a value from the microprocessor's register, it (0) sets bit `C` to 1, (1) waits until `C` is set to 0 (by the microprocessor, to indicate that a new value is in its register), and then reads the register. This is how the host computer and the microprocessor communicate and synchronize.

Because of its built-in security measures, Java lacks the ability to manipulate the contents of specific memory addresses explicitly, so it is not well suited to real-time programming. However, Java does provide the tools needed to simulate a real-time computing environment. Our real-time simulation of a home heating system is an example. In it, we simulate this aspect of communication using a method `readValue()`.

4 Simulating a heating system

Besides a system output window, our heating-system simulation has five small windows, each of which represents one component of the home heating system:

1. **A clock**. You can see the clock "ticking" away. The period 5000 means that the clock ticks every 5,000 milliseconds, or every 5 seconds. You can change this to a smaller or larger number of milliseconds by clicking in that text field, typing a new number, and pressing button `Read period`. (Minimum period is 100 milliseconds.)
2. **The outside temperature** is initially 32 degrees, since this is Ithaca. Typically, a microprocessor would be attached to a sensor that would detect the outside temperature. In our simulation, the user changes the outside temperature by typing a new integer and pressing button `Read temperature`.
3. **The furnace** is switched on or off by the program as needed. This window just displays its status. There is a button for turning output on or off. Press it; thereafter, at each clock tick, a single line of output is printed in the system output window, telling you the status of the furnace, the change in the inside temperature due to the furnace being on, the change due to the difference in the temperature inside and outside, the total change for this tick, and finally the new inside temperature.
4. **The desired temperature** is set initially at 68, rather than 72, to save energy. Change the desired temperature, as you change the outside temperature. Go ahead; raise it if you're cold.
5. **The inside temperature** is initially 60 because the thermostat was at 55 for the night and we just woke up and changed it to 68. Don't worry, it will warm up soon. The inside temperature is simulated by the program; how much it changes at each clock tick depends on whether the furnace is on and the difference between the inside and outside temperatures.

Experimenting with this home-heating simulation. Become familiar with the five components on the screen.

5 Threads in Java

Typically, several programs may be executing on a computer at one time. Suppose your laptop is connected to a printer and a modem. One program on your computer could be printing something, another one could be faxing a file, a third could be computing something in the background, and a fourth could be some game that you are playing. Thus, (at least) four programs are executing at the same time on your laptop. Each is called an *execution thread*.

There is only one CPU (central processing unit) on your computer, so the four programs can't execute simultaneously. Instead, your computer allocates a

bit of execution time to each thread. Your operating system uses a priority scheme to decide which thread should execute next and how much time it should get. This switching is so fast and frequent that it gives the illusion of simultaneous execution.

These threads (of execution) can be independent (e.g. the printer program and your game don't interact), in which case allocation is fairly easy.

Sometimes, threads have to communicate with each other and synchronize in some fashion. For example, one thread may maintain windows on the screen, so when your Java program —which has at least one thread of execution— draws something in a window, it communicates with the thread that maintains the window.

In Java, synchronization is provided using three features. Suppose c is a class instance that is a thread. Then, execution of `c.wait()` in another thread b (say) tells the system to stop executing thread b until thread c executes a statement `notifyAll`. In other words, execution of `c.notifyAll` tells all threads that are waiting on c that they can now continue.

There is also a "synchronize" property, which can be attached to methods or to individual statements; it says that all other processes must be deterred from executing code in this class until this method or statement has finished –other processes are locked out.

A thread is an instance of class `Thread`. Here are some of the methods of class Thread that you can use:

- public void run(). When a thread is created by implementing interface `Runnable`, which we do, this method is called to start the thread executing.
- `public static Thread currentThread()`. This returns the thread that is currently executing.
- `public void start()`. Call this to start executing the thread –this method, which should not be overridden, calls method run.
- `public static void sleep(long millis)`. Make this thread sleep for millis milliseconds.
- `public void interrupt()`. Interrupt this thread, so that another thread can execute.
- `public final boolean isAlive()`. = "this thread is alive –has not died".

There are two ways to create a `Thread`. The first, which we don't use, is to use a constructor of class `Thread`. The other way is to implement interface `Runnable`, which requires the class to have a method `run()`, which is called by the system to execute the thread whenever your program calls method `start()`. This is the method we use in the home-heating simulation.

Here's a comment from interface `runnable`: In addition, `Runnable` provides the means for a class to be active while not subclassing `Thread`. A class that implements `Runnable` can run without subclassing `Thread` by instantiating a `Thread` instance and passing itself in as the target. In most cases, the `Runnable` interface should be used if you are only planning to override method `run()` and no other `Thread` methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

In our Java simulation, we have six programs —or six threads of execution— executing simultaneously.

6 The SystemClock

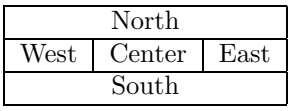
File SystemClock.java simulates a clock. At the end of the line `public class SystemClock...` appears the phrase `implements Runnable`. Interface Runnable defines one method, `run()`, which is called to execute a class instance as a thread. We'll look at `run` later.

So, each instance of this class is a thread of execution.

The class has two variables that deal with the clock itself: `counter` contains the number of clock ticks that have happened thus far; `period` is the number of milliseconds to wait between clock ticks.

Four fields are components that go in the Frame, which you see on the screen: two Labels, a Textfield, and a Button.

Look at the constructors for `SystemClock` (including method `initialize`). We use the default layout manager for Frames, `BorderLayout`. This layout manager allows us to place components in five places: North, West, Center, East, and South:



In method `initialize`, the four calls on method `add` place the four components in the Frame.

Two methods in `SystemClock` are fairly obvious: `reset` sets the clock back to zero, and `ticks` is used by other processors to get the clock's value.

Method `run` is of special significance. The main program (method `main` of `Thermostat`), calls `cThread.start()`, where `cThread` is the variable that contains the instance of `SystemClock`. In turn, method `start` will call `run` to start execution of this thread.

Method `run` is basically a loop in which each iteration does the following:

- Make the thread sleep for `period` milliseconds. Then wake up.
- Notify all the other threads that may be waiting on this one (for example, to read the tick) that the thread is now awake.
- Increase the time counter (and display it properly in the Frame).

The code to notify the other processes should be performed only when no other process is referencing variables associated with this thread. This "lock out" is indicated by the phrase **synchronize**. We don't go into any more detail here on this notion of synchronization.

Finally, take a look at method `action`, which is called when a button is pressed. There is only one button in the Frame, so there is no need for code to determine what the action was. The pressing of the button is processed by (0) reading `TextField periodField`, (1) trimming whitespace from both ends of it, (2) converting this `String` value to an integer, and (3) finally storing the integer in `period` (if the value is at least 100).

7 InsideTemperature

We discuss just two other classes in this project, because the others are similar.

First, an instance of class `ClockedFrame` is a thread of execution that synchronizes with the clock. Thus, within this class, you see a method `run`, which is a loop that at each iteration synchronizes with the clock. That is all.

Also, this class is a Frame on the monitor screen. This class is a superclass of `InsideTemperature` as well as of several other classes, so an instance of `InsideTemperature` is an executable thread.

Now turn to subclass `InsideTemperature`. Besides some components for the Frame (two labels), there is only one variable, `temperature`, which contains the current inside temperature in the simulation. The constructor just constructs the Frame, shows it, and returns; there is nothing special about it.

Methods `getValue` and `setValue` can be called to reference and set the inside temperature.