Assignment 5-6

COMP 2401

Date: November 18, 2019

Due: on December 6, 2019 before 18:00

Submission: Electronic submission on cuLearn.

Objectives:

a. Using binary files: opening and closing a file (fopen() and fclose(), reading from a file (fread())

b. Process spawning: differentiating between parent and child processes (fork()), examining the return code of the child process(), waiting for a child process (wait() and waitpid())

c. Program Morphing – changing from one program to another (excevp())

d. Inter process communication – using signals and return code

e. Usage of signals – signals and function pointers.

f. Threads – creating, synchronizing using mutex

Submission (10 pts.)

- Submission must be in cuLearn by the due date.
- Submit a single tar file with all the c and h files. The tar file name is A5.tar
- The tar file should include all programs and code (code that you wrote and code that was provided as part of the assignment)
- Submit a Readme.txt file explaining
    - Purpose of software
    - Who the developer is and the development date
    - How the software is organized (partitioned into files)
    - Instruction on how to compile the program (give an example)
    - Any issues/limitations problem that the user must be aware of
    - Instructions explaining how to use the software
- Reminder
    - Assignment must run on the course VM in the school's labs.
    - After loading your assignment (the tar file) do the following:
        - Create a temporary dirctoriy
        - Download the tar file
        - Extract the files, and make sure that all the files are available.
        - Compile and the test the code to ensure that you loaded the correct files.

Coding Instructions:

1. Comments in Code – as provided in the slides given in class

2. No usage of global variables. All data must be passed or received via function parameters except when specifically stated.

Provided Resources

The following files are provided in assignment_5.tar

1. *numPrimeFactors.c* – the base program for returning the number of prime factors of a given input number
2. *prime.bin* – a file that contains 12 unsigned numbers in binary format
3. *prime.txt* – a file that contains 12 unsigned numbers in ASCII format. The numbers correspond to the numbers in the file prime.bin
4. *createBinary.c* – a program that creates a binary file from a given set of numbers. Assuming that the executable is *a.out* the usage is: *a.out filename num1 num2* etc. where filename is the binary file name, num1 is the first unsigned number num2 is the second unsigned number etc.
5. *prime_thread*.c – skeleton program for Task II (threads)

Grading:

Assignment is graded out of 200 Note that the maximum grade that you can receive for the all sections excluding the bonus sections is 200 even though the number of allotted points for the different sections (excluding the bonus sections) is more than 200. Thus, the maximum points that you can earn for this assignment is 230 (200 for assignment sections + 30 for bonus sections).

The grade may be reduced to 0 if the correct file names are not used. Part or all the assignment may be tested by another program.

You will be graded with respect to what you have submitted. Namely, you earn points for working code and functionality.

Grading will look at how the functions are coded, whether functions meet the required functionality, memory allocation and de-allocation, setting pointers to NULL as needed, etc.

1 Background

You are tasked to write a program that accepts a set of unsigned long integers and computes the total number of prime factors of all the input numbers. In order to complete the task you are given a program (*numPrimeFactors.c*), which accepts a single unsigned integer as a command line parameter and return:

- the number of prime factors
- -2 if the command line argument did not include a number

The given program assumes that the input, if provided to the program, is correct and therefore the program does not require to check whether the input is an unsigned integer.

Not all programs can fully utilize the computation power of the cpu. Often a program is required to wait for slow resources (e.g., reading from or writing to a disk).

2 Task 0 Understanding the base code (0 points)

In this task you will understand the prime factor counting program that was provided.

1. Review the program numPrimeFactors.c. Make sure that you understand the program and how it works. Do the following:
   1.1. Compile the program and create an executable called numPrimeFactors.
   1.2. Test the program as follows:
      1.2.1. Test 1 numPrimeFactors 1535068679
      1.2.2. Test 2 numPrimeFactors 1535068677
      1.2.3. Test 2 numPrimeFactors 39821

   Since <u>you will not see any output</u> use the debugger to ensure that your code is working by putting breakpoints at each of the return() function calls. Alternatively, you can **<u>temporarily</u>** add a print statement indicating the number of prime factors.

3 Task I Spawning multiple children (85 points)

Task overview

In this task you will write a program that will spawn multiple child processes. Each child process will morph itself into the numPrimeFactors program. The parent program will wait until all the child processes have completed their work and then prints the total number of prime factors of all the input numbers.

You can assume that the binary file (if it is exists) contains at least 10 unsigned long numbers. For simplicity the program will need to process only the first 10 numbers. You can also obtain additional points by doing the bonus section in this task.

To do

1. (25 pts.) Create a program *multiFactor* that accepts the binary file name as a single command line parameter. Here you can use the tutorial tools that you developed. Name the code file *multiFactor.c*,
   1.1. (5 pts.) The program will check that a file name was provided as command line parameter. If no file was provided it will print how to use the program – *Usage: multiFactor fileName*. See below for correct program return code.
   1.2. (10 pts.) The program will check if the file exists. If the file does not exist then the program should produce an error message: *file file.bin does not exist*, where *file.bin* is the provided file name. See below for correct program return code.
   1.3. Reading data from file (10 pts.) - The program shall read the first 10 numbers into an array of unsigned integers using a single call to fread().
2. (10 pts.) Create a function morph() which will take as input a string (the input number) and morph itself to the numPrimeFactors program using the excev or execvp system function call.
   2.1. Prototype *int morph(char *number);*
   2.2. Input: unsigned number to be checked
   2.3. Return -1 if the morph failed
3. Spawning children (25 pts.)
   3.1. (10 points) The program should spawn a child process for each of the input numbers. Each child should morph itself into the numPrimeFactors program.
   3.2. (5 pts.) Each child process needs to return the number of prime factors of the assigned number
   3.3. (10 pts.) The program needs to spawn all child processes before it starts to collect the responses from the child processes.
4. (20 points) The parent program should sum up the number of prime factors that were returned by all child processes
   4.1. (10 points) Here you will have to use the waitpid() function, which waits for child processes to complete their tasks. You will invoke the function as waitpid(-1, &status, 0) where -1 indicates wait for any child process to complete their task.
   4.2. (10 points) The parent program also needs to know when to quit. This can be done in two ways a. check the return code from waitpid(); If it is -1 then there are no more children or an error has occurred. In this case the program can quit. B. count how many times it has received input from the children and quit once argc-1 children have responded.
5. (5 points) Create a make file for the program. Makefile name should be Makefile1. The program name should be multiSpawn
6. Test your program. Here you are provided with: a. a binary file prime.bin which contains 12 unsigned integers in a binary format. b. a utility to create a binary file (the utility is createBinary.c).

7. Bonus Section (10 points)

   7.1. Reading and processing all the numbers in the file (in case the file consists of more than 10 numbers). Here you will need to allocate memory in Section 1.3. This can be done by determining how many numbers are in the file and then allocating memory for the array of numbers and array of process ids. Determining how many numbers are contained in the binary file is accomplished by first finding the size of the file: a. moving the file pointer to the end of the file using fseek() and then b. determining the file size using ftell(). Once the file size is found the number of records is found by fileSize/sizeof(unsigned long)

8. Bonus Section (10 points)

   8.1. In this task you will write enhance the program in Task I to use a simple signal SIGUSR1
   8.2. Add a counter to your program from Task III, which tracks/collects the number of processes that have responded so far.
   8.3. If signal SIGUSR1 is invoked then the program should print how many processes have finished their execution and how many processes are still working. Note, that in order to accomplish this you will need to use global variable within the file

scope. For example you can either declare it as static *int countFinished* or *int countFinished*. You will also need to keep as a global variable the total number of child processes that were launched. Otherwise you will not be able to complete access the information from the interrupt function.

4 Task II Threads (120 points)

Task overview

In this task you will write a program that uses threads to solve the same problems as Task II. You can use the skeleton file *prime_threads.c*

You can assume that the binary file (if it is exists) contains at least 10 unsigned long numbers. You can also obtain additional points by doing the bonus section in this task.

To do

1. (10 points) The program will accept as command line arguments the following:
   1.1. Parameter 1 - binary file name
   1.2. Remaining parameters are indices of the numbers that must be processed. If the index value is -1 then all numbers in the file must be processed.
   Examples:
      A. Command line is *multiThread prime.bin -1*. In this case the program *multiThread* shall compute the number of prime factors of all the numbers stored in the file.
      B. Command line is *multiThread prime.bin 1 5 3*. In this case the program *multiThread* shall compute the number of prime factors for the numbers that appear in positions 1, 5 and 3.
      Thus, if the file contains the following number (in binary) 55 34 1001 2 14 37 21 then the program will only process the numbers 55 14 and 1001
   1.3. If the number of parameters is not greater than two then the program shall print how to use the program. For example: *multiThread <filename> index1 index2 …*

2. Reading data from file (40 pts.) -
   2.1. (5 pts.) The program shall read the required numbers to be processed into an array.
   2.2. (10 pts.) The program shall allocate memory for the array that will hold the numbers to be read from the file. The global variable of the array is *recordsToProcess*
   2.3. The program shall store the array size in the variable *numRecords*
   2.4. (10 pts.) If all the numbers in the file must be read then the program shall do it in one read operation (single call fread())
   2.5. (15 pts.) If a list of indices is provided then the program shall navigate to each number and read it (using fread()).

3. Processing (65 points)
   3.1. (5 pts.) The program shall create five threads to process the numbers in the array
   3.2. (35 pts.) Each created thread will take a job (a number from the array *recordsToProcess*), compute the number of prime factors for that number and add it to the global variable *totalNumPrimeFactors*.
   Notes:
   1. Here you need to use the function *numPrimeFactors*().
   2. The two actions 1. Obtaining the next number to process (next job) and 2. Adding the number of prime factors to the total, are critical path. Namely, all threads may try to access the same variables at the same time. Here you will have to use mutex.
   3. Hint obtaining the next job has to be done carefully. Here you can use the number of records in the array of numbers as a stack.
   3.3. (15 pts.) When a tread completes to process a number then the thread will update the total number prime factor found so far.
   3.4. (10 pts.) When there is no more work (namely, all the numbers in the array were processed) then the threads complete

the work and return. The main thread will wait until they all complete and then print the total number of prime factors for all the numbers.
For example: if the program is asked to process the numbers 11 12345671 23 24545 2345 976543 22 44444555 66666777541 93780343 93780383 938026471 then the program shall print something like
Total number of prime factors is 25

4. (5 points) Create a make file for the program. Makefile name should be Makefile2. The program name should be multithread

5. Bonus (10 points)

In this task you will write enhance the program in Task III to use a simple signal SIGUSR1

5.1. Add a counter to your program from Task III, which tracks/collects the number of threads that are still working.
5.2. If signal SIGUSR1 is sent to the program then the program should print how many **threads are still working**. Note, that in order to accomplish this you will need to use global variable within the file scope. Otherwise you will not be able to complete access the information from the interrupt function.