Assignment 2

Due: 8:30 A.M., Wednesday, June 23, 1999.
Weight: 8% of your course grade.

Preliminaries

As with all assignments in this course, a small part of the assessment for this assignment reflects how your work is presented. Your work must be easy to read. This means:

- Your work must be clear, succinct, grammatically correct, and correctly spelled.
- Your work must be laid out sensibly. In text, lines should be short enough to be readable (i.e., don't fill the entire width of the page with text—appropriate use of margins and other blank space on a page can dramatically improve the readability of your work).
- Your program code should be indented sensibly, and be read easily by anyone familiar with UNIX and C/C++. Well-written code that implements straightforward functionality may not need any additional comments.
- Unless there is a compelling reason to do otherwise, all pages should be on letter-size paper, with content in the portrait orientation. Pages should be held together with a single staple in the top lefthand corner.
- Avoid unnecessary embellishment—you should not use presentation folders.

Most students find it easier to create their answers to assignments using a word processor or equivalent program. Although you may handwrite your answers, it is much easier to check and correct grammar and spelling on a computer screen than on a handwritten page. If you choose to handwrite your answers, they must be neat, tidy and legible.

The late policy for all assignments is as follows: 10% penalty for being one day late, 20% penalty for being two days late. Assignments may be handed in early. No assignments will be accepted more than two days late. The late period begins immediately after the beginning of class on the due date (i.e., handing in your work at 9 A.M. on the Wednesday it is due counts as one day late).

This assignment may be refined or clarified in class or e-mail.

Programming Exercises

For this programming assignment, you will be modifying C++ source code. This source code is available on the Web at http://www.cs.sfu.ca/CC/300/oneill/Homework/assign2.tar.gz and on CSIL in /gfs1/CMPT/300/src/assign2.tar.gz . You can expand this archive file using the command gzcat assign2.tar.gz | tar xvf – on any UNIX system. You can also find the contents of this archive file in the directory /gfs1/CMPT/300/src/assign2/.

For each of the programming exercises below, develop the program as indicated in the question, and then include the source listing for the program in your report along with any commentary you feel is necessary to explain it. Your programs must be *short* (your changes should easily fit on a single page), but *clear* and *robust*.

You may program in either C or C++. The source code provided is in C++, but does not use advanced C++ features—it should be straightforward to port the code to plain C.

P1. This question concerns the files hitme.cc and hit.cc, provided with this assignment.

The program hitme waits for a simple message to be sent by the program hit. When it receives the message, hitme writes "Ouch!" to standard output.

Currently, the hitme program receives its messages synchronously (i.e., it does nothing while it is waiting for messages to arrive). Your task is to modify the program so it receives these messages asynchronously. In your revised version, the program will run the juggle function while messages are not being received. You *may not* modify the juggle function.

Modify hitme.cc so that it uses pthread_create to create a thread that will listen for the messages.

For bonus credit, create an alternate version that uses the Unix SIGIO or SIGPOLL facilities rather than threads (on systems supporting the BSD programming interfaces or the latest POSIX interfaces (NEXTSTEP and Linux), see the description of the FASYNC (or O_ASYNC) in your reference for the fcntl system call; on System V systems (Solaris), see the I_SETSIG ioctl in the streamio documentation). Because the LocalPort class hides implementation details, including the file descriptor it uses to receive messages, you will need to extend it. Compare the thread-based and SIGIO-based approaches and contrast their strengths and weaknesses.

Written Exercises

W1. What are *race conditions?* Explain, using the message-server and message-test programs from the project as examples.

W2. Explain why race conditions can still occur even if every data item is protected by a mutual-exclusion lock and the data item is always locked when it is read or written.

W3. Suppose we have a counter which we must increment atomically. Two solutions are proposed:

(a) Protect the variable using a simple spin lock. We assume an atomic test_and_set operation, where test_and_set(&*flag*) sets *flag* to be true and returns the value *flag* held just before it was set to true.

```
struct Counter {
        bool  mutex;
        int   value;

        void inc() {
                while (test_and_set(&mutex)) {
                        /* do nothing */
                }
                value = value + 1;
                mutex = false;
        }
}
```

(b) Use compare_and_swap, where compare_and_swap(&*var*, *oldval*, *newval*) atomically compares the contents of *var* with *oldval*, and if they are the same, updates *var* to hold *newval*. compare_and_swap returns the original value it found in *var*. (In other

words, *oldval* is our "guess" for the value of the variable *var* (it is only a guess because some other process might modify *var* "behind our backs"). If our guess is right, compare_and_swap will modify *var* to hold *newval* and return *oldval*. If our guess is wrong, it will return the value that is really in *var*, which we can then use as a guess if we want to try again.)

```
struct Counter {
    int    value;

    void inc() {
        int old_value, swap_value, new_value;

        swap_value = value;
        do {
            old_value = swap_value;
            new_value = old_value + 1;
            swap_value = compare_and_swap(&value, old_value, new_value);
        } while (swap_value != old_value);
    }
}
```

The second method saves the storage space of a boolean variable, since the class does not need a mutex field, but there is another reason why the second method is better.[1] Determine and explain the problem that exists in (a) that does not exist in (b), and discuss whether replacing the spin lock with semaphores would address the problem.

w4. Consider the following set of running processes, where the arrival time and the length of the next processing burst are given in milliseconds:

Process	Arrival Time	Burst Time	Priority (if applicable)
A	0	10	3
B	0	1	1
C	0	2	3
D	0	1	4
E	0	5	2

The processes are assumed to have arrived in the order A, B, C, D, E, all at time zero. This question concerns the behaviour of these processes under the following scheduling algorithms:

i. First Come, First Served (FCFS)

ii. Shortest Burst First (SBN (aka SJF))

iii. Nonpreemptive Priority (where zero is the highest priority)

iv. Round Robin (RR) (where quantum = 1)

(a) Draw a Gantt chart illustrating the execution of these processes for each of the four scheduling algorithms given above.

(b) What is the turnaround time of each process for each of the four scheduling algorithms?

(c) What is the waiting time of each process for each of the four scheduling algorithms?

1. *Hint:* Assume that processes can be preempted.

(d) Which of the schedules results in the minimal average waiting time (over all processes)?

w5. Although state diagrams for processes usually show a single blocked (or waiting) state, there is rarely a single queue of blocked processes associated with that state. Why?

w6. Multiprocessor operating systems often have a ready queue for each processor, with their designers citing performance improvements for this design. Why should multiple ready queues improve performance and how does this design complicate matters?

w7. Consider the following preemptive–priority-scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the processor (in the ready queue, but not running), its priority changes at a rate α; when it is running, its priority changes at a rate β. All processes are given a priority of zero when they enter the ready queue. The parameters α and β can be set to give many different scheduling algorithms.

(a) What is the policy that results from $\beta > \alpha > 0$?

(b) What is the policy that results from $\alpha < \beta < 0$?

(c) It might seem like updating the priorities of all the processes in the ready queue could take an amount of time dependent that is on the length of the ready queue (i.e., if the size of the ready queue is n, it would take $O(n)$ time), but in fact it is possible to do better. Explain how a clever implementation could make the time to update these priorities independent of the length of the ready queue.

In your scheme, would the following operations become dependent on the length of the ready queue, and if so what would be the relationship?

i. Add a new process to the ready queue

ii. Switch from running one process

Express your answer using O notation if you know it. If these operations do not depend on the length of the ready queue, explain why.

(d) The algorithm above is underspecified—for example, it is unclear exactly when the priorities of processes are changed.[2] Similarly, there are some cases that could occur in which it is not clear how the scheduler should behave. Explain one such case and specify an appropriate behaviour.

(e) Can this algorithm provide a timesharing scheduling policy?

- If it *is* possible, state values of α and β that provide such a policy and explain the scheduling behaviour that will be observed—does it correspond to Round Robin, or is it different?
- If it *is not* possible, explain why and suggest an enhancement to the algorithm that would allow it to support timesharing scheduling policies.

w8. Contrast the following kernel designs (which may overlap):

(a) Monolithic non-reentrant kernel

(b) Reentrant kernel

(c) Multi-threaded kernel

(d) Microkernel

(This question may require you to read about some topics we have not covered in class.)

2. Presumably priorities are updated at discrete intervals in time, rather than continuously.