Introduction to UNIX

EE 361

Sasaki

August 2020

Contents

1 Introduction

1.1 Computer Programs

A computer is a digital circuit that will run programs that are stored in its memory. A computer can execute simple instructions called *machine instructions*. Machine instructions are simple operations such as adding two numbers, storing a number into a memory location, and having the computer jump from one instruction to another. Programs that the computer can run are made up of machine instructions. These programs are sometimes called *executable or machine programs*. Different computer processors may have different machine instruction sets. For example, the Intel Pentium and the ARM Cortex processors have their own instruction sets. An executable program for the Intel Pentium will not run on the ARM Cortex. Some processors do run on the same instruction set such as the Intel Pentium and Celeron, and the AMD Athlon.

The programs you write in the C language are not executable, i.e., they cannot be executed by the computer. They have to be *compiled* to be converted to an executable program. The programs you write in C are called *source programs* or source code. Tthe compiler will convert the source code into an executable program for a particular processor with a particular instruction set. Sometimes this processor is called the *target machine.* For example, a C program can be compiled to a machine program for the Intel Pentium and can also be compiled (with another compiler) to another machine program for the ARM Cortex. Though the two machine programs are different, they will accomplish the same task, as specified by the C program. Note that the C program is portable since it can be used on different machines.

1.2 Operating Systems

Since computers can only execute simple instructions, it's difficult to have them do complicated tasks. Programs that you use on your laptop or desktop have tens or hundreds of millions of machine instructions. To simplify developing and running software, computer architects try to separate computer operations, making them modular. Organizing a large system into modules makes it easier to understand and manage, and supports reusability of modules.

For laptops and desktops, the operating system is the main software that is run. It takes care of all the details of running a computer to make it easier for the user. For example, it manages memory and allocates space for programs and data. It interacts with the Internet, receiving and sending Internet packets over the Ethernet ports. It deals with all of the devices attached to your computer such as the keyboard, video display, thumb drives, joy sticks, etc. It also provides *system calls* which are subroutines that often access computer hardware. For example, printf("Hello world\n") is a system call in the C programming language. It accesses your computer screen to print the sentence "Hello world". The user doesn't know the details of the hardware interface to the video screen of the particular computer. In this way, the user has a reliable interface to use a computer screen on any computer. This is also a reliable interface for a designer of an operating system. The designer knows that if he/she can write a printf subroutine to the computer's screen, any source program using this system call will work. Subroutines such as printf are often called *application programming interfaces* (API).

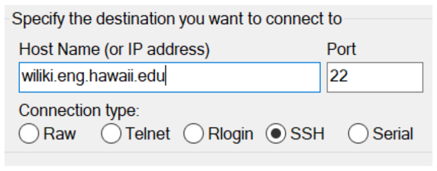
Windows, MacOS, and UNIX are popular operating systems. In this course, we will use the UNIX operating system. Operating systems that are related to UNIX are Linux, which is popular in computer servers, and Android, which is used in smart phones.

Finally, note that some computers have no need for an operating system. Usually they have small programs designed for very specific tasks. Examples are a microcontroller used to regulate fuel flow in a car or a microcontroller that does signal processing is for very specific tasks.

1.3 Remote terminals

To remotely access a Linux server, such as wiliki, you can use a remote terminal software such as PuTTY.

Here's the login:



Specify the destination you want to connect to

Host Name (or IP address): wiliki.eng.hawaii.edu

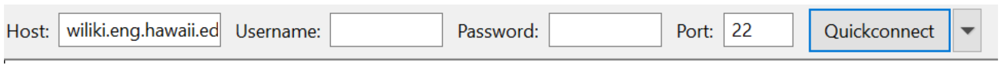
Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

To download a free copy, go here: http://www.putty.org/

Note that you can open more than one window into your Linux account. This will allow you to have several windows while you develop and debug your code, which can be very helpful.

Here's the login:



Host: wiliki.eng.hawaii.ed Username: Password: Port: 22

If the network connection fails, you may need a Virtual Private Network (VPN) access. Here are the instructions to set that up: https://www.hawaii.edu/askus/819

You can transfer files from your laptop (or desktop) using FileZilla

To download a free copy, go here:

https://filezilla-project.org/

The parameters are the same as Putty, where the port number is 22.

2 Introduction to UNIX

The UNIX operating system is made up of three parts: the kernel, the shell and the programs.

The Kernel

It is the hub of the operating system:

- Allocates time and memory to programs
- Handles the filestore and communications in response to system calls

The Shell

- The shell acts as an interface between the user and the kernel.
- When a user logs in, the login program checks the username and password, and then starts another program called the shell.
- The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out.
- The commands are themselves programs: when they terminate, the shell gives the user another prompt.
- The adept user can customise his/her own shell, and users can use different shells on the same machine.

As an illustration of the way that the shell and the kernel work together, suppose a user types **rm myfile** (which has the effect of removing the file **myfile**). The shell searches the filestore for the file containing the program **rm**, and then requests the kernel, through system calls, to execute the program **rm** on myfile. When the process **rm myfile** has finished running, the shell then returns the UNIX prompt % to the user, indicating that it is waiting for further commands.

Files and Processes

Everything in UNIX is either a file or process:

- A process is an executable program identified by a unique PID (process identifier).
- A file is a collection of data, e.g., a document, program, a directory (which is explained below). Here, we should distinguish the difference between the program as a file and the program as a process. The program as a file is one that is stored away but not run. It is usually stored in the hard disk or thumb drive. The program stored as a process is a running program. It is usually stored in RAM.

Files are organized into *directories*. Directories are similar to folders in the Windows operating system. Directories can contain files and other directories. In UNIX, the main directory is the "root", and all other directories are subdirectories of it. Note that each user in a UNIX system has an account. This account has files in a directory created for the user. For example in the wiliki computer system, if you type "pwd" in your account, you will get the directories your account is in. It will look something like this:

```
/users10/ee/fac/sasaki
```

In this case, the operating system has its root directory divided into subdirectories, of which one is "users10". This subdirectory is for users of this computer system. The "users10" directory is divided into subdirectories, of which one is “ee”. This is for users in the EE department. Then the “ee” directory has a “fac” directory for faculty members, and my directory is “sasaki”. The sequence /users10/ee/fac/sasaki is called a “path” since it shows how to get from the root to my directory.

2.1 Some Unix Commands

2.1.1 Managing directories and files

Command	**Description**	**Example of use**
ls	List files and directories	Just type ls. There are other options which you can invoke by typing ls -<the option>. An example option is ls -a which lists “all” files and directories.
mkdir	Make a directory.	Type: mkdir <name of new directory>
rmdir	Remove directory	Type: rmdir <name of directory>
cd	Change directory	This command allows you to navigate through the directories. If you are in a directory then you can move to a subdirectory by typing cd <name of subdirectory> If you want to move up in directory, type cd .. If you want to move to your account directory then simply type cd
pwd	Display the path to the current directory	Type: pwd
more	Displays a file a page at a time	Type: more <filename>
cat	Concatenate.	This concatenates files together. The concatenated files are output somewhere, and by default to the standard output. Recall that the standard output is your computer screen. You can use this to display a file by just typing cat <filename> This concatenates one file (so no concatenation at all) and outputs to the standard output which is your computer screen.

Command	**Description**	**Example of use**
cp	Copies files	If you type cp <file1> <file2>

		It will copy the file1 to file2. Note that file1 should be a file that already exists, and file2 is one you created. You can also create files in a directory. For example, cp <file1> <subdir1/file2> will create file2 in the subdirectory subdir1. You can also type cp <file1> <subdir1/> will create file1 in subdirectory subdir1. Note that you can also type cp <path1/file1> <path2/> or cp <path1/file1> <path/file2> In the first case, the new file is named <file1> and in the second case, the new file is named <file2>
rm	Remove file. Delete a file. No trash bin so once it's deleted that's it. Be careful.	Type rm <file>
chmod	Change mode of file or directory. Files and directories can have modes such as read only.	chmod <mode value> <file or directory> A file or directory's mode is specified by 3 digits. Each digit represents 3 bits so the digit's value ranges from 0 through 7. Each bit corresponds to permission to read, write, or execute the file. Each digit correspond to permissions at different levels: user, group, and others. Note that transferring files and directories can sometimes change the permissions. For example if you transfer a program or data file from your laptop to wiliki or spectra, the permissions may change to no reading or writing or execution. Then you have to change the mode.

2.1.1.1 Text editors

You can create a file by using a text editor such as vi or pico. Pico is an easy editor to learn, but vi is more efficient (faster). You are encouraged to learn vi. Here are references to vi tutorials. If these are not enough then take the time to search the web (with google or other search engine) for better tutorials.

VI tutorials:

- https://engineering.purdue.edu/ECN/Support/KB/Docs/ViTextEditorTutorial
- http://www.eng.hawaii.edu/Tutor/vi.html
- http://www-ee.eng.hawaii.edu/~sasaki/EE260/Labs/labVi.html
- http://heather.cs.ucdavis.edu/~matloff/UnixAndC/Editors/ViIntro.html

PICO tutorial:

- http://www.ncsu.edu/it/essentials/managing_files/text_editors/pico_tutor/

2.1.1.2 File access and chmod

Each file and directory has access limitations, some which can be controlled by the user. There are three classes of access: user, group, and other. User just means your access, group is the group access, and other means everyone outside of you and the group. You can check which groups you are in by typing “groups”.

The access (for user, group, or other) is represented by a 3 bit number (basically an octal digit), each bit corresponds to read, write, and execute permissions. A value “1” means it’s permitted, and “0” means it’s not. Thus, 111 means you can read, write, and execute the file, while 000 means you cannot do anything with this file. The three bits are read, write, and execute. Thus, 110 (or the octal digit 6) means you can read and write but not execute. Permissions for a file can be represented by three octal digits, where the first digit is the user, the second digit is the group, and the third digit is other. Thus, 660 means read and write permissions for the user and group, but no permissions for other.

Here’s more on file system permissions and the octal notation:

- http://en.wikipedia.org/wiki/File_system_permissions
- http://en.wikipedia.org/wiki/File_system_permissions#Octal_notation

An example of changing the mode of a file or directory is

```
chmod  660  <file or directory>
```

To check the permissions type

```
ls -l
```

or more specifically

```
ls -l  <file or directory>
```

Alternatively, you can chmod using another notation, e.g., to change the user mode to allow reading a file, type

```
chmod u+r  <file>
```

The symbol “u” is for user, “+” means to add, and “r” is read permission. Here is a more complete list of the symbols:

- Classes: “u” is user, “g” is group, “o” is other, and “a” is all (everyone, and it’s the same as “ugo”
- Permissions: “r” is read, “w” is write, and “x” is execute
- Actions: “+” is add and “-“ is subtract (or delete)

The format for chmod is

```
chmod  <classes><action><permissions>   <file or directory>
```

Example:

```
chmod   ug+rw  myfile
```

will grant read and write permissions to the user and group to myfile.

```
chmod a-x  myfile
```

will remove permission to user, group, and others to execute myfile.

Here’s a reference to chmod

- http://en.wikipedia.org/wiki/Chmod

2.1.2 Input/output

Command	**Description**	**Example of use**
>	Direct standard output to a file	<command> > <file> For example, cat <file1> <file2> > <file3> will concatenate <file1> and <file2> and the result is stored in a new created file <file3>. Note that “command” can be a program that you have written. Instead of having your program display on the computer screen, you can direct the output to a file.
>>	Append standard output to a file	command >> <file> For example, cat <file1> <fil32> >> <file3> will concatenate <file1> and <file2> and the result is appended to the existing file <file3>.
<	Redirect standard input from a file	command < <file>
|	command1 | command2	Pipe (or send) the output of command1 to the input of command2.

2.1.3 Informational

Command	**Description**	**Example of use**
who	Lists the users currently logged in	who
man	Online manual of Unix commands	man <command>
whatis	Brief description of a command	whatis <command> Available on wiliki but not on spectra at the time of this writing
apropos	Searches for man pages	apropos <word> Example: apropos search to find man pages for commands that may have something to do with searching. Available in wiliki but currently not available in spectra
whoami		Let’s you know who you are
finger	Finds information about an account	finger george@wiliki.eng.hawaii.edu will give you information about George at wiliki

2.1.4 Processing

This is about launching programs to be executed and managing these processes. When a program is launched, it has an ID number. Also, programs can be run in the "foreground" or "background". A program run in the foreground is the one you interact with via the computer screen and keyboard. Only one program can be in the foreground. When you log out, these programs are terminated even if they did not complete their task.

Programs run in the background are running but you cannot interact with them directly. However, they can continue running after you log out. This is good for number crunching programs. You can also run multiple programs in background. Since these programs do not interact with you directly, their inputs and outputs often come from files.

Command	**Description**	**Example of use**
./<executable>	To run your program, you must prefix with "./"	Type ./a.out
jobs	Lists current jobs	
ps	Lists current processes	Similar to jobs. It has options. For example, ps -a lists all processes. Note that each process has an ID number. You can use the ID number to kill jobs -- as shown below.
^c	Kill the job running in the foreground	Type ^c
^z	Suspend the job running in foreground	Type ^z
bg	Background the suspended job	Type bg This will have the job continue running in background.
fg	Bring a background job into the foreground	Type fg %1 to foreground job number 1
&	Puts a job or process in the background	Type command & or ./a.out &
kill	Kills a job or process	kill %1 to kill job 1 kill 26152 to kill process 26152
Ls -lag	Lists access rights for all jobs	

2.1.5 Searching, File Processing, and File Transfer

Command	**Description**	**Example of use**
grep	Search a file or files for keywords	grep hand test.txt will search the file test.txt for all occurrences of the word “hand” even if it is part of another word, e.g., “handle”.
wc	Counts the number of lines, words, or characters in a file	
find	Finds files anywhere on the system	find . -name “<filename>” will display all files and directories with the name <filename> find . -size +1M -ls will find files over 1Mb in size and display the result in a list.
pwd	This shows the path of the current directory	pwd
tar	Combines files, e.g., in a directory into a single file.	This is useful to store a directory away. You first transform the directory into a single file and then you can transport it as a file. It’s a good idea to compress it before transporting it to save space and communication time. tar cvf <directory.tar> <directory> will tar <directory> into a single file and call it <directory.tar> tar xvf <directory.tar> will untar the file <directory.tar>. There are other options that are useful so explore.
gzip gunzip	Compression algorithms. Compressed files usually have the suffix .gz	To compress a file type gzip <file> Note that the file can be a tar’d directory. To uncompress a file type gunzip <file>
ssh	Secure shell	This is the up to date method for a secure connection to a computer. It has two applications, one to transfer files and the other is a remote terminal.

2.1.6 Environment and C Shell

The next set of commands is useful for setting up the environment. UNIX runs a *shell* called C shell which which provides a user interface. For example, it interprets commands that are typed in by the user. C shell can interpret instructions called *scripts*, which has a language of its own. The scripting language is similar to C, which is why the shell is called C shell. C shell is also referred to as “csh”. The up to date version is “tcsh”.

Here are some commands that are useful for the shell environment

Command	**Description**	**Example of use**
alias	It allows you to type one string instead of another.	alias remove rm -r This means that instead of typing “rm -r”, you just type “remove”, e.g., type “remove file” rather than “rm -r file”. Its also useful for replacing control keys. For example, if control-h is used for deletion you can use an alias to replace control-h with Delete.
echo	Echoes whatever you type after it	If you type echo hello world The computer screen will immediately display hello world
set	Sets shell variables	Shell variables apply only to the current instance of the shell and are used to set short-term working conditions By convention, shell variables have LOWER CASE names. Examples of shell variables are “home”. “history” and “path”
setenv	Sets environment variables	Environment variables have a far-reaching significance, and those set at login are valid for the duration of the session. By convention, environment variables have UPPER CASE names. Examples are “TERM”, “EDITOR” and “PATH”.
date	Displays the current date and time	date
cal	Shows a calendar of the current month	cal 10 2011 shows October2011 cal 2011 shows the calendar for the whole year
*	Wild card, e.g., *.txt means all files that end with “.txt”, while	More about this below
.	Current directory	./a.out means the file a.out in the current directory
~	Path from root to user’s home directory	More about this below

The wild card is useful as a short cut. For example, suppose you want to copy all files with the suffix “.txt” to a directory Directory. Then you can type

```
cp  *.txt   Directory/
```

The operating system will look for all files where "*" can be anything but ".txt" has to be a perfect match. Another example is if you want to copy all files with the suffix "ditto*", then the operating system will look for all files that starts with "ditto" but the rest of the file name can be anything.

The "~" can be used as part of a path. It means a path from the root to the user's home directory. If you type

```
ls  ~/ee367
```

you will list all members of the "ee367" directory in the user's account.

2.2 Environments and shells

Here we explain how you can customize your working environment in UNIX.

2.2.1 Variables

There are two types of variables, environment and shell variables.

Environment variables

These are variables available while you are logged in. Some environment variables in UNIX

- PATH: This is a single variable (character string) that lists all the paths the operating searches to find an executable program. The paths in PATH are separated by colons.
- HOME: Indicates the user's home directory
- TERM: Indicates the terminal being emulated

A "$" prefix implies the value of the variables. You can see the contents of the variable by using echo, e.g.,

```
echo $PATH
```

You can set environment variables by the setenv command, e.g.,

```
setenv TERM xterm
```

Note UNIX environment variables are in upper case.

Shell variables

C shell has its own set of variables. You can create them by using the set command, e.g.,

```
set  var1=10
```

This creates a variable var1 and sets it to the value 10. There are also built-in shell variables, e.g.,

- argv: Special variable used in shell scripts to hold the value of arguments

- autologout: Contains the number of minutes the shell can be idle before it automatically logs out
- history: Sets how many lines of history (previous commands entered by the user) to remember
- path: Contains a list of directories to be searched when running programs or shell scripts
- prompt: Sets the prompt string
- term: Contains the current terminal type
- home: Contains path to home directory

Variables can be used by typing the dollar sign before the variable name, e.g.,

```
echo   $history
```

will display the value of "history".

You can set built-in variable values using "set", e.g.,

```
set history=50
```

will set the built-in history variable to 50, so that the last 50 commands will be remembered.

2.2.2 Shell scripts

Shell scripts are a list of commands and executable programs. A shell script can be executed by the shell. The shell then becomes an *interpreter* of the script. An interpreter will read the script line by line and execute the lines as they are read. It's as if the user is typing in the shell script lines into the computer for the operating system to interpret.

You can create a shell script using a text editor. The first line must begin with the string

```
#!/bin/csh
```

This tells the operating system which shell you are executing, in this case the C shell.

Note that "#" is the symbol for "comment to the end of the line". This is true except for the first line.

The rest of the script is comments (prefixed by "#"), commands, and executable programs.

If you would like to include more paths into your "path" variable, then do the following

```
set path=($path /usr/bin /usr/sbin)
set path=($home/xpg/bin $path)
```

The first line will include /usr/bin and /usr/sbin into the set of paths, while the second line will include "$home/xpg/bin" into the set of paths.

The wrong way to include paths is "set path=(/usr/sbin)" because this will erase the existing paths and replace it with /usr/sbin. The previous paths (that were written over) may have been important but now they're lost.

After you've written the script document, make sure it can be run by changing its mode

```
chmod  u+x  <filename>
```

Then you can run the script by typing

```
./filename
```

as if you are launching an executable file.

Another detail is that the line #!/bin/csh will cause an instance of the C shell to get started. When the script is completed then the C shell stops. For example, suppose you had the shell script

```
#!/bin/csh
alias displaypath  echo Hello world
displaypath
```

When you run the script, you will get displayed

```
Hello world
```

Now after the script is run, the alias isn't valid anymore because the shell running the script is terminated. Thus, if you type

```
displaypath
```

You won't get the display.

2.2.3 .login and .cshrc files

All of the C shell commands are good while you are logged in. After you log out, you need to redo them again. To get around this, use the following two files. They are similar to shell scripts. Note that both have a period as a prefix in their names. Due to this naming, these files won't appear when you try to list their directory.

- .login - This is the login file. It has commands that will be run by UNIX whenever you login. So you can write all your aliases, settings of variables, etc and basically all commands that initialize your environment in there. An example of a .login file is shown below

- .cshrc - This file is run whenever the C shell is started, so for our situation, whenever we login. It is useful for the following reasons. When you type in a command, the operating system will check where the executable program is located. By default, it only checks the current directory, /bin and /usr/bin; so if your command is in another directory, it'll give you an error message -- command not found. If you want it to check other directories, you must set up paths to these directories in the .cshrc file. An example .cshrc file is shown below.

Both files are created (or copied from another file) by you, and you can edit them using a text editor.

Example .login file:

```
# The hash symbol means a comment to the end of the line
resize
setenv   TERM   vt100
echo  Welcome to wiliki
date
```

Example .cshrc file

```
# Sample .cshrc file
setenv EXINIT 'set smd sw=4 wm=2'
set history=50
set savehist=50
set ignoreeof noclobber
alias f finger -R
alias lo logout
# The “-i” option for the commands “rm”, “cp”, etc, means that the
# operating system will query you about whether you really want to
# remove the file.  This reduces the chance of you accidentally
# removing (or copying over) a file.  The following alias will
# automatically put you into this mode when you try to remove a file.
alias rm rm -i
```

Other examples:

- http://unlser1.unl.csi.cuny.edu/tutorials/viunix/subsubsection3_3_6_1.html
- http://physics.wm.edu/unix_intro/loginmht/cshrc.html
- http://agecon.ucdavis.edu/department/computer_services/unix/login/thelogincshrfiles.php

2.3 Other useful commands

Command	**Description**	**Example of use**
du	Disk usage	du <file or directory name> Shows how much space is used for the file or directory
quota -v	Disk quota	quota -v
passwd	Change password	passwd

3 Compiling

To compile a C program you can use *cc* or *gcc* compilers. We'll use gcc.

To compile a program in a file program.c, just type

```
gcc program.c
```

and an executable program *a.out* will be created. To run an executable, just type

```
./a.out
```

You can rename a.out by using the "mv" command, e.g., mv a.out   myprogram. Alternatively, you can compile by typing

```
gcc   program.c   >   myprogram
```

Larger programs that are in multiple files require more complicated compiling steps.

3.1 Organizing a C program

3.3.1 Header files and prototypes

Large C programs are divided into smaller files, which are usually of two types: source code files which have source code, and header files which have source code that can be used by program files. Program files usually have the suffix ".c", and header files usually have the suffix ".h". The code in a header file can be included in source code with the line

```
#include "<header.h>"
```

Header files are useful for definitions, such as data types and constants, and *prototypes* of functions. Before discussing what a prototype is, we will review the compiling process. The compiler will scan the source code file line by line. Generally, it doesn't like to find a function call before the function is defined. For example, the following has a function defined after the function call.

```
main( )
{

test( );  // function call to function test( ).  It is before the function is defined.

}

void test( )
{
printf("This is a test\n");
}
```

The compiler doesn't like to be surprised like this. One way to get around this is to define the function before the function call, but this can be impractical for large programs. Another approach is to use prototypes as follows.

```
void test( );  // prototype for function test( )

main( )
{

test( );  // function call to function test( ).  It is before the function is defined.

}

void test( )
{
printf("This is a test\n");
}
```

The prototype gives a "heads up" to the compiler that there is a function test( ) defined somewhere. The prototype has all the information the compiler needs to implement the call, such as defining the input and output parameters and their data types. Note that the prototype of the function is the same as the first line of the function but it also includes a semicolon. Prototypes are also used to let the compiler know of functions in other program files.

You can create libraries of functions and list prototypes in a header file. Other source code can use these functions by including the header file.

3.3.2 Object files and linking

Compiling multiple source code files into a single executable is a little more complicated. Basically, you compile each source code file, creating *object files*. Then you *link* the object files together to create the executable. An object file has the machine program for the source code, and it also has additional information used for linking.

To create an object file "file.o", type from the source code "file.c", type

```
gcc  -c  file.c
```

To create an executable from object files file1.o, file2.o, and file3.o, type

```
gcc   file1.o, file2.o, file3.o   -o   project
```

where "project" is the executable.

3.2 Makefile

The *make* command simplifies compiling. When you type "make" it will read a file named "makefile", which has a list of tasks which are typically compiling and linking commands. The format of a task is

```
target:  source(s)
(tab)command
```

The target is the result (or output file) of the task, e.g., an object or executable file. Note that the target is followed by a colon.

The source(s) are the input files to the task. If the task is to compile, the source(s) are all the source code files and header files. If the task is to link, the source(s) are all the object files to be linked together.

The command is usually either a compile or link command. Note that the command must be preceded by a tab.

The list of tasks in the makefile is ordered in the reverse so the last task is listed first and the first task is listed last -- weird, I know. Typically, all the compiling to object files is done at the bottom, and the linking of the object files is done at the top. Here's an example:

```
#  This is an example comment line.
project:  program1.o program2.o
	gcc  program1.o  program2.o   -o  project
program2.o:  program2.c   useful.h
	gcc  -c  program2.c
program1.o:  program1.c   useful.h  anotherheader.h
	gcc -c program1.c
```

In this example, program1.c includes two header files: "useful.h" and "anotherheader.h". The program program2.c requires just one header file "useful.h".

Note that in any directory there is just one file called "makefile".

The "make" command is useful because it only compiles what is necessary. Before running any task, it checks the sources and targets to check for any changes. If these files have not changed then it doesn't redo the task.

Sometimes these tasks can have lengthy lines. For example, lists of files can be quite long. You can use the "\" symbol to break a line so you can continue on the next line.

4 References

There are many resources for UNIX on the Internet and you should search and explore them using search engines such as google Here are some examples:

- http://www2.ocean.washington.edu/unix.tutorial.html
- http://www.hawaii.edu/itsdocs/cen/unxbasic.pdf
- http://www.hawaii.edu/itsdocs/cen/unxbasic.pdf
- http://www.ee.surrey.ac.uk/Teaching/Unix/
- http://acm.unl.edu/resources/tutorial.php