Abstract

This document’s objective is to provide preliminary introduction to Unix and C programming for students taking the Algorithms course offered at Georgia State University. The first section provides a basic introduction to UNIX commands. The second section provides details of the utilities available to write, compile and execute a C program. Later sections include suggestions for debugging and performance tuning of a program. Last section contains an example C program with appropriate library calls to find the execution time of a portion of code on randomly generated input.

Contents

1 Introduction to UNIX .......................... 2
  1.1 Logging on to cheetah ---------------------- 2
  1.2 Directory structure. ---------------------- 2
  1.3 File and Directory Manipulation .......... 2
    1.3.1 Commands to manipulate a directory .. 2
    1.3.2 Commands to manipulate a file ... 3
    1.3.3 Commands to set the access permissions for files and directories .. 3

2 Compiling and Running C Programs ........ 4
  2.1 Compiling .................................. 4
  2.2 Running Programs .......................... 4
  2.3 Execution Monitoring ...................... 4

3 Debugging tools ................................ 5

4 Performance Enhancing Tools .............. 7
  4.1 Handling your Programs Efficiently: The Make Utility: ...................... 7
  4.2 Improving the Efficiency of your Programs - Pixie and Prof ......... 8
    4.2.1 Pixie Utility ................................ 8
    4.2.2 Prof Utility ................................ 8

5 An Example C Program ....................... 9

¹Most of this guide has been put together by past Graduate Research Assistants Hrishikesh Joshi and Nikhil Junankar.
1 Introduction to UNIX

1.1 Logging on to cheetah

Cheetah are to be used for programming and panther for email and other network related activities. All machines can be accessed from the computer labs on campus particularly the Wells Computer Center in the Classroom South building in room 109. You can also access them using a modem from your home. Computer consultants are available at the Wells Computer Center 24 hrs to assist you either in the lab or help on using the modem. Particularly you can get the dial-in numbers from here and numerous other hand-outs. The phone number of the Wells Computer Center is 404-651-4542.

Use the user ids given to you once you get the login prompt and press enter and then the password.

Here we give a brief introduction to the necessary UNIX commands (IRIX in particular, the version of UNIX running on Cheetah). Use man command to see manual pages, and man man to learn about man itself. man -k keyword will list all available commands which relate to the keyword.

1.2 Directory structure.

The directory structure of IRIX is similar to DOS. We can create a tree of directories and access them with pathnames. Concept of a path is also same as in DOS except that the ‘/’ character in DOS is replaced by ‘///’. For example a typical path in DOS is given as

```
usr
bin
```

whereas in IRIX it is given as /usr/bin.

**Absolute path:** Absolute path to a file or a directory starts with a root directory /. For example the path /usr/include/stdio.h is an absolute path to the file stdio.h because it starts with root directory.

**Relative path:** Relative path to a file or a directory does not start with root directory but it is relative to the present directory we are in.

For example let us assume that we are already in /usr directory. Now relative path to the file stdio.h in the include directory is include/stdio.h. Notice that in a path if we do not have first character as / then present directory is taken as the starting point.

1.3 File and Directory Manipulation

The basic IRIX commands required for our day-to-day usage can be broadly classified as follows:

- Commands to manipulate a directory
- Commands to manipulate a file
- Commands to set the access permissions for files and directories

1.3.1 Commands to manipulate a directory

**Home directory:** Every user account on IRIX account has a unique home directory. This directory acts as a root for all the files and directories created by us. This is setup by the system administrator at the time of creating the user account. This is the default working directory every time we log in.

To give pathnames with our home directory as the starting point we should use ~ as our first character in the path name. For example the path ~/doc/mydoc.doc specifies the file mydoc.doc in the sub-directory doc which is under our home directory.

Present working directory: The command `pwd` tells us what directory we are currently in.

Changing directory: The command `cd directory_names` changes the present working directory to the new directory, if it exists. The command, `cd` takes us to our home directory.

Creating a directory: A new directory can be created by `mkdir` command. For example the command `mkdir testDir` will create a new directory testDir in the present sub-directory, we are in.
Moving a directory: `mv` command allows the user to change the location of a directory and/or its name.

Deleting a directory: A directory can be deleted by the following command: `rmdir directory\name` The directory is deleted only if it is empty. If we want to delete a directory along with all its subdirectories then the command `rm -i -r directory\name`. This command recursively deletes all the subdirectories under the given directory and finally deletes the directory itself.

Listing the files in a directory: The command `ls` lists the files and subdirectories in the present directory. The command `ls directory\name` lists the files and subdirectories in the specified directory.

### 1.3.2 Commands to manipulate a file

Creating a file: We are mainly interested in creating C program files also called as C source files. These files contain the C source code in the form of ASCII text and have a `.c` extension. We create these files using the editor program. The simplest editors are `’pico’` and `’vi’`.

The command `pico myfile.c` will open a file that may or may not exist. If it already exists then we can make changes to it and save it either under the same name or under a different name.

By using a C compiler `’cc’`, we can compile our source file `’myfile.c’` and generate. This file is created by the compiler program, by taking out C source file as input. The procedure to compile C programs is described in a later section. Unlike in DOS executable file in IRIX need not have the extension `.exe`. We can execute a file by simply giving its name on the command prompt.

For example, suppose we have an executable named `myfile`, then we can execute it by simply giving `myfile`.

Moving a file: The command is `mv`. It is the same command that was used in moving directories.

Removing a file: The command is `rm -i file\name`.

Copying a file: The command is `cp file\name1 file\name2`.

### 1.3.3 Commands to set the access permissions for files and directories

IRIX provides us a method to define the access privileges for users other than the owner of the file or directory. Normally, owner of the file or directory is the user who actually creates a file or directory by logging into his own account.

Let us assume that the user account `matnajx` has created a C source file `example.c`, on `cheetah`. Following operations can be performed on this file:

1. A file can be modified by changing its contents. This operation is known as write operation. (Deletion is a special write operation in which all the contents of the file are destroyed.)

2. A file can simply be read without changing any contents. This is known as the read operation.

3. A file can be executed (if it is indeed an executable file).

The access privileges for a file can be set by the command `chmod access\bits file\name` Where `access\bits` can be the octal digits LMN (L, M, N can be from 0 to 7)

L: specifies Access privileges for the owner

M: specifies Access privileges for the users of the same group as the owner

N: specifies Access privileges for all the other users

Every digit L, M or N can specify following types of permissions:

0 : Neither read nor write nor execute

1 : Only execute
2 : Only write
3 : Only write and execute
4 : Only read
5 : Only read and execute
6 : Only read and write
7 : All the three privileges

For example, the command `chmod 775 example.c` sets the following privileges for example.c.

- Read, write and execute by the owner.
- Read, write and execute by the users of the same group.
- Read, write by all the other users.

2 Compiling and Running C Programs

2.1 Compiling

To compile the source code in a file called `prog.c` and to obtain the executable in file called `prog` you would enter the following IRIX command, `cc prog.c -o prog -lm`. We normally use the library `-lm` to access the math library.

If you plan to use the debugger (`dbx`, described in the next section) use the following command for compilation:

`cc -g prog.c -o prog -lm`.

2.2 Running Programs

Continuing from the example above, the executable `prog` obtained above is now used to run the program. At the cheetah prompt type `prog` and press ENTER.

If you want to capture the output of `prog` in a file which you can edit later type in the following format:

`prog > output_file`

Using the `&` character you can run your program in background. This means you will return to the command prompt as soon as you press enter. You can now continue with other tasks e.g., reading mail! For example:

`prog > output_file &`

2.3 Execution Monitoring

`ps` command:

- If you type `ps` at the cheetah command prompt and type enter you get a list of processes which belong to you and are currently alive.
- If you type `ps -ef` at the prompt you get a complete listing of all the processes currently alive in the system which will not all belong to you. Some are system processes while others are those initiated by other users.
- To isolate processes while using `ps -ef` which belong to you use `ps -u user_id`
- In the output of the above commands we specifically get information about the process id which is of importance to us. Try to identify the pids of the processes by referring the corresponding command cols. Refer to the man pages for an explanation of other details shown.

`kill` command:

The kill command can be used to terminate processes you identified using `ps`, because of a variety of reasons, primarily because they are erroneous, requiring debugging.
To use kill first obtain the process ids using `ps` by matching the PID column with the COMMAND column. Assuming that you have a PID of a process which you want to kill, type in the following command and press enter `kill -9 PID`. The -9 option stands for sure kill. Others are discussed in the man pages.

`top` command:

The top command displays information about the top cpu processes alive at anytime. To use top, type `top` at the cheetah prompt and press enter. Type `q` to quit. Top displays the top 30 processes and ranks them according to raw cpu percentage usage.

Read the man pages to get more details on `ps`, `kill` and `top`.

3 Debugging tools

The IRIX system provides a source level debugger called `dbx`. It can be invoked from the command line as `dbx exec_program` The `exec_program` is the name of the executable program to be debugged. This program must have been compiled by the `-g` option when compiling the C source code. After we run `dbx` we get a prompt `>` on which we issue the various commands to monitor the execution of the program.

Starting the execution: The program is executed by the command `run [argument list]`. Here argument list is the list of arguments that is normally passed when running the program on the command line. The program execution continues till a break point.

If at any point of time we want to reexecute the program from start, we can use the command `rerun`.

Setting a breakpoint: We can set the break point at any line in the source code by giving the a line number (line numbers are shown in dbx). The command is, `stop at NN`, where NN is the line number at which you want to set the break point.

We can set a break point just after entering a function. The command is, `stop in func_name`, where func_name is the name of the function at which the break point is to be set.

Single stepping: After the program encounters a break point, the execution is controlled by us, normally single stepping. The command is `next` or `n`. This command will execute the statements sequentially. So we have to just keep on doing `next` every time.

Stepping inside a function: The command `step` allows to single step inside a function call.

Continuing execution: When we are single stepping, we can set new break points or continue execution by the command `cont`. This command continues execution till the next break point.

Printing a value of a variable: The command `print variable_name` prints the present value of variable_name. We do not have to specify the type of the variable so it is properly taken care of by the `dbx` itself.

Determining the type of the variable: The command `what is variable_name` gives us the type of the variable variable_name.

Selecting a source file: The command `source_file` edits a particular source file.

Listing the source file: The command `list` lists the next ten lines of the source file currently selected. If we say `list line_number`, the next ten lines starting with the specified line number is shown.

Editing a source file: The command `edit source_file` allows the editing of source_file. It invokes the vi editor for editing the source file.
Debugging the core dumps: Core dump is the abnormal termination of the program due to malicious memory handling, due to a bug in the program. When a core dump occurs during the execution of a program, a file named core is created containing some information about the abnormalities. dbx uses this file to extract the information about the location of the core dump. We do not have to control anything because generation of the core file, is automatic.

To get the name of the function which caused the problem, we issue the command trace or simply t. This gives us the list of functions in their calling sequence with the most recently called function listed first, this function itself is the cause of the core dump.

The next logical step is to actually go inside the function by setting the break point and single step to determine the actual location of the problem. To facilitate the debugging the normal functions should not be more than 30 lines.

Following are possible causes of a core dump:

1. An array is accessed beyond its maximum size.
2. A pointer is accessed without allocating it by a call to malloc or calloc.
3. A pointer is freed twice.
4. A pointer being freed is null.
5. Trying to access the memory through a pointer that is null.
6. Trying to access the memory through a pointer that is uninitialized.

Debugging infinite loops: The most ubiquitous problem is a program going in an infinite loop. For this do the following steps:

1. Run dbx program name.
2. Execute the program by run [argument list]. At this point the program is in infinite loop.
3. Press ctrl-c and break the loop.
4. See the trace by the command t. The very first function listed in the trace contains the infinite loop.
5. Single step inside the function.

Following are the possible causes of an infinite loop:

1. In do-while loops the condition to break the loop never evaluates to 0.
2. In while loops the condition to break the loop never evaluates to 0.
3. In for loops the loop variable is never changed.
4. In for loops the terminating condition is never reached.

Some useful hints to debug programs:

1. dbx can be used to step through the code and find the problem.
2. Generous use of printf statements to print the various information at the various steps can be helpful. It is better if the information printed is only one line long otherwise output is very cluttered.
3. All the debugging statements can be enclosed in #ifdef _DEBUG_ and #endif pairs. Then use #define _DEBUG_ at the beginning of the source file to enable debugging. For example,
#define _DEBUG_    /* Only once and at the beginning of any source file */

 ifdef _DEBUG_
 printf("If I am here, there is a problem. Count = %d\n", Count);
 endif

 The debugging statements can be disabled by removing the statement #define _DEBUG_.

4 Performance Enhancing Tools

4.1 Handling your Programs Efficiently: The Make Utility:

As long as you are working with one C source file say progc you will typically use the following command to compile cc progc -o proglm. When you work with multiple files i.e. your total program is split in more than one file, you need a compilation or a linking procedure which is not cumbersome. Here’s why. Say you have two files main.c and average.c. There are some functions in the average.c that you access in main.c. You will first compile main.c to obtain an object file cc -c main.c (note the -c option creates the object file only)

At this time if you do a ls you will notice main.o in the directory along with the other files. Similarly obtain average.o. Now you need to link them to obtain the final executable because in .o files external linkages are not resolved. cc -o average main.o average.o

Now if there is a change in main.c or object.c you will again have to go through the procedure of obtaining the object file and then linking it with the others. Imagine doing this for 5 files out of the 10 you worked on.

The make utility helps in situations like above. First a special file is created called the Makefile (note the capital M) and kept in the same directory as all other files, .c and .o. Once we create this file by a procedure described next, we type make at the command line. The make program will read the compilation and linking instructions in the Makefile and create the executable thus saving time for the programmer.

Preparing the Makefile: A make file consists of a series of entries of the following form

[...targetfile...]: [.....dependencies.....]
[One Tab Space] [.....commands........]

The first line of each entry is a list of targetfiles separated by spaces, then a colon and then a list of files called the dependencies. A dependency is a file which if changed will alter the functionality of the target file. The colon is followed by a tab before any text is inserted. If the names of the dependencies overflow in the next line then that list is followed by a semi-colon before starting the commands.

A typical Makefile for our running example of main.c and average.c is as given below

average: main.o average.o
 cc -o average main.o average.o -lm
main.o: main.c
 cc -c main.c
average.o: average.c
 cc -c average.c

The make program interprets the first entry in the following manner: average is the target file and it depends on main.o and average.o and it is obtained by executing the command on the next line. The other two entries are interpreted in the same manner. An example of a Makefile is given along with the example programs. Hence the make utility proposes the following methodology... edit...make...run and repeat...
4.2 Improving the Efficiency of your Programs - Pixie and Prof

4.2.1 Pixie Utility

The pixie program adds profiling code to a program. It reads an executable program, partitions it into basic blocks, and writes an equivalent program containing additional code that counts the execution of each basic block. A basic block is a region of the program that can be entered only at the beginning and exited only at the end. Pixie is invoked on an executable file as pixie prog where prog is a executable file from our running example.

After the above command is executed you should find an executable file in your directory called prog.pixie and a file called prog.Addrs.

When you run this new executable file prog.pixie like any other file, it will generate a file containing basic block counts. This file has the name of prog.Counts. This file will go as input to prof command which is explained next.

4.2.2 Prof Utility

Having obtained the prog.Addrs and prog.Counts files which contain information on addresses of blocks and counts of the blocks respectively, by using pixie, we then use the prof utility to analyze this profiling data and produce a listing. This listing typically gives information on how many times a given procedure is invoked, what is it’s execution time, lines or procedures which did not execute and other information used for optimization. The use of prof command is explained below using an example. For more details do a man prof.

Example:

```
cc -o myprog myprog.c      /* generates executable called prog */
pixie -o myprog.pixie myprog   /* generates myprog.Addrs */
myprog.pixie               /* generates myprog.Counts */
prof -pixie myprog myprog.Addrs myprog.Counts > myprog.prof
```
5 An Example C Program

/* this is a sample program which finds maximum and calculates the execution time */

/* compilation: cc filename.c -lm -o filename
   running:     filename
   help:       man cc -- c compiler
               man times -- timing routine
               man vi -- full screen editor
               man dbx -- c debugger

   Refer to this handout or a unix and/or a C book, if needed. */

/* Include Files */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <sys/param.h>
#include <sys/types.h>

/* Constants */
#define maxSize 1024 /* max array size */
#define seed 10 /* seed for random number generator */
#define loop 1000 /* repeats loop times for timing accuracy */

main()
{
    struct tms buffer; /* needed for timing function */
    double StartTime, EndTime, ElapseTime;
    int A[maxSize]; /* input array */
    int max; /* variable to contain the current maximum */
    int i,j; /* index variables */
    srand(seed); /* initialize random number generator */

    for (i=0; i<maxSize; i++)
        A[i] = rand() % 1000; /* fill A with random number in the range 0 through 999; % is mod function */
    StartTime = times(&buffer); /* get current time */

    for (j=1; j<loop; j++) /* repeat max-finding 1000 times so that the time taken would be large enough to measure */
        max = A[0]; /* initialize max */
    for (i=1; i < maxSize; i++) /* repeats loop 1000 times */
        if (max < A[i]) max = A[i];

    EndTime = times(&buffer); /* current time */
    ElapseTime = EndTime - StartTime;
    ElapseTime = ElapseTime / loop; /* time for a single run */
    ElapseTime = ElapseTime / HZ; /* to convert into seconds */
    printf("Max = %d, time taken = %f seconds \n", max, ElapseTime);"