1 OPERATING SYSTEMS LABORATORY I - UNIX Tutorial - Additional

In this text, UNIX will be used controversial with Linux. This tutorial is intended to make an introduction to the operating system concepts which will be discussed in lecture hours.
• Understanding commands and processes

- when you enter a command it invokes a program. While this program is running it is called a *process*. It is important to grasp that although there is only one copy of a program held in the file system, any number of processes can be invoked which run this program.
- when the operating system is started after a boot, a single process is started. This process is the parent of all subsequent processes. Each process created on the system has a unique number, known as its PID, associated with it.
- when you login to the system a process is started to run your shell program. Any processes that are started from within your shell - such as entering a command - are the children of this process. A process can have many children, but only one parent.
- Every computer has an operating system.The UNIX operating system has three important features;
  - kernel, (1.0.1)
  - shell, (1.0.2)
  - filesystem (1.0.3).

1.0.1 The Kernel

- As its name implies, the kernel is at the core of each UNIX system.
- Basically, the kernel is a large program that is loaded in whenever the system is started up - referred to as a boot of the system, and it controls the allocation of hardware resources from that point forward.
- The kernel knows what hardware resources are available (like the processor(s), the on-board memory, the disk drives, network interfaces, etc.), and it has the necessary programs to talk to all the devices connected to it. It manages the entire resources of the system, presenting them to you and every other user as a coherent system.

- You do not need to know anything about the kernel in order to use a UNIX system. This information is provided for your information only.
- **Location of the kernel file**; the root directory contains both the boot program and the file containing the kernel for the system. The name of this kernel file varies from one manufacturer's machine to another. You can search for it as;
  $ls -l /boot/vmlinuz*
  -rw-r–r– 1 root root 2359087 Sep 22 00:20 vmlinuz-2.4.26

  In this example; Linux kernel is given, but a similar name can be found for the other variants of UNIX.

1.0.2 The Shell

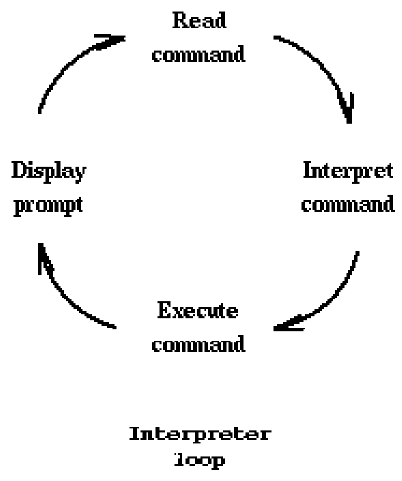

Figure 1: The shell

- Whenever you login to a Unix system you are placed in a program called the shell. The shell is perhaps the most important program on the UNIX system, from the end-user's standpoint.
- The shell is your interface with the UNIX system, the middleman between you and the kernel. The shell acts as a command interpreter; it takes each command and passes it to the operating system kernel to be acted upon. It then displays the results of this operation on your screen.

- The shell is a program that the UNIX kernel runs for you. Many basic shell commands are actually subroutines built in to the shell program. The commands that are not built in to the shell require the kernel to start another process to run them.
  - **Features provided by the shell;** create an environment that meets your needs, write shell scripts, define command aliases, manipulate the command history, automatically complete the command line, edit the command line, etc.
  - **Special characters in UNIX**; UNIX recognizes certain special characters as command directives. If you use one of the UNIX special characters in a command, make sure you understand what it does. The special characters are:
    / < > ! $ % ^ & * | { } ~ and ;
    When creating files and directories on UNIX, it is safest to only use the characters A-Z, a-z, 0-9, and the period, dash, and underscore characters.
  - **Getting help on UNIX;** to access the on-line manuals, use the **man** command, followed by the name of the command you need help with. **EXAMPLE:** Type
    - **$man ls** ; to see the manual page for the "ls" command.
    - **$man man**; to get help on using the manual.
- **Description of different types of shell;** there are several different shells available for Unix. You can use any one of these shells if they are available on your system. And you can switch between the different shells;
  - Bourne shell (sh),
  - C shell (csh),
  - TC shell (tcsh),
  - Korn shell (ksh),
  - Bourne Again SHell (bash).

  Changing your shell; you can switch to another shell for the remainder of your login session. To do this enter the shell command name at the system prompt. For example:
  $csh This switches you from your current shell to the C shell.

- **The environment variables** are passed to all programs that are not built in to the shell, and may be consulted, or modified, by the program. By convention, environment variables are given in upper case letters. To view all the environment variables, use the command ;
  $printenv
  You can also view a particular environment variable using the echo command:
  $echo $TERM
  The above command echos the value of the TERM environment variable to the standard output. Important environment variables

  - **TERM;** environment variable defines the type of terminal that you are using. Most UNIX systems have a database of terminal types, and the capabilities of each terminal type.
  - **PATH;** variable contains the names of directories to be searched for programs that correspond to command names. When you issue a command to the shell, the shell searches sequentially through each directory in the PATH list until it finds an executable program with the command name you typed,
  - **USER;** variable contains your username. Any time you access a file or directory, the access permissions are checked against the value of USER,
  - **HOME;** variable contains the name of your home directory. When you issue the cd command with no directory argument, you will be placed in the directory defined in the HOME environment variable. The HOME variable is also where the shell will look for the user login scripts,
  - **EDITOR;** variable is used by programs that must invoke a text editor to provide the ability to edit or compose documents,
  - **HOST;** environment variable contains the name of the host machine that is running your shell program. When you connect to a remote host through telnet or ftp, the name of your host is relayed to the remote machine, so the administrators of the remote machine can keep track of who is connecting, and from where.

- **Setting environment and shell variables;** the exact mechanism for setting the environment and shell variables depends upon the type of shell you're using;

  - **sh, or ksh**; *EDITOR=emacs;export EDITOR*,

  - **csh**; *setenv EDITOR emacs.*

- **Summary of shell facilities;** See table 1. The sign * means which is

Table 1: Summary of shell facilities

Facility	Bourne	C	TC	Korn	BASH	Explanation
command history	No	Yes	Yes	Yes	Yes	(Allows previous commands to be saved, edited, and reused)
command alias	No	Yes	Yes	Yes	Yes	(Allows the user to rename commands)
shell scripts	Yes	Yes	Yes	Yes	Yes	(Allows the user to shell programming)
filename completion	No	Yes*	Yes	Yes*	Yes	(Allows automatic completion of partially typed file name)
command line editing	No	No	Yes	Yes*	Yes	(Allows the use of an editor to modify the command line text)
job control	No	Yes	Yes	Yes	Yes	(Allows processes to be run in the background)

not the default setting for this shell.

1.0.3 UNIX file system

- All the stored information on a UNIX computer is kept in a *filesystem*.
- The place in the filesystem tree where you are located is called the *current working directory*.
- Every item in the UNIX filesystem tree is either a file, or a directory.
- A directory is like a file folder. A directory contained within another is called the *child* of the other. A directory in the filesystem tree may have many children, but it can only have one parent.
- A file can hold information, but cannot contain other files, or directories. The file is the smallest unit in which information is stored.

- The UNIX file system has several important features.
  - *Different types of file;* to you, the user, it appears as though there is only one type of file in UNIX - the file which is used to hold your information. In fact, the UNIX filesystem contains several types of file.
    - ***Ordinary files;*** this type of file is used to store your information, such as some text you have written or an image you have drawn. Files which you create belong to you - you are said to "own" them - and you can set access permissions to control which other users can have access to them. Any file is always contained within a directory.
    - ***Directories;*** a directory is a file that holds other files and other directories. You can create directories in your home directory to hold files and other sub-directories. Directories which you create belong to you, too.
    - ***Special files;*** this type of file is used to represent a real physical device such as a printer, tape drive or terminal. It may seem unusual to think of a physical device as a file, but it allows you to send the output of a command to a device in the same way that you send it to a file. For example:
      $cat scream.au > /dev/audio
      This sends the contents of the sound file *scream.au* to the file */dev/audio* which represents the audio device attached to the system.
    - ***Pipes;*** UNIX allows you to link two or more commands together using a *pipe*. The pipe acts as a temporary file which only exists to hold data from one command until it is read by another. The pipe takes the standard output from one command and uses it as the standard input to another command.
      $ command1 | command2 | command3
      The | (vertical bar) character is used to represent the pipeline connecting the commands. With practice you can use pipes to create complex commands by combining several simpler commands together.
  - *Structure of the file system;* the UNIX file system is organized as a hierarchy of directories starting from a single directory called *root* which is represented by a / (slash). Immediately below the root directory are several system directories that contain information required by the operating system. The standard system

directories are given below. Each one contains specific types of file. The details may vary between different UNIX systems, but these directories should be common to all.

  * / ; The *root* of **ALL** files and directories
  * /bin/ ; Executable system utilities, like ls, cp, rm
  * /boot/ ; The kernel program
  * /dev/ ; Where special device files are kept
  * /etc/ ; System configuration files and databases
  * /home/ ; Where the personal files and directories of all users are kept
  * /lib/ ; Operating system and programming libraries
  * /lost+found/ ; Where the file system checker puts detached files
  * /usr/bin/ ; Additional user commands
  * /root/ ; The home directory of *super user*. The contents of this directory is usually hidden for other users available on the system.
  * /tmp/ ; System scratch files (all users can write here)
  * /usr/include/ ; Standard system header files
  * /usr/lib/ ; More programming and system call libraries
  * /usr/local/ ; Typically a place where local utilities go
  * /usr/man ; The manual pages are kept here

- *Home directory;* any UNIX system can have many users on it at any one time. As a user you are given a home directory in which you are placed whenever you log on to the system. User's home directories are usually grouped together under a system directory such as */home*. A large UNIX system may have several hundred users, with their home directories grouped in subdirectories according to some schema such as their organizational department.
- *Pathnames;* every file and directory in the file system can be identified by a complete list of the names of the directories that are on the route from the root directory to that file or directory. Each directory name on the route is separated by a / (forward slash). For example: **/usr/bin/gcc**. This gives the full pathname starting at the root directory and going down through the directories *usr* and *bin* to the file gcc - the GNU c compiler.

1.1 History of the UNIX operating system

UNIX is not one single operating system, it is a *family of operating systems*. Different computer manufacturers produce their own versions of UNIX. Although these are mostly similar, there are small differences which can cause problems. The most obvious examples are the layout of the file system and the exact format of certain commands.

The first version of UNIX was created in 1969 by Kenneth Thompson and Dennis Ritchie, system engineers at AT&T's Bell Labs, in an effort to provide a multiuser, multitasking system for use by programmers. The philosophy behind the design of UNIX was to provide simple, yet powerful utilities that could be pieced together in a flexible manner to perform a wide variety of tasks. It went through many revisions and gained in popularity until 1977, when it was first made commercially available by Interactive Systems Corporation.

At the same time a team from the University of California at Berkeley was working to improve UNIX. In 1977 it released the first Berkeley Software Distribution, which became known as **BSD**. Over time this won favor through innovations such as the C shell.

Meanwhile the AT&T version was developing in different ways. The 1978 release of Version 7 included the Bourne Shell for the first time. By 1983 commercial interest was growing and Sun Microsystems produced a UNIX workstation. **System V** appeared, directly descended from the original AT&T UNIX and the prototype of the more widely used variant today.

1.1.1 Modern variants of UNIX

There are two main versions of UNIX in use today: **System V** and **BSD**. System V is the more popular of the two.

From a user's perspective they are very similar and you are unlikely to have difficulty unless you use more than one type of system. In this case you might notice differences in the structure of the file system or in how certain commands behave.

1.2 Debugging C/C++ Programs Using gdb

1.2.1 Creating Debug-Ready Code

- Normally, when we write a program, we want to be able to debug it
  - that is, test it using a debugger that allows running it step by step,

  - setting a break point before a given command is executed,
  - looking at contents of variables during program execution, and so on.

- In order for the debugger to be able to relate between the executable program and the original source code, we need to tell the compiler to insert information to the resulting executable program that'll help the debugger. This information is called "debug information". In order to add that to our program, lets compile it differently:
  $gcc -g code1.c -o code1
  The "-g" flag tells the compiler to use debug info, and is recognized by mostly any compiler out there.

- You will note that the resulting file is much larger than that created without usage of the "-g" flag.

- The difference in size is due to the debug information. We may still remove this debug information using the **strip** command, like this:
  $strip code1
  You'll note that the size of the file now is even smaller than if we didn't use the "-g" flag in the first place. This is because even a program compiled without the "-g" flag contains some symbol information (function names, for instance), that the **strip** command removes. You may want to read **strip**'s manual page (man strip) to understand more about what this command does.

1.2.2 Creating Optimized Code

- After we created a program and debugged it properly, we normally want it to compile into an efficient code, and the resulting file to be as small as possible.

- The compiler can help us by optimizing the code, either for speed (to run faster), or for space (to occupy a smaller space), or some combination of the two. The basic way to create an optimized program would be like this:
  $gcc -O code1.c -o code1
  The "-O" flag tells the compiler to optimize the code. This also means the compilation will take longer, as the compiler tries to apply various optimization algorithms to the code.

- This optimization is supposed to be conservative, in that it ensures us the code will still perform the same functionality as it did when compiled without optimization.
- Usually can define an optimization level by adding a number to the ”-O” flag. The higher the number - the better optimized the resulting program will be, and the slower the compiler will complete the compilation.

1.2.3 Getting Extra Compiler Warnings

- Normally the compiler only generates error messages about erroneous code that does not comply with the C standard, and warnings about things that usually tend to cause errors during runtime.
-
- However, we can usually instruct the compiler to give us even more warnings, which is useful to improve the quality of our source code, and to expose bugs that will really bug us later.
- With gcc, this is done using the ”-W” flag. For example, to get the compiler to use all types of warnings it is familiar with, we’ll use a command line like this:
  $gcc -Wall code1.c -o code1
  This will first annoy us - we’ll get all sorts of warnings that might seem irrelevant.
- However, it is better to eliminate the warnings than to eliminate the usage of this flag. Usually, this option will save us more time than it will cause us to waste, and if used consistently, we will get used to coding proper code without thinking too much about it.
- One should also note that some code that works on some architecture with one compiler, might break if we use a different compiler, or a different system, to compile the code on. When developing on the first system, we’ll never see these bugs, but when moving the code to a different platform, the bug will suddenly appear. Also, in many cases we eventually will want to move the code to a new system, even if we had no such intentions initially.

1.2.4 Compiling A Single-Source "C++" Program

- Now that we saw how to compile C programs, the transition to C++ programs is rather simple. All we need to do is use a C++ compiler, in place of the C compiler we used so far.

- So, if our program source is in a file named code1.cc ('cc' to denote C++ code. Some programmers prefer a suffix of 'C' for C++ code), we will use a command such as the following:
  $g++ code1.cc -o code1

1.2.5 Getting a Deeper Understanding - Compilation Steps

Now that we've learned that compilation is not just a simple process, lets try to see what is the complete list of steps taken by the compiler in order to compile a C program.

1. Driver - what we invoked as "gcc". This is actually the "engine", that drives the whole set of tools the compiler is made of. We invoke it, and it begins to invoke the other tools one by one, passing the output of each tool as an input to the next tool.

2. C Pre-Processor - normally called "cpp". It takes a C source file, and handles all the pre-processor definitions (#include files, #define macros, conditional source code inclusion with #ifdef, etc.) You can invoke it separately on your program, usually with a command like:
   $gcc -E code1.c
   Try this and see what the resulting code looks like.

3. The C Compiler - normally called "cc1". This is the actual compiler, that translates the input file into assembly language. As you saw, we used the "-c" flag to invoke it, along with the C Pre-Processor, (and possibly the optimizer too, read on), and the assembler.

4. Optimizer - sometimes comes as a separate module and sometimes as the found inside the compiler module. This one handles the optimization on a representation of the code that is language-neutral. This way, you can use the same optimizer for compilers of different programming languages.

5. Assembler - sometimes called "as". This takes the assembly code generated by the compiler, and translates it into machine language code kept in object files. With gcc, you could tell the driver to generated

only the assembly code, by a command like:
$gcc -S code1.c

6. Linker-Loader - This is the tool that takes all the object files (and C libraries), and links them together, to form one executable file, in a format the operating system supports. A Common format these days is known as "ELF". On SunOs systems, and other older systems, a format named "a.out" was used. This format defines the internal structure of the executable file - location of data segment, location of source code segment, location of debug information and so on.

As you see, the compilation is split in to many different phases. Not all compiler employs exactly the same phases, and sometimes (e.g. for C++ compilers) the situation is even more complex. But the basic idea is quite similar - split the compiler into many different parts, to give the programmer more flexibility, and to allow the compiler developers to re-use as many modules as possible in different compilers for different languages (by replacing the preprocessor and compiler modules), or for different architectures (by replacing the assembly and linker-loader parts).

The explanations given here are specific to the "gdb" debugger, since there are no real standards regarding the activation and usage of debuggers, but once you know what features to expect from a debugger, it's not too hard to adapt your knowledge to different debuggers.

1.2.6 Invoking the "gdb" Debugger

- Before invoking the debugger. make sure you compiled your program (all its modules, as well as during linking) with the "-g" flag.
- Compile the debugme.c program, and then invoke "gdb" to debug it:
  $gcc -g debugme.c -o debugme
  $gdb debugme
  Note that we run the program from the same directory it was compiled in, otherwise gdb won't find the source file, and thus won't be able to show us where in the code we are at a given point. It is possible to ask gdb to search for extra source files in some directory after launching it, but for now, it's easier to just invoke it from the correct directory.

1.2.7 Running A Program Inside The Debugger

- Once we invoked the debugger, we can run the program using the command "run".

- If the program requires command line parameters (like our debugme program does), we can supply them to the "run" command of gdb. For example:
  (gdb) run "hello, world" "goodbye, world"

- Note that we used quotation marks to denote that "hello, world" is a single parameter, and not to separate parameters (the debugger assumes white-space separates the parameters).

1.2.8 Setting Breakpoints

- The problem with just running the code is that it keeps on running until the program exits, which is usually too late. For this, breakpoints are introduced.

- A break point is a command for the debugger to stop the execution of the program before executing a specific source line.

- We can set break points using two methods:

  1. Specifying a specific line of code to stop in:
     (gdb) break debugme.c:9
     Will insert a break point right before checking the command line arguments in our program.
  2. Specifying a function name, to break every time it is being called:
     (gdb)break main
     this will set a break point right when starting the program (as the function "main" gets executed automatically on the beginning of any C or C++ program).

1.2.9 Stepping A Command At A Time

- So lets see, we've invoked gdb, then typed:
  (gdb break main
  (gdb)run "hello, world" "goodbye, world"

- Then the debugger gave something like the following:

```
Starting program: /mnt/usb/home/ozdogan/cfiles/debugme "hello,world"
"goodbye, world"
Breakpoint 1, main (argc=3, argv=0xbffff054) at debugme.c:16
16 if (argc ¡ 2) { /* 2 - 1 for program name (argv[0]) and one for a
param. */
(gdb)
```

Now we want to start running the program slowly, step by step. There are two options for that:

1. "next" - causes the debugger to execute the current command, and stop again, showing the next command in the code to be executed.
2. "step" - causes the debugger to execute the current command, and if it is a function call - break at the beginning of that function. This is useful for debugging nested code.

- Now is your time to experiment with these options with our debug program, and see how they work. It is also useful to read the debuggers help, using the command "help break" and "help breakpoints" to learn how to set several breakpoints, how to see what breakpoints are set, how to delete breakpoints, and how to apply conditions to breakpoints (i.e. make them stop the program only if a certain expression evaluates to "true" when the breakpoint is reached).

1.2.10 Printing Variables And Expressions

- Without being able to examine variables contents during program execution, the whole idea of using a debugger is quite lost. You can print the contents of a variable with a command like this:
  (gdb) print i
  And then you'll get a message like:
  $1 = 0
  which means that "i" contains the number "0".

- Note that this requires "i" to be in scope, or you'll get a message such as:
  No symbol "i" in current context.

- For example, if you break inside the "print_string" function and try to print the value of "i", you'll get this message. You may also try to print more complex expressions, like "i*2", or "argv[3]", or "argv[argc]", and so on.

- In fact, you may also use type casts, call functions found in the program. Again, this is a good time to try this out.

1.2.11 Examining The Function Call Stack

- Once we got into a break-point and examined some variables, we might also wish to see "where we are".

- That is, what function is being executed now, which function called it, and so on. This can be done using the **where** command. At the gdb command prompt, just type "where", and you'll see something like this:
  #0 main (argc=3, argv=0xbffff054) at debugme.c:16
  This means the currently executing function is "main", at file "debugme.c", line 23.

- We also see which arguments each function had received. If there were more functions in the call chain, we'd see them listed in order. This list is also called "a stack trace", since it shows us the structure of the execution stack at this point in the program's life.

- Just as we can see contents of variables in the current function, we can see contents of variables local to the calling function, or to any other function on the stack. For example, if we want to see the contents of variable "i" in function "main", we can type the following two commands:
  (gdb) frame 1
  (gdb) print i
  The "frame" command tells the debugger to switch to the given stack frame ('0' is the frame of the currently executing function). At that stage, any print command invoked will use the context of that stack frame.

- Of-course, if we issue a "step" or "next" command, the program will continue at the top frame, not at the frame we requested to see.

- After all, the debugger cannot "undo" all the calls and continue from there.

1.2.12 Attaching To an Already Running Process

- It might be that we'll want to debug a program that cannot be launched from the command line. This may be because the program is launched from some system daemon (such as a CGI program on the web), and we are too lazy to make it possible to run it directly from the command line. Or perhaps the program takes very long time to run its initialization code, and starting it with a debugger attached to it will cause this startup time to be much much longer.

- In order to do that, we will launch the debugger in this way:
  $gdb debugme 9561

Here we assume that "debugme" is the name of the program executed, and that 9561 is the process id (PID) of the process we want to debug.

- What happens is that gdb first tries looking for a "core" file named "9561", and when it won't find it, it'll assume the supplied number is a process ID, and try to attach to it.
- If there process executes exactly the same program whose path we gave to gdb (not a copy of the file. it must be the exact same file that the process runs), it'll attach to the program, pause its execution, and will let us continue debugging it as if we started the program from inside the debugger.
- Doing a "where" right when we get gdb's prompt will show us the stack trace of the process, and we can continue from there.
- Once we exit the debugger, It will detach itself from the process, and the process will continue execution from where we left it.

1.2.13 Debugging A Crashed Program

- A core file contains the memory image of a process, and (assuming the program within the process contains debug info) its stack trace, contents of variables, and so on.
- A program is normally set to generate a core file containing its memory image when it crashes due to signals such as SEGV or BUS.
- Provided that the shell invoking the program was not set to limit the size of this core file, we will find this file in the working directory of the process (either the directory from which it was started, or the directory it last switched to using the **chdir** system call).
- Once we get such a core file, we can look at it by issuing the following command:
  $gdb /path/to/program/debugme core
  This assumes the program was launched using this path, and the core file is in the current directory. If it is not, we can give the path to the core file.
- When we get the debugger's prompt (assuming the core file was successfully read), we can issue commands such as "print", "where" or "frame X".

- We can not issue commands that imply execution (such as ”next”, or the invocation of function calls).
- In some situations, we will be able to see what caused the crash. One should note that if the program crashed due to invalid memory address access, this will imply that the memory of the program was corrupt, and thus that the core file is corrupt as well, and thus contains bad memory contents, invalid stack frames, etc.
- Thus, we should see the core file's contents as one possible past, out of many probable pasts.