Homework Assignment #5
System Development in the UNIX Environment (605.614)

0. *Your* assignment

You might not know this, but I am actually teaching a second section of this class at the same time as I am teaching this one. This class has completed a 'homework5', and, since I am pretty busy, I am asking you to grade this assignment for me. You should have a pretty good idea from the first four assignments on what I expect.

You should check for:

- Correctness of the solution (the programs need to work!) Be sure to test all cases.
- If the program fails, you should be able to find out why the program failed, so you can deduct the proper amount of points (a one-line fix should merit a smaller deduction than missing functionality or a totally confused implementation).
- Efficiency of the solution (as specified in the assignment)
- Implementation - are the programs well organized?
- Source code (e.g. how well the programs adhere to the style guide)

How many points should you deduct for each error or deficiency you find? Some guidance is provided in the assignment itself. Overall, make your best judgment, and you can certainly ask for further direction from me - but please don't ask me to find the problems in the code! You are the grader, not me.

The good news is that there are only two students in this class. You will find their assignments in the class notes area under the STUDENTS directory. (~jcn/unix_class/STUDENTS)

I am not requiring these students to pack up their homework using the normally obligatory Perl or Python scripts (these students are not quite as capable as you). Don't worry about permissions. You can just copy the homework5 into your directory and (hopefully) build and grade it. If you can't get access to the homework's, let me know and I'll make sure you get access. Thanks for your grading help. :-)
The specification for the homework you are grading follows.

When you are completed, you should submit them in BlackBoard. This can either be done as an attachment in BlackBoard or in the text segment in the assignment area. Be sure to submit the grades to me before the due date. We will be discussing the assignment during class, so they must be in at that point. The format of the grades should take the format of the grade reports that I have provided you over the semester.

1. Overview

The fifth and final homework assignment for this class is to create a set of programs that interact with a network server and place information from the server into a shared memory area. Completion of the assignment involves submitting to the instructor, on or before the due date, the following materials:

1. Printed listings of the Makefiles, programs' and libraries' source code are **not** necessary.
2. A comment block at the beginning of each program and library source file shall include the following information: the student, the course, the name of the file, and a short description of the file's purpose.
3. The homework should be compiled and run on the SunOS (UNIX) system (*dev4*) at the Computer Center. The executable programs must be installed in the structure mandated in the last section of this document. Execution on the Linux system (*absaroka*) is **not** required.

The fundamental rules governing the homework are:

1. The programs must be written in the 'C' or the 'C++' programming language.
2. No copying or plagiarism. If copying or plagiarism occurs, the person or persons involved will receive no credit. They will be treated as not having submitted the assignment.
3. Assignments are due by midnight of the due date stated above.
4. Extensions are only granted if arrangements are made with the instructor well in advance of the due date of the assignment. No last minute extensions will be given; if a student has not completed the assignment by the due date, the student has the choice of receiving a grade based on the work completed to that point, or receiving a 5% penalty for each day (or part of a day) it takes to complete the assignment. In these cases, the assignment is not considered complete until the instructor is notified by the student of the completion of the assignment.

2. Permissions

To allow your homework to be graded, you must allow group access to your source and executable files. To do this, first assure that your home directory has group read access:

chmod g+rx ~

Next, assure that your homework5 directory has both read, write and appropriate access permissions:

find homework5 -exec chmod g+rw {} \; -type d -exec chmod g+x {} \;

There is an executable file in the class notes directory called FIX_HOMEWORK5, which you can execute to accomplish this. *Failure to do this by the due date of the assignment will result in an automatic 5 point deduction.*

3. Grading of the Homework

The homework will be graded based on 100 points according to the following criteria:

1. **10** points for project directory structure and Makefiles. (A release script is **not** required.)
2. **30** points for the correctness of the libraries.
3. **60** points for the **control_probe** program itself. Of this **60** points, **40** points will be based on the program organization, style, and general correctness; **15** points will directly reflect the percentage of the cavern which was mapped, and **5** points will reflect the move/space ratio (how many moves it takes to explore the cavern).

   The grader wrote a successful probe program (that is, it always maps out the entire cavern) in less than 400 lines of code. This program has an average move/space ratio of 4.684.

4. Homework Requirements

4.1. Overview

You have just arrived at the site of a mining disaster. Several people have been trapped in a deep underground cavern. You must program a mining probe to explore the cavern and find all the people.

Your predecessor, Sub-commander T'Pol, attempted to write the program but failed. The pressure proved too much for her; she went stark raving mad, and is now venturing where no person has gone before. Now it's up to you.

The probe inserts itself somewhere in the cavern. (Note that you do not know the absolute position of the probe in the cavern.) Following your instructions, it explores every crevice. The probe is controlled by two simple instructions: MSG_INSERT and MSG_MOVE. In response to MSG_INSERT, the probe burrows into the ground and digs down until finding the cavern. The MSG_MOVE command is one of either MOVE_NORTH, MOVE_SOUTH, MOVE_EAST, or MOVE_WEST. The response to this command is a MSG_MV_RESPONSE message which will be YES if the move was successful, NO if it was not. The probe automatically picks up people when it finds them.

Once the probe has finished its exploration, you send a MSG_EXPLODE message. This causes a concussion bomb to be exploded so we can sonically map the cavern. If you've explored every crevice and found every trapped person, you'll receive a pat on the back and a hearty handshake. But if your program botches the job and the concussion bomb goes off with unrescued people remaining in the cavern...well, let's just hope you know a good attorney.

The executable must be named **control_probe**. The **control_probe** program will communicate with the probe via a beamed Ethernet. Your program must establish a stream connection with the probe over a socket.

How big are these caverns anyway? They are no larger than 24 spaces north and south, and 80 spaces east and west. This is exactly the size of a terminal screen. What a coincidence. Fortunately, T'Pol finished a display program before her unfortunate illness (she cleverly called it **display**). This display program assumes you have put a picture of the cavern into a shared memory segment. Whatever characters you put into shared memory (using your **control_probe** program) will be displayed on the screen; in this fashion you can get a picture of the cavern.

4.2. Communicating with the probe

Communication with the probe is performed via a stream socket. The machine address of the probe is called *localhost*, and it listens for connection initiation on port 10000. Once a connection is made to the probe, communication is performed via the exchange of messages. The first message you send is 'INSERT_PROBE' to initialize the probe and insert it into the cavern. Immediately after inserting the probe, you can move in via the MOVE message. The MOVE message will include a direction (NORTH, SOUTH, EAST or WEST) to tell the probe in which direction to move. (Sorry, the probe is an early prototype, and no diagonal moves are allowed.) After sending the probe a MOVE message, the probe will respond with a MV_RESPONSE message which will tell you if the move you requested was performed. If the move was not possible, you may assume that you just told the probe to move into a solid wall of rock.

As soon as you believe that the probe has visited all areas of the cavern, guaranteeing that all of the trapped souls were found and rescued, you then destroy the probe by sending the EXPLODE command. Upon receipt of this command, the probe will send a RESULTS message, and then explode, terminating the connection. The results message will include how many areas of the cavern that your search missed, and how many trapped people were in these areas (hopefully none).

4.2.1. Probe Message Formats

Here are the message formats that the probe understands. This information is included in the file .../probe/include/messages.h.

```
typedef enum {
        MSG_INSERT_PROBE   = 0, /* from control_probe to the probe */
        MSG_MOVE           = 1, /* from control_probe to the probe */
        MSG_MV_RESPONSE    = 2, /* from the probe to control_probe */
        MSG_EXPLODE        = 3, /* from control_probe to the probe */
        MSG_RESULTS        = 4 /* from the probe to control_probe */
} COMMANDS;

typedef enum {
                MOVE_NORTH      = 0,
                MOVE_SOUTH      = 1,
                MOVE_EAST       = 2,
                MOVE_WEST       = 3
} DIRECTIONS;

typedef enum {
                YES     = 1,
                NO      = 0
```

```
} MOVE_RESPONSES;

typedef struct _msg_insert {
        int     cavern;
} PROBE_INSERT_MSGS;

typedef struct _msg_move {
        DIRECTIONS      move;
} MOVE_PROBE_MSGS;

typedef struct _msg_mv_response {
        MOVE_RESPONSES success;
} MOVE_RESPONSE_MSGS;

typedef struct _msg_results {
        int cavern_size;
        int amt_mapped;
        int lawsuits;
        int moves;
} RESULT_MSGS;

typedef struct msg_generic {
        COMMANDS        msg_type;
        union {
                RESULT_MSGS             results;

                MOVE_RESPONSE_MSGS      mv_response;
                MOVE_PROBE_MSGS         move;
                PROBE_INSERT_MSGS       insert;
              } msg;
} PROBE_MESSAGES;
```

Note that all messages are contained in PROBE_MSGS via a union. Thus all messages are the same size.

4.3. control_probe

The **control_probe** program will be invoked as follows:

control_probe -m <shared memory key> -P <port number> -c <cavern id>

The -P option must be followed by the port number; this is the port number used to connect to the server; that is the probe server will be listening on this port. This option is not required; if omitted, the default port number is 10000. (There is an option provided to the probe server to change its port; see below.)

The -m option is followed by a number that represents the shared memory key that is used by the cavern display program. This argument is required.

The -c option represents a cavern id. This number should be placed in the **cavern** field of the PROBE_INSERT_MSGS structure, which is sent to the probe server. Every time you set the cavern field to say, 5, you will be inserted into the number 5 cavern in the same place. This way you can test your program over and over again on the same cavern.

Using a cavern of 0 puts you into random caverns of many different sizes and shapes, and is the ultimate test of your program. Until you're sure of yourself, use a fixed cavern. Be sure to test your program on several caverns. It is possible to entirely map cavern 5 while failing miserably on cavern 7 (and vice versa). This option is used for testing; the cavern associated with a given cavern id will always be the same. This argument is optional.
The **control_probe** program generally will perform the following steps:

- Process the arguments
- Attach to the probe server
- Attach to the display shared memory
- Initialize the probe
- Move the probe, mapping out the entire cavern. For each space of the cavern you examine, the display shared memory will be updated as described below.
- When you are finished mapping the cavern, explode the probe.
- Read the RESULT_MSGS and log the results of your attempt to your logfile.
- Clean up and exit. Clean up must include closing the socket and deleting the shared memory segment.

All significant events (connecting to the server, attaching to the shared memory, etc.) as well as any detected errors, should be logged to a log file. You must use the log_mgr library you completed in homework 2, along with any subsequent fixes, to log these messages. You must also use the *setup_client( )* function provided by the instructor to connect to the server, and the *connect_shm( )* function you defined in the shared memory library you developed in homework 4 to attach to the shared memory segment.

One aspect of the behavior of the probe that you must be aware of: after 5000 moves or 30 minutes, the probe runs out of energy and dies. You will receive a MSG_RESULTS message upon its death. Therefore, you must be able to handle the receipt of a MSG_RESULTS messages in such a circumstance.

4.4. probe

The **probe** program, which is provided for you, runs as a server. Perhaps it will be running when you want to use it but for various reasons it may not. If you experience any trouble with **probe**, the executable is in the /probe/bin directory under the class notes. If it is not running (which you can check with a "ps ax | grep probe" command), just execute it with "probe &". If it is frozen, you won't be able to use the regular port. Fortunately, **probe** lets you specify a port with the **-P** option. So you could execute, say, "probe -P 5000 &". Since your program also accepts the **-P** option you can execute it to match ports with **probe**.

4.5. display

The **display** program is also provided for your use. It resides in probe/bin. This program is invoked as follows:

display <shm_key>

Where *shm_key* represents the key to the shared memory segment. This number should

match the number given following the -M option of **control_probe**. This program simply inspects the contents of this shared memory segment every second, and updates a screen oriented display based on the contents of this segment. Whenever the character ')' is detected in any cell in the shared memory area, the display program cleans up and exits.
Both source and executable are available; the source for the display program resides in: probe/src/display in the class notes.

Although you are not required to modify the source, you must include the display source in your homework5 directory tree; appropriate Makefiles must exist to build and install your display program.

4.5.1. Format of the shared memory
The shared memory is simply defined to be a two dimensional array of characters. The number of rows in this array are twice the number of rows that a cavern can extend over (which would be 2*24, or 48). The number of columns would be twice the number of columns that a cavern can extend over (which would be 2*80, or 160). The **control_probe** program must be careful not to exceed these bounds. The shared memory segment is of this size for the convenience of the **control_probe** program, since the **control_probe** program does not know in absolute terms the initial position of the probe. Whatever characters are entered in the shared memory segment are displayed on the screen.

4.6. Some Examples

There is a sample **control_probe** program in the .../probe/bin directory. You can run it to see how a reasonable program performs. Perhaps you can even beat it (that is, search the entire cavern in less moves).

Here are some scenarios you may encounter as you develop the **control_probe** program:

EXAMPLE 1: Your program cannot establish a streams connection with the probe server. A "ps ax | grep probe" command reveals the probe server is indeed running, so either it is frozen or your program has a bug. You issue a ".../probe/bin/probe -P 5400; control_probe -P 5400" command to try to execute it on a different port. If it still doesn't connect, you probably have a bug.

EXAMPLE 2: Your program cannot establish a streams connection with the probe server. A "ps ax | grep probe" command reveals that probe is NOT running, so you execute:
".../probe/bin/probe&" to start it.

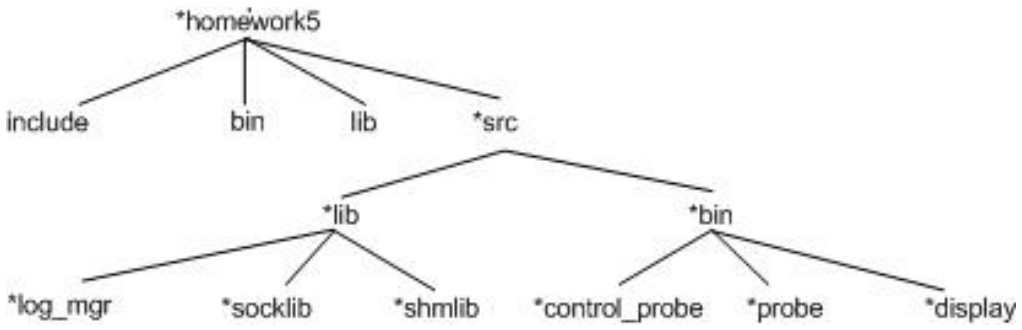
EXAMPLE 3: Your program establishes a streams connection with the probe server. It sends the MSG_INSERT_PROBE message, then a MSG_MOVE message with MOVE_NORTH in the move field. The MSG_MV_RESPONSE message contains YES, so the move was successful. Overconfident, you send a MSG_EXPLODE message. The

MSG_RESULTS message informs you, that the cavern_size was 900; you explored 2 spaces in it (the one you started on and the one you moved to); in 1 move; with about 9 lawsuits. The goal of the probe controller is for amt_mapped to be equal to the cavern_size and 0 lawsuits.

EXAMPLE 4: Your program establishes a streams connection with the probe server. It sends the MSG_INSERT_PROBE message, then a series of MSG_MOVE messages. The program thinks it has finished and sends a MSG_EXPLODE message. The MSG_RESULTS message informs you that the cavern_size was 756; you explored 428 spaces in it; in 2526 moves; with about 3 lawsuits. Fortunately, you set the cavern field in the MSG_INSERT_PROBE message to 55 instead of 0, so you can make changes to your code and explore the exact same cavern again and again until you get it right. You could even write a program, which takes input from the keyboard and allows you to map cavern 55 manually.

4.7. Homework Implementation

The design and implementation of the homework shall follow these guidelines:
- The homework shall consist of the two programs and three libraries organized as described in this document. There shall be a minimum of three header (include) files that shall reside in $(HOME)/homework5/include (one header file for each of the three libraries).
- The following directory structure shall be enforced:
- The directory hierarchy under this directory shall look as follows:



The directory "homework5/lib" will contain your project-specific libraries (libshmlib.a, libsocketlib.a and liblog_mgr.a).
The directory "homework5/bin" will contain your project-specific programs (control_probe and display).
The directory "homework5/src" will contain the source code for the project, organized as shown.
The source for each program will be in the deepest directories of the above hierarchy. Each of these directories will contain a Makefile; the function of the Makefile will allow

the UNIX command “make" to make the executable (or the library) and install it in the appropriate installation directory (either homework5/bin or homework5/lib).
In addition, each directory marked with an (*) above should contain a Makefile; these Makefiles should allow 'make' to be run in these directories, the effect being to run make with the specified argument in each of the directories underneath.
All Makefiles should be called ``Makefile'' and should contain support for the following targets at a minimum:

- it
- install
- depend
- clean

The directory ``homework5/include'' will contain project-specific include files (include files that need to be shared among distinct programs).