Assignment 1

UBC | EECE 494 | Spring 2010

Released on January 11
Due on January 18
This assignment is for individual work.

1. **Preliminary tasks**

   (a) **Extracting the skeleton code**

   To help you with this assignment, a lot of support code has been made available on the course website in the *Assignments* section. Copy the tarball to the directory `eece494` in your home directory (preferably on `ssh-linux.ece.ubc.ca`. After extracting the tarball, you will see that a directory called `hw1` has been created. This will contain the code that you can use for this assignment.

   (b) **Verifying `POSIX` support**

   `POSIX` defines a set of standards for operating systems. Operating system vendors are not obligated to include the entire `POSIX` standard, and many do not. The version running on `ssh-linux.ece.ubc.ca` does support most `POSIX` features but it never hurts to check it anyway. If you are using your home version of Linux, you certainly want to check it. If you have access to a Unix machine (such as `ug1` to `ug10`) you can log in and check those too (they support a slightly different subset of `POSIX`, but probably still enough for our purposes).

   To check the `POSIX` components supported on your version of Linux/Unix, compile and run the program in `hw1/POSIX`. When you run the program, the results will scroll off the screen; you can redirect the results to a file using the "redirection" operator:

   ```
   a.out > outputfile
   ```

   and then use any text editor (or the more command) to view outputfile.

   Look for the line `POSIX.1c pthreads are supported`. You might also want to check for `POSIX.4 PRIORITY SCHEDULING Supported`. Some of the other status messages should also make sense.

   You will probably want to use this program later, if and when you need to use more advanced `POSIX` features. You may also need to use the `-lrt` option.

   (c) **Study the examples**

   Many examples – including code for handing multiple threads – are available in the directory `hw1/examples`. It will be well worth your time to understand these examples. You need to use the appropriate compiler flag to include the `pthreads` library. Also note that you may need to compile some programs with the `-lrt` compiler option to include the real-time support library.

2. **The UBC Rover [80 points]**

   Several years ago, at a cost of $820,000,000, the UBC IEEE Student Branch launched a spacecraft to Mars from the attic of the Cheese Factory. The spacecraft contained a Rover (Figure 1) that was intended to scour the surface of Mars and search for signs of life. Unfortunately, when the Rover descended to the surface of Mars, things went wrong. We are not actually sure what happened, but

we have determined that we have only limited control over the Rover, and only a few sensors are working.

There are four sensors working. One sensor detects life-forms (this sensor is not shown on the diagram). If a Martian is sensed in the immediate area, the sensor returns `true`, otherwise it returns `false`. Two other sensors are used for navigation; one at the front of the Rover and one at the back. Consider the sensor at the front of the rover. This is used to examine the ground ahead of the rover. As long as the ground is relatively flat, the sensor returns a `false`. If, however, a cliff is seen immediately in front of the rover, the sensor returns `true`. Judicious use of this sensor's results are needed to prevent the Rover from tumbling over a cliff. There is also a sensor on the back of the Rover which performs exactly the same function; this is required if the Rover is driving backwards.

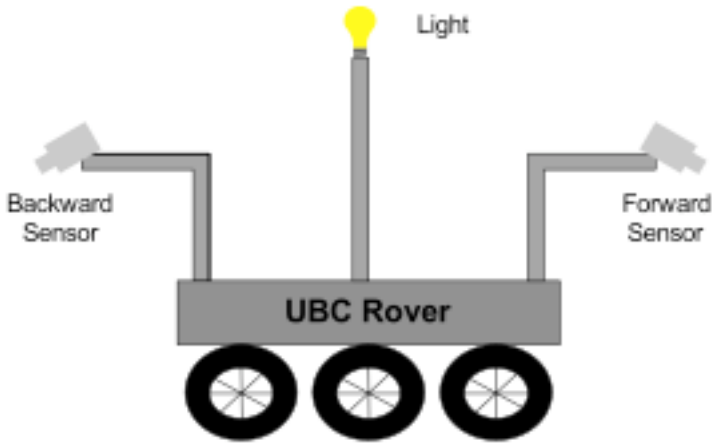

Figure 1: The UBC Rover

There is a fourth sensor working, which simply indicates whether the Rover is still alive. If this sensor ever returns a `false` (which might happen if the Rover drives off the edge of a cliff, for example), the Rover is dead, and the IEEE Student Branch is out $820,000,000.

Although the Rover was designed to be very maneuverable, many of the actuators appear to have been damaged during the entry to the Martian atmosphere. In particular, we have been able to detect that the Rover is moving, but we can not stop it. The only thing we can do is control the direction (forward or backward). It also appears that our digital camera is working; there is an actuator, that when triggered, takes a digital picture.

Unfortunately, we do not have the resources to go and rescue the Rover. We have decided that the best course of action is to flash a `HELP` pattern using the light, hoping that some intergalactic transport ship will pick the Rover up. At the same time, we have to make sure that the Rover does not go over a cliff.

Your program must:

(a) Keep the Rover from driving off the edge of a cliff. When moving forwards, you should monitor the Forward sensor, and when it returns `true`, cause the Rover to start moving backwards. When moving backwards, you should monitor the Backward sensor, and when it returns `true`, cause the rover to start moving forwards. You must respond to a change in direction fast enough that the Rover does not drive over the cliff. Note: to avoid unneeded stress on the Rover's mechanical systems, you should not change directions more often than you have to. In other words, you are not allowed to switch between backwards and forwards every clock cycle. You can only change directions when information from the sensors indicates that it is time to do so.

Table 1: Sensors

Forward sensor	`true` ⇒ obstruction or a cliff immediately ahead of the rover `false` ⇒ ground ahead is relatively flat
Backward sensor	`true` ⇒ obstruction or a cliff immediately behind of the rover `false` ⇒ ground behind is relatively flat
Lifesign	`true` ⇒ Martian nearby `false` ⇒ no Martian nearby
Alive	`true` ⇒ the Rover is alive `false` ⇒ the Rover is dead

Table 2: Actuators

Direction	`true` ⇒ move forward `false` ⇒ move backwards
Camera	when triggered, the camera takes a snapshot
Light	`true` ⇒ the light is `on` `false` ⇒ the light is `off`

(b) Flash `HELP` in Morse Code using the light. Morse Code for `HELP` is: dot dot dot dot break dot break dot dash dot dot break dot dash dash dot break (and repeat). A dot will be indicated by turning the light on for one second. A dash will be indicated by turning the light on for two seconds. Between each dot or dash, you turn off the light for one second. Between letters (denoted "break" above), you turn off the light for one additional second (so that is two seconds, including the second after the final dot or dash in the letter).

(c) If a Martian is detected, the Rover should take a picture of it as soon as possible (before it disappears). This will be useful if the Rover is rescued, and we need to prove that life does exist on Mars.

Use three threads (and the *main* thread of execution). One thread should control the light (turning it on and off), the second thread should use the directional sensors and control the movement of the rover, and the third thread should monitor the Martian surface for signs of life and take a picture if a Martian is detected. The main function will terminate if the rover dies.

You cannot use *busy waiting*. You are only allowed to check each sensor or modify each actuator once per second. Any busy waiting in your code will cost you marks.

To help you write your code, stubs are available, as well as a Rover simulator and a graphics subsystem. the code for the UBC Rover will be in the directory `hw1/rover`.

To use the graphics subsystem, you will need X-Windows. Although you will be running your program on one of the department Linux servers, you will not actually be sitting in front of the Linux machine. Instead, you will access the Linux machine through one of the Windows computers in MCLD 309/358, or from home. If you are running this in MCLD 309/358 (recommended), do the following. On the local machine in MCLD 309/358, log in, and run two programs:

(a) First run X-Win32 (from the Start menu). This is an X-server program which will receive drawing commands from the remote server (the Linux machine) and use these commands to draw windows on your screen in MCLD 309/358.

(b) Run the SSH tool (also from the Start menu in the computers in MCLD 309/358). The remote machine you want to connect to is called `ssh-linux.ece.ubc.ca`, and you can use your ECE id and password. If all goes well, SSH will start a terminal window. Commands you type in this window will be executed on the Linux machine.

You then *might* have to set up your display variable, to tell the linux machine where to display your windows. The name of your local machine is on a yellow sticker on the computer itself. As an example, if your local machine is called `castlegar.ece.ubc.ca`, then enter the following command to set up your display variable:

```
setenv DISPLAY castlegar.ece.ubc.ca:0.0
```

If you are working from home, you will need X-Windows server software (such as MI-X) running. You probably already have such software, but if not, you can either get free or cheap versions on the web. *Note that the TAs will not be able to help you debug your home computer set-up; if you cannot get X-Windows working at home, you should use the machines within the department. Further, note that the IP address of a machine can be used in place of its machine name.*

There are four source files that you have copied (in addition to header files). The file `main.c` contains the main program loop. It sets up the graphics, initializes the rover, and then calls your control routine. The files in `rover.c` contain the Rover simulator. These routines read the actuator values, adjust the position of the Rover, and set the sensor values. The files in `graphics.c` contain the graphics subsystem. The file `control.c` is where you will write your controller. You do not need to look at any files other than `control.c` (but if you are curious, go ahead; they are not very complicated, and you will learn a lot). To compile the program, a `makefile` is provided; to compile the code, execute the command `make` and the executable `sim` will be created.

Within `control.c`, you will see where you should write your code. Routines to set the actuator values and read the sensor values are provided at the top of the file. Make sure you use either `pthread_detach` or `pthread_join` before the `main` function exits.

You will know your assignment works when the Rover can flash out `HELP` many times, moves back and forth without driving over a cliff, and takes pictures of Martians whenever they appear. Since intergalactic transport ships do not exist yet (a key flaw in our rescue plan) you will have to end the simulation with a `Ctrl-C`.

3. **Measuring execution times**
Determining the (worst-case) execution time of tasks is an important aspect in the design and development of a real-time system. Most schedulability analyses assumes we know the execution times of tasks rather accurately.

You need to examine some baseline code for a Discrete Cosine Transform (do not worry if you know nothing about this operation). DCTs are used in image processing and you need to examine the running time of the baseline code that has been made available to you (in the tarball for this assignment). This code (`task3/baseline.c`) finds the discrete cosine transform of a 8×8 pixel block. The original block is the `img` array (which is hardcoded into the program) and the output is the `f` array (which is then printed out to the screen). To make it easier to measure the time to perform the DCT, repeat the DCT 10,000 times (on the same input data) - the time to perform one DCT is the time to perform all 10,000 divided by 10,000.

Compile this code using gcc. To compile this, you have to link in the math library (otherwise the linker will complain that it can not find the `cos` routines):

```
gcc baseline.c -lm
```

Do not use any optimization flags (e.g., `-O2`). Measure the time to execute the resulting executable on `ssh-linux`. To do this, the simplest way is:

`time a.out` (or `time ./a.out` depending on your shell environment)

This will run the executable `a.out` and print the user and system time that it takes to run the executable. It is the user time (the first number) that you care about. It should be on the order of a few seconds.

4. **Reducing execution times [20 points]**
Sometimes it is necessary to rewrite modules of a real-time system to reduce the execution time and make the system schedulable. One of the slowest portions of the DCT algorithm is the cosine function. To reduce this inefficiency, replace the call to `cos( )` with a table lookup:

```
static const int COS_TABLE[8][8] = {
{ 32768, 32138, 30273, 27245, 23170, 18204, 12539, 6392 },
{ 32768, 27245, 12539, -6392, -23170, -32138, -30273, -18204 },
{ 32768, 18204, -12539, -32138, -23170, 6392, 30273, 27245 },
{ 32768, 6392, -30273, -18204, 23170, 27245, -12539, -32138 },
{ 32768, -6392, -30273, 18204, 23170, -27245, -12539, 32138 },
{ 32768, -18204, -12539, 32138, -23170, -6392, 30273, -27245 },
{ 32768, -27245, 12539, 6392, -23170, 32138, -30273, 18204 },
{ 32768, -32138, 30273, -27245, 23170, -18204, 12539, -6392 }
};
```

The value of $\cos(\pi(2a+1)b/16)$ can then be approximated by `COS_TABLE[a][b]/32768.0`.

Copy `task3/baseline.c` to `task4/task4.c`. Then, in `task4.c`, add the above table, and replace all the calls to `cos( )` with an access to the table as described above. Note: The above table is stored in `task4/table.txt` (to save you the effort of typing it in yourself).

You need to measure two quantities:

(a) The impact on run-time. Hopefully, the modification described above will reduce the runtime of the algorithm. Measure the runtime (to perform 10,000 DCTs) and compare it to the number you obtained in Task 1. I would expect that most people would see a significant speed-up here. Record the speedup in the file `~/eece494/hw1/info.txt`.

(b) The impact on accuracy. This modification will also have an impact on accuracy, since fewer bits are used to represent the value of the cosine function. One way to quantify the accuracy is by using the following metric:

$$\frac{1}{N}\sum_{i=0}^{N-1}(x_i - y_i)^2,$$

where N refers to the number of samples the program calculates (64 in this case). The quantity x_i refers to the ith sample calculated by the original program, and the quantity y_i refers to the ith sample calculated by the program as modified in this task. Clearly, if there was no impact on accuracy, this quantity would be 0, however, I will expect that you will obtain a non-zero value. Record this value in the file `~/eece494/hw1/info.txt`.

5. **Submitting your assignment**

(a) **Make sure your work runs on** `ssh-linux.ece.ubc.ca`
We will be marking your assignment on the machine `ssh-linux`. If this is where you have been doing your development, then you do not need to do anything for this task. If you have been doing your development at home, with your own version of Linux, or have been doing your development on a Sun Unix station, you will need to make sure that your program works on `ssh-linux`. Since there are a lot of people in the class, it will be impossible for the TAs to attempt to run each assignment on multiple platforms.

(b) **Handin your assignment**

Login to `ssh.ece.ubc.ca` (not `ssh-linux`). You do not need to copy any of your files to this machine because your directory is maintained using `NFS` and will be the same on both systems. If you have created your files in the directory structure as described in each task, everything should be ready to go. The only thing left you need to do is to tell us who you are. To do this, please add your name and student number to the file `info.txt` in the `eece494/hw1` directory. This is very important. Although your user id is recorded when you submit your files, sometimes it is difficult to figure out your name from your user id.

To hand in your files: `handin eece494 hw1`

6. **Takeaways**

This assignment provides you an introduction to controlling a system using multiple threads. The three threads that you create for the UBC Rover are, in fact, periodic tasks. The speed of the rover will, in reality, dictate the frequency with which you need to sense movement and redirect the rover. Triggering the camera is an aperiodic task that needs a short response time. Unfortunately, we do not perform any explicit scheduling in this assignment. Scheduling requires the ability to change the priorities of threads and that would require administrative permissions on the ECE systems, which you do not have. (You can assign priorities to the user-level threads, but that is not enough; your process is still being scheduled by the operating system and you do not have control over that scheduler.) If you did have admin access, you would be able to pick the periods for the tasks and assign priorities appropriately, and even perform schedulability analysis to determine the best possible periods. You could then implement an aperiodic task server for the camera task and capture a digital image of a Martian as soon as possible.