Instruction

In this project, you will implement a Process ID (PID) manager in C for Linux. An operating system's PID manager is responsible for managing process identifiers. When a process is first created, it is assigned a unique PID by a PID manager. The PID is returned to the PID manager when the process completes execution, and the manager may later reassign this PID. Process identifiers are discussed more fully in Section 3.3.1. What is most important here is to recognize that process identifiers must be unique; no two active processes can have the same PID. You can find the basic description of this programming project from Exercise 3.20 and 4.20. You need to further develop your solution ensuring that the data structure used to represent the availability of process identifiers is safe from race conditions. Use Pthreads mutex locks, described in Section 5.9.4 is a must. A start code is provided at the bottom of the page.

Step 1.

Implement the following APIs (functions) for obtaining and releasing a PID:

• int allocate_map(void)—Creates and initializes data structure for representing pids; returns -1 if unsuccessful, 1 if successful (5 pts)
• int allocate_PID(void)—Allocates and returns a PID; returns -1 if unable to allocate a PID (all pids are in use) (5 pts)

You may use any data structure of your choice to represent the availability of process identifiers. One strategy is to adopt what Linux has done and use a bitmap in which a value of 0 at position i indicates that a process id of value i is available and a value of 1 indicates that the process id is currently in use.

Step 2.

Writing a multithreaded program (at least 100 thread) that tests your PID manager implemented in Part 1. Ensure that the data structure used to represent the availability of process identifiers is safe from race conditions using Pthreads mutex locks, described in Section 5.9.4. Write down the number of line where you use these methods in your PDF file, (10 pts)

pthread_mutex_init
pthread_mutex_lock
pthread_mutex_unlock

In your test program you will need to create 100 threads causing a racing condition. One racing condition example can be each thread constantly requests a PID in multiple iterations. Upon having a PID a thread sleeps for a random period of time, and then releases the PID. (Sleeping for a random period of time approximates the typical PID usage in which a PID is assigned to a new process, the process executes and then terminates, and the PID is released on the process's termination.) Output to the shell to reflect this random acquiring/releasing PID process. Finally, add this line to your output. Make a screenshot of your terminal window once your test program finishes. (10 pts)

```
printf("***DONE***\n");
```

Tips:

1. Use the following constants to identify the range of possible PID values:

   ```
   #define MIN PID 300
   #define MAX PID 350
   ```

2. On UNIX and Linux systems, sleeping is accomplished through the sleep() function, which is passed an integer value representing the number of seconds to sleep.

3. Since you will use pthread library in Linux environment again you must include pthread.h and compile the source code with -lpthread option because GCC doesn't do auto-linking of libraries triggered by header inclusion. Here is an example,

   ```
   gcc  -lpthread -o testpid test.o pid.o
   ```

   At the bottom of the start code there is Makefile which includes all commands you need to compile the 3 files.

Submission Instruction:

Combine all source code to a single .c file and should be named as yourCSUNID.c.
The PDF file comprising numbers of lines and your screenshots needs to be named following this format,
YourCSUNID_YourLastName.pdf
Both files need to be submitted here in Moodle.

Submission failed to meet the submission requirement will not be graded. Grade may be forfeited.