EXPERIMENT 3

PROCESS SYNCHRONIZATION

1. OBJECTIVES

After completing this lab, you will be able to:

- Understand how to synchronize processes/threads.
- Understand interleavings and race conditions, and master some way of controlling them.
- Know how to use *locks/semaphores* to solve a critical section problem.

2. LABORATORY

Software Lab 1 (N4-01a-02)
Software Lab 3 (N4-B1c-14)
SPL (N4-B1b-11)

3. EQUIPMENT

Pentium IV PC with Nachos 3.4.

4. MODE OF WORKING

You should be working alone. No group effort.

5. THREAD OPERATIONS

The thread is a light-weight process in Nachos. In the first experiment we have learnt the basic feature of thread execution in Nachos, and in the second experiment we have learnt how to control the scheduling of those threads using time quanta. In this experiment, we need to use the following thread operations to finish the exercises. Please read the source code under the directory `threads` (particularly `thread.h` and `thread.cc`).

```
void Fork(VoidFunctionPtr func, int arg, int joinP);
// Make thread run (*func)(arg) and
// know if a Join is going to happen.
```

`Fork` creates a new thread of control executing in the calling user thread address space. The function argument is a procedure pointer. The new thread will begin executing in the same address space as the thread calling `Fork`. The new thread must have its own user stack in the user address space, separate from all other threads. It must be able to make system calls and block independently of other threads in the same thread.

```
void Join(Thread *forked);     // Waits for specified thread to
                               // finish before continuing
```

A parent thread may call Join to wait for a child thread to complete (e.g., to Exit).

```
void Yield();                  // Relinquish the CPU if any
                               // other thread is runnable
```

This function was discussed in Experiment 2.

```
void Sleep();                  // Put the thread to sleep and
```

```
                                        // relinquish the processor
```

A thread gives up the CPU ownership and allows another thread to execute. Its state shifts from running to waiting (waiting for the wake-up event). When the wake-up event is triggered, the thread is put to the ready queue.

6. CONTEXT SWITHCHES AND RACE CONDITIONS

On a multiprocessor, the executions of threads running on different processors may be arbitrarily inter-leaved, and proper synchronization is even more important. In Nachos, which is uniprocessor-based, interleavings are determined by the timing of context switches from one thread to another. On a uniprocessor, properly synchronized code should work no matter when and in what order the system chooses to run the threads. The best way to find out if your code is "properly synchronized" is to see if it breaks when you run it repeatedly in a way that exhaustively forces all possible interleavings to occur. This is the basic idea to check whether multiple threads have the problem of race condition. To experiment with different interleavings, you must somehow control when the executing program makes context switches.

Context switches can be either voluntary or involuntary. Voluntary context switches occur when the thread that is running explicitly calls a yield or sleep primitive (Thread::Yield or Thread::Sleep) to cause the system to switch to another thread. A thread running in Nachos might initiate a voluntary switch for any of a number of reasons, perhaps in the implementation of a synchronization facility such as a semaphore.

In contrast, involuntary context switches occur when the inner Nachos modules (Machine and Thread) decide to switch to another thread all by themselves. In a real system, this might happen when a timer interrupt signals that the current thread has consumed its allocated time quanta (as in Experiment 2) or is hogging the CPU.

7. EXERCISES

1. Copy a complete set of Nachos 3.4 source code to your home directory by typing ***cp -r /shares/nachos-exp3-4 ~***

2. In this exercise, we will conduct the following steps to understand race condition problem in Nachos.

   a) Change your working directory to Experiment 3 by typing ***cd ~/nachos-exp3-4/exp3***

   b) Read the Nachos thread test program ***threadtest.cc*** carefully. There is a shared variable named ***value*** (initially zero). There are two functions, namely ***void Inc(_int which)*** and ***void Dec(_int which)***, which increases and decreases ***value*** by one, respectively. In this exercise, you need to consider different interleaving executions of `Inc` and `Dec` so that the shared variable value is equal to a predefined value after the threads complete.

   c) You need to implement the following three functions. When all the threads (two `Inc_v1` threads and two `Dec_v1` threads) complete in `TestValueOne()`, ***value=1***.

      ```
      void Inc_v1(_int which)
      ```

```
void Dec_v1(_int which)
void TestValueOne()
```

In `Inc_v1` and `Dec_v1`, you need to use `Yield` primitive in Nachos to induce context switch. `Inc_v1` *and* `Dec_v1` *should have the same logic as* `Inc` *and* `Dec`, *respectively. You are only allowed to add* `Yield` *into those two functions.* You need to implement `ThreadValueOne()` by creating two threads with `Inc_v1` and two threads with `Dec_v1`. The current thread should wait for all those threads to complete. At the end of `TestValueOne()`, a checking is performed on whether the value is ***1***. If the checking is passed, you should get the message "congratulations! passed.". Otherwise, an error message is printed.

d) After you finish implementing the above-mentioned functions, you can demonstrate the result of `TestValueOne()`, by commenting other test functions in `ThreadTest()` like below.

```
//for exercise 1.
TestValueOne();
//TestValueMinusOne();
//for exercise 2.
//TestConsistency();
```

e) Compile Nachos by typing ***make***. If you see "***ln -sf arch/intel-i386-linux/bin/nachos nachos***" at the end of the compiling output, your compilation is successful. If you encounter any anomalies, type ***make clean*** to remove all object and executable files and then type ***make*** again for a clean compilation.

f) Test this program by typing ***./nachos***. If you see “congratulations! passed.” at the end of the debugging messages, your program is successful. Otherwise, “failed.” will be displayed.

g) Repeat Steps 3)—6), and implement the following three functions. When all the threads (two `Inc_v2` threads and two `Dec_v2` threads) complete in `TestValueMinusOne()`, ***value=-1***. At Step 4), you need to test `TestValueMinusOne()`.

```
void Inc_v2(_int which)
void Dec_v2(_int which)
void TestValueMinusOne()
```

3. In this exercise, we will conduct the following steps to understand process synchronization problem in Nachos.

1) Change your working directory to Experiment 3 by typing ***cd ~/nachos-exp3-4/exp3***

2) You need to implement the following three functions. When all the four threads (two `Inc_Consistent` threads and two `Dec_Consistent` threads) complete in `TestConsistency()`, ***value=0***. You need to achieve consistent result (***value=0***), regardless of different interleaving execution orders in `Inc_Consistent` and `Dec_Consistent` as well as different thread fork orders in `TestConsistency()`.

```
void Inc_Consistent (_int which)
void Dec_Consistent (_int which)
```

```
void TestConsistency ()
```

In `Inc_Consistent` and `Dec_Consistent`, you use `Yield` interface in Nachos to induce context switch. You need to implement `TestConsistency()` by creating two threads with `Inc_Consistent` and two threads with `Dec_Consistent`. The current thread should wait for all those threads to complete. At the end of `TestConsistency()`, a checking is performed on whether the ***value*** is ***0***. If the checking is passed, you should get the message "congratulations! passed.". Otherwise, an error message is printed.

3) After you finish implementing the above-mentioned functions, you can demonstrate the result of `TestConsistency()`, by commenting other test functions in `ThreadTest()` like below.

```
//for exercise 1.
//TestValueOne();
//TestValueMinusOne();
//for exercise 2.
TestConsistency();
```

4) Compile Nachos by typing ***make***. If you see "***ln -sf arch/intel-i386-linux/bin/nachos nachos***" at the end of the compiling output, your compilation is successful. If you encounter any anomalies, type ***make clean*** to remove all object and executable files and then type ***make*** again for a clean compilation.

5) Test this program by typing ***./nachos***. If you see "congratulations! passed." at the end of the debugging messages, your program is successful. Otherwise, "failed." will be displayed. You should consistently get the same result irrespective of the thread interleavings; it is a good idea to test your implementation for different interleaved executions of `Inc_Consistent` and `Dec_Consistent` as well as for different thread fork orders in `TestConsistency()`.

8. ASSESSMENT

- Assessment of your implementation. Please leave your code in the **exp3** folder for TA/Supervisor to review. Deadline is 1 week after your lab session (e.g., if lab session is from 10AM-12PM on a Monday, then deadline is 9:59AM on the next Monday).
- **Quiz 2**, which is an online multiple-choice quiz, will be administered through NTULearn on 12th November, 2020, 9:30AM – 10AM (Week 13).