**Theory**

Given this reference string

334444121122323314432255543

Compute the page fault frequency and the mean resident set for the Working Set strategy with control parameter τ = 3.

Given this reference string

334444121122323314432255543

Compute the **page fault frequency and the mean resident set** for the VMIN with control parameter τ = 3.

The VMIN strategy is identical to the Working Set strategy, but knows the future, i.e. at time t it knows if the current page will be used within τ references: it is this case the page is kept in the resident set, otherwise it will leave the resident after it has been used (in the next reference time).

Write the sequence of instructions that allow the bash process to interpret and execute the command

**p1 | p2**

where **p1** and **p**2 are two executable processes.

Describe the syntax of the system calls **close** e **unlink**, and list with precision what they do, with reference to the internal structures of the file system.

What is a kernel module, how it is created, installed and removed?

(3.0 marks) **Detail** the steps performed by the system call **close**, with reference to all the **objects** and **data structures** involved by its operations.

(3.0 marks) Explain all the arguments of this command line:

**qemu -hdb fs.img xv6.img -serial mon:stdio -S –gdb \ tcp::26000 -smp 2 -m 512**

Write the sequence of instructions of a parent process that

* Creates a child, which executes, as a process, the command **cat –n,** which prints what it receives from its standard input on a file **foo.txt**.
* then, it loops forever sleeping 2 seconds, increments an integer number and writes its value on its standard output
* The parent process redirects its standard output to the standard input of the child process when it receives a **SIGUSR1** signal from a bash window, and recovers the original redirection to the standard output when it receives again the same signal. (Code in **dup\_restore.c**)

Illustrate the use of the **popen** system call.

Describe the solution to the Mutual Exclusion problem for a single CPU system without special instructions.

**Programs**

Write a C program using Pthreads to implement an election algorithm that elects a leader thread.

Assume a system with **N** threads. Each thread has its thread identifier and an integer rank value generated random by the main thread (in the range **1-N**).

Threads **cannot access the rank value of other threads**, only the current best thread rank value is available in a global variable **best\_rank** together with the corresponding thread identifier.

Each thread must compare its own rank value with the current value in **best\_rank** to decide if it is the leader or not, but it must synchronize with all the other threads to be sure that they have updated the value of **best\_rank.**

When each thread has obtained the rank value of the leader thread it simply prints:

* its identifier and its rank value,
* the leader thread identifier and its rank value.

Hint: think to a voting algorithm, and use a global variable to count the number of threads that completed their voting process.

Code in **election.c**

The binomial coefficient is defined as:

**n** and **k** are integers, and **n >= k >= 0**.

For example

Write a concurrent C program in the Unix environment which receives two arguments **n**, **k**, and computes and print the binomial coefficient.

The computation has to be performed in concurrency by a **numerator** thread a **denominator** thread.

The **numerator** thread loops by taking **two factors at a time** that must be multiplied (for example 15 and 14), computes their product, and save its result in its variable **numer**.

The same procedure is performed by the **denominator** threads, which save its result in its variable **denom**.

After the **numerator** and the **denominator** threads have performed their computation, **they must synchronize**: **one of them has to perform the division** of the partial contributions stored in **numer** and **denom,** and save it in a global variable **result.**

Then, **numerator** and **denominator** threads continue their job with the next pairs of factors that have not yet been processed.

Since it is possible that the number of remaining factors in the numerator and in the denominator is less than 2, care has to be taken to synchronize the threads also considering this possibility, so that the synchronization among the **numerator** and **denominator** threads is correct and each thread can terminate.

The main thread will print the **result** of the binomial coefficient computation.

Code in **binomial.c**

Write a concurrent C program in the Unix environment which receives two arguments: a directory pathname and a filename.

The main thread creates **N** threads which list the content of the directory (first argument), and its subdirectories on the file passed as the second argument.

In particular, each thread has to print its number and the pathname of all the files that it finds in a directory. The pathname of the regular files must also be written in the filename given as second argument.

Code in **directory\_search.c**