New York University Abu Dhabi

CS-UH 3010 - Spring 2023

Programming Assignment 2

Due: **March 16^{th}, 2023**

Preamble:

In this assignment, you will write a program termed primes that computes all the **prime numbers** found in an interval of integers [a, b]. The task is accomplished with the help of processes enmeshed in a hierarchy. The latter is created using `fork()` calls and subsequently nodes/processes may opt to deploy `exec*()`s to load up diverse executables. The processes participating in the hierarchy help collaboratively carry out the computation in question.

In general, the concurrent operations of all processes in the hierarchy follow a ***divide-and-conquer*** style of action: a number of nodes (processes) are expected to work on their own and identify primes in sub-intervals of [a, b]. Other processes will operate as delegators of work and help ***asynchronously*** collect produced results as well as contribute in compiling a sorted outcome.

Processes in the hierarchy may communicate among themselves using either *signals* or *pipes* to pass information about events and/or results. It is highly desirable that the use of *blocking pipes* be avoided as much as possible [Ker10, Del23].

Overall in this project, you will:

- create a hierarchy of processes by invoking `fork()` multiple times (as needed),
- allow the execution of different piece(s) of code by (some) nodes in the hierarchy by invoking `exec*()` calls,
- use a number of useful system calls including `fork()`, `exec*()`, `read()`, `write()`, `wait()`, `dup()`, `dup2()`, `getpid()`, `getppid()`, `pipe()`, `poll()`, `select()`, `mkfifo()`, etc.

Procedural Matters:

◇ Your program is to be written in `C/C++` and *must run* on `NYUAD's Ubuntu` server `bled.abudhabi.nyu.edu`.
◇ You have to first submit your project via `brightspace.nyu.edu` and subsequently, demonstrate your work.
◇ Dena Ahmed (`daa4-AT+nyu.edu`) will be responsible for answering questions as well as reviewing and marking the assignment in coordination and collaboration with the instructor.

Project Description:

Figure 1 depicts a sample process hierarchy your program may generate whose objective is **to find all prime numbers in range** [a, b]. Overall, the process hierarchy generated is **always of fixed-height** 3 **and a *full n-ary tree* where** n **can take a fixed value** when primes is invoked. Figure 1 portrays a 4-*ary* hierarchy where every parent has 4 children.

In Appendix 1, we provide the `C` code for 2 different implementations of algorithms that find all prime numbers encountered in an interval of integers. Using these 2 implementations, every leaf-node process from $W_0, W_1, W_2, ..W_{15}$ finds the prime numbers in the sub-interval is assigned to work on. These are the *Worker* nodes which find prime numbers. Our 2 independent implementations for finding primes are fitted at the leaf-level modes in a "**circular**" fashion i.e., $W_0, W_1, W_2, ..W_{15}$ work respectively with implementation $1, 2, 1, 2, 1, 2, ...$

As a worker node W_i goes about her work, she finds a prime number within the sub-interval she is assigned to. For each such prime number, W_i has to pass the result to her parent. In this regard, every *Delegator (D)* will receive ideally in an asynchronous fashion all results from its subordinate workers. For example, D_1 will "collect" all results from nodes W_4, W_5, WW_6 and W_7 as the individual results of the workers become available. Any transfer of data between process nodes occurs with the help of *pipes* that you have to design. The passage of information or results between nodes in 2 levels in the tree should take place ideally in an *asynchronous manner*.

Before it completes its work, every W_i node does the following:

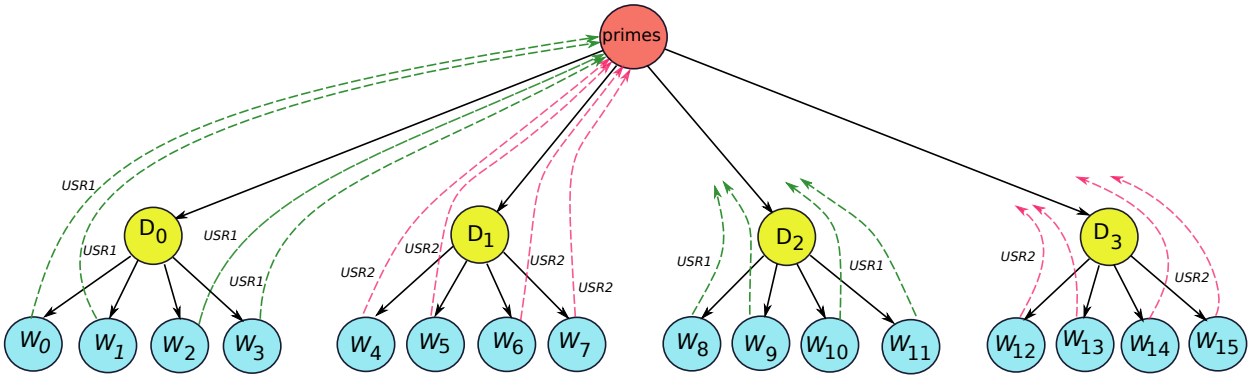

Figure 1: The process hierarchy that collaboratively identifies primes in a in an interval of integers

1. sends to its managing *Delegator* the time (in *msecs*) that took to complete its work of identifying all primes in its sub-interval, and
2. depending on the batch W_i is in, dispatches either a *USR1* or *USR2* signal to the root of the hierarchy –i.e., the executable primes– indicating completion. The 1^{st} group of n workers (e.g., in Figure1, processes $W_0, ..W_3$) will dispatch *USR1*, the 2^{nd} group of n leaf-level node processes will issue *USR2*, the 3^{rd} group of n workers will send *USR1* and so on. All signals will be collected by the root –primes– and their specific numbers will be reported.

In the process hierarchy, there are 3 types of nodes that carry out distinct jobs:

1. **root node**: this is your executable primes. It functions as the *anchor* for the entire hierarchy, but more importantly, it orchestrates the collective work of finding primes within an interval. At first, it creates n **delegator** nodes D_i with i=0..n-1 and asks each one to undertake the job of identifying prime numbers within a sub-interval.

   The delegation of parameters vital to the work of every D_i from the root occurs through a pipe. Similarly, any results collected by D_i are ideally received by the root through a non-blocking piping mechanism that you have to design.

   The *root node* **receives results from all n-delegators, sorts and presents them** on the `tty`. Moreover, it receives performance statistics that have to be reported along with the numbers of *USR1* and *USR2* signals received by worker processes W_i.

2. **delegator node**: this type of node takes in its assigned sub-interval and splits the work either *equally* or *randomly* among n subordinate worker nodes W_i with i=0..n-1 that it first creates.

   Through piping every D_i sends the job to each of its by individual players in the hierarchy subordinate W_i worker nodes and awaits to collect partial results. This collection should also be ideally facilitated with a non-blocking piping mechanism. Once a delegator creates its subordinate n-leaf-level processes, it deploys to each of her worker a distinct independent executable.

3. ***worker node***: each worker node is empowered with a different executable to identify primes; 2 sample such programs are provided in Appendix 1.

   As soon as each worker finds a result, it lets its respective D_i know *ideally* through a non-blocking piping mechanism. Every worker has also to measure its time of execution and dispatch this piece of information to its coordinating D_i.

   Finally, depending on the batch a worker finds itself to be and *just before* it concludes its work, it dispatches either a *USR1* or *USR2* signal to the root.

The main benefit of the aforementioned approach is that finding prime numbers may proceed asynchronously for all sub-intervals and thus, there is an opportunity for your program to run overall faster than a strictly

serial version. All the hierarchy processes in Figure 1 are to run **concurrently and progress should ideally take place in an asynchronous manner**.

Any time there is a need to create a new offspring(s) in the hierarchy a `fork()` has to be apparently involved. If there is need to replace the address space of these new processes with other executables an exec*() of your choice should be invoked along with the proper parameter list (that you have to come up with).

How primes Should be Invoked and Its Output:
Your program could be invoked as follows:

```
./primes -l LowerBound -u UpperBound -[e|r] -n NumOfNodes
```

where `./primes` is the name of your (executable) program and its flags are:

- `-l` indicates the minimun integer or `LowerBound` of the interval,
- `-u` indicates the maximum integer or `UpperBound` of the interval,
- `-e` indicates that work is to be distributed equally among all delegators and workers. If the flag `-r` is used instead, sub-intervals are arranged in a random manner for both delegators and workers, and
- `-n` provides the fixed number of offspring processes every node has to create at the top–2 levels of the process hierarchy.

Your program should report:

- All primes found in a provided interval sorted,
- The minimum, maximum and average time that all workers took to carry out their work, and
- the number of SIGUSR1 and SIGUSR2 signals the *root* –`primes`– has caught (or "seen") just before the end of its execution.

What you Need to Submit:

1. A directory that contains all your work including source, header, Makefile, a readme file, etc.
2. A short write-up about the design choices you have taken in order to design your program(s); 1-2 pages in ASCII-text would be more than enough.
3. All the above should be submitted in the form of "flat" `zip` or `7z` file bearing your name (for instance `AlexDelis-Proj2.tar`).
4. Submit the above tar/zip-ball to `brightspace.nyu.edu`.

Grading Scheme:

Aspect of Programming Assignment Marked	Percentage of Grade (0–100)
Quality in Code Organization & Modularity	35%
Correct Execution for Queries	20%
Addressing All Requirements	30%
Use of Makefile & Separate Compilation	7%
Well Commented Code	8%

Miscellaneous & Noteworthy Points:

1. You have to use *separate compilation* in the development of your program.
2. If you decide to use `C++` instead of plain `C`, you *should not use* STL/templates.
3. Although it is understood that you may exchange ideas on how to make things work and seek advice from fellow students, sharing of code is *not allowed*.

4. If you use code that is not your own, you will have to provide *appropriate citation* (i.e., explicitly state where you found the code). Otherwise, plagiarism questions may ensue. Regardless, you have to fully understand what such pieces of code do and how they work.

5. We would like to see this project be done in **groups of two students**. If you very strongly wish, you could also develop this assignment by working individually. Regardless, your code should run on the LINUX server: `bled.abudhabi.nyu.edu`

6. Pieces of code that are developed by groups should display the full names of both group members in all source files as part of a commented-out-line.

7. You can access the above server through `ssh` using port `4410`; for example at prompt, you can issue: "`ssh yourNetID@bled.abudhabi.nyu.edu -p 4410`"

8. The *Unix Lab* Saadiyat [Uni23] offers limited Unix support.

References

[Del23] A. Delis. `www.alexdelis.eu/3010S23`. Known Credentials, 2023.

[Ker10] M. Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming*. No Starch Press, San Farnsisco, CA, 2010.

[Uni23] NYUAD UnixLab. `https://unixlabnyuad.github.io/`. C2 Building, 2023.

Appendix 1: Algorithms (Programs) for Finding prime numbers

`Program 1: Finding Prime Numbers`

```
#include <stdio.h>
#include <stdlib.h>

#define YES 1
#define NO  0

int prime(int n){
        int i;
        if (n==1) return(NO);
        for (i=2 ; i<n ; i++)
                if ( n % i == 0) return(NO);
        return(YES);
}

int main(int argc, char *argv[]){
        int lb=0, ub=0, i=0;

        if ( (argc != 3) ){
                printf("usage: prime1 lb ub\n");
                exit(1); }

        lb=atoi(argv[1]);
        ub=atoi(argv[2]);

        if ( ( lb<1 )  || ( lb > ub ) ) {
                printf("usage: prime1 lb ub\n");
                exit(1); }

        for (i=lb ; i <= ub ; i++)
                if ( prime(i)==YES )
                        printf("%d ",i);
        printf("\n");
}
```

Program 2: Finding Prime Numbers

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define YES 1
#define NO  0

int prime(int n){
        int i=0, limitup=0;
        limitup = (int)(sqrt((float)n));

        if (n==1) return(NO);
        for (i=2 ; i <= limitup ; i++)
                if ( n % i == 0) return(NO);
        return(YES);
}

int main(int argc, char *argv[]){
        int lb=0, ub=0, i=0;

        if ( (argc != 3) ){
                printf("usage: prime1 lb ub\n");
                exit(1); }

        lb=atoi(argv[1]);
        ub=atoi(argv[2]);

        if ( ( lb<1 )  || ( lb > ub ) ) {
                printf("usage: prime1 lb ub\n");
                exit(1); }

        for (i=lb ; i <= ub ; i++)
                if ( prime(i)==YES )
                        printf("%d ",i);
        printf("\n");
}
```

Appendix 2: An Example of Timing in LINUX

```
#include <stdio.h>      /* printf() */
#include <sys/times.h>  /* times() */
#include <unistd.h>     /* sysconf() */

int main( void ) {
    double t1, t2, cpu_time;
    struct tms tb1, tb2;
    double ticspersec;
    int    i, sum = 0;

    ticspersec = (double) sysconf(_SC_CLK_TCK);
    t1 = (double) times(&tb1);
    for (i = 0; i < 100000000; i++)
            sum += i;
    t2 = (double) times(&tb2);
    cpu_time = (double) ((tb2.tms_utime + tb2.tms_stime) -
                         (tb1.tms_utime + tb1.tms_stime));
    printf("Run time was %lf sec (REAL time) although we used the CPU for %lf sec (CPU time)
        .\n", (t2 - t1) / ticspersec, cpu_time / ticspersec);
}
```