CPSC 213: Assignment 7

Due: Monday, March 12, 2012 at 6pm.

Late assignments are accepted until Wednesday, March 14 at 6pm with a 25% penalty per day (or fraction of a day) past the due date. This rule is strictly applied and there are no exceptions.

Goal

In this assignment you will investigate interrupts and asynchronous programming by reading and modifying a program that uses signals to trigger asynchronous actions. This assignment is closely modeled on the asynchronous disk read example discussed in the lecture and gives you a chance to experience this sort of programming first hand, using a software environment that mimics hardware interrupts.

Interrupts, Signals and Asynchronous Programming

Included with this assignment is a small C program called `async.c`. This program uses a Unix OS feature called "signals" to mimic hardware interrupts. In the `boot` procedure, the program registers `interruptServiceRoutine` as a signal handler for the `SIGALRM` signal and then tells the OS to deliver this signal to the program once every second. The program includes a method called `doAsync` that schedules an asynchronous event, sort of like a disk-read request. These events complete in order, one at a time, each time the `SIGALRM` is delivered to the program. The `doAsync` procedure enqueues events on a circular completion queue and `interruptServiceRoutine` dequeues these completion events when `SIGALRM`s arrive and delivers the completion by calling the completion routine with two parameters, a pointer and an `int`, whose meaning is determined by the completion routine. You will note the use of the type `void*`. This type is called an "opaque pointer" and is used to store pointers of any type. The program includes a small example of the use of this system to asynchronously print three strings.

I'd like you to pretend that what is actually happening is that the OS is an I/O controller that is doing some real work for your program as a result of `doAsync` and that it uses `SIGALRM` to signal that this work has completed. The fact that the OS isn't really doing anything other than regularly delivering signals is necessary because we are emulating complex behaviour with a simple program.

You will note that this program ends with an infinite loop and so it will run forever unless you (or someone else) kill it. Be sure to kill it when it is done (e.g., by typing `Control-C`).

You have two tasks. First, read, compile and run the program to understand what it does. Insert detailed comments in the program to carefully explain all of the data structures and procedures. Do not add comments to individual lines of code, but ensure that your other comments are detailed enough to fully explain what this code does. Use the `man` command as necessary to get the documentation for functions such as `signal` and `ualarm`, etc.

Then, modify this program to use the `doAsync` procedure and this framework to implement a program with the following asynchronous operations (each implemented by a procedure that is never called directly but that is instead caused to run by `doAsync`). First, define a `struct Triple` with struct members `arg0`, `arg1`, `result`, and `complete`, that hold the arguments and result for a simple computation with `complete` indicating whether the computation has been completed.

1. `add (void* xp, int n)` that casts `xp` to a `struct Triple` pointer and computes `xp->result = xp->arg0 + xp->arg1`.
2. `sub (void* xp, int n)` that casts `xp` to a `struct Triple` pointer and computes `xp->result = xp->arg0 - xp->arg1`.

Write a program that uses only these two procedures and the `doAsync` procedure to compute the value of the expression: "`((1+2)-(3+4))+7`" and store the final result in a global variable. The program should terminate after printing the value of this global variable.

Note that a key challenge here is that some of operations use results from previous operations (e.g., you can't do the subtraction until the additions for `(1+2)` and `(3+4)` have completed). You will thus need to synchronize your program to some extent. Do not synchronize any more than necessary (e.g., `(1+2)` and `(3+4)` do not need to be synchronized with each other). Implement this synchronization using a shared variable whose value indicates whether the computation can continue and then "spin" on this variable until it has this value. For example, the following code waits until the variable `n` has the value `1`: `while (n != 1) {}`. This is an example of "polling" a variable for a value and this particular strategy has a special name: "a spin lock".

Requirements

Here are the requirements for this week's assignment.

1. Carefully comment the `async.c` program to explain every procedure and data structure in detail.
2. Compile `async.c` with the `-g` option and then type `gdb a.out` (or, replace `a.out` with the name you give to your program) to run the program in the debugger. Type `b printString` to set a breakpoint at the `printString` procedure and then type `run` to start the program. When the program stops at the breakpoint, type `backtrace` to have `gdb` display the current contents of the runtime stack. There is one line for every activation frame on the stack, with the current frame on top. Carefully explain what you see. Then modify `async.c` to call `printString` directly from the `main` function. Repeat this process and compare this stack trace to the original one. Carefully explain the difference between the two stack traces.
3. Modify `async.c` as specified above and test your program.

Material Provided

The code for `async.c` in included in the file `code.zip`.

What to Hand In

Use the **handin** program. The assignment directory is **a7**. Please hand in exactly the following files with the specified names. Do **NOT** hand in class files, or your entire Eclipse project, or a README in formats like `.doc` or `.rtf`.

1. `async.c` commented and modified to perform the specific task as specified above for Requirement 3.

2. `README.txt` that contains:

   - header with your name, student number, four-digit cs-department undergraduate id (e.g., the one that's something like a0b1)

   - statement that "I have read and complied with the collaboration policies" at http://www.ugrad.cs.ubc.ca/~cs213/winter11t2/policies.html

   - detailed description of the two stack traces and explanation for the difference, as specified in Requirement 2.

   - brief description of the testing of your modified `async.c` program, i.e. whether it produces the desired result.