March 19, 2008
Operating Systems
LIACS
Spring Semester 2008
Assignment #4

Deadline: Tuesday, April 1, 2008
You are encouraged to work in teams of 2 persons.

Goal

Learn about threads; experiment with race conditions.

Introduction

In this exercise we will continue learning about the key concept of OSs: the notion of a process. In the first weeks we learned about some process management system calls of Linux. This week we will focus on the use of the POSIX thread API for C/C++. It allows one to spawn a new concurrent process flow. It is most effective on multiprocessor systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing. Threads require less overhead than "forking" or spawing a new process because the system does not initialize a new system virtual memory space and environment for the process. While most effective on a multiprocessor systems, gains are also found on uniprocessor systems which exploit latency in I/O and other system functions which may halt process execution. (One thread may execute while another is waiting for I/O or some other system latency.) All threads share the same address space. A thread is spawned by defining a function and its arguments which will be processed by the thread.

Assignment

a) In this part of the exercise we will set up a race condition. Moreover we will set up things in such a way that we will be able to see the nasty effects of this race condition. We assume the use of POSIX Pthreads. The POSIX standard does not prescribe the exact concurrency requirements among threads. Assume that the implementation supports concurrent threads with RR scheduling. That means, each thread is automatically preempted after it ran for some time and another thread is started. Thus, all threads within a process proceed concurrently at unpredictable speeds.
To set up a race condition, create two threads, *thread0* and *thread1*. The parent thread of the two threads – *main* – defines two global variables *account1* and *account2*. Each variable represents a bank account; it contains a single value – the current balance – which is initially zero. Each thread emulates a banking transaction that transfers some amount of money from one account to another. That means, each thread reads the values in the two accounts, generates a random number *amount*, adds this number to one account and subtracts it from the other.

The following code skeleton illustrates the operation of each thread:

```
counter = 0;
do {
    temp1 = account1;
    temp2 = account2;
    amount = rand();
    account1 = temp1 - amount;
    account2 = temp2 + amount;
    counter++;
} while (acount1+account2 == 0);
print(counter);
```

Both threads execute the same code. As long as the execution is not *interleaved*, the sum of the two balances should remain zero. However, if the threads are interleaved, one thread could read the old value of *account1* and the new value of *account2,* or vice versa, which results in the loss of one of the updates. When this is detected, the thread stops and prints the step (i.e., the *counter*) at which this occurred.
Write the code for *main* and the two threads. Then measure how long it takes to see the effect of the race condition.

b) In this part of the exercise you will translate the following program:
Cooperating Processes:

```
procA {
while (TRUE) {
        <compute section A1>
        update(x);
        <compute section A2>
        retrieve(y);
    }
}

procB {
while (TRUE) {
        retrieve(x);
        <compute section B1>
        update(y);
        <compute section B2>
    }
}
```

Processes A and B share the variables *x* and *y*. Process A writes *x* and reads *y*, while process B writes *y* and reads *x*. The two processes need to cooperate so that B does not read *x* until after A has written it, and so that A does not read *y* until B has written a new value to *y*.

Use threads and possibly mutexes to accomplish a good working translation of the given consumer-producer program.

c) Implement the program of b) using threads and instead of mutexes use Peterson's algorithm for a correct solution.

d) Think of a concrete problem/situation where a solution to the problem can be achieved with a program described in b).

Hints:

i Read the pthread man pages to become familiar with pthreads and mutexes. In particular, the functions `pthread_create`, `pthread_exit`, and `pthread_join` are useful when working with pthreads, and `pthread_mutex_init` and `pthread_mutex_lock/unlock` are useful when working with mutexes.

ii In order to compile your programs with the pthread library, add `-lpthread` to the gcc command line.

Deliverables:

i a README file which contains a list of items you turned in + in what fashion they are to be used

ii All C programs – part a, b, c, and the example for part d.

iii A one-page lab report detailing what you have done in the lab and more importantly what you have learned from this and what you have learned from your lab partner.

Make sure each deliverable contains the names of the authors, student IDs, assignment number and date turned in!

Due date: as stated before April 1st.

Put all files you want to deliver into a separate directory (e.g., *assignment4*), free of object files and/or binaries, and create a gzipped tar file of this directory:

```
      tar -cvzf assignment4.tgz assignment4/
```

Mail your gzipped tar file to Sven van Haastregt (e-mail: svhaastr at liacs dot nl) and *let the subject field of your e-mail contain the string* ``OS Assignment 4".