**CSCI.258 (63.258) - Introduction to Operating Systems Using UNIX**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Professor Krishnamoorthy, Computer Science Dept., Framingham State College**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Homework #3 – 150 Points**

**Coverage: Chapters 14, 16, 17, 18 and 19 from Das Textbook**

**Refer Syllabus for due date**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Objectives**

1. Learn about compiling, linking, creating executable and running program
2. Get more experience about process, fork, getpid, and getppid
3. Learn to create *makefile* and use make utility, which are used in software development
4. Learn to modify PATH shell variable to search a new directory
5. Learn to use *ftp* to do file transfer between two systems
6. Learn to use *whereis* and *which* commands
7. Learn to use networking commands (utilities) such as *ftp, rwho, finger, mailx, mount,*  and *ping*
8. Learn to create a backup and restore using *cpio* and *find* commands with pipe or *tar* command

**How to get the answers to the questions in the homework?**

Use the method explained in Homework #1.

**You have to use the UNIX server at FSU to get output for the commands.**

**How to get the answers to the questions in the homework?**

Use the method explained in Homework #1.

**IMPORTANT NOTE: You must study about the commands needed for each question and know how to use them before you begin answer the question.**

**Q1. Compiling, linking, and executing program – cc, vi, cat, running program**

A C program is given at the end of this question. It is the program in your textbook in the Systems Programming chapter. I have made few minor changes and added more documentation. This program reads the content of a directory given as an argument when running the program.

Create a program file with extension .c (for example, hw3q1.c) with the given program as the content. To create the file, you can use one of the two methods described below: **NOTE: Do not issue script command while creating this source file.**

**Remember to exit from script before using the vi command.**

1. You can type the given program in the vi editor.

Alternately, you can copy and paste the program from this Word homework file into Notepad or Wordpad on your PC. Then, delete and retype each and every double quotes (“) one by one. You must do this because the double quote in Word is different from double quote in Notepad or Wordpad. Your program will not work without this change. After changing ALL the double quotes, save the file. Finally, transfer the file from your PC to the UNIX server using ftp command.

After creating the program source file on the UNIX server, issue script command and do the following.

1. Compile and create an executable file for this program using the *cc* command.

The following command will produce a.out as the executable file assuming hw3q1.c is the program file name:

cc hw3q1.c

The following command will produce hw3q1 as the executable file assuming hw3q1.c is the program file name. The file name after the –o option is the executable filename.

cc hw3q1.c –o hw3q1

Use one of the two methods. The later one is preferred choice by professionals.

1. Display the content of the program file, which is also called the source code file.
2. Run the program by typing the executable file name followed by a directory name as argument. The argument could be current working directory or any directory.
3. Look at the output of the program, then using echo command answer what are the two information stored in a directory file.
4. Using echo command, answer what are . and .. displayed in the output

**C program to read the content of a given a directory:**

**Note**: This C program is in your textbook in the Systems Programming chapter. I made couple of minor changes: replaced arg\_check by an if statement. I have added documentation to the program to show you how to document a function and a program. You must create the program file as given including documentation. You should practice to properly document your program and the functions in it. Refer to the book for details about this program. This program should compile without any error, if you type it correctly.

If the cc command produces syntax errors, it is because of your typing mistakes. With each syntax error, the line number where the error occurred will be displayed. To fix the problem, carefully compare the line in your file with the line in the program below. Fix any differences and recompile. Keep doing it until all the errors are fixed.

/\*

\* Program filename: \*\*\* type the actual filename you chose here like in the textbook. \*\*\*

\*

\* Program Description:

\* This program reads the content of a given directory and displays it.

\* Uses the system call function opendir( ) to open the directory file.

\* Uses the system call function readdir( ) to read the directory content one file entry at time

\* in a the while loop until there is no more entry (end of file).

\* Finally, after reading all the entries, the directory file is closed using the

\* system call function closedir( ).

\*

\* Function: main ( )

\* Input parameter:

\* Directory name

\* Output parameter:

\* None

\*

\* Return value:

\* 0, if successful

\* -1, if a directory name is not given as argument

\* -2, if opening the directory file (opendir) fails

\*

\* Program Defect:

\* This program does not explicitly check for readdir system call failure.

\* But returns successful status.

\*/

#include <dirent.h> /\* For DIR and struct dirent \*/

#include <stdio.h> /\* For standard input and output \*/

int main (

int argc, /\* Number of arguments supplied when running the program \*/

char \*\*argv /\* Values of arguments \*/

)

{

DIR \*dir; /\* Directory file pointer returned by opendir \*/

struct dirent \*direntry; /\* Information for one file in the directory \*/

/\* returned by readir \*/

/\* Check number of arguments supplied \*/

if (argc != 2)

{

printf(“Run the program specifying one directory name as argument.\n”);

exit (-1); /\* return error status \*/

}

/\* open the directory file \*/

if ((dir = opendir(argv[1])) == NULL)

{

printf(“%s directory file open failed.\n”, argv[1]);

exit (-2); /\* return error status \*/

}

/\* Read directory entries one file entry at a time until end of file or error \*/

while ((direntry = readdir(dir)) != NULL) /\* until entries are exhausted \*/

{

printf (“%10d %s\n”, direntry->d\_ino, direntry->d\_name);

}

/\* Close directory file \*/

closedir (dir);

exit (0); /\* return successful status \*/

} /\* end main ( ) \*/

**Q2. Using system call functions: fork, getpid and getppid and commands: cc, vi, cat**

**Remember to exit from script before using the vi command.**

1. Without issuing the script command, create a C program file by typing the program given below. It is in your textbook. Check the Systems Programming chapters, if not found then check other chapters. You can name the file like hw3q2.c instead of fork.c since UNIX has a command called fork. Hence, if you name your executable file as fork, it could clash with system command file when you run your program.

/\* Program: fork.c -- A simple fork \* Shows PID, PPID in both parent and child

\*/#include <stdio.h>#include <sys/types.h>

int main (void)

{ pid\_t pid;

printf("Before fork\n"); pid = fork(); /\* Replicate current process \*/ if (pid > 0)

{ /\* In the parent process; make sure \*/ sleep(1); /\* that parent doesn't die before child by sleeping for one second \*/ printf("PARENT -- PID %d PPID: %d, CHILD PID %d\n", getpid(), getppid(), pid); } else if (pid == 0) /\* In the child process \*/ printf("CHILD -- PID %d PPID: %d\n", getpid(), getppid()); else

{ printf("Fork error\n"); exit(1); } printf("Both processes continue from here\n"); /\* In both processes \*/

exit(0); /\* exit from main. Both process will end \*/

}

Issue the script command and then do the following:

1. Compile, link and create an executable file using cc command
2. Display the content of the C file
3. Run the program by typing the executable program filename as the command.
4. Issue ps command
5. Look at the output of the program and the output of the ps command. Then using echo command indicate what is the process name corresponding to the PPID of the parent?

If you make typing mistakes, you have to fix them by **carefully comparing** each line of your program with the above program or the program in the textbook.

**Remember to exit from script before using the vi command.**

**Note**: Go through the program and understand the operations performed by the system call functions: fork, getpid, and getppid. It will help in your quiz.

**Q3. Make Utility – Commands: make, pwd, mkdir, cd, ls, touch, cat, PATH**

1. Create the directory *hw3q3make* under your login directory.
2. Copy one of your working C programs in homework #3 Q1 or Q2 program into *hw3q3make* directory and name the new file as hw3q3.c. If you do not have a working program, then create a simple and small/trivial program, like the “Hello World” that you see in a programming book to do this homework.
3. Change working directory to *hw3q3make*.
4. Exit from script, if it enabled.
5. Using vi editor, create the *makefile* that will produce an executable file for the C programs in the directory. In the *makefile*, there should be a target to create the object module ( .o file ) with compile only command ( cc –c ) and another target to generate the final executable file named hw3q3 by linking the object module.
6. Enable script to start capturing the output for the steps below.
7. Display the current working directory to ensure it is *hw3q3make* directory
8. Display the attributes of the files in the directory using long option
9. Display the content of *makefile*
10. Build the executable by issuing *make* command.
11. Display the attributes of files in the directory
12. Modify the PATH shell variable to search *hw3q3make* directory for command as shown below (type it on the command line and do not modify .profile):

PATH=$PATH:/usr/home/your login name/hw3q3make

1. Change working directory to another directory outside your login/home directory like a system directory
2. Display current working directory
3. Run the program by typing just the executable program filename hw3q3. Supply argument to the program (command), if necessary. The shell should search the hw3q3make since it has been added to the PATH shell variable, find the program and run it.
4. Change working directory to *hw3q3make*
5. Issue *make* command again
6. Issue *touch* command on the C source program file to change the modification date to current time
7. Issue the *make* command again. Observe that the C program gets compiled; object file and the executable files are recreated since the dependent C program file has been modified by the touch command.

**Tips on make utility and makefile**

"*makefile*" is the name of a text file. When you issue the *make* command without any argument, it reads this file and performs the operations specified in it to produce target file(s), where a target file could be an executable file like the executable file hw3q1 in Q1 above that can be run or compiled binary object module like xyz.o that cannot be run but could be linked to produce an executable file. The statements in the *makefile* looks like the following:

**Target** file: one or more **dependent** files separated by space needed to create the target

**Tab key as first character** **in this line** followed by **command** to produce the target

The following is an example where xyz.o is the target file to be created, which dependents on xyz.c file, cc is the command that will compile xyz.c file and produce xyz.o file

xyz.o: xyz.c

    cc -c xyz.c

Note that I typed the “tab” key before the cc in the above line. It is not spaces. Effect of typing space and tab key looks similar on the monitor. If you type space instead of tab, *make* utility will give strange error, which is hard to understand. If the dependent file xyz.c is changed after creating the target, then when the *make* command is typed, it will automatically run the command and produce the executable file.

Make sure the final target file appears first and then other target files since *make* command without any argument creates the first target only. If the final target is not the first target in the *makefile*, then you have to type the final target as argument to the make command like “make hw3q3”.

Using the above example and a working program file that you have, create the *makefile*. Use the vi editor to create it. Put appropriate commands to create the executable file.

Make sure your makefile has two targets: (1) hw3q3.o on object module which cannot be run, and (2) an executable target file hw3q3. There will be 2 lines for each target. You have to use cc command to create each target file.

Read about *make* command and *makefile* in your textbook.

**Q4. ftp – ftp commands**

You should try to do this question from your PC (laptop). Run *cmd* command to start the DOS command interface window on your laptop by clicking on *Start*, then *run* and then typing *cmd* and pressing Enter.

Issue ftp command from DOS window with UNIX server IP address to start the client ftp, establish a session (connection) with the server and login to the server. Issue appropriate ftp commands to perform the following tasks.

1. Issue *help* or *?* command to get the list of ftp commands.
2. Issue *stat* ftp command and make sure the transfer mode is ASCII. If current mode is BINARY, issue ASCII command to change mode to ASCII.
3. Issue ls (lst) ftp command
4. Using *get* (or *recv*) ftp command, get a file from your login directory into your PC local directory. Give name to the file to be stored on the PC different from the file on the server.
5. Create a directory called *ftptest* under your login directory using *mkdir* command
6. Change the working directory to ftptest using *cd* (or cwd) command
7. Issue *pwd* ftp command.
8. Using *put* (or *send*) ftp command, send the file you have received in Step (3) above to the *ftptest* directory.
9. Issue *ls* (lst) ftp command.
10. Delete the file in the *ftptest* directory using *delete* ftp command.
11. Issue *ls* (lst) ftp command.
12. Issue *cd* (or *cwd*) ftp command to change working directory to your login directory.
13. Delete the *ftptest* directory using *rmdir* ftp command.
14. Issue *ls* (lst) ftp command.
15. Issue *quit* ftp command to logout from the ftp session.

Copy and paste the output of the *ftp* session in the DOS window into your file to be submitted.

To exit from DOS window, type *exit* command after copying and pasting the output.

**NOTE**: If you are unable to login to the server from your ftp session on the PC, then login to the server using PuTTY. Start the script. Then, issue ftp command and perform the above steps. After issuing the quit ftp command, stop the script command and save the output.

**Q5. Networking commands – mailx, finger, rwho, ps, grep, ping, mount, netstat, cat**

1. Send mail to yourself using *mailx* command
2. Read the mail you sent to yourself
3. Issue *finger* command with your login name. Finger will work for local and remote users.
4. Issue *rwho* command. The *rwho* command will display users logged on remote systems and local systems, if *rwhod* daemon is running and system is configured to send the info over the network.
5. Issue *ps –e* command and pipe the output to *grep* command with *rwhod* as the search pattern/string.
6. Look at the output of the *grep* command. With echo command, answer whether the *rwhod* daemon is running on the UNIX server.
7. The *ping* command can be used to check the network working condition between two systems. You may have to find the path name of the *ping* command file using *which* or *whereis* command and then type the full/absolute path name of the *ping* command. Issue *ping* command with UNIX server IP address as argument. If the command does not stop after displaying five packets transfer and receive information, terminate the command using ctrl-c. Otherwise, the command may never stop.
8. The *mount* command without arguments will display the list of mounted local and network file systems on the UNIX system. Find the path name of the *mount* command file using *which* or *whereis* command and then by typing the full/absolute path name of the *mount* command to display the list of mounted file systems.
9. Identify the file that has the list of file systems to be mounted when the UNIX system boots. **Hint:** This file is located in the /etc directory and the filename ends with tab. Change to the /etc directory and then issue *ls* command to display filename ending with tab ONLY. Display the content of the appropriate file containing the list of file systems to be mounted.
10. The file named *hosts* in the /etc directory contains names of computers (hosts) and their corresponding IP addresses. The first name in the list is called the official name and the remaining names are called aliases. Display the content of the hosts file.
11. Look at the content of the hosts file displayed above, and then using echo command, answer what is the name of the UNIX server at FSU that you are using.
12. The *netstat* command displays information about your network. To test this command, start the DOS command line interface on your PC by running *cmd* command like you did before. (1) Issue *netstat* command. (2) Issue *help* ( ? ) when you are in *netstate* to see the various options it supports. (3) Identify the option to display the routing table and then issue netstat with that option to display the routing table. Copy and paste the output of *netstat* session from the DOS window into your file to be submitted. To exit from DOS window, type *exit* command after copying and pasting the output.

**Q6. Backup and Restore Using *cpio* and *find* Commands.**

1. Make your login directory as your working directory
2. Using the *find* command and the *cpio* command, create a backup file called *backup* containing all the files in your login directory. To do backup, pipe the standard output of the *find* command to the *cpio* command and redirect the *cpio* standard output to the file named, *backup* like *find . | cpio* *options* > *backup*. Use appropriate options in the place of *options* in the *cpio* command to do backup. **Must use verbose option** in addition to other required options so that the list of files being backed will be displayed on the monitor. Make sure you **use relative path in the *find* command** to generate the list of files with relative path; otherwise you will not be able to restore the files in the restore directory.
3. Display the attributes of the *backup* file
4. Create the directory *restore* under your login directory
5. Copy the *backup* file from your login directory to the *restore* directory
6. Change working directory to *restore*
7. Issue *pwd* command
8. Display the attributes of the files in the *restore* directory
9. Restore the files from the *backup* file into the *restore* directory. Redirect input using < operator from *backup* file to the *cpio* command. **Must use verbose option and create directory option** in addition to other required options so that the list of files being restored will be displayed on the monitor and subdirectories will be created.
10. Display the attributes of the files restored files in the *restore* directory