New York University Abu Dhabi
CS-UH 3010 - Spring 2018
Programming Assignment 2
Due: March 24^{th}, 2018

Preamble:
In this assignment, you will write a program termed `mysorter` that initially creates a *binary hierarchy of processes* using `fork()`s and then, it uses `exec*()` system calls to have nodes accomplish *sorting of records* by employing diverse and self-sustained programs (executables).

Once started, the program along with all its created offsprings at different logical layers, will collectively provide a mechanism that carries out the sorting of a (arbitrarily-long) *binary file* of data; we assume that this file features taxpayer records. The sorting in question is accomplished in a *divide-and-conquer* style: a binary hierarchy of processes will be created and its leaf-node processes are expected each to sort a portion of the records provided.

As soon as leaf-level nodes in the hierarchy produce sorted chunks of records, their results are orderly furnished into the internal nodes which collaboratively merge partially results, present their combined outcome to their parent(s), and ultimately to the root.

The leaf level nodes use diverse independent programs to sort the portion of the input data file they are assigned to. The requisite communication between consecutive levels of processes takes place with the help of either simple *pipes* or *named-pipes* (*FIFOs*). Processes may also communicate with the root via *signals*.

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/programs by the elements of the resulting hierarchy through the invocation of `exec*()` calls, and
- use system calls you may deem necessary for the management of LINUX processes including `fork()`, `exec*()`, `read()`, `write()`, `wait()`, `waitpid()`, `poll()`, `getpid()`, `getppid()`, etc.

Procedural Matters:
◇ Your program is to be written in `C++` or `C` and must run on the `NYUAD Linux (CentOS)` Server.
◇ You will have to first submit your project electronically and subsequently, demonstrate your work.
◇ Nabil Rahiman (`nr83-AT+nyu.edu`) will be responsible for answering questions as well as reviewing and marking the assignment.

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-provided condition; records are provided to the root in the form of a binary file [1].

There are three types of nodes that try to accomplish different types of tasks. In particular:

1. *root node*: this is your program and functions as the "anchor" for the entire hierarchy; more importantly, it orchestrates the entire sorting operation.

   At first, it may create a single *splitter/merger* node and the root passes to this node –named sm_0– the range of records whose sorting is to oversee (i.e., the entire range).

   The depth of the tree (and consequently the number of levels internal nodes can be found) is designated by a parameter at the command line of your program (`mysorter`)

2. *sorter nodes*: each of these leaf-level nodes is provided with a set of records (possibly in the form of a file name/descriptor, as well as the range of the records in the file) that it will have to sort.

[1] evidently, you will be provided with the specification of the data records so that you can readily fetch records of interest

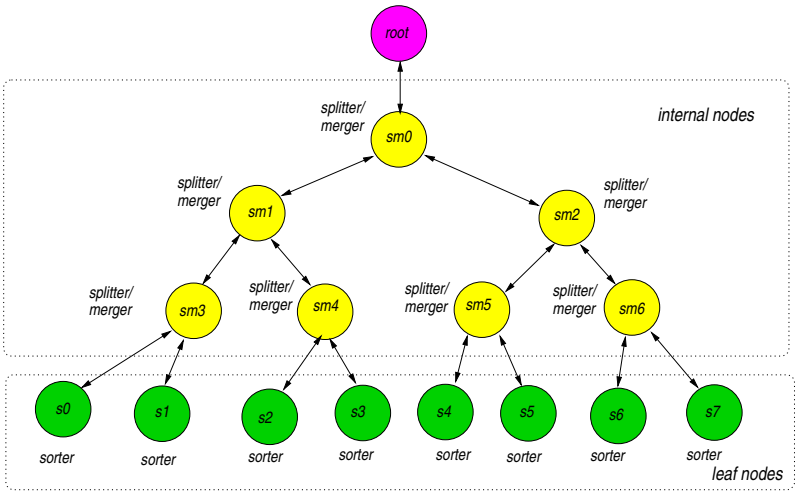

Figure 1: A sample process hierarchy that accomplishes sorting

To achieve their goal, *sorters* deploy different programs and return to their parent their outcome in the form of either *pipe* or *named-pipe* (*FIFO queue*).

Three different *individual* programs are used to do sorting at leaf-level, namely:

(a) *Shell-Sort (SH)*

(b) *Quick-Sort (QS)*

(c) *Bubble-Sort (BS)*

You will have to realize all above three programs as self-sustained executables that receive appropriate *inline parameters* to accommodate the specific needs of `mysorter`. In Fig. 1, leaf-nodes $s_0, s_3, s_6, \ldots$ use *SH*, nodes $s_1, s_4, s_7, \ldots$ use *QS*, and nodes $s_2, s_5, s_{10} \ldots$ use *BS*.

At the end of its work phase, every *sorter* sends a signal to the `root`. However this is done following a small protocol depending of *how* the sorter went about its work: *a)* if the sorter uses *SH*, it dispatches an USR1 signal, *b)* if the sorter is *QS*, it sends a USR2 signal, and finally, *c)* if the sorter is *BS*, it sends a a ALRM signal with appropriately modified service handler.

3. *internal nodes*: every *splitter/merger* process receives from its parent the name/descriptor of a data file, the record range for which the process undertakes the oversight as well as the field on which the sorting is to take place (condition).

   Every internal node may produce exactly *two* children.

   A *splitter/merger* awaits until the results from its own kids are available and then proceeds to merge the partial outcomes. The merged outcome is passed to the parent with the help of a named-pipe. For example, as soon as sm_1 receives partial results from its two children (sm_2 and sm_3), it merges the respective records according to the condition used and passes the result of its work to sm_0.

Any time there is 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. The latter should also contain the depth of the hierarchy.

All processes in the hierarchy of Figure 1 should run *concurrently* and should progress in an *asynchronous manner*.

How Your Application Should Be Invoked:
Your mysorter application can use a number of flags (in-line parameters) that you may deem required. A possible invocation of your application could be:

```
NYUAD-CentOS> ./mysorter -d TreeDepth -f RecsFile -a AttrNum -o OutFile -r
```

where `./mysorter` is the name of your (executable) program, `TreeDepth` is the depth of the process hierarchy, `RecsFile` is the binary file that contains the data records and `AttrNum` is a valid numeric ID that designates the attribute on which sorting is to be carried out.

The *root* will be responsible for producing the combined sorted output. If a `-o` flag is present the output of `mysorter` will be directed to the file named `OutFile`. Otheriwise, the standard output of the *root* (i.e., the `tty` used) will be used to display the results.

When used, the (random) flag `-r` indicates that each *sorter* should work on a batch of data records whose number is random; each batch however contains consecutive records from `RecsFile` and we assume that there are no common records among all batches leaf-nodes work on. If the flag `-r` does not appear, we can assume that all leaf-nodes work on equally-populated batches of records; this population is $\sim N/2^{TreeDepth}$ with N representing the number of records found in `RecsFile`.

You may opt for using additional flags of your choice in the invocation line of your program.

`TreeDepth` may range from 1 to 6; 0 is the degenerate case and it not of much interest. If `TreeDepth` is 1, then we have only 1 internal node and 2 sorters. In the case of Figure 1, we have depth 3, $\sum_0^2 2^i$ internal nodes (*splitter/merger*s) and 2^3 leaf-nodes (*sorter*s).

We assume that the data file consists of taxpayers records with each record having 4 attributes: taxation number (`int`), first-name (`char(25)`), last-name (`char(35)`) and taxable income (`float`) in *AED*. More specifically, the file could appear as:

```
999883333 Lise Chen  91560.34
888123456 Hayat Cherniekov 345678.23
222334444 Kulvider Kanitkar 320000.00
456678811 Mohamed Ali 1300023.88
333445555 Kathy Sarif 74500.03
```

In the command-line, `AttrNum` may take value between 0 and 3 for the above type of data file.

Finally, `mysorter` has to report the time each *sorter* took to sort its batch of records, the time each *merger* took to complete its work and the turnaround time required for the entire sorting job. Moreover, the *root* should report how may of the three distinct types of signals (USR1, USR2, and modified ALRM) it has received before it terminates operation. Before termination, the *root* should report how many and of what kind signals has received in an effort to ascertain how many signals were actually delivered. The number and type of missing signals should be reported.

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 `tar` or `zip` file bearing your name (for instance `AlexDelis-Proj2.tar`).
4. Submit the above tar/zip-ball using *NYUclasses*.

Grading Scheme:

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

Noteworthy Points:

1. The project is to be done ***individually***.
2. You **have to use *separate compilation*** in the development of your program.
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 and how such pieces of code do.

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);
}
```