Problem 0

In this assignment, you will explore the **/proc** filesystem in linux. The **/proc** filesystem provides a means to get and set various information about the kernel and about particular processes.

You have to first write a program that will read the /proc file system and print out the following (with an appropriate message in each case):

1. The number of CPUs in your machine and their clock speed, number of cores.
2. The version of Linux kernel running on your system
3. The time in day:hr:min:sec when the system was last booted
4. The average load on the system in the last 15 minutes
5. The total usable and currently free memory in the system
6. The total swap space and the currently used swap space in the system
7. The swap partitions and their sizes
8. The time the CPU spent (over all processes) in the user mode and kernel mode
9. The number of context switches made by the system so far
10. The number of interrupts handled by the system so far

Next write another program that gets the following information specific to a process. The program takes the *pid* of the process as a command line argument.

1. The command line with which the process was started
2. The time spent by the process in running and in waiting
3. The time spent by the process in the user mode, kernel mode
4. The environment of the process
5. The contents of the address space of the process

In order to answer the above questions, the files of the /proc filesystem that will be relevant for you are *cpuinfo, uptime, loadavg, cmdline, stat, meminfo*, *mem, schedstat, maps*. Some of these files will be under /proc directly, some will be under the directory for the specific process, and some will be under both. You will need to read and understand what is contained in these files and implement the above program. Note that the exact format of the files vary somewhat between different versions of Linux, so you should try to write your program in as format-independent manner as possible (the names of things are mostly standard).

Problem 1:

Write a program that uses threads to perform a couple operations on Boolean matrices. All shared data should be declared as global variables. Mutual exclusion and thread synchronization must be achieved by pthread library calls only (mutexes and condition variables). You are *not allowed* to use shared memory, semaphores or any other inter-process communication mechanism (like pipes).

Part 1: Generating a Boolean matrix

The master thread creates an $M \times M$ Boolean matrix A with random entries. Each entry of the matrix is zero or one with probability half. The matrix is to be stored in a global two-dimensional array, and may be statically or dynamically allocated (your choice).

Part 2: Creating the worker threads

The master thread creates N worker threads among which the following Boolean matrix operations will be equitably shared. Notice that the same threads will take part in *all* of the following operations. That is, you must not create a fresh set of threads for each individual matrix operation or in each iteration. The master thread must create all necessary resources (like mutexes and condition variables to be used later) *before* the creation of any worker thread. The worker threads must be joinable.

Part 3: Counting the number of ones in A

Each of the N worker threads computes the number of ones in an $(M / N) \times M$ submatrix, and adds this count to a shared (that is, global) counter variable. The master thread initializes the counter to zero before any worker thread accummulates its count in the counter. The updating of the counter by the N threads must be mutually exclusive. After all the worker threads accummulate their respective counts, the master thread prints the value of the counter.

Part 4: Computing the transitive closure of A

For two Boolean matrices U and V, define the product UV as the Boolean matrix W such that the (i,j)-th entry of W is one if and only if there is a k for which the (i,k)-th entry of U and the (k,j)-th entry of V are both one. This multiplication is not the standard modulo-two product of U and V.

The worker threads compute A, A^2, A^4, A^8, . . . , A^{2^e} for some e satisfying $2^e \geq m$. In each squaring step, the current matrix stored in A is multiplied with itself, and the result is temporarily stored in a second global matrix B. Each of the N worker threads computes M / N rows of the product. After all the worker threads complete their respective parts in the computation of B, they collectively copy back the product stored in B to the matrix A. Each worker thread copies the portion of B computed by it back to A.

Since the locations of A and B modified by the worker threads are disjoint from one another, no mutual exclusion is necessary during squaring or copying back. However, the copying of B to A must start *after* all computations involving the old A are over. Moreover, the next squaring step is allowed to start only *after* the entire copy of B to A is over. Use condition variable(s) to synchronize the worker threads.

After A^{2^e} is computed (and stored in the global array A), the master thread prints the matrix A.

Part 5: Winding up

Each worker thread individually terminates at this point. The master thread waits for all the worker threads to join. After that, the master thread cleans up thread resources, and exits.

Take M = 1000 (dimension of the matrix A and N=4 in your submission)

Problem 2

Design and Implementation of a Filesystem

In this problem, you will design a filesystem of your own. Your filesystem will reside physically in a file on disk. The name of the file (for example, *$HOME/myFS*) will be the name of your filesystem. In the rest of this assignment, we will refer to the name of the filesystem as *FS_Name*. The raw size of your filesystem should be not less than 100 KB.

Your filesystem will store files and directories. The directories will also be treated as files. Each file will have the following attributes: *name, type (file or directory), size (in bytes), time of creation,* and *mode* (*read-only*, *read-write*). The *mode* attribute is valid only if the file is a regular file. You will have to maintain a tree-structured directory organization. You should be able to create and delete files and directories in your filesystem, and read and write to the files as well.

Think of your filesystem as a sequence of bytes, divided into 64-byte long blocks. The first block should be the *boot* block, and should be filled with all 0's as your filesystem is not bootable. The next block (or blocks if you need more than one) will be the *superblock*(s). Your superblock(s) will contain at least the following information: the name of the filesystem, its size, information about the root directory, number of blocks used, number of free blocks.

The filesystem will support a *myformat*() function call to create the filesystem. This is the same as logical formatting of a disk. It should create the superblock(s) on your filesystem, create and initialize any persistent data structures you need to store on the filesystem, and set to 0 all other locations in the filesystem. The *myformat*() function can be called from within a program to "format" your filesystem at any time.

After the filesystem is created, any application program should be able to use it by making the following calls:

1. *mount(FS_Name)*: This function creates and initializes the in-memory data structures that you need for subsequent operations. This will also need to read the superblock of the filesystem and cache it in memory. This function should be the first function called, and returns 0 on success. Trying to mount an already mounted filesystem should return an error. Making any of the following operations on an unmounted filesystem should also return an error.

2. *unmount(FS_Name)*: This function frees the in-memory data structures created, making sure that any cached data is properly written back to disk if not already so. This should be the last function called.

3. *myopen(filename, mode)*: This opens a file by name. Mode can be read-only or read-write (use "r" and "w"). If the file does not exist, it is created with the mode specified, with 0 size. If an existing file is attempted to be opened in the wrong mode, an error occurs. The function returns an integer (> 0) file descriptor to the file on success, 0 otherwise.

4. *myread*: This has the same syntax, meaning, and return value as the *read* call you know.

5. *mywrite*: This has the same syntax, meaning, and return value as the *write* call you know.

6. *myclose*: This is same as the *close* call.

7. *mymkdir(dirname)*: This creates a new directory *dirname.*

8. *myrmdir (dirname)*: This deletes an existing directory *dirname*. If the directory is not empty, an error occurs.

9. *myrm (filename)*: This deletes the file *filename* from your filesystem.

10. *myls(dirname)*: This lists all files and directories under the directory *dirname* in your file system. If this command is called without the parameter *dirname*, contents of the root directory is listed.

Your filesystem should implement these calls efficiently. You should keep a file descriptor table in memory to keep track of open files and their information. In addition, you can cache any other info from disk that you want in order to efficiently implement your calls. However, if the cache copy is modified, you should have a scheme for updating the disk copy within reasonable time.

You can assume that only a single process will access the filesystem at one time, so there is only one file descriptor table.

Problem 3

In this problem, you will need to simulate different page-replacement algorithms and compare their performances. Your simulations should be able to take in the required parameters from a data file in the following format:

- First line of the file contains two integers in this order - the no. of pages in the reference string and the no. of page frames
- The rest of the file contains page reference string as a sequence of integers that are the page numbers (in virtual address space) accessed by the process in sequence.

The simulator will simulate the behavior of the following page replacement algorithms on the reference string and report the number of page faults generated for each algorithm:

FIFO
LFU
LRU

The simulator should take the following command line arguments (in this sequence): *the name of the data file*, a sequence of strings (max. 3) from the following sets: *FF, LF, LR* (meaning the above 3 algorithms respectively). The simulator simulates only the algorithms specified in the command line. If no algorithm is specified, all 3 are simulated.

Problem 4:

File system defragmenters can improve performance by laying out all the blocks of a file in sequential order on disk. In this project, you will write a disk defragmenter for a Unix-like file system.

Your defragmenter will be invoked as follows:

$ ./defrag <fragmented disk file>

Your defragmenter should output a new disk image with "-defrag" concatenated to the end of the input file name. For instance,

$ ./defrag datafile

should produce the output file "datafile-defrag"

On disk, the first 512 bytes contain the *boot block* (and you can ignore it). The second 512 bytes contain the superblock. All offsets in the superblock start at 1024 bytes into the disk and are given as blocks.

Simulate a random set of processes with random set of memory requirements. Allocate memory to them as needed, maintain a free list or something else, invoke the defragmenter periodically.