Lab 5: Programs and Pipes
CMSC 342
Due: 4/17/2014

This lab will follow two threads which you can work on in any order (or alternate between). The first thread introduces you to some fancy and cool features of vim. The second thread involves doing some fancy programming with pipes and the fd_sets.

Thread 1: Using Vim

Many schools teach their students to use an integrated development environment (IDE) for writing programs. IDEs are (usually graphical) programs that have lots of development features, such as code-completion, graphical debugging, tracing, and so forth. Eclipse (written in Java and used primarily for Java programming) is one of these programs. The downside to IDEs is that they are slow to load which means that you have to follow a software development work-flow where you have many windows open all the time. That can make it cognitively difficult to remember which window has which part of your code.

An alternative work-flow is to use a text editor, like Vim, to compile your code. Because vim is quick to launch, you can open your program, make a change, then save, quit, and recompile, and then open your program back up very quickly.

However, vim can be a very inefficient way to write programs if you don't know how to use its (very powerful) features and commands. In this thread of the lab, I will introduce you to a whole bunch of powerful commands and then ask you to apply them to a document. I will give you a set of tasks to perform and ask you to write down the sequence of vim keystrokes youwould use to do them in an “answers” file.

Step 0. Getting Started

Create a folder named “Lab5”. In that folder, create a file named “ANSWERS.TXT” Be sure the name of the file is in all-caps. Write your name and “student” login at the top of the file. Then number the next 8 lines 1 through 8 like this:

```
1.

2.

3.

4.

5.

6.

7.

8.
```

Each of these lines corresponds to one task you will carry out in vim.

Background

Vim has several modes:

Insert mode allows you to enter text
Replace mode allows you to overwrite existing text (but backspacing restores the old text)
Visual mode allows you to select a block of text
Command mode allows you to type in commands like :wq
Normal mode is the default mode and allows you to type in key combinations.

We are going to learn some things you an do in normal mode with those key combinations.

Every normal mode command consists of the following parts (some of which are optional):

a count or range, a register, a command, a motion

The count is a number and indicates how many times the command should be repeated. For example, the “dd” command deletes a single line. The “10dd” command deletes 10 lines.

Some commands can take a register – a temporary place to store the result. (For example, when you copy a line, you can put it in a register instead of the clipboard – and when you paste you can paste from that register instead of the clipboard). Registers start with a double quote (i.e. "a)

The motion identifies which text the command should be applied to. Here are some of the most important motions:

k – up j – down h – left	l – right	
w – next beginning of a word	b – back to beginning of a word	
e – next end of a word	ge – back to end of a word	
{ – beginning of paragraph	} – end of paragraph	
(– beginning of sentence	) – end of sentence	
$ – end of a line	0 – beginning of a line	^ – first character of a line
G – end of file	gg – beginning of file	

:number – jump to line number “number”

% – matching curly brace, quote mark, or parenthesis to the one under the cursor
/term – search for “term” in the document and position the cursor on it
n – repeat the previous search forwards (n stands for “next”)
N – repeat the previous search backwards

You can also type a colon followed by a line number to jump to that line number.

So you could type 5w to move 5 words ahead. You can type 5dw to delete the next five words of the document. Of course, if your cursor is in the middle of a word, it would only delete part of the word you're on (the part to the left of the cursor would be left alone). If you want to delete ALL of the word, you can type 5daw instead. (The a stands for “all”).

If you repeat a command twice it is applied to the entire line. So for example, “dd” deletes a line, “yy” yanks (copies) it and “>>” indents it.

Here are a few useful commands you can combine with these counts and motions:

>: indent to the right	<: unindent (to the left)	
y: copy (yank)	d: delete	c: change (deletes and goes to insert mode)
p: put after	P: put before	
u: undo	CTRL-R: redo	
r: replace		

The change command and replace command are a little different than the others. The change command deletes the current text and puts you in insert mode so you can type in a new value. This saves you a keystroke. Instead of typing dw, then "i" to replace a word, you can just type "cw".

The replace command replaces the symbol under the cursor with whatever character you type next. so if you type "ra" vim will replace the current character with an "a". The advantage of this is that you never leave normal mode, which saves a few keystrokes if you only need to change that one letter.

Switching Modes

There are actually several different keystrokes that can put you into insert mode. The "i" keystroke puts you into insert mode and leaves the cursors where it is. You can save yourself some time in certain circumstances by using the "a" keystroke instead. The "a" keystroke "appends". That is, instead of inserting text in front of the character under the cursor, it adds text following that character. If you type SHIFT-A instead, vim appends to the end of the current line. It jumps the cursor to the end of the line and puts you in insert mode.

The "o" keystroke opens a new line on the one below the cursor, positions the cursor at the beginning of that line, and puts you in insert mode. This is often much faster than typing "i", going to the end of the line, and then hitting enter.

Visual Mode

Visual mode allows you to select a block of text and then perform a command on it. Vim supports several different visual modes, but the most important one is visual-line mode. To enter visual line mode, type SHIFT-V. Then use motions or the arrow keys to select a block of text. You can then delete the entire block by pressing "d", yank the entire block by pressing "y", or indent by pressing ">".

Pressing 'v' (without shift) puts you in "visual character" mode in which you can highlight part of a line.

Step 1. Creating a document

Save the following text into a document named "story.txt":

Once in a wolgrum's life, he discovers his true nature. It is then, and only then, that he is allowed to take the pesh – the great journey – beyond the borders of the stahk and into the wide world. The world is a dangerous place, filled with evil grimelings and terrible clawrocks. A wolgrum is by nature a very careful creature, but even so, many foul and hideous things can happen on the pesh.

Jurgle curled his tail around a stout wooden staff, squared his shoulders, and shuffled off along the path toward the city of Wom. He felt very alone and very scared. But he was also determined -- determined to complete his year of wandering and return from his pesh a master of the arts. As everything he had ever known grew slowly smaller behind him, he wondered if he would ever see Lelitha again.

Step 2. The eight-fold tasks

Okay, so that wasn't very well written prose. However, we can use it to pratice a little vim-fu. Here are eight tasks for you complete. For each one, try to find the shortest sequence of keystrokes that will allow you to perform the task. Then write that sequence in your ANSWERS.TXT file.

Each task should be considered independent of the others. (That is, you should undo each one before continuing to the next task).

You should assume that the cursor starts in the upper left corner of the document.
Task 1: Replace every instance of the word "pesh" with the word "mettletest".

Task 2: Indent the entire second paragraph one level.

Task 3. Delete the entire last sentence, including the two spaces before the word "As".

Task 4. Remove the apostrophe from the word "wolgrum's"

Task 5. Add a quote mark to the beginning of the document and a quote mark to the very end of the document.

Task 6. Copy the second line of the file to the bottom of the file.

Task 7. Change the order of the words "evil" and "grimelings".

Task 8. Change the first letter of the word "Wom" to a lowercase "w".

Thread 2: Using Pipes

Unix provides several different ways for processes to communicate with each other. We already looked at how related processes can use signals to communicate.   The problem with this is that signals are unreliable – if two signals get delivered at the same time, we can lose one. Also, signals can only transmit a small amount of information – they tell us that an "event" has occurred, but that's all.

Another way for processes to communicate is to use IPC (inter-process communication) functions. Of these, the easiest to use are "pipes". Pipes allow related processes (such as a parent and its child), to send information in first-in first-out order (FIFO).

Whenever you have multiple inputs or outputs, you need a way to select which one to use for communication. Unix provides a "select" function that lets you wait for input on several file descriptors and then read from the first one that has data available. This is called I/O multiplexing.

In this lab, you will use pipes, multiplexing, and shared memory to implement a multi-player game.

Step 0. Getting Started

Download the file "Lab5.tar.gz" from my web site and extract it inside your Lab 5 folder:

```
wget http://robert.marmorstein.org/Spring2014/cs242/Lab5.tar.gz
tar xzvf Lab5.tar.gz
```

This will give you the source code for a complete, working game involving ghosts and treasure. The game is split up into three main parts. The files shmlib.h and shmlib.c form a small library which automates some of the more tedious tasks of creating a block of shared memory.

The file client.c uses the shared library to implement a computer player for the game.

The file server.c uses the shared library to implement the game engine. The game engine draws a 20x20 grid to the screen. The grid consists of empty spaces (represented by dots), a treasure (represented by a $), the player (represented by a + sign) and some number of “ghosts” (computer players) represented as question marks.

The game engine uses shared memory and pipes to communicate with the computer players. Whenever the computer player makes a move, the move is written to a pipe. The game engine also handles input from the user on the standard input. It uses I/O multiplexing to respond to either of these inputs. When the player presses a key or data is received on a pipe, the game engine responds by moving the appropriate character. Each time the player moves on top of a treasure, their score increases by one and a new treasure is generated. If the player comes in contact with one of the ghosts, the game is over and the engine ends the game, signals the computer player processes to quit, and prints out a “Game Over” message.

Your job in this lab is to extend this game in various ways.

Before you begin, you should edit ALL of the files so that the comments at the top reflect the fact that you are modifying the source code. Simply add your name and e-mail address to the Copyright notice immediately after my name and e-mail (you can add the word “and” if you like).

Step 1. Improving the Client

Right now the client isn't very smart. It moves about randomly unless it is RIGHT next to the player. Add some code that searches through the shared memory map (the “grid”) for the plus sign. Then determine whether the ghost is above the player, below the player, to the left of the player, or to the right of the player. Then make the ghost move in an appropriate direction TOWARD the player.

Step 2. Giving the player a chance

With smarter ghosts, it will be MUCH harder for the player to survive for very long. Modify the server so that at the beginning of the game, it generates six “pitfalls”, represented on the map by lowercase “o” characters. If a ghost moves onto the pitfall, it should be teleported to the farthest corner of the map. Make sure the pitfall doesn't get deleted when this happens.

Step 3. Giving the ghosts a clue

Right now, communication on the pipes is totally one-way. The ghosts can send commands to the server, but the server can't send any kind of reply. The teleportation of a ghost will totally ruin your player-hunting algorithm, because there is no way for the server to inform the “client” process that the ghost isn't where it used to be.

Add a new set of “output” pipes that allow the server to send data to the client. Every time the server receives a command from a client on an input pipe, it should write the new x and y coordinates back on the output pipe for that client. Then modify your client so that it reads the x and y coordinates every time it sends a command and updates its internal state variables (x and y) appropriately.

Step 4. Greedy Ghosts

Add code that uses shared memory to keep track of the TOTAL number of times any ghost either:

1. Stomps on the treasure
2. Encounters another ghost on the same square.

You will need to allocate space for a single (unsigned long) integer. Have the server initialize it to zero. Each client should keep track of whether or not it's projected move takes it to a square already occupied by the treasure or another ghost. If so, it should increment the shared variable. For the purposes of this lab, you may assume that "increment" is atomic (although this probably isn't true – really, we should probably use semaphores, but we'll save that for later).

Modify the server to print out the total (and then deallocate the memory) at the end of the game.

Glitter

Make a copy of the inner Lab 5 folder and name it "Glitter":

```
cd ..
cp -r Lab5 Glitter
cd Glitter
```

Then extend the project in a creative and interesting way. You will need to turn in BOTH folders (Lab5 and Glitter) as well as your ANSWERS.TXT file.

You can create a tarball containing them all like this:

```
cd ../..
tar czvf Lab5.tar.gz Lab5
```

Submitting

As usual, create a tarball of your entire Lab5 folder:

```
cd ..
tar czvf Lab5.tar.gz Lab5/
```

Then turn your work in through the course web site:

http://marmorstein.org/~robert/submit/