CS342 Operating Systems – Fall 2022
Project #2 –Synchronization and Scheduling

Assigned: Oct 27, 2022.
Due date: Nov 14, 2022, 23:59. Document version: 1.1

This project will be done in groups of two students. You can do it individually as well. You will program in C/Linux. Programs will be tested in Ubuntu Linux. Please start as soon as possible.

Part A (80 pts):

In this project you will implement a multi-threaded program, called **`cfs`**, that will simulate processes and their scheduling in a system. The scheduling algorithm will be CFS (completely fair scheduler). The simulated system will have one CPU and one runqueue. For simplicity, we will assume that simulated processes will not do I/O operations. Therefore, each process will have a single cpu burst, which will be also the length of the process (i.e., required cpu time).

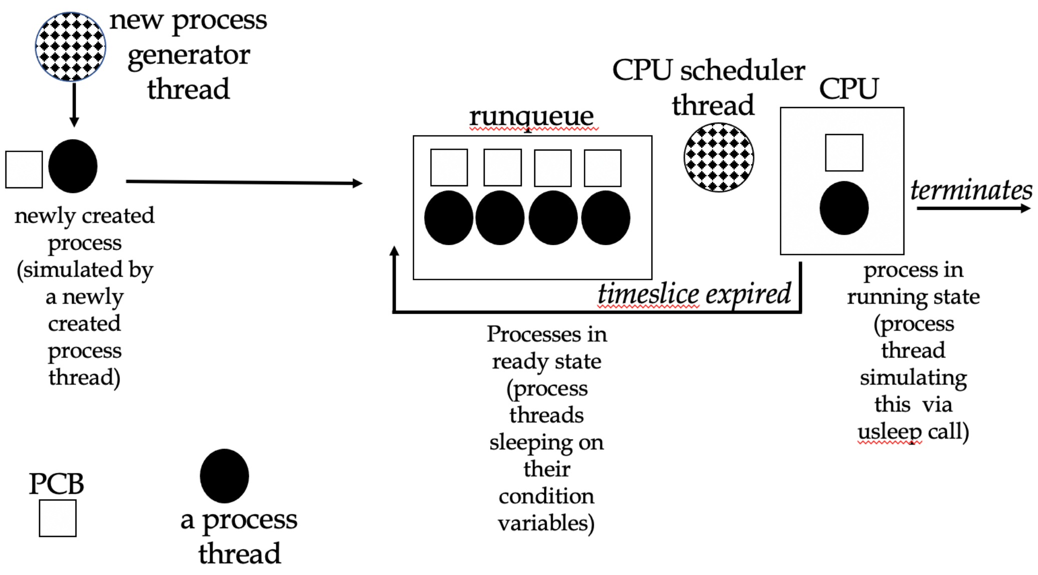
Processes will be simulated with threads. We will call these threads as *process threads* or simply *threads* or *processes*. Hence, if we want to simulate the execution of *N* processes in the system, then there will be *N* threads created. Each process thread will have its own condition variable to sleep on, while it is waiting in the runqueue for CPU to be assigned. Besides the process threads simulating processes, we will have two control threads: a generator thread and a scheduler thread.

The *generator thread* will create process threads. Before creating a process thread, it will wait for some amount of time (interarrival time) using `usleep()` function. Then it will create a process thread by using `pthread_create()` function. Then it will wait for some amount of time again. It will repeat this behavior until desired number of process threads are created. Then no more process threads will be created.

The *scheduler thread* will schedule the process threads for CPU. It will also have an associated condition variable, and when not scheduling, it will sleep on its condition variable. When woken up, the scheduler thread will check if scheduling is needed. If not needed, it will go to sleep again. Otherwise, it will select a process thread from the runqueue according to CFS algorithm. Then it will wake up the selected thread by signaling on its condition variable. After scheduling a process thread, the scheduler thread will sleep again. The scheduler thread is woken up in the following cases:

1. when the running process thread terminates and leaves the CPU.
2. when the timeslice for the running process thread expires and it gets added to the runqueue again.
3. when a new process thread is created and added to the runqueue.

A *process thread* starts its life when it is created by the process generator thread (see figure). A process thread will simulate a process. Each process thread will have a C structure allocated. We can call this structure as PCB. It will keep information about a process thread, like its pid, state, corresponding thread id, process length, total time spent in the CPU, a condition variable to sleep on, etc. After being created, a process thread will add itself (its PCB) to the runqueue and start sleeping on its condition variable until it is selected for CPU. Before sleeping, it can signal the scheduler thread. Note that while accessing the runqueue, or any other shared data, we need to get the related lock so that no race conditions will happen.


new process generator thread
newly created process (simulated by a newly created process thread)
runqueue
CPU scheduler thread
CPU
terminates
timeslice expired
Processes in ready state (process threads sleeping on their condition variables)
process in running state (process thread simulating this via usleep call)
PCB
a process thread

When selected for CPU, a process thread is woken up by the scheduler thread. The woken up thread will remove itself (its PCB) from the runqueue and will start using the CPU. Before using the CPU, it will first calculate its timeslice and its remaining time to finish. Let *x* be the minimum of these two values. Then, running in CPU for *x* ms will be simulated by calling the `usleep()` function and sleeping for *x* ms. For example, if *x* is 30 ms, then the thread will call `usleep(30000)` and will sleep for 30 ms. Then we consider the process has used the CPU for 30 ms. When the process leaves the CPU, i.e., the process thread returns from `usleep()`, it will update its virtual runtime. If its remaining cpu time for the process did not become zero, it will add itself to the runqueue and start sleeping on its condition variable again. Otherwise, it will terminate. Before termination, a process thread will deallocate its resources and will destroy its synchronization variables (for example, its condition variable). After leaving the CPU, a process thread will signal the scheduler thread.

The program will have the following command line parameters.

cfs C minPrio maxPrio distPL avgPL minPL maxPL distIAT avgIAT minIAT maxIAT rqLen ALLP OUTMODE [OUTFILE]

or

cfs F rqLen ALLP OUTMODE INFILE [OUTFILE]

The first argument after the command line will specify the whether the workload parameters will be taken from command line (C) or from an input file (F).

If workload parameters will taken from command line, minPrio and maxPrio are numbers between `-20` and `19` (priority/nice values). For a new process thread, a priority number will be selected between minPrio and maxPrio randomly (uniform random). If minPrio is equal to maxPrio, then all process threads will have the same priority.

The distPL parameter is used to specify the length of a newly created process. If distPL is "`fixed`", then process length (cpu burst length) will be avgPL. If distPL is "`uniform`", then a uniformly distributed random value (in ms) between minPL and maxPL will be selected. In this case the avgPL parameter will not be used (it can be set to 0 at command line). If distPL is "`exponential`", then an exponentially distributed random value will be chosen as process length. The λ parameter of the exponential distribution will be equal to 1/avgPL.

The distIAT parameter is used to specify how long the process creator thread will sleep (interarrival time) before creating another process thread. If distIAT is "`fixed`", then interarrival time will be avgIAT. If distIAT is "`uniform`", then a uniformly distributed random value (in ms) between minIAT and maxIAT will be selected. In this case, the avgIAT parameter will not be used (it can be set to zero at command line). If distIAT **is** "`exponential`", then an exponentially distributed random value will be chosen as interarrival time. The λ parameter of the exponential distribution will be equal to 1/avgIAT.

An exponentially distributed random value as the length of a new process can be generated as follows. 1) Set the parameter λ of the exponential distribution to be equal to 1/avgPL. 2) Select a random number u between 0 and 1 (*uniformly* distributed). 3) Let $x = ((-1) \cdot ln(1-u))/\lambda$. 4) If x is between minPL and maxPL we are done. Round x to an integer (ms) value and it will be the length of the new process. If x is not in the desired range, go to step 1 and repeat until x is between minPL and maxPL (we are *approximating* exponential distribution). We can use the same method for selecting an exponentially distributed random value for interarrival time (IAT) at process creation thread.

The rqLen parameter specifies the maximum number of processes we can have in the runqueue (not including the running process). If that number is reached, process creation thread will not create a new thread, but will wait for some time (interarrival time) and will try again.

The ALLP parameter specifies the total number of processes (process threads) that will be created by the process generator thread.

OUTMODE specifies what should be printed out to the screen while simulator is running. It can be 1, 2, or 3. If OUTMODE is 1, nothing will be printed out. If it is 2, the next running thread will print out the current time (in ms from the start of the simulation), its pid, its state (RUNNING), and how long it will run in the CPU (ms) until it will be preempted. It should do this before going to sleep (using `usleep`). An example output can be as follows (use this format):

```
3450 7 RUNNING 50
```

If OUTMODE is 3, then a lot of information will be printed out while the simulation is running. The format is up to you. But please print related information (in a separate line) for the following events at least: new process created, process will be added to the runqueue, process is selected for CPU, process is running in CPU, process expired timeslice, process finished, etc.

Example invocations are shown below.

```
./cfs C -20 19 exponential 300 100 1000 exponential 400 100 2000 20 100 1
./cfs C -10 10 uniform 0 100 1000 exponential 600 100 2000 10 200 2 out.txt
./cfs C 3 15 fixed 100 100 100 fixed 200 200 200 10 200 2
./cfs F 20 100 2 infile.txt
./cfs F 20 100 2 infile.txt out.txt
```

If INFILE parameter value is specified, then process lengths and interarrival times will be read from an ascii text file with name INFILE. A line starting with PL will indicate a process length (ms) and a priority value for a process. A line starting with IAT will indicate an interarrival time (ms) between two processes. An example input is shown below. You must follow the same format. Note that in this case, the process generator thread immediately creates a process thread, without waiting (sleeping) some amount of time (i.e., we do not have an IAT before first process creation - arrival).

```
PL 1000 2
IAT 3000
PL 300 -2
IAT 1000
PL 700 0
IAT 5000
PL 2000 5
IAT 300
PL 1000 -10
.....
```

When the simulator is started, time will be assumed to be 0 (ms). You can record the real time when the simulator has started, and then, whenever you need, you can measure the elapsed time (in ms) between the current real time and the recorded start time. To obtain the real time you can use the `gettimeofday()` function. The elapsed time calculated as t (ms) at a moment will be the current time in ms at that moment.

At the end of the simulation, when the last process (thread) has left the system (terminated), the program will write the following information to the screen before terminating. For each process, it will write the pid, time process arrived (ms – from the start of the simulation), time process finished (ms), the priority of the process, total time process has run in the CPU (ms) (i.e., process length), total time process waited in the runqueue, turnaround time (ms), and number of context switches experienced (>= 0). At the end of the output, you will print the average waiting time. An example output can be as follows (use this format strictly):

```
pid  arv dept  prio  cpu   waitr  turna  cs
1    10  5010    2  1000   2000   5000   4
2    20  7020   -3  1500   1500   7000   2
3    40  8000    5  2500   7960   10     7
avg waiting time: 3000
```

The numbers above are just to give the format. They don't have to make sense.

The OUTFILE argument is optional. If it is specified, all output will go to an ascii text file whose name is specified as the value of the OUTFILE argument. In this case, nothing will be written to screen.

You will use POSIX pthreads mutex and condition variables. You will use locks (mutex variables) while accessing shared structures or variables (like the runqueue). Be careful about deadlocks. Write your code in such a manner that there will be no deadlocks and race conditions.

Part B – Experiments (20 pts):

Set the distribution of process lengths and interarrival times to be exponential. Fix a large enough rqLen value. Run your program and measure the average waiting time in the runqueue for processes. Repeat the experiment with various values of the rate parameter of the exponential distribution (for process length and interarrival time). Plot graphs or create tables. Try to interpret the results. Write a report and have it in PDF form in the folder that you tar and gzip and upload.

Submission:

Put all your files into a directory named with your Student ID. If the project is done as a group, the IDs of both students will be written, separated by a dash '-'. In a `README.txt` file, write your name, ID, etc. (if done as a group, all names and IDs will be included). The set of files in the directory will include `README.txt`, `Makefile`, and program source files. We should be able to compile your program by just typing `make`. No binary files (executable files) will be included in your submission. Then tar and gzip the directory, and submit it to Moodle.

For example, a project group with student IDs 21404312 214052104 will create a directory named "21404312-214052104" and will put their files there. Then, they will tar the directory (package the directory) as follows:

```
      tar cvf 21404312-214052104.tar 21404312-214052104
```

Then they will gzip the tar file as follows:

```
      gzip 21404312-214052104.tar
```

In this way they will obtain a file called 21404312-214052104.tar.gz. Then they will upload this file into Moodle. For a project done individually, just the ID of the student will be used as file or directory name.

References:

1. Operating System Concepts, A. Silberschatz et al. Wiley.
2. CS342 Lecture slides.
3. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt
4. https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-lottery.pdf
5. https://developer.ibm.com/tutorials/l-completely-fair-scheduler/
6. https://lxr.linux.no/linux+v5.17.4/kernel/sched/fair.c (source code of CFS)
7. https://lxr.linux.no/linux+v5.17.4/kernel/sched/sched.h (source code of CFS - data structures)
8. https://lxr.linux.no/linux+v5.17.4/lib/rbtree.c (red black tree implementation).

Tips and Clarifications:

- The simulator will keep a pid counter. When a process thread is started, it will be assigned the next integer pid. The first pid is 1. Pids will *not be recycled*.
- The process generator thread and the CPU scheduler thread will be created when the simulator program is started. You may want to do some initialization before you create these two threads. These control threads will be terminated before the program terminates.
- You need to initialize any mutex or condition variables you are using. And you need to destroy them as soon as you are finished with them.
- CFS minimum granularity value (`min_granularity`) will be 10 ms.
- CFS targeted latency value (`sched_latency`) will be 100 ms .
- Any timeslice value calculated will be rounded to an integer value (ms). In CFS, timeslices are calculated dynamically. For a process that is selected to run next in the CPU, the timeslice will be calculated just at that time, before running the process (according to the formula given in the slides). If the calculated timeslice is smaller than `min_granularity`, then timeslice will be set to be equal to `min_granularity`.
- We will use the Linux kernel CFS weight table (shown in cs342 lecture slides - CFS), which is called `prio_to_weight` table.
- Virtual runtime value of a process will be 0 when it is created. It will be updated each time the process leaves the CPU, according to the formula given in the slides. You should keep the virtual runtime value as a high-precision floating pointer number (i.e., double). The unit will be ms.

- You can implement the runqueue as a linked list or as a read-black tree. If you wish, you can use Linux kernel red-black tree implementation. It is in lib/rbtree.c in kernel source code tree. If you choose to implement a linked list, then it should be your implementation.
- Maximum value of rqLen can be 100.
- Max value of ALLP is 2000. Min value is 1.
- The minimum value of process length can be 10 ms. The maximum value of process length can be 100000 ms. minPL and maxPL can be any integer value (in ms) between these two numbers (minPL <= maxPL).
- The minimum value of interarrival time can be 10 ms. The maximum value can be 100000 ms. minIAT and maxIAT can be any integer value (in ms) between these two numbers (minIAT <= maxIAT).
- Start early, work incrementally.
- "`man pthreads`" will you give information about POSIX threads API. You can also find a lot of information on Web about POSIX threads API.