CIS 452 Lab 1
Introduction to UNIX and System Development Tools

Overview

The purpose of this assignment is to re-familiarize yourself with UNIX commands and system calls, with the syntax and use of C library functions, and with some of the programming utilities and development tools that will be useful during the course of the semester. This lab should largely review of CIS 241 (Systems-Level Programming). If you have not been exposed to this material, then it will serve as a brief introduction and tutorial.

This lab tutorial concentrates on tools and utilities available on *any* Linux-based system - including simple terminals and telnet/SSH sessions. Many of the programs and utilities mentioned have graphical equivalents available on our systems. Consult any Linux guide for more information on using our current distro (Arch Linux), or use the on-line help system, or just ask the person sitting next to you.

Note that other operating systems (e.g., Windows, Mac) will have similar tools available.

Submit:

- A word document containing the answer to the numbered questions. (The questions should also be included either in bold or italicized).
- Any requested screenshots embedded in the word document.
- Corrected valgrind source code (no zip files)

Online Help

The first resource to familiarize yourself with on any system is the Help pages. The most convenient (but certainly not the easiest to understand) source of information on a UNIX system is the online documentation, or manual pages. With sufficient practice, man pages actually become readable. The following table lists relevant portions of the man pages; in this course we will primarily reference chapters 2 and 3.

Chapter	Contents
1	Shell commands

2	System calls
3	Library functions
8	System administration

The following commands are a starting point for productively using the man pages :

- **`man man`** - describes how to use the man command
- **`man num intro`** - displays information from the introduction to Chapter `num`
- **`man num topic`** - displays information about `topic,` which is found in Chapter `num`
- **`man -k keyword`** - lists commands which have `keyword` in their synopsis

For example, if you want to know what information is found in Chapter 1 of the man pages, you would type "**`man 1 intro`**". If you want to find all entries that deal with PDF files, type "**`man -k pdf`**".

Unix man pages are also available online in various formats. I frequently find myself using google to search 'man <topic>' where <topic> is the man page I'm looking for. Some examples of online man pages:

- https://man7.org/linux/man-pages/
- https://linux.die.net/man/

Answer the following questions:

1. Briefly describe the purpose of (what is found in) any two sections commonly found on a man page (e.g. the synopsis section of the `malloc()` system call)
2. Describe the difference between the UNIX shell command **write** and the UNIX system call `write()`

Include Files

A frequent point-of-reference when programming is the include file associated with a particular system call. It defines function prototypes, specifies the makeup of data structures, defines macros, and includes other useful information and documentation. The most useful directories for this course will be "**`/usr/include`**", "**`/usr/include/sys`**" and "**`/usr/include/bits`**". These directories of course will vary from system to system.

When compiling a program you can use the '-v' option to help determine where the include files are for your system.

Answer the following question:

3. What is the meaning of the stream-based `SEEK_SET` macro?

    - Start by using a keyword search on the man pages to find possible "seek" function calls applying to a stream
    - Use the man pages to discover which include files are referenced by those function calls
    - Scan the include files to discover the meaning of the macro

Useful UNIX Fundamentals

You should be fluent with all file handling commands and familiar with the various options these commands provide. In particular, you should know how to:

- `cp` - copy a file
- `mv` - move or rename a file or directory
- `rm` - remove (delete) a file
- `mkdir` - make a new directory
- `ls` - list the contents of a directory
- `cd` - move to another directory
- `rmdir` - remove (delete) a directory
- `pwd` - print the name of the current working directory
- `chmod` - change access permissions on files and directories

Also familiarize yourself with these miscellaneous commands:

- `script` - record a terminal session
- `telnet/ssh` - Telephone Network protocol / login program
- `cat` - print a file to the display
- `more/less` - print a file to the display, a screenful at a time
- `lpr` - send a file to the printer
- `grep` - pattern match a string in files

If you are unfamiliar with any of these commands, consult an introductory UNIX text or user guide (e.g. "*A Practical Guide to Linux Commands, Editors and Shell Programming*", 4th Ed., by Mark Sobell and Matthew Helmke). Alternatively, you can utilize the help system as described above.

Note: the desktop interface has a Windows-like file manager that graphically facilitates many of the above file tasks.

Answer the following questions:

4. What is the command to list the contents of a directory in long mode, including hidden files?
5. What is the command syntax to make a directory readable/writable/executable only by you?

Program Development Process

The typical program development cycle consists of Edit/Compile/Run/Debug sessions that repeat until the program functions correctly.

Edit - there are several built-in editors available on the system. For example, **vim** (Vi IMproved) is an improved version of the venerable **vi** editor **-** arcane but available on every UNIX system. Likewise for those who prefer a superior editor, **emacs** is a great solution. The user interface also includes several simple, graphical text editors (e.g. **Kate**, **Pluma, gedit**).

Compile - use either the GNU C compiler (**`gcc`**) or the GNU C++ compiler (**`g++`**). Check out the man pages for useful command line options. *Always* compile with full warnings (i.e. **`-Wall`**), this may save you on troubleshooting your application. You may also need to link in specific libraries if you use functions included in them (e.g. use **`-lm`** to link in the math library). One of the biggest differences between C and C++ or Java is the C Standard I/O library, so you should also do a man on **`printf()`** for output and **`scanf()`** for input to understand C-style I/O (most sample programs presented in the lab will be written in C). Note: several "MS Visual Studio"-like development environments are also available (e.g. **Eclipse** with **CDT**). Visual Studio Code has become a popular, easy to use development environment also.

Run - unless specified otherwise by the "**`-o`**" (output) switch, a successful compilation will produce a file called **`a.out`**, which is your executable. Simply type the program name to execute the program. Depending on how your path is configured, you may need to path-qualify the name, as in "**`./a.out`**". I always suggest naming your output to something useful, typically the program name without the extension. For example, if my program was named 'mySimpleProgram.c', the output should be 'mySimpleProgram'.

Debug - the EOS systems have the GNU debugger, **`gdb`**. Debuggers can be very useful, not only to determine the source of problems in a program but also to understand program execution better. Remember that you need to use the **`-g`** command line option when compiling in order to enable debugging. Note: the **Kdbg** debugger is a graphical version of

the gdb debugger. Visual Studio Code also has a very easy to use debugger but isn't readily available on all systems.

You can use the following shortcuts in **`gdb`** or type **`help`** to find out how to:

`b` - set a breakpoint (your program stops at the specified program line or function name)
`n` - step to the next line in a program (steps *over* function calls)
`s` - step to the next line in a program (steps *into* function calls)
`p` - display (print) the value of a variable
`watch` - display (watch) the value of a variable whenever it changes
`bt` - display (backtrace) the current program execution stack
`finish` - finish the current function. (Useful if you have accidentally used `s` where you meant `n`)

Program Debugging

When we're trying to figure out exactly what's causing a problem (like a bug) in a program, we often use something called a debugger. Debuggers are helpful not only for finding bugs but also for checking the values of variables as the program runs.

Some computer tools like Integrated Development Environments (IDEs) and specific editors, such as Visual Studio Code, have very good debuggers built into them. These are usually the best choice because they're practical and have lots of features. However, these fancy tools might not be available on every computer.

On the other hand, if you have a UNIX system, you automatically have access to a debugger called gdb. This part of the guide will show you a quick exercise to help you get the hang of using it.

sampleProgramOne

```
#include <stdio.h>
#include <math.h>

int main()
{
   double num = 0.0;
   printf ("Hello, world.\n");
   num = pow(2, 28);
   printf ("You are the %f person to write this program!\n", num);
   return 0; // The normal exit of a program has a return value of 0
}
```

6. Perform the following operations:

- Create the above sample program (via copy/paste)
- Compile the program (remember to include debugging information and link all necessary libraries)
- Run the sample program
- Start the debugger on your program (**`gdb sampleProgramOne`**)
- Set a breakpoint at the function **`main`**
- Take a screenshot (screenshotOne)
- Run your program within the debugger (**`run`**) and step through it
- Use the debugger to print the value of **`num`** before and after it changes
- Take a screenshot (screenshotTwo)

Dynamic Memory Debugging

Static program analysis refers to the ability to detect errors (and other characteristics of a program) at compile-time. In computer science, the term dynamic always refers to execution, or "run-time" analysis. This section introduces a tool for performing dynamic memory debugging, in order to examine a program's use of memory while it is executing. The idea is to catch a simple but common memory error (i.e., a memory leak) before releasing your code. This is an important component of professional code development.

Finding Memory Leaks

The tool is called **valgrind**, and with practice it can be a powerful aid in detecting memory problems. Consider the following sample code:

sampleProgramTwo

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 16

int main()
{
    char *data1, *data2;
    int i;

    do {
       data1 = malloc (SIZE);
       printf ("Please input your eos username: ");
       scanf ("%s", data1);
       if (!strcmp (data1, "quit"))
```

```
            break;
         data2 = malloc (SIZE);
         for (i=0; i<SIZE; i++)
            data2[i] = data1[i];
         free (data1);
         printf ("data2 :%s:\n", data2);
      } while (1);
      return 0;
}
```

Study the code until you are certain of what it does (you should compile and execute it). Although it appears to run correctly, there is a memory leak within the loop that will eventually cause a problem if the loop is executed a sufficient number of times. (Note: there is also another instance of "unrecovered" memory; it is subtle and a less serious problem).

A memory-checker can help detect these types of problems. Read the man pages for the **valgrind** utility program. One of its advantages is that it simply monitors memory location accesses, hence it does not require any instrumenting of your code and is relatively easy to use:

- At the command prompt, type: **`valgrind --leak-check=full sampleProgramTwo`**

It takes some practice, and some good guessing, to understand the errors being identified in the output from valgrind. Note that problem details are given in inverse order because of the function call stack. Remember to compile your program for debugging (i.e. use the **`-g`** command-line option when compiling); **valgrind** can be more informative when given more information about your program (symbol table, etc.). Once you know how to use it, **valgrind** is a nice tool with a number of different applications; you can find more information at http://www.valgrind.org.

Run the program, executing the loop several times before you exit. Use the line numbers provided by valgrind and your understanding of the code and the pattern of memory losses to determine the sources of error.

7. Describe precisely (nature of problem, location) the memory leak(s) in sample program2. Fix the problem(s) and submit your corrected source code. Note: it would be a good idea to use **valgrind** to verify that you have fixed the problem(s)!

System Profiling

In addition to memory debugging, there are similar tools designed to profile the flow of execution through a program and report its use of system/library calls (providing useful

debugging info). The utility program used in this part of the lab is called **`strace`**; please read the man pages to discover its operation and use.

Run the **strace** utility on the executable of the *original* sampleProgramTwo.

Now read the man pages again to discover how to use command-line options to get a less cluttered output from `strace/ltrace`. Look for the option that reports a *summary* for each call. Then use the output to answer the following questions:

8. How many times is the `write()` system call invoked when running sample program 2? Experiment with executing the loop a different number of times. Then express your answer as a formula.

9. Examine the source code and output to answer the question: what is the primary 'C' library subroutine that causes the `write()` system call to be invoked while executing sample program 2?