CSC112 Operating System – Assignment 2 – Programming Part

Due [All Parts]: Wednesday, March 25th, 2020, 11.59 pm

Only submissions on Blackboard will be graded

What to Turn In (for all parts):

A single zip file named `Firstname_Lastname_As2.tar.gz` submitted to Blackboard under `Assignment2`, which contains the following files:

(a) Written.txt (or **.pdf**)

(b) Programming.txt (or **.pdf**)

(c) Nthreads.c

(d) drawingStraws.c

(e) stats.c **[OPTIONAL – EXTRA CREDIT]**

Commands for generating the ".tar.gz" file (assuming that your current directly is the parent folder of As2/):

```
tar -cvf Firstname_Lastname_As2.tar  As2/
gzip Firstname_Lastname_As2.tar
```

The purpose of this assignment is to help you review the **notion of threads** and get you familiar with the POSIX Thread library functions that are related to thread management as well as refresh your memory on C programming and pointers.

Programming Part

[P0 – SETUP]. You can use any Linux distribution that has the POSIX library (**pthreads**). Your assignments will be graded on the VM provided on Blackboard so make sure that at least you check your programs there before submission.

- Any design choices or any answers (sample outputs) you need for this part you should write into a file **ProgrammingPart.txt (or.pdf)**

[P1.-EXPLORE YOUR VM (or LINUX-BASED MACHINE)] Use the `top` and `ps` commands on the Linux terminal to find information about your VM. Read the manual pages to see the different options you can use for both of them.

To support your answer to the following questions, you should provide the **output (screenshots)** of the commands you used every time (include a screenshot of the command and its corresponding output (or unless otherwise instructed *copy-paste* the exact output in your **Programming.txt** (or **.pdf**))

a. Approximately how many **processes** and **threads** are currently present on your system?
   - Provide a screenshot of your findings.

b. Which process (PID or processname) has consumed the **most CPU time** so far? (Use the "`top -o cpu`" command to help you answer this question).
   - Provide a screenshot of your findings.

c. Open a text editor application (e.g., `gedit, vim, …`) and use the `ps` command to find the `PID` of this process. What is the **priority** of this user process assigned by the OS?
   - For example, if the PID of the editor process was 861, then you could see the priority assigned to it by using the command: `ps -lp 861`

d. What is the **priority** assigned to user processes on your system? You need to explain/justify **how** you came up with your answer.
   - What is the **default priority** assigned to user processes on your system? You need to explain **how** you came up with your answer.

e. Show all the processes running on your system by **you**. (*Hint*: the –u option will be useful.)
   - Provide a screenshot of the command and your findings.

f. Identify a **system process** and mention its **priority**. You need to explain **how** you came up with your answer.

g. Use the command `ulimit` to find the maximum allowed user processes on your (virtual) machine.

[P2. NTHREADS] Write a multi-threaded program (**`Nthreads.c`**) that does the following:

a. The **master (main)** thread "handles" the command-line input `N` (should be `< 10`) and creates **N threads** inside a `for`-loop.
   - The number **`N`** should be provided at program invocation as a command line option, as shown below:

```
$ ./Nthreads 5
$ ./Nthreads 8
```

     - Reminder: you can access the command-line arguments within your code, using the `argc` and `argv` arguments of your **main** function:

```
int main(int argc, char *argv[])
```

   - The **master** thread should be doing the **proper error checking** for the command line option `N`.
     - Make sure that one-command line argument is provided at program invocation.
     - Make sure your program doesn't crash if no command-line argument is provided.
     - Make sure the command-line option `N` provided is in the valid range.
     - In all these cases, your program should print relevant error-messages (describing the problem) and gracefully terminate.

b. All spawned threads should be executing the **same** function `drawNumber`, which takes as argument the **thread number** (NOT the thread-id).
   - Each thread should **print** its **corresponding number** and a **number** randomly selected from the ones in the **global array** `numbers`:
     - `numbers = {2,6,32,11,12,50,55,64,4,100}`
   - So for example, the Thread0 and Thread1 could print the following messages:

```
Thread 0: drew 55
Thread 1: drew 32
```

   - Insert the statement "`usleep(rand()% 1000000);`" ***before*** each thread `printf` statement, to make threads sleep a random amount of time (and see more variation in the output).

c. Make sure your code does **not** include **any restrictions** on the **order** the spawned threads execute (and print).
d. You should **not** include **any other restrictions** on the **order** that the two spawned threads are executing
e. Make sure that **all** spawned threads are guaranteed to execute to completion. Program should terminate after this.

✠ Your program should produce *similar* output to the **sample outputs** shown below (assume `N=3` in this example):

   - Sample Output 1:

```
Thread 0: drew 55
Thread 1: drew 32
Thread 2: drew 12
All threads completed!
```

   - Sample Output 2:

```
Thread 1: drew 6
Thread 2: drew 50
Thread 0: drew 6
All threads completed!
```

✠ Compiling your program (as shown below):

- `$ gcc -g -Wall -pthread` **`Nthreads.c -o Nthreads`**
  - Note that the "`-pthread`" option links the **pthreads** library to your program. Depending on the flavor of Linux (or Mac OSX) kernel you use, you may need to use the option "`-lpthread`" instead.
  - Make sure your program **compiles** and **runs** (without warnings).

✠ Include in your submission for this part:

- Your program **Nthreads.c**
- Submit **sample outputs** of at least **six (6) different executions** of this program (in your **Programming.txt** or **.pdf**) with the **same** and **different** `N`'s.
  - To run your program multiple times from command line, type the following command at the **Terminal** prompt `$` (in the example below N was `3`):

    `$` **`for((n=0;n<10;n++)); do ./Nthreads 3; done`**
  - Make sure you include the output of an execution, where there is no command-line argument at program-invocation or `N` is not in the acceptable range.

[P3. DRAWING STRAWS] Make a *copy* of your **`Nthreads.c`** program and name the new file **`drawingStraws.c`**

a. As in **P2**, your **master** thread should "handle" the command-line input `N` and create **`N` spawned** threads, where N gets its value at **program invocation** as a **command line option**, as shown in the example below (where `$` is the Terminal's command line):
   - `$` **`./drawingSraws`** `8`

   - Once again, make sure your program does **proper input-validation** for `N`.
   - The **master** thread should be creating the `N` threads from before.
     - All `N` threads should be executing the `drawStraw` function (see details below).
   - An **extra** thread (N+1)th should be spawned for deciding and **printing** the **loser** among the previous `N` spawned threads.
     - This thread should be executing the thread-function, named `find_loser`.
       - `void * find_loser(void * param)`
     - The thread with the **smaller** randomly picked number from the array `numbers` is the "loser" thread.

b. Rename the `drawNumber` function to `drawStraw`:
   - This function will be executed by the *first* `N` spawned threads (not the thread that will decide the final winner).
   - Change your code from before so that **no two threads** are **randomly** picking the **same element** from the **global** array `numbers = {2,6,32,11,12,50,55,64,4,100}`.
   - Note: You will need to describe/explain in 2-3 sentences in your **Programming.txt** (or **.pdf**) your implementation for this.

   - For example, make sure that you do **NOT** have output like the following (i.e. Sample Output 2 (in [**P2**])):

```
Thread 1: drew 6          → Thread1 should not be allowed to pick the same number(6) as Thread0!
Thread 2: drew 50
Thread 0: drew 6
All threads completed!
```

c. You should **not** include **any other restrictions** on the **order** that the spawned threads are executing, apart from the ones indicated in the description.

✠ Your program should produce *similar* output to the **sample outputs** shown below (assume `N=3` in this example):

- Sample Output:

```
Thread 2: drew 6
Thread 3: drew 32
Thread 1: drew 4
Thread 0: drew 55
Thread 4: drew 12
Loser: Thread 1 drew the smallest straw!
All threads completed!
```

→ Last line should be printed by the **master** thread

✠ Include in your submission for this part:

- Your program **drawingStraws.c**
- Short explanation of how your code **guarantees** that **no two** threads are picking the same element from the global array `numbers`.
- Submit **sample output** of at least **six (6) different** executions of this program (in your **Programming.txt** or **.pdf**) with the **same** and **different** `N`'s.
  - To run your program multiple times from command line, type the following command at the **Terminal** prompt `$` (in the example below N was `3`):

    ```
    $ for((n=0;n<10;n++)); do ./drawingStraws 3;  done
    ```

  - Make sure you include the output of an execution, where there is no command-line argument provided at program-invocation or `N` is not in the acceptable range.

[EXTRA-CREDIT: OPTIONAL]

[P4. STATS] Write a multithreaded program (**`stats.c`**) for computing statistics on all the numbers (integers) from two different arrays.

a. The **master (main)** thread should process the sizes of the two arrays and spawn **six** more threads.

- You should be implementing two **global** arrays of integers, but the size of these arrays will be determined at program invocation.
- The sizes of the two arrays are passed as command-line arguments at program invocation.
  - For example, the following command, would invoke your program and create two arrays of size **10** and **20** respectively.

    ```
    $ ./stats 10 20
    ```

  - ***Suggestion***: Start your implementation with one array of fixed-size, say 10, make all the threads work for this one array and then work with the second array and the rest of the requirements.
- **Master** thread should include code for proper **input-validation checks** on these two numbers.
- M**aster** thread should spawn two threads (one for each array) that will be taking care of the ***filling*** of the two arrays with **random** numbers (see details of `fill`() function below).
- M**aster** thread should spawn two *more* threads (one for each array) that will be taking care of the ***summation*** of the numbers in each array (see details of `sum`() function below).
- M**aster** thread should spawn two *more* thread that will be computing and returning to the master thread the **minimum** of each of the two arrays. The master thread should be **printing** the absolute minimum of the numbers in both arrays.

- **Master** thread should be computing the **average** of the two arrays using the results of the **summation**-threads.

b. A separate thread should *fill* **each array** *with numbers*. Each of these two spawned threads should execute the same function **`fill`**`()`, that should have as parameter the **size** of the array that it will be working on.
   - Function **`fill`** should be filling the array with random numbers in range `1..100`.

c. A separate thread should be spawned to compute the ***sum*** of the numbers in **each array**.
   - Both of these threads should execute the same function **`sum`**`()` but work on a different array and they should both **return** to **master**-thread the result of these summations.

d. A separate thread should be spawned to compute the ***min*** of the numbers in **each array**.
   - Both of these threads execute the thread-function **`minimum`**`()` and it should return to **main** the corresponding *minimum* number for each array.

e. You should **not** include **any other restrictions** on the **order** that the spawned threads are executing, apart from the ones indicated in the description.
   - **Note**: the goal is to run as many threads in parallel as possible.
   - Make sure you **justify/explain** in your **Programming.txt** (or .pdf) for this part, the restrictions (if any) in the order the threads described above will execute.

✠ Include in your submission for this part:
   - Your program **stats.c** (for this part you will be graded also on the **efficiency** and **good-coding** practices of your programs**,** i.e**,** if you are allocating more memory than you should etc**.**
   - Submit **sample output** of at least **six (6) different** executions of this program (in your **Programming.txt** or **.pdf**) with the **same** and **different** `N`'s.