CSL204: Operating Systems Lab

List of exercises

☞ For all menu-driven programs, the menu should be repeatedly displayed until the user opts to quit.

I Familiarization of Linux commands

1. Explore the usage of following commands. For each of the command, write an illustrative example along with its output.

`ps, pstree, strace, gdb, strings, objdump, nm, file, od, xxd, fuser, top, awk, cal, ls, chmod, chown, chgrp, mkdir, rmdir, locate, nftw, touch, cat, more, less, cp, mv, rm, grep, tail, head, find, sort, stty, sed, uniq, du, df, man, help, pr, tr, diff, wc, bc, gzip, history, groups, cut`

Also learn about *redirections* using `>, >>, <` and *piping* using '|'.

Many of the above commands have multiple options that you can pass along with the command itself, for example `ls -a` or `ps -L`. You are supposed to explore such options as well.

II Shell scripting

Create a menu driven script to print the following:

(a) OS name, its version and kernel version

(b) number of user accounts

(c) currently logged user and his login name

(d) all available shells

(e) your current shell (the terminal in which you are running the script)

(f) your home directory

(g) your current path setting

(h) your current working directory.

2. Google a bit on *hyperthreading*. Now create a menu driven script to print the following processor information using the `proc` file system:

(a) vendor (manufacturer) id

(b) model name

(c) processor generation

(d) number of processor chips

(e) number of processor cores

(f) is your processor hyperthreaded?

(g) number of logical processors

(h) id of the core to which each logical processor is mapped

(i) speed of each logical processor

(j) cache size.

3. Create a menu driven script to print the following memory information using the `proc` file system:

(a) total memory in the system

(b) amount of memory left unused

(c) amount of memory available

(d) amount of memory used as cache.

How do you differentiate between parts (b) and (c) above?

Given the basic pay of an employee, calculate the salary as follows.

$$salary = basic + dp + da + hra + ma–pf$$

where

dp is 50% of $basic$

da is 35% of $(basic + dp)$

hra is 8% of $(basic + dp)$

ma is 3% of $(basic + dp)$

pf is 10% of $(basic + dp)$

Determine the average of a set of N numbers.

4. Implement a menu driven scientific calculator. You should consider arithmetic operators, sin, cos, tan, power and square root.

5. Print all prime numbers in a range.

6. Write a menu driven program to convert a positive decimal number to binary, octal or hexadecimal based on the user's choice. Each of the conversions should be implemented using separate functions.

Read two numbers via command line and display the result of dividing the first number by the second. If the second number is zero, your script should prompt the user to give proper inputs, and then perform the operation with the new inputs.

Input a list of numbers via command line, count the even and odd numbers and then write these counts to a file `mycounts.txt`

7. Read an integer N, a string *word* and a filename via command line. If *word* occurs more than N times in the file, remove all its occurrences in the file.

You are given a file `results.csv`, containing the register number and grades obtained for six courses in an examination. First line is the header row with titles "register number" and the course codes. Following lines contain register number of student and his/her grades. Perform a result analysis which shows the following details :

- count of students registered
- count of students who passed in all courses
- *pass count* and *fail count* for all the courses along with the course wise pass percentage in a tabular form.

Create a command `printstats` which is to be invoked as

```
printstats dty
```

This command has to display the following:

1. total number of files in the directory `dty`
2. list the extensions of all the files present in `dty`
3. count of files having each extension.

The command has to work recursively for all the subdirectories as well.

8. You have a file `s4csb1.txt` with the following format: `AdmnNo,Name,Address,PhoneNo,Email`. Now create a command `recedit` which is to be invoked as:

```
recedit op ID
```

Here `op` is one among the operations `add,search,update,delete` and `ID` is an admission number. The command behaviour is summarized below:

- raise an error if sufficient number of arguments are not supplied.

- to add a new student with the given `ID`, get the other pertinent details and create a new row for the student in the file `s4csb1.txt`. If such a student already exists, raise an error.
- if the operation is `search`, the pertinent details should be printed if such a student exists or raise an error otherwise.
- if the operation is `update`, get the details and update the student record.
- if the operation is `delete`, remove the student record.

9. Create a menu driven script to print the following process information:

(a) Number of processes forked since the last boot

(b) Number of processes currently in the system

(c) Number of running processes

(d) Number of blocked processes

(e) PID of the current shell

(f) Number of context switches performed by this shell. How many of these were forcibly taken?

III Process management

10. Google a bit on the **`getrusage()`** system call and the **`system()`** library call. Now, write a simple program to print the system date and time. Have your program also print how long it (in the strict sense, the corresponding process) ran in user and kernel modes. Next add a loop in your program that runs for a long time (some loop which simply counts up to a huge value) and then rerun the program. See how the two time values have changed.

11. Create a new process using the **`fork()`** system call. Have the processes introduce themselves with their process IDs as follows:

Parent: `I am <myID>, and my child is <mychildID>`
Child: `I am <myID>, and my parent is <myparentID>`

Use the **`pstree`** command to display the process tree for the child process starting from the **`init`** process. The child process is to be "highlighted" in the tree.

12. Write a program to add two integers (received via command line) and compile it to an executable **`myadder`**. Now write another program that forks a new process. Make the child process add two integers by replacing its image with the **`myadder`** image using **`execvp()`** system call.

Fork a new process. The child process prints the string "`CSL204`" and the parent process prints "`Operating Systems Lab`". Use **`wait()`** system call to ensure that the output displayed is "`CSL204 Operating Systems Lab`".

13. Fork two processes. The parent process inputs an integer array. One child prints all the even numbers in the array and the other prints all the odd numbers. Use **`waitpid()`** system call to ensure that all odd numbers are printed first followed by the even numbers.

IV Interprocess communication

Two processes use message passing for IPC. The first process sends three strings to the second process. The second process concatenates them to a single string (with whitespaces being inserted between the individual strings) and sends it back to the first process. The first process prints the concatenated string in the flipped case, that is if the concatenated string is "`Hello S4 Students`", the final output should be "`hELLO s4 sTUDENTS`".

14. The **`time`** command can be used to know the actual time taken to execute a command. For example

```
time ls
```

will tell you how much time it took to execute the **`ls`** [Actually, **`time`** command outputs three time values. Google about this to know further!]. Let us now build our version of **`time`**. You need to create a command **`exectime`** that will be run as

```
exectime cmnd
```

and will report the amount of elapsed time to run the specified command `cmnd`. (For this, you need to write a program `time.c` (or with any other name) and compile to an executable **`exectime`** and then copy it to the **`/usr/bin`** directory as was discussed in the class.)

The strategy is to fork a child process that will execute the specified command using **`execvp()`**. However, the child, before it executes the command, will record a timestamp of the current time (call it `starting time`). The parent process will wait for the child process to terminate. Once the child terminates, the parent will record the current timestamp (call it `ending time`). The difference between the starting and ending times represents the elapsed time to execute the command.

As the parent and child are separate processes, they will need to arrange how the starting time will be shared between them. For our case, you will use *shared memory*. The child process should write the starting time to a region of shared memory before it calls **`execvp()`**. After the child process terminates, the parent will read the starting time from shared memory and proceed with calculating the elapsed time. You should use the **`gettimeofday()`** system call to record the current timestamp.

V Multithreading

15. Write a multithreaded program that calculates the mean, median, and standard deviation for a list of integers. This program should receive a series of integers on the command line and will then create three separate worker threads. The first thread will determine the mean value, the second will determine the median and the third will calculate the standard deviation of the integers. The variables representing the mean, median, and standard deviation values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited.

VI Process scheduling

16. Input a list of processes, their CPU burst times (integral values), arrival times, and priorities. Then simulate the following scheduling algorithms on the process mix:

(a) FCFS

(b) SJF

(c) SRTF

(d) non-preemptive priority (a larger priority number implies a higher priority)

(e) RR (with a quantum of 3 units).

Determine which algorithm results in the minimum average waiting time (over all processes).

VII Process synchronization

Use semaphores to solve the producer consumer problem.

Use semaphores to solve the readers-writers problem with writers being given priority over readers.

17. A university computer science department has a teaching assistant (TA) who helps undergraduate students with their programming assignments during regular office hours. The TA's office is rather small and has room for only one desk with a chair and computer. There are three chairs in the hallway outside the office where students can sit and wait if the TA is currently helping another student. When there are no students who need help during office hours, the TA sits at the desk and takes a nap. If a student arrives during office hours and finds the TA sleeping, the student must awaken the TA to ask for help. If a student arrives and finds the TA currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chairs are available, the student will come back at a later time. Using semaphores, implement a solution that coordinates the activities of the TA and the students.

VIII Deadlock handling

18. Obtain a (deadlock-free) process mix and simulate the banker's algorithm to determine a safe execution sequence.

Obtain a process mix and determine if the system is deadlocked.

19. Implement the deadlock-free semaphore-based solution for the dining philosophers' problem.

IX Memory management

20. Simulate the address translation in the paging scheme as follows: The program receives three command line arguments in the order

- size of the virtual address space (in megabytes)
- page size (in kilobytes)
- a virtual address (in decimal notation)

The output should be the physical address corresponding to the virtual address in ⟨ frame number, offset⟩ format. You may assume that the page table is implemented as an array indexed by page numbers. (NB: If the page table has no index for the page number determined from the virtual address, you may just declare a page table miss!)

21. Simulate the FIFO, LRU, and LFU as follows: First, generate a random page-reference string where page numbers range from 0 to 9. Apply the random page-reference string to each algorithm, and record the number of page faults incurred by each algorithm. Assume that demand paging is used. The length of the reference string and the number of page frames (varying from 1 to 7) are to be received as command line arguments.

22. At one instant, the memory map of a 4MB (4000KB) RAM looks as in Figure 1. Processes (P1, P2, etc.) request the operating system for memory and also release the allocated memory after completing execution, A sample execution trace is shown below:

```
P7 requests for 115 KB
P10 requests for 650KB
P3 completes execution
P1 completes execution
P6 completes execution
P8 requests for 200KB
P5 completes execution
P2 completes execution
P9 requests for 37KB
P10 completes execution
P9 completes execution
P4 completes execution
```

Which strategy among the first fit, best fit, and worst fit performs the best here?

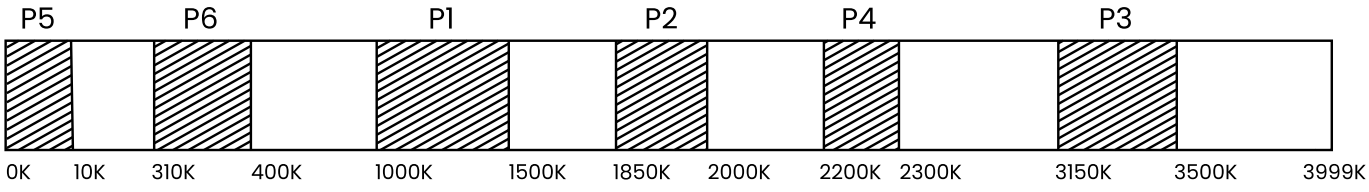

Figure 1: A memory map (Picture courtesy: Pridhu Raj)

X File management

23. Create a command **`listdir`** that works like the **`ls`** command and will be invoked as

```
listdir path
```

where `path` is the path of some directory in your system. The output should be a list of files and directories in that path along with the following information for each file or directory:

- file permissions and file-type information
- the user identity of the file owner
- the time of last access
- the time of last modification to contents.

You may have to use system calls like **`opendir()`**, **`readdir()`**, **`stat()`** etc.

Repeat the above exercise, but this time, write the output to a file **`myoutput.txt`** using the pertinent system calls.

XI Disk management

24. Simulate the FCFS, SCAN, LOOK and CSCAN disk-scheduling algorithms as follows: Your program will service a disk with 5,000 cylinders numbered 0 to 4,999. The program will generate a random series of 10 cylinder requests and service them according to each of the algorithms listed earlier. The program should be invoked with the initial position of the disk head and the last request served as command line arguments and should report the total number of head movements required by each algorithm.

Repeat the above exercise with SSTF and CLOOK algorithms.