NachOS Project Stage 1
Appetizer: Input and Threads
Total Points: 150

Assigned: January 21, 2016
Due: February 2, 2016

Objectives

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

1. Read and understand NachOS code pertaining to thread management.
2. Translate given project specifications into an implementation using NachOS threads.
3. Integrate and test their solution to ensure robustness.

Overview

This first stage of the NachOS project will help you get familiar with the NachOS operating system and learn how to solve synchronization problems through the implementation of the following tasks:

1. Input Identification
2. Shouting Threads
3. Command Line
4. 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 input identification and validation, thread creation, and synchronization.

Before you begin, it is strongly advised that you read through the introductory NachOS material posted on Moodle. Also read and understand the files in the *threads* directory, especially *thread.cc*, *threadtest.cc*, *synch.cc*, and *system.cc*. Understanding these files will help you complete the assignment.

Task 1: Input Identification

Summary

For this task, you must fork a single thread to a function that will prompt the user for arbitrary input. You will then determine which of the following categories the input belongs in, and display output indicating such:

- Integer
- Decimal
- Negative
- Character

You are not expected to evaluate fractions, mathematical expressions, or other unusual notation.

Procedure

Have the user enter some arbitrary input and store it in a character array. Inspect each character and determine which category the input belongs to. You must consider the input as a whole; 42 is a number, 42ASDF is not. 3.14 is a decimal, -3.14 is a negative decimal, 3.AD is neither. If the input belongs to multiple categories, display each category it belongs to.

For the purposes of this task, use the following definitions:

- A **number** can be an **integer**, a **decimal**, or a **negative**.
    - An **integer** is a sequence of 1 or more digits.
    - A **decimal** is a sequence of 1 or more digits followed by a period ('.') followed by a sequence of 1 or more digits.
    - A **negative** is a dash ('-') followed by an **integer** or **decimal**.
- A **character** or **character string** is anything that is not a **number**.

You do not need to fork multiple threads for this task. However, you should still not call the task function directly. Fork a thread to the function and let the main thread die, possibly with *currentThread->Finish()*. The forked thread may call helper functions as needed.

You are expected to apply what you learn in this task to all future tasks and assignments. Consider what types of input make sense, then consider if any specific values in acceptable input types can still cause trouble, such as zero. Failure to properly handle invalid input in all future tasks and assignments will result in a penalty.

Report

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

1. Why is the ability to check input so important?
2. Other than simply providing the wrong type of input, what other ways can you think of for bad input to cause an error? Consider situations other than typing input when prompted.

Task 2: Shouting Threads

Summary

For this task, you will implement a shouting match between threads. Prompt the user for the number of threads, *T*, and the number of times each thread should shout, *S*, for a total of *T* * *S* shouts. Each thread must be identified by a unique ID number when shouting.

Procedure

After prompting the user for *T* and *S*, fork *T* threads to a shouting function. Shouting consists of displaying a random phrase as output. Whenever a thread shouts, it enters a busy waiting loop where it yields the CPU for 2-5 cycles. Each thread repeats this cycle until they have shouted the proper number of times.

A busy waiting loop consists of a loop where a thread repeatedly yields the CPU with some condition where the thread may eventually break out of the loop and continue running, like so:

```
while(booleanCondition)
        currentThread->Yield();
```

For this task, the condition is that the thread yields 2-5 times before continuing. For future projects, the condition may be waiting on some resource to become available. Note that even though the above example uses a *while* loop, any loop structure may be used. In any case, both the loop and the yield statement are crucial; without the loop, the thread doesn't stay in place, and without the yield, other threads never get a chance to use the CPU. Other code may be present in the loop in addition to the yield if the situation calls for it.

The shouts themselves can be hard coded and stored however you wish. However, there should be a minimum of 5 distinct shouts, and the shouts themselves must be chosen at random by the threads. Threads should pick a shout whenever it's their time to shout, instead of shouting the same thing every time.

You should not under any circumstances use the main thread as a shouter, nor should you call the shouting function directly. Instead, fork the appropriate number of threads to the shouting function and let the main thread die with *currentThread->Finish()*. Forked threads may call helper functions as needed.

You are expected to use what you learned from Task 1 to validate user input for this task. Improper input should result in an error message and a prompt to try again. NachOS should not crash due to bad input. If you are having difficulty converting the analyzed input to a useable integer format, you may find it helpful to research the *atoi()* function.

Report

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

1. Remove the busy waiting loop used whenever a thread shouts and run the task with 5 shouters and 5 shouts per shouter. Then have each thread yield once after shouting and run another test with the same parameters. Note your results and explain your observations. Undo any changes made to accommodate this question before submitting your assignment.
2. Temporarily disable your input validation, run a minimum of 5 tests with garbage input, and note the results. How would an end user react to this? Undo any changes made to accommodate this question before submitting your assignment.

Task 3: Command Line

Summary

Implement a new command line option, *-A*, in order to select which task to run. The command syntax and valid arguments are as follows:

Command Line	***Task***
./nachos -A 1	Task 1: Input Identification
./nachos -A 2	Task 2: Shouting Threads

Any other value should result in an error message.

Procedure

In *system.cc* you will find the implementation of existing command line options such as *-rs*. Use one of these options as a template for your own *-A* option. You will need to use a global variable in order for *threadtest.cc* to see anything you set in *system.cc*.

The *-A* option should not conflict with the *-rs* option. Commands such as the following should still work:

```
./nachos -A 1 -rs 1234
```

You need to validate the input provided with *-A*. If the input is invalid, or if *-A* is not used at all, display an error message and do not run any of the tasks. It is acceptable to terminate the main thread or otherwise leave it to die and let NachOS quit on its own.

This is a higher standard than what NachOS uses for its own command line options. However, you are not responsible for retroactively applying it to these already existing options.

Report

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

1. What other solutions can you think of to improper input on the command line?

Task 4: 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 did you learn from working on this assignment?

3. 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 3, all of your code will be in *threadtest.cc*. Task 3 will be implemented in *system.cc*.

If you are using a freshly downloaded copy of NachOS, type *make all* while in the *code* directory once and only once. For all subsequent compilations, you should compile with *make* or *gmake* from the appropriate subdirectory. For this assignment, compile from the *threads* directory. Continuing to compile from the *code* directory can lead to errors at both compile time and run time.

To run and test your code, type *./nachos* from the *threads* directory after compiling. Once you complete Task 3, select the appropriate task with *./nachos -A X* where *X* is 1 or 2.

Remember that you are expected to validate the input provided for Task 2. Consider what types of input make sense, and reject all input not of that type. In addition, you must consider the edge cases of acceptable input types, and what implications those values have for the algorithms as written.

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.

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.

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 point in the course.

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 Matt Wallace
...
//End code changes by Matt Wallace
```

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 project01_CLID1_CLID2_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 the file name includes 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.