

## Homework 1

*Reading: Tanenbaum, Chapter 1*

*Due: Sep. 11, 2008*

The homework is not typical of other CSE 422S homeworks in that it is almost all programming. The problems are primarily my reaction to the results from Assessment 1 and are intended to give you a chance to review concepts that have become encrusted from inactivity and get everyone up to some minimum speed. A few preliminary comments:

- *Purpose:* The purpose of homework assignments is to provide focus and structure to the practice of learning. You should be asking yourself why you are working on a problem and how you can improve on the process of working on a problem ... even after you have submitted your solution.
- *0-Point Problems:* 0-point problems are usually (but not always) warmup problems. Often, they provide background for the non-zero point problems. Sometimes, they are shorter, simpler problems and often resemble exam questions. But this homework is atypical, and you will not get any of the 0-point questions on an exam. The solution to 0-point problems is posted at the time the assignment itself is posted. My advice to you is that if a 0-point problem seems trivial, you should still try to sketch out the solution before you look at the solution. But even if you don't, you should read the solution some time before the due date.
- *Discussion With ...:* I encourage discussion of homework problems and their solution with your classmates, TAs, and me. The intent of the discussion should be to get a better understanding of the material or to clarify the assignment. But **if you just copy someone else's solution**, I will treat that as cheating.
- *FAQs:* Each assignment has its own FAQ. Check the FAQ before asking a question. **Q:** What is the source of FAQ content? **A:** You and your classmates. I post my replies to relevant email in the FAQ after some editing.

Now for a commentary about each problem.

- *Problem 1:* Many students were confused by Problems 4-6 on Assessment 1. I have provided the source code for the solution to these problems. Compile and run them. Experiment with variations if you wish.
- *Problem 2:* If you haven't had much experience with a command-line interface and Unix, you should plan to spend some time learning some basic Unix commands and utilities.
- *Problem 3:* The Standard Template Library `map` class covered in CSE 422S is useful for representing a symbol table; i.e., a string-string map or `char*-char*` map. This concept is not critical now, but this problem will allow students who have never used `map` to start developing their skill with it. Homework 3 will give you another chance to practice with `map`. Since this problem is unrelated to Problem 4, you may want to look at it after doing Problem 4.
- *Problem 4:* This problem will give you practice in dealing with multiple representations in C/C++ and force you to think about what is really happening when you try to store data into a buffer (an area of memory).

**Problem 1** (0 Points)

Read the solution to Assessment 1 and write a C/C++ program that demonstrates Problems 4, 5 and 6 on Assessment 1.

**Problem 2** (0 Points)

If you are new to the Unix environment, do reading suggested on the course Web page. You will find the reading by following the **Introduction to Linux** link under **Online Notes and Tutorials**.

**Problem 3** (0 Points)

Write a small C++ test program that will accept plain text lines each consisting of two words and either update or show the *value* of a symbol. A word is any sequence of symbols separated by one or more white spaces (space or tab character). The symbol table is defined as a map from one symbol to second symbol which is considered the value of the symbol:

```
struct ltstr {
    bool operator()(const char *s1, const char *s2) const {
        return strcmp(s1, s2) < 0;
    }
};
...
map<const char *, char *, ltstr> value;
```

There are two kinds of lines. If the first word is "show", the second word is considered to be a symbol for which you want to display the symbol and its value. Otherwise, the first word is the name and the second is its new value.

In the following example, the meaning is shown to the right:

```
x      123y      # set the value of "x" to "123y"
show x          # should output "x 123y"
894 xxx        # set the value of "894" to "xxx"
x      abc      # set the value of "x" to "abc"
show 894       # should output "894 xxx"
```

**Problem 4** (6 Points)

Write a C/C++ program that will eventually evolve into a special command-line interpreter that reads from `stdin` and writes to `stdout`. For now, the program will just do some simple lexical analysis and outputting. In the description below, a command-line is a sequence of characters separated by one or more consecutive white-space characters. A white-space character is either a space or a tab character. Each non-white-space character sequence is called a *word*. Here are three examples:

```
set x 3-12
spawn myprog hello 8 57
wait 367
send 367 hello mom
```

The commands have 3, 5, 2 and 4 words respectively. Note that 3-12 is one word since there are no intervening white space characters.

- The program has the following synopsis:

**hw1-4**

In this assignment, we are only interested in the syntax of the language and the semantics of the language is left undefined.

- Your program should do the following for each command line:
  - Display on **stdout** the command line exactly as it appears on input.
  - If the first word of the command is **spawn**, **wait** or **send**, form a buffer with the format described below. Suppose that there are  $n$  words in the command; i.e.,  $W_1, W_2, \dots, W_n$ . The buffer starts with an *unsigned long* command number that is 1, 2 or 3 depending on whether the first word ( $W_1$ ) is **spawn**, **wait** or **send** respectively. This number is immediately followed by  $n$  pointers and then the  $n - 1$  words  $W_2, \dots, W_n$  in C-string format (with the NUL character terminating each string). The  $n$  pointers point to each word in the buffer except the last one is the NULL pointer indicating the end of the pointer array.
  - Then if a buffer is formed, display the contents of the buffer. *Note that this means display what IS in the buffer, NOT what you think is in the buffer; i.e., the values should come from the buffer.* In this format, the command number is displayed in human readable form in a 3-character field; pointers are displayed in each character is displayed in hexadecimal (each output field is separated by a space character. For example, the command **'spawn foo 32'** might be displayed as follows if the buffer begins at address **0xbfffe7ec**:
 

```
1 0xbfffe7fc 0xbfffe800 (nil) 66 6f 6f 0 33 32 0
```
- Your program should exit when either the first word is **quit** or an *end-of-file* condition is encountered on **stdin**.

Here is a detailed description of the language:

- Each command is separated by a newline character (**'\n'**).
- All commands have a variable number of arguments which are separated from each other by white-space. However, you can assume that a command never has more than five words nor will overflow a 100-byte buffer.

Submit:

- Source listing
- Test output (the preferred method is to redirect **stdin** and **stdout**, but you can use some other method such as editing the output from the **script** command if you wish)
- A short explanation of why the test output indicates that your program is operating correctly.

The following man pages might be useful: **strcmp(3)**, **memset(3)**, **isdigit(3)**, **atoi(3)**, **strtol(3)**, **strtok(3)**, **printf(3)**, **exit(3)**, **memcpy(3)**, **fgets(3)**, **strlen(3)** and **ascii(7)**. The digit in parentheses indicates the probable section where you can find the man page; i.e., **'man 3 memset'** looks for **memset** in section 3. Note that in Solaris, you will need the **-s** flag; i.e., **'man -s 3 memset'**.