Operating Systems, Sprint 2002

CS 5/43201
Department of Computer Science
Kent State University

Project Nachos

Suggested Reading: To do this first assignment, you will need to understand Nachos. Read the article "A Roadmap Through Nachos" by Tom Narten (it is in the class web site) very carefully. Also read the following source code files in the thread/ directory.

- main.cc threadtest.cc
- thread.h thread.cc
- scheduler.h, scheduler.cc
- synch.h, synch.cc
- list.h, list.cc
- synclist.h, synchlist.cc
- system.h, system.cc
- utility.h, utility.cc
- switch.h, swith.cc

You many also want to see the following files which implements the MIPS Virtual Machine:

- intterupt.h, interrupt.cc
- time.h, timer.cc
- stats.h
- console.h, console.cc
- disk.h, disk.cc

Preparation: The first thing you will need to do is to copy, install and test run a copy of Nachos in your directory. I have kept a ready to use copy of Nachos in my account. The link "**Begin here**.." in "**Project Page**" contains step by step instructions to get it. Make sure you can run the simple test program in Nachos by typing at the end of following those instructions:

% nachos

There is a function called *ThreadTest()* in file *threadtest.cc*. This is a sample user program and your starting point. This function is called by Nachos automatically for you from *main.cc*. In this assignment you will need to write many versions of this function. Inspect the file and the function, study and understand the output.

Your next step is to trace the execution path of this program. There are three ways to trace. (i) you may insert *printf()* statements inside, recompile and run it. (ii) you may use *gdb* debugger or (iii) you can use the built-in debug of Nachos. To use Nachos' debug run it with "-d" flag:

%nachos –d

If you browse you will see that Nachos code has already many DEBUG() (i.e. conditional *printf())* statements inserted in various routines. The "–d" flag activates them. For a complete listing of all these pre-inserted *printf()* statements in the source files in threads directory you may type:

%grep DEBUG *h *cc

If you run Nachos with "-d" flag, you will see that all DEBUG statements have 't' flag in them. The files in machine directory similarly have "i" and "m" flags. You may choose to restrict the trace only to the thread or the machine routines by further specifying flags in following manners:

%nachos –d t
%nachos –d ti

This assignment is about concurrently running threads. Concurrent threads voluntarily relinquish CPU by calling the function *thread::Yield().* when *Yield()* is called the calling function is temporarily suspended by Nachos scheduler and a second thread from the ready list is activated. If the concurrent threads are written correctly then the exact point where a thread yields does not matter. Conversely, one way of ensuring the correctness of a concurrent threaded system is to insert Yields() at different places and see how their running behavior changes. To help in such testing Nachos has provided a handy command line tool. It can insert random yields inside user threads on user's behalf. You can invoke this feature by using –rs flag. Type:

%nachos –rs 100

Here the number 100 is the seed. It can be 200, 300 or any integer. Every time you invoke Nachos with the same seed the yields will be inserted at the same set of points. Experiment running Nachos with different seeds and try to understand any change is the output.

Road Map Questions For Nachos Threads

Here I have compiled a set of questions which will lead you through the Nachos system. You must know the correct answers for them to make sure that you have understood the Nachos System. These are not assignments and you do not need to turn in the answers. Instead, in about a week from now, most likely an unannouced quiz will verify if you have solved these problems or not.

1. How many registers Nachos virtual machine has?
2. What are the physical locations of the *Stack Pointer* and *Current Program Counter* registers?
3. What are the preconditions for executing *machine::Run()?*
4. What is the name of the variable that remembers the ticks in Nachos? Which header file defines it?
5. List the three event points at which Nachos' time advances.
6. In Nachos, can you place an interrupt anywhere in time?
7. Will *SetLevel()* tick the clock if a process wants to:
   a. Disable interrupt?
   b. Enable interrupt from disabled state?
   c. Re-enable interrupt when the interrupt is already enabled?
8. Why *OneTick()* checks SystemMode variable?
9. Why *StackAllocate* places a sentinel value at the top of allocated stack?
10. It is not physically possible for a thread to kill itself. Explain the mechanism how a Nachos' thread terminates.
11. Why *ThreadRoot()* and *Switch()* functions are implemented in Machine Language? Why there are so many versions of these two routines in *switch.s*?
12. Trace the routines invoked from the point an user process calls the routine *Fork().*
    a. Describe the exact point at which the new process starts running.
    b. What is the purpose of *ThreadRoot*?
    c. When exactly a child starts execution?
13. Trace the routines invoked from the point an user process calls the routine *Yield().*
    a. Describe the exact point at which context switch has taken place.
    b. Which thread removes the carcass of a terminating thread?
14. What is "Noff"? Convert a "coff" file into a "Noff" file and execute it in Nachos.

Operating Systems, Spring 2002

CS 5/43201
Department of Math and Computer Science
Kent State University

Project#1: Due Date: 3/13/02

Objective: In this project we will learn about threads, how concurrent threads can be started, and how such threads can cooperate, communicate and synchronize. In this first project we will be a user process in Nachos and explore how it can help us in managing concurrent cooperating threads.

Assignment: In the problems 1-4 you should write a function called *ThreadTest()* (and replace the initial one given in *threadtest.cc*), which takes no argument. This function is called by Nachos automatically for you from *main.cc*. In function *ThreadTest(),* fork two new threads named Mary and Anna. The Mary and Anna should look like below:

```
int  letter = 0;
while(1)
{
        printf("Mary gets paper, pen, envelop, stamp for letter # %d \n", letter);
        printf("Mary write letter # %d \n", letter);
        printf("Mary seals envelop # %d \n", letter);
        printf("Mary mails to Anna # %d \n", letter);
        /*PLACE#1*/
        letter++;
}
```

Anna should look like this:

```
int letter=0;
While (1)
{
        printf("Anna receives Mary's mail # %d in her mailbox\n", letter);
        printf("Anna opens and reads the letter # %d\n", letter);
        printf("Anna thinks about Mary# %d\n", letter);
        /*PLACE#1*/
        letter++;
}
```

The two threads will represent two entities.

1. (200 points) In the file **pa1.cc**, write a function *ThreadTest*, which forks a Mary and a Anna thread as described above. At the end of the loop in both the routines (as marked as PLACE#1 comment in the programs) insert the line "*currentThread->Yield()".*

   Copy the file **pa1.cc** over the file *threadtest.cc*, compile and run Nachos. Then answer the following questions in the files **proj1.txt**:

   a. What happens when it runs? It is desirable that before a particular letter can be received by Anna, it must be mailed by Mary. Does the output correspond to the desirable solution? Include a few line of output and explain what is happening.

b. Now let Nachos insert random yields, by calling it with "-rs" command line option as described above. Is the production/consumption is synchronized now? Explain what is happening including few lines of output.

2. (200 points) In the file **pa2.cc**, write a new function *threadtest()* which creates a new semaphore named s with an initial value of 1, and then forks the Mary and Anna threads as described. Now place the semaphore in the right place so that the desired solution can be achieved (Each letter should be mailed by Mary before it is received by Anna).

   Copy the file **pa2.cc** over the file *threadtest.cc*, compile and run Nachos. Now in the files **proj1.txt** answer the following question:

   a. What happens when the program runs? Include a few line of output and explain the behavior.

   b. Now, let Nachos insert the random Yields. What happens now? Include few lines of output and explain the behavior. (hint: how to insert random yield has been explained in the handout "Project Nachos")

   c. Now try inserting your own Yields in several places. Can you make things worse? Explain what is happening including few lines of output.

   d. If Mary decides, she will write a new letter only if Anna reads the earlier one, will the above one semaphore solution work? Explain your answer (Hint: you may want to do some experimentation like b and c).

3. (200 points) Let us assume that Anna's mailbox can hold only 3 letters. As a result, US post-office will not receive any mail from Mary if it finds that Anna's mailbox is full. Modify pa2.cc into **pa3.cc** so that that Mary will wait if Anna's mailbox is full. Now, let Nachos insert the random Yields. In file **proj1.txt** include few lines of output from several such runs and explain the behavior.

For the next assignment read the implementation of semaphores in *synch.cc* file. Backup the file synch.cc on synch.bak, and then modify the synch.cc into synch1.cc:

4. (200 points) Implement the condition variables directly using interrupt enable and disable to provide atomicity in the file **synch1.cc**

   a. Complete the null functions *condition::Condition, condition::~condition, condition::Wait and condition::Signal* functions. Copy the file *synch1.cc* on *synch.cc*, recompile and run.

   b. Write a new file **pa4.cc** to synchronize the operations of the Mary and Anna threads of problem #2 (one letter at a time) using the condition variables approach.

   d. Now test the system by let Nachos insert the random Yields. What happens now? In file **proj1.txt** include few lines of output from several such runs and explain the behavior.

5. (200 points) In this assignment we will implement a barbershop. A Barbershop has a waiting room with 4 chairs and 2 barbers with two barbers-chairs. If there is no customers to be served, the barbers go to sleep in the barber's chair. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is as sleep, the

customer wakes him up. Below are the two routine which describes the actions of the barbers and that of the customers:

```
#define CHAIRS 4
int waiting=0 /* number of customers waiting */
void barber (void)
{
while (1) {

        waiting = waiting –1;      /*invite a customer*/
        cut_hair();
}
}

void customer(void)
{
if (waiting< CHAIRS) {              /*If there is no free chair then just leave*/
waiting=waiting+1;        /*occupy a chair and increment count of a chair*/
get_haircut();
}
}

cut_hair()
{
int pid;
pid=getpid();
printf(" #d Cutting Hair\n"l,pid);
}

get_haircut()
{
int pid;
pid=getpid();
printf(" #d Getting Haircut\n", pid);
}
```

a. You can simulate the Barbers shop in your computer by forking-off two barber processes and N number of customer processes. Write/complete a new **pa5.cc** program with the parent, the above two routines, and any other routines to simulate the barbershop with 2 barbers and 15 customers. Test the system with random yield of Nachos. Include some output line to explain the operation of the system and add it to **proj1.txt**.

b. However, the above simple routines will not work properly. Because although they describe individual entities, but there is no co-ordination between the customers and the barbers. Customers do not know if any of the barbers is free. Conversely, the barbers also don't know if there is a customer waiting. As a result the barbers may end up cutting air when there is no customer. A customer may end up sitting on an already occupied barber's chair. Modify the codes with semaphore(s) so that none of these happens (Hint: you can use two semaphores, which keeps track of the free barber and number of waiting customers).

c. Besides the above, there is yet another problem. There is the possibility that, when there is only one chair empty, accidentally two of the customers may simultaneously find one empty chair. In such a case they will both try to sit on the 4^{th} chair (without knowing that there are others who are trying to sit on the same chair). It may result in an embarrassing situation.

Use semaphore(s) to avoid such situation. One solution is to make sure that two customer processes should not be allowed to access the variable waiting. Make a new file with **pa6.cc** with both the routines modified which corrects the above mutual exclusion problems using semaphore(s). (Hint: you can use a third semaphore to ensure mutual exclusion).

How to Submit:

In this assignment you have created a set of program files *.cc and one answersheet proj1.txt which contains all your explanations.

On top of each file include your name, data and project number. Add:

```
/*************************************************
Name:
Date:
Project/Question Number:
OS CS 5/43201, SPRING 2002
Instructor: KHAN, KSU
*************************************************/
```

For source files (*.cc) comment them.

You now need to mail all of them into one package using the following procedure:

%tar cvf project1.tar pa1.cc pa2.cc pa3.cc sync1.cc pa4.cc pa5cc pa6.cc proj1.txt
After that you should have file project1.tar file, which contains above files.
Send this file to okomogor@cs.kent.edu with subject "Project 1 *your group number*"
and attach file project1.tar to that e-mail

Check thoroughly before you submit. If you need to re-send, for any reason inform TA (okomogor@cs.kent.edu) beforehand. Keep a copy of all the files including project1.tar in your directory. Do not modify them afterward. If need arises, TA may want to check these files. Any modification afterward (reflected in the file date) will result in late submission penalty.

Grading:

See notes to grader in the website.

Cheating and Copy:

Projects have to be done individually. If a copy is caught, all involved submissions (original as well as the copies) will be penalized. So it is your responsibility to guard your work. Secure the read/write access of your directories. Any copy will result in ZERO grade for the assignment for both party. Only exception is when you report the theft of your work in advance.