EXPERIMENT 1

NACHOS THREADS

1. OBJECTIVES

After completing this lab, you will be able to:

- Understand how context switch works in operating systems.
- Trace the execution flow of a thread (i.e., a lightweight process) in a multi-threading environment.

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. WHY NACHOS?

In teaching operating systems at an undergraduate level, it is very important to provide a lab that is realistic enough to show how real operating systems work, yet simple enough for students to understand and modify in significant ways. Nachos [1] is a good Instructional Operating System for teaching basic operating systems. The use of Nachos is motivated by the complicated use of the current operating systems such as LINIX, MINIX [2], or even much earlier operating systems such as MULTICS [3]. Nachos illustrates and takes advantage of modern operating systems technology, such as threads and remote procedure calls.

The introduction of minicomputers, and later, workstations, also aided the development of instructional operating systems. Rather than having to run the operating system on the bare hardware, computing cycles became cheap enough to make it feasible to execute an operating system kernel using a simulation of real hardware. The operating system can run as a normal UNIX process, and invoke the simulator when it would otherwise access physical devices or execute user instructions.

However, recent advances in operating systems, hardware architecture, and software engineering have left many operating systems projects developed over the past three decades out of date. Networking and distributed applications are now commonplace. Threads are crucial for the construction of both operating systems and higher level concurrent applications. And the cost-performance trade-offs among memory, CPU speed and secondary storage are now quite different from those imposed by core memory, discrete logic, magnetic drums, and card readers.

For example, a complete UNIX-like file system would be too complicated for students to understand in only a few weeks. The approach was to build the simplest implementation for each sub-system of Nachos; this provides students a working example, albeit overly simplistic, of the operation of each component of an operating system. As a result of the emphasis on simplicity, the Nachos operating system is

about 2500 lines of code, about half of which are devoted to interface descriptions and comments.

It is thus practical for students to read, understand, and modify Nachos during a single semester course.

5. NACHOS OVERVIEW

Like many of its predecessor instructional operating systems, the Nachos kernel and hardware simulator run together in the same UNIX process. Nachos has several significant differences from earlier systems:

- The simulation is deterministic. Debugging non-repeatable execution sequences is a fact of life for professional operating systems engineers. Instead of using UNIX signals to simulate asynchronous devices such as the disk and the timer, Nachos maintains a simulated time that is incremented whenever a user program executes an instruction and whenever a call is made to certain low-level operating system routines. Interrupt handlers are then invoked when the simulated time reaches the appropriate point.
- Nachos is implemented in a subset of C++. Object-oriented programming is becoming more popular. It was a natural idiom for stressing the importance of modularity and clean interfaces in building operating systems. To simplify matters, certain aspects of the C++ language are omitted: derived classes, operator and function overloading, and C++ streams.

6. THREAD MANAGEMENT

In much the same way as pointers for beginning programmers, understanding concurrency requires a conceptual leap on the part of students. Contrary to Dijkstra [4], the best way to teach concurrency is with a “hands-on” approach. Nachos helps in two ways. First, thread management in Nachos is explicit: students can trace, literally statement by statement, what happens during a context switch from one thread to another, both from the perspective of an outside observer and from that of the threads involved. This experience is crucial to demystifying concurrency. Precisely, because C++ allows nothing to be swept under the covers, concurrency may be easier to understand (although harder to use) in these programming languages than in those explicitly designed for concurrency, such as Ada [5], Modula-3 [6], and Concurrent Euclid [7].

A working thread system, as in Nachos, allows students to practice writing concurrent programs and to test out those programs. Even experienced programmers find it difficult to think concurrently; a widely used operating systems textbook had an error in one of its concurrent algorithms that went undetected for several years.

The thread system is based on FastThreads [8]. The primary goal was simplicity, to reduce the effort required for students to trace the behavior of the thread system.

For simplicity, thread scheduling is normally non-preemptive, but to emphasize the importance of critical sections, there is a command-line option that causes threads to be time-sliced at ”random”, but repeatable, points in the program. Concurrent programs are correct only if they work when “a context switch can happen at any time”.

In this lab, you are required to read and understand the partial thread system that has been written for you. This thread system implements thread fork, thread completion, along with semaphores for synchronization. Run the program Nachos for a simple test of the code. Trace the execution path (by hand) for the simple test case provided.

When you trace the execution path, it is helpful to keep track of the state of each thread and which procedures are on each thread's execution stack. You will notice that when one thread calls `SWITCH`, another thread starts running, and the first thing the new thread does is to return from `SWITCH`. This comment will seem cryptic to you at this point, but you will understand threads once you understand why the `SWITCH` that gets called is different from the `SWITCH` that returns.

7. EXERCISES

1. Copy a complete set of Nachos 3.4 source code to your home directory by typing ***cp -r /shares/nachos-exp1-2 ~***

2. Change your working directory to Experiment 1 by typing ***cd ~/nachos-exp1-2/exp1***

3. Read the Nachos thread test program ***threadtest.cc*** carefully to understand how multiple threads are created and executed concurrently and to predict the output from the test program.

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. Trace a run of this Nachos test program by typing ***./nachos -d > output.txt***. Option ***–d*** is to display Nachos debugging messages.

6. Fill in the following table (one of the example is shown below) whenever:
   - the ready list (i.e., ready queue in your textbook) of Nachos is changed, or
   - the current thread is changed, or
   - a new message is printed in method `SimpleTest(int which)`.

ready list	**current thread**	**printf message**
Empty	main	
child1	main	
child1, child2	main	
Child1, child2	main	thread 0 looped 0 times
child2, main	child1	

   In column *ready list*, you need to list the names of the threads in the ready list (with the leftmost being the thread at the front of the queue). Column *current thread* should show the current thread. The messages generated in method `SimpleTest(int which)` should be provided in column *printf message* when appropriate. Fill as many rows as necessary until Nachos exits (each row corresponds to 10-tick output except the first row).

7. List all context switches occurring in the test run above. Indicate from what thread to what thread is the context switching.

8. ASSESSMENT

- Assessment of your implementation. Please leave the file **output.txt** as well as a **table.pdf** file containing the above (populated) table in the **exp1** 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 1**, which is an online multiple-choice quiz, will be administered through NTULearn on 1st October, 2020, 9:30AM – 10AM (Recess week).

9. QUESTION

Describe what and how a short-term CPU scheduler is implemented in Nachos by examining relevant source code files in the `threads` directory. Does the output from the test program reflect this scheduling discipline? Justify your answer.

10. References

[1] http://homes.cs.washington.edu/~tom/nachos/
[2] http://www.minix3.org/
[3] http://www.multicians.org/
[4] E. W. Dijkstra, Solution of a problem in concurrent programming control, Communications of the ACM, v.8 n.9, p.569, Sept. 1965
[5] http://www.adahome.com/
[6] http://www.modula3.org/
[7] J.R. Cordy and R.C. Holt 1980. Specification of Concurrent Euclid. Technical reports CSRI-115 (July 1980) and CSRI-133 (August 1981), Computer Systems Research Institute, University of Toronto.
[8] Thomas E. Anderson FastThreads User's Manual. University of Washington. Seattle. January 1990.