CS 331 Lab #4
Process & Thread Communication

Spring 2003

Week #1: 3/19/2003 – 3/21/2003
Week #2: 4/2/2003 – 4/4/2003
Report due in lab: 4/9/2003 – 4/11/2003

1 Overview

It is bad practice to let a real-time loop directly read from or write to the screen since the read and write operations are time consuming. If read waits for the user's input, it can cause deadlines to be missed. Even using write by itself can lead to unpredictable waiting since screen writes are typically first-in-first-out. Your task could get blocked by other tasks' screen activities, including non-real-time I/O. So your task may still miss deadlines even though it does very little I/O. This precaution is especially important when a graphical user interface is used.

In this lab you will create two tasks. The I/O task will periodically read the position of the water seesaw and send the value to the display task. The display task will receive the position, perform some calculations on it, and print information to the screen.

In part 1 of the lab you will implement these tasks as two processes communicating with a message queue. In part 2 you will implement them as two threads within one process communicating with shared memory and a mutex.

2 Procedure

2.1 Process Communication with a Message Queue

1. Make a `lab4` directory to put your code for this lab in. Copy `~cs331/lab4/Makefile` to your new directory.
2. Turn on the op-amp box and execute `~cs331/lab4/display` and then `~cs331/lab4/io` in a different window to see an example of the programs you now need to write.
3. Write a C++ program called `io.cc` that reads the position of the water seesaw 10 times per second and sends the raw reading as two `unsigned char`s to another process using a message queue.
   - (a) The queue should hold a maximum of one second's worth of messages.
   - (b) The `mode` argument of `mq_open()` should be "0600" (*zero six zero zero*). The leading zero tells the compiler that an octal value follows, and 600 makes the queue private.
   - (c) Check `mq_send()`'s return value for errors and delete the queue with `mq_unlink()` if something is wrong.
   - (d) When Ctrl+C is pressed, a SIGINT signal will automatically be sent to the process. Use `sigaction()` to setup an asynchronous handler which deletes the queue with `mq_unlink()` when Ctrl+C is pressed.
4. Write a C++ program called `display.cc` that receives the messages sent by `io.cc`. The data should then be printed to the screen in three ways: raw 12-bit value, voltage, and seesaw angle in degrees.
   - (a) To get reference points for the angle calculation, you may perform several A/D conversions directly from `display.cc` before opening the message queue.

(b) The seesaw's range of movement is $-15°$ to $+15°$.

(c) As in `io.cc`, the queue should hold a maximum of one second's worth of messages, and the `mode` should be "0600" (*zero six zero zero*).

(d) If `mq_receive()` blocks when the queue is empty, the maximum rate of consumption will automatically be governed by the rate of the producer (`io.cc`).

(e) As in `io.cc`, you should delete the message queue when Ctrl+C is pressed.

5. When finished writing `io.cc` and `display.cc`, compile them by running "`make part1`". Then execute the programs in different windows with "`./display`" and "`./io`".

2.2 Thread Communication with Shared Memory

1. Make sure that op-amp box is on and execute `~cs331/lab4/threads` to see an example of the program you now need to write.

2. Write a C++ program called `threads.cc` that contains two threads. The I/O thread should act like `io.cc` and the display thread should act like `display.cc`. Use shared memory (global variables) and a mutex to pass the raw reading as two `unsigned char`s 10 times per second.

   (a) For this lab you can have both threads run at the same default priority by passing `NULL` as the second argument to `pthread_create()`.

   (b) Each thread requires its own timer. These timers must generate different signals so that the program can distinguish between them. The following code shows how to change the signal that a timer generates.

```
/*  Create a timer that sends SIGUSR1 instead of the default SIGALRM  */
timer_t timer;
struct sigevent timer_evp;
memset(&timer_evp, 0, sizeof(timer_evp));
timer_evp.sigev_signo = SIGUSR1;
timer_evp.sigev_notify = SIGEV_SIGNAL;
timer_create(CLOCK_REALTIME, &timer_evp, &timer);
```

   (c) The `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions are not asynchronous-signal safe. This means that calling those functions from a `sigaction()` signal handler may cause deadlock. To avoid this we will use `sigwait()` instead of `sigaction()`. `sigwait()` takes a set of signals, and blocks until one of the signals is received. A signal handler can therefore be simulated as in this example.

```
/*  This thread will execute the handler_function() function  */
/*  every time a SIGALRM signal is received.                  */
sigset_t signals_to_wait_on;
sigemptyset(&signals_to_wait_on);
sigaddset(&signals_to_wait_on, SIGALRM);
int sigval;
while (1)
{                                            /* Note that even though we     */
    sigwait(&signals_to_wait_on, &sigval);   /* don't use it, passing the    */
    handler_function();                      /* sigval pointer is required   */
}                                            /* for sigwait() to work right. */
```

3. When finished writing `threads.cc`, compile it by running "`make`". Then execute the program with "`./threads`".

3 Demonstration

Notify the TA when you are done with the assignment and ready to demonstrate your code. The TA may randomly pick one of you to do the demonstration and ask another to explain how it was done. *Make sure that everyone in your group understands how all of the code works before your demonstration.* Some lab topics will appear on the final exam.

4 Lab Report

Your group should turn in one typed lab report when Lab 5 starts (4/9/2003 – 4/11/2003). The report should include the following items:

1. An organized header that, at the minimum, contains your names and NetIDs, as well as your lab section (day and time), workstation, and username. For example:

   CS 331 Lab Report #4

   Tim Eriksson (eriksson@uiuc.edu)
   Xiaolei Li (xli10@uiuc.edu)

   Section: Friday at 10:00am
   Workstation: emb8
   Username: group38

2. An overview of how you implemented both parts 1 and 2.
3. A discussion of any problems you encountered while working on this assignment, and how you overcame them.
4. Responses to the following questions.
   (a) In `io.cc`, did you open the message queue as blocking or non-blocking? Why?
   (b) In `display.cc`, did you open the message queue as blocking or non-blocking? Why?
   (c) Why was a special handler for Ctrl+C used in `io.cc` and `display.cc`? Explain.
   (d) Why is a mutex required in `threads.cc`?
5. A printout of your commented code (`io.cc`, `display.cc`, and `threads.cc`).