New York University Abu Dhabi
CS-UH 3010
Programming Assignment 2
Due: March 8th, 2021

Preamble:
In this assignment, you will write a program that creates a hierarchy of processes using `fork()` and uses the `exec*()` system call(s) to have nodes accomplish potentially diverse tasks.

The initial program along with all its created offsprings will collectively provide a mechanism that carries out the sorting of a (arbitrarily-long) file of (say taxpayer) records based on a designated attribute. This is to be accomplished in a *divide-and-conquer* style: a number of nodes (processes) are expected each to sort a portion of the records in the provided file and another node (process) will perform the assembly of the final result.

The nodes of the hierarchy that undertake the sorting of the records execute a specific user-provided program (executable) whose objective is to produce the records in either ascending or descending order for the designated attribute. 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, Del21].

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(), getpid(), getppid(), pipe(), dup(), dup2(), 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 `newclasses.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. The overall goal of the hierarchy is to create a sorted listing of all data-records based on a user-designated attribute.

Each data-record consists of a number of lexemes offering information for an individual taxpayer including *resident-id, first name, last name, number of dependents, income* and *postal code*[1]. Data-records are provided to the your program as a data file in *ASCII*-text format.

In the process hierarchy, there are 4 types of nodes that carry out distinct jobs. More specifically:

1. *root node*: this is your program termed `myhie`. Although it functions as the *anchor* for the entire hierarchy, more importantly, it orchestrates the collective sorting task. At first, it creates a single *coordinator* node and the root passes to this node –named *coord*– the file containing the data-records to be sorted, the (variable) number of k workers to be used for the work, the numeric attribute on which sorting should be carried out (i.e., 0, 3 4 or 5), the executable program to be used for sorting data-records and anything else it might be deemed necessary for the successful completion of the job at hand.

   The *root node* also receives SIGUSR1/SIGUSR2 signals dispatched by other processes in the hierarchy and prints the numbers of the signals caught just before the *root* terminates its operation.

[1] these 6 are also called attributes

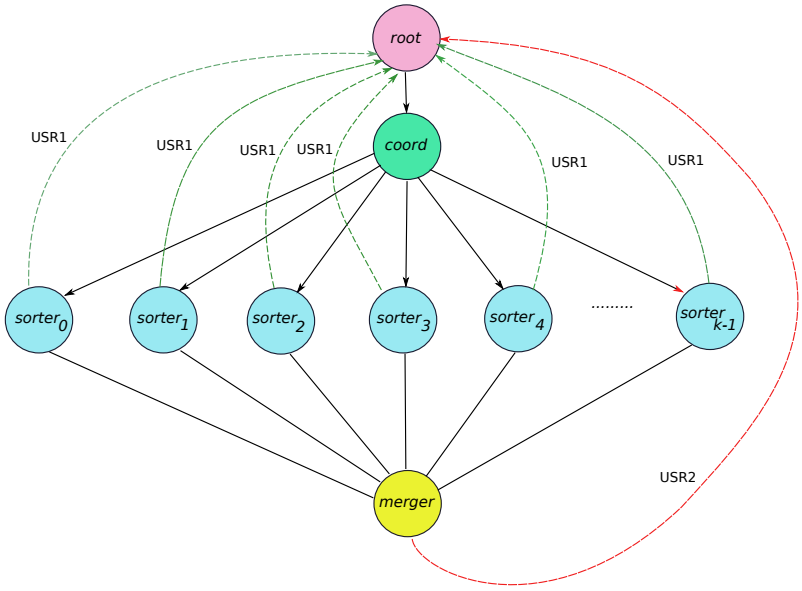

Figure 1: The process hierarchy that accomplishes sorting

2. *coordinator node*: its purpose is to split up the sorting job among k sorters (or workers). The *coord* passes to each sorter the name of the file, the "range" of data-records on which the worker is going to work on, as well as the sorting domain (i.e., which attribute to use for sorting). More importantly, it provides the name of the executable to be used to carry out the sorting. The "range" can be either the same for all *sorters* or it can be randomly set for each *sorter* (provided that the sum of all ranges is equal to the total number of data-records we attempt to sort).

3. *sorting nodes*: to achieve their goal, *sorters* deploy 2 programs of your choice. For instance if you selects *Heap-Sort (HS)* and *Bubble-Sort (BS)* then odd-indexed workers use the *HS* and even-indexed workers the *BS* program. You may elect to implement any 2 sorting algorithms that you find interesting that can function as independent programs and for which *you will have to write the code*. For simplicity, you can implement your 2 sorting programs assuming the sorting occurs only on valid numeric fields only (i.e., not alphanumeric).

   As indicated earlier, each *sorter* is furnished with a set of data-records (possibly designated in the form of a file name/descriptor along with the range of records in the file) and any additional parameter deemed required for its work.

   Every sorter does pass the result of its labor to the *merger* node through a (named) pipe [Ker10, Lov07]. Sorters do pass on their timing statistics to the *merger node* and before they can complete their work send a SIGUSR1 to the *root*.

4. *merger node*: this single node receives (or reads) ordered partial results from all workers through pipes and creates the outcome of the entire sorting process. This node can be *coord*–created and as soon as it generates its outcome to standard output, it

   – prints out statistics that consisting of the time taken for each of the workers to complete their individual work, and
   – sends a signal –say a SIGUSR2– to the *coord* pointing out that the entire workflow is all but completed. As soon as a signal SIGUSR2 is caught by *coord*, it signifies the end of the hierarchy and the graceful release of any acquired resources.

The main benefit of the aforementioned approach is that sorting may proceed asynchronously for all individual segments of data and thus, there is an opportunity to run matters faster. All the hierarchy processes (Figure 1) are to run concurrently and progress should occur 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 your application should be invoked:
Your program could be invoked as follows:

```
./myhie -i InputFile -k NumOfWorkers -r \
           -a AttributeNumber -t Type -o Order -s OutputFile
```

where `./myhie` is the name of your (executable) program, `InputFile` the file name containing the data-records, `NumOfWorkers` is the number of *sorters* to be spawned, `r` is the flag that instructs the program to have workers work on "random ranges" or not-equally-sized batches of data-records, `AttributeNumber` is a *valid numeric-id* that designates the field on which sorting is to be carried out, `Order` is either ascending (`a`) or descending (`d`), and `OutFile` is the file in which the outcome of `myhie` could be saved. There is no pre-determined order with which the above flags can be entered in the command line.

Assume that data files consist of taxpayer records with each record (or line in a text file) featuring 5 attributes: resident-ID, first-name, last-name, income, number-of-dependents, and postal-code. The input file consists of data-records (one-per-line) and its format is as follows:

```
222334444 Kenan Barbieri 3 120000 20742
333445555 Kathy McAllister 1 145000 11201
111223333 Dema Allaster 5 87000 20654
999883333 Lise Williams 2 91560 20742
....... ...
```

In the command-line and for this specific file the `AttributeNumber` may take value 0, 3, 4, or 5 designating corresponding numeric fields in the data-records.

Finally, your program *root*) has to report:

- the time each *sorter* took to sort its batch of records, provide the length of time that the *merger* took to complete its work and furnish the turnaround time required for the entire sorting task to complete,
- the number of SIGUSR1 and SIGUSR2 signals the *root* has caught (or "seen") just before the 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" `tar` or `zip` file bearing your name (for instance `YaserFarhat-Proj2.tar`).
4. Submit the above tar/zip-ball using *NYUclasses*.

Grading Scheme:

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*.

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%

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. The project is to be done individually as the syllabus indicates and should run on the LINUX server: `bled.abudhabi.nyu.edu`

6. 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`"

Timing in Linux: an example

```
#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);
}
```

References

[Del21] A. Delis. `www.alexdelis.eu/abudhabi`. Known Credentials, 2021.

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

[Lov07] R. Love. *Linux System Programming*. O'Reilly, Sebastopol, CA, 2007.