NachOS Project Stage 2
Soup and Salad: Synchronization
Total Points: 225

Assigned: February 4, 2016
Due: February 19, 2016

Objectives

After successful completion of this assignment, students will be able to:

1. Understand classic synchronization problems like Dining Philosophers.
2. Implement threaded solutions to synchronization problems.
3. Compare and contrast busy waiting loops and semaphores as synchronization solutions.

Overview

This second stage of the NachOS project will help you understand synchronization problems and solutions to them through the following tasks:

1. Dining Philosophers (Busy Waiting)
2. Dining Philosophers (Semaphores)
3. Post Office Simulation (Busy Waiting)
4. Post Office Simulation (Semaphores)
5. Command Line
6. Report

This document will give a detailed specification of the above tasks. In addition, it will provide design and implementation hints as preparation for further work. The focus of this assignment will be synchronization problems and various means of solving them.

Before you begin, it is strongly advised that you read through the NachOS material posted on Moodle. Also read and understand the files pertaining to synchronization, especially *semaphore.h/.cc*.

Task 1: Dining Philosophers (Busy Waiting)

Summary

For this task, you must first understand the Dining Philosophers problem. After doing so, you must implement a solution that uses only busy waiting loops in its implementation. Prompt the user for *P* and *M*, where *P* is the number of philosophers and *M* is the total number of meals to be eaten.

Procedure

After prompting the user for *P* and *M*, fork a single thread for each philosopher, similar to the Shouting

Threads task from Stage 1. There will be a number of chopsticks available equal to the number of philosophers. Each thread will then run through the Dining Philosophers algorithm, which is as follows:

1. Sit down at table.
2. Pick up left chopstick.
3. Pick up right chopstick.
4. Begin eating.
5. Continue eating for 2-5 cycles.
6. Put down left chopstick.
7. Put down right chopstick.
8. Begin thinking.
9. Continue thinking for 2-5 cycles.
10. IF all meals have not been eaten, GOTO 2.
11. ELSE leave the table.

As this is an extraordinarily polite group of philosophers, they hold themselves to the highest standards of etiquette. All philosophers must enter the room together and none should sit down until all are present. Likewise, no philosopher should get up from the table to leave until all are ready to do so.

The chopsticks can be represented as a simple array of booleans, with as many chopsticks as there are philosophers. If you wish, you may use another, more complex means to represent them, but it's crucial that there is a way to distinguish between a chopstick that is available and one that is not.

The philosophers themselves are seated in a circle such that philosopher *N* has access to chopstick *N* on their left and chopstick *(N + 1) % P* on the right. These chopsticks are the shared resource that you must control access to as part of your solution. A philosopher can only pick up a chopstick when it is available. If the chopstick is not available, then the philosopher must enter a busy waiting loop until it becomes available. For additional details on busy waiting loops, please see the Stage 1 specification.

As the philosophers progress through the algorithm, they must produce output. Specifically, they must produce output at steps 1, 2, 3, 4, 6, 7, 8, and 11 in the algorithm outlined above. All output must specify which philosopher is producing the output, as well as any other identifiers involved such as chopstick numbers, or number of meals eaten so far.

Once the total number of meals have been eaten, no philosopher should eat any additional meals. It is acceptable for such a philosopher to proceed normally through the algorithm, with associated output, as long as no additional meals are eaten.

This implementation of a Dining Philosophers solution must use busy waiting loops for all synchronization control. No semaphores are to be used.

Report

As part of your report on this project, answer the following questions about Task 1:

1. When implementing and testing your solution, did you notice any deadlock? Deadlock occurs when threads cannot progress due to improperly controlled access to critical resources. (In this case, the critical resources are the chopsticks.) How did you solve any deadlock problems?

2. Make the philosophers yield between attempting to pick up the left and right chopsticks. Run at least 3 tests with various parameters and -rs seeds. Record your findings and explain your results. Undo any changes made to accommodate this question before submitting your assignment.

Task 2: Dining Philosophers (Semaphores)

Summary

For this task, you will implement another solution to the Dining Philosophers problem as explained in Task 1. Unlike Task 1, this solution will use semaphores instead of busy waiting loops for all synchronization control. The only busy waiting loops to be used are the ones that allow a philosopher to eat and think for random amounts of time.

Procedure

The requirements of this task are identical to Task 1 in all respects with the exception of using semaphores instead of busy waiting loops for all synchronization issues. As this task uses semaphores instead of busy waiting loops, you will need to represent the chopsticks as an array of semaphores instead of an array of booleans.

Using busy waiting loops for any part of this task, with the exception of a philosopher eating or thinking for a random amount of time, will result in a penalty.

Report

As part of your report, answer the following questions about Task 2:

1. Run your solutions to Task 1 and Task 2 with the same parameters, including -rs seeds. Note the number of ticks that NachOS runs. Do this for at least 3 different sets of parameters. Record your results in a table, including -rs seeds, number of philosophers, and number of meals. Explain your findings.
2. As with Task 1, make the philosophers yield between attempting to pick up the left and right chopsticks. Run the same tests used in Task 1 and record the results. Were the results the same or different? Why? Undo any changes made to accommodate this question before submitting your assignment.

Task 3: Post Office Simulation (Busy Waiting)

Summary

For this task, you will implement a simulation of a post office. Prompt the user for *P*, *S*, and *M*, where *P* is the number of people participating in the simulation, *S* is the number of messages a person's mailbox can hold, and *M* is the total number of messages to be sent before the simulation ends.

Procedure

After prompting the user for *P*, *S*, and *M*, fork a number of threads equal to the number of people involved in the simulation. Each person then proceeds according to the following algorithm:

1. Enter the post office.
2. Read a message in that person's mailbox.
3. Yield.
4. IF there are more messages to read, GOTO 2.
5. ELSE compose a message addressed to a random person other than themselves.
6. IF recipient's mailbox is not full, place the message inside it.
7. ELSE enter a busy waiting loop until recipient's mailbox is no longer full.
8. Leave the post office.
9. Wait for 2-5 cycles.
10. GOTO 1.

For this task, the mailboxes are the critical resource that must be protected via synchronization techniques. Each mailbox must have an associated semaphore to protect access to it. Like the chopsticks from Task 2, this can be represented as an array of semaphores. The messages that each person sends should be randomly selected from a list, similar to the shouts from Stage 1.

The mailboxes themselves can be represented as 2D arrays. One dimension of the array represents the person who owns the mailbox, while the other dimension represents the slots in the mailbox that can hold messages addressed to that person.

The algorithm as written is vulnerable to deadlock. It is possible for two or more people to attempt to send mail to each other, but their mailboxes are both full. In such a situation, they would both enter a busy waiting loop for the mailbox to empty, which can no longer happen now that they are both waiting. You must address this deadlock somehow, but is NOT acceptable to modify or remove existing messages in the mailbox. Regardless of the method used, a person must attempt to send their message at least 3 times before any deadlock prevention kicks in.

Like Task 1 and 2, you must provide output as the people move through the algorithm. In particular, there must be output associated with steps 1, 2, 5, 6, and 8. In addition, you must provide output detailing the contents of read, composed, and sent messages, as well as the senders and recipients of all messages. You must also provide output whenever deadlock prevention comes into play.

Each person must be aware of the total number of messages sent and should not attempt to send a message if this limit has been reached. It is acceptable for such a person to proceed through the algorithm as usual, with associated output, as long as no messages are sent.

Report

As part of your report, answer the following questions about Task 3:

1. Explain the method you used to resolve the deadlock problem. Why did you choose this particular method?

Task 4: Post Office Simulation (Semaphores)

Summary

For this task, you will implement the same post office simulation as in Task 3. However, you must use semaphores to control mailbox access when attempting to send a message to another person.

Procedure

The algorithm for this task is similar to that for Task 3, with the exception of using additional semaphores. The steps are as follows:

1. Enter the post office.
2. Read a message in that person's mailbox.
3. Call *V()* on a semaphore corresponding to that person's mailbox.
4. Yield.
5. IF there are more messages to read, GOTO 2.
6. ELSE compose a message addressed to a random person other than themselves.
7. Call *P()* on a semaphore corresponding to the recipient's mailbox.
8. Place the message in their mailbox.
9. Leave the post office.
10. Wait for 2-5 cycles.
11. GOTO 1.

Note the addition of a semaphore in steps 3 and 7. This semaphore represents the space available in the mailbox. Calling *V()* represents the action of freeing up room in the mailbox when a message is read, and calling *P()* represents the action of taking up a slot in the mailbox when inserting a message.

With the exception of changes due to the semaphore, the requirements of this task are identical to Task 3.

Report

As part of your report, answer the following question about Task 4:

1. Did you experience any deadlock when testing this task? How was it different from Task 3?

Task 5: Command Line

Summary

Extend the existing *-A* command line option from Stage 1 to accommodate the tasks from this project. The list of valid inputs is as follows:

Command Line	*Task*
./nachos -A 1	Stage 1: Input Identification
./nachos -A 2	Stage 1: Shouting Threads
./nachos -A 3	Task 1: Dining Philosophers (Busy Waiting)
./nachos -A 4	Task 2: Dining Philosophers (Semaphores)
./nachos -A 5	Task 3: Post Office Simulation (Busy Waiting)
./nachos -A 6	Task 4: Post Office Simulation (Semaphores)

As implied by the table, it should still be possible to access previous tasks from Stage 1. Invalid values should result in an error message.

Procedure

You should already have a working *-A* option implemented from Stage 1. If not, refer to the specification sheet for that stage. All you need to do is extend the range of valid parameters as detailed in the table above. As usual, make sure you validate the parameter for sanity and correct value range, including whether or not the parameter actually exists. (For example, *./nachos -A* should not cause a segfault or similar.)

Task 6: Report

Summary

You must turn in a report on this assignment along with your code. In addition to the questions listed under each task, the report should answer the following:

1. In your own words, explain how you implemented each task. Did you encounter any bugs? If so, how did you fix them? If you failed to complete any tasks, list them here and briefly explain why.
2. What sort of data structures and algorithms did you use for each task?

Procedure

Your report should be in *.pdf*, *.txt*, *.doc*, or *.odt* format. Other formats are acceptable, but you run the risk of the TA being unable to open or read it. Such reports will receive 0 points with no opportunity for resubmission.

Your name and CLID must be clearly visible. For group assignments, include the name and CLID of all group members. The questions in the report should be arranged by their associated task, then numbered. There is no minimum length, although insufficient detail in your answers will result in a penalty.

Hints

Except for Task 5, all of your code will be in *threadtest.cc*. Task 5 will be implemented in *system.cc*.

To run and test your code, type *./nachos* from the *threads* directory after compiling, just like with Stage 1. Once you complete Task 5, select the appropriate task with *./nachos -A X* where *X* is 3, 4, 5, or 6. Make sure that you can still run old tasks from Stage 1 as well.

Like Stage 1, you are expected to validate user input. The same concerns about acceptable input types and edge cases within that type apply.

To clarify when busy waiting loops vs. semaphores should be used:

- In Task 1, no semaphores are to be used at all. You should use busy waiting loops for any and all synchronization problems. This includes etiquette, where the philosophers enter and leave as a group.
- In Task 2, the reverse is true: use semaphores and only semaphores, with no busy waiting loops, with the exception of philosophers who are idling while they eat or think. Again, this includes etiquette.
- In Task 3, you should use semaphores, with the sole exception of when a person attempts to put mail into someone else's mailbox. At this point, and only at this point, you should use a busy waiting loop to check if the mailbox is full or not and keep the thread in place if it is. Remember to resolve the deadlock issue somehow.
- In Task 4, the busy waiting loop from Task 3 should be removed and a semaphore should use *P()* and *V()* to represent mail being inserted and removed from the mailbox. This is outlined in further detail under the Task 4 notes.

If you look at the implementation of *Semaphore*, you will notice that the threads are queued up neatly while they wait their turn. You do not need to duplicate this neat queuing structure in a busy waiting loop. This means that a busy waiting loop is less structured than a semaphore and that a thread inside a busy waiting loop may have to wait an unusually long amount of time if it's unlucky. Similarly, a busy waiting loop does not inherently have anything to prevent interrupts or random *Yield()* statements from *-rs*. This is normal and part of the reason why it's crucial to keep deadlock in mind when designing your solutions. If you're not careful, a badly timed *Yield()* from *-rs* can cause deadlock. Test your solutions thoroughly!

Under no circumstances should your code segfault, assert out, or initiate a stack dump. It will be thoroughly tested with a wide range of valid and invalid inputs to check for this.

When testing your code, make sure it works with the *-rs* option. *-rs* is a command line option that takes a positive integer as a seed and uses it to force the current thread to yield the CPU at random times. You can use *-rs* like so:

```
./nachos -rs 1234
```

Use a wide variety of *-rs* seeds to ensure your code works properly. A badly coded assignment can break if *-rs* forces a thread to yield at an inopportune time. After completing Task 4, you can append *-rs* as usual:

```
./nachos -A X -rs 1234
```

Your code should gracefully handle any errors that crop up. Graceful handling means not letting NachOS

crash or segfault, but instead catching and managing the error so that, at the very least, NachOS exits normally. Any crashes, segfaults, etc. will result in a penalty, no matter the cause, so it is in your best interest to avoid them.

NachOS has a built in RNG function you can use called *Random()*. It acts like *rand()* from the C Standard Library, returning an integer between 0 and some large upper limit, which you will need to apply arithmetic operators to in order to get a workable value. If you don't use *-rs* in testing, or if you use the same *-rs* seed, it will always return the same sequence of values. This is expected behavior. Mix up the seeds you give to *-rs* to observe different behavior in anything that uses *Random()*.

Although NachOS is written in C++, it doesn't play very well with the C Standard Library, most notably data containers like *vector*. Although you may attempt to integrate it into NachOS, it's generally not worth the hassle and we will be unable to assist you if you run into any difficulty. It may be helpful to think of NachOS as C code with some C++ sugar on top, instead of straight C++.

Due to the complexity of NachOS in general, you may wish to set up a version control system so that you can readily revert to a previous version if you accidentally break your code. However, this is not a requirement at any stage.

Submission

Throughout your code, use comments to indicate which sections of code have been worked on by which student. For example:

> *//Begin code changes by Devin Rooney.*
> *...*
> *//End code changes by Devin Rooney.*

Inside the *nachos-3.4* folder, create a subfolder called *submission_documents*. Place your report inside this folder. In addition, put a copy of all files you created or modified for this project into this folder. Tar and gzip together the *nachos-3.4* and *gnu-decstation-ultrix* folders with the following command:

> *tar -czvf project02_CLID_nachos.tar.gz ./nachos-3.4/ ./gnu-decstation-ultrix*

Submit the resulting *tar.gz* archive on Moodle. When unpacked, it should be ready to run with no setup required.

For group submissions, make sure to include the CLID of all students involved. Only one student per group should submit. Do not submit a paper copy. Any paper copies will be shredded into confetti and dumped over the student at a random point in the semester.

Late and improper submissions will receive a maximum of 50% credit.