CSE 473 Project 1: Thread Schedulers

(due Tue, Oct 11 by 11:59PM thru Angel - NO EXTENSIONS WILL BE GIVEN!)

Please direct all your project-related questions/clarifications to the TAs, either in person or by email. Piazza is also available for queries (but no exchange of code is permitted).

Description

In this project, you will implement four thread schedulers for a uni-processor environment:

1. First Come First Served (FCFS) - no preemption
2. Shortest Remaining Time First (SRTF) - with preemption
3. Priority Based Scheduling (PBS) - with preemption, and
4. Multi Level Feedback Queue (MLFQ) - with preemption and aging.

Deliverables

Your scheduler should implement the following interfaces in C programming language, and all these four functions should be provided in a new file called scheduler.c.

`void` **`init_scheduler`**`(int` **`sched_type`**`);`

This function will be invoked once in the beginning for initializating anything that you may want in your code. The parameter **`sched_type`** will contain values from 0 to 3, denoting 0-FCFS, 1-SRTF, 2-PBS, 3-MLFQ.

`int` **`schedule_me`**`(float` **`currentTime`**`, int` **`tid`**`, int` **`remainingTime`**`, int` **`tprio`**`);`

This function should implement the actual scheduler functionality based on the initialized scheduler type in the **`init_scheduler`**`()` function. It takes 4 input arguments from the calling thread. They are:

1. **`currentTime`** indicates the time at which the call to **`schedule_me`**`()` is made by some thread. No two **`schedule_me`**`()` invocations will have the same **`currentTime`** value. However, it is possible that when a thread is executing your **`schedule_me`**`()` function, another thread invokes the same function, albeit with a higher **`currentTime`**. Hence your implementation should be multi-thread safe. This **`currentTime`** can take float values.
2. **`tid`** indicates the thread identification number.
3. **`remainingTime`** indicates the amount of remaining CPU time requested by the thread for its current CPU burst. The **`remainingTime`** will always be an integer number.
4. **`tprio`** indicates the priority of the thread. **`tprio`** can take a value between 1 and 5, 1 being the highest priority.

Your scheduler should maintain a global current time value which should be updated based on the **`currentTime`** value passed by the **`schedule_me`**`()` invocations of different threads. You need to return from this function ONLY when this thread gets the CPU (otherwise it will need to wait until then). When it eventually returns from **`schedule_me`**`()` function, the function should return the global current time value to the thread. Note that we are simplifying your implementation

so that time quanta are always assigned at integral values, and these quanta will never start at non-integral time values (e.g. a thread will never relinquish the CPU for I/O at non-integral time values, even if it arrives in the system at non-integral time values). This function is further explained below.

`int` **`num_preemeptions`**`(int` **`tid`**`);`

This function should return the total number of preemptions incurred by the thread whose **`tid`** is passed as argument, over its entire lifetime until the point this function is being called.

`float` **`total_wait_time`**`(int` **`tid`**`);`

This function should return the total number of cycles that a thread (denoted by its corresponding **`tid`**) waits in any READY queue over its entire lifetime until the point this function is being called.

At every integral time unit, a thread that still needs the CPU, will come and call your **`schedule_me()`** function, along with the 4 mentioned parameters (the **`currentTime`** is the time at which this call is being made and the **`remainingTime`** is the remaining amount of time that this thread needs on the CPU for the rest of its CPU burst). **`schedule_me()`** needs to maintain a READY queue, so as to determine whether the calling thread should get the CPU or not as per the specified scheduling disciplines. If the thread is not the next to get the CPU, then it will wait/block it until that point. Else, it should return with the current time. The thread will then resume and run for exactly 1 integral time unit before making the next **`schedule_me()`** call. Scheduling decisions should be made only at these integral time units. Note that these could also be pre-emption points based on the type of scheduler.

The global current time maintained by your **`schedule_me()`** should be updated based on the **`currentTime`** being passed from the threads, and also the elapsing of time (every 1 integral time unit for every call in the CPU burst). For instance, say T1 arrives at time 1.1, and calls **`schedule_me()`** with a **`remainingTime = 2`**. Initially the globalTime will be 0. When T1 calls **`schedule_me()`**, it should get the CPU at globalTime = 2 and return from the function with this value (since scheduling decisions are made only at integral values, it is as though this thread arrived at time 2). It will subsequently make two more invocations to **`schedule_me()`**, at times 3 and 4, with a corresponding remaining time of 1 and 0 respectively. **`schedule_me()`** should return back time values of 3 and 4 respectively at these 2 points.

When a thread invokes **`schedule_me()`** function and the CPU is currently being used by another thread, your scheduler should block the calling thread until the next integral time unit (using a pthread routine). The blocked thread will be made to wait in the READY queue.

Your scheduler should not use any system timer interrupt functionality - the only notion of time for your scheduler is the **`currentTime`** indicated by the threads. When a thread has finished its CPU requirements, it invokes the **`schedule_me()`** function with **`remainingTime = 0`**, after which the thread is no longer part of the READY queue - it has either left the system or gone for I/O. Return from the **`schedule_me()`** function indicates that the thread is either granted the CPU or has completed its CPU burst (when it calls the **`schedule_me()`** function with **`remainingTime = 0`**). Since multiple threads could be simultaneously accessing common data structures (like the READY queue), make sure you use pthread synchronization primitives at appropriate places to ensure atomicity. pthread calls should also be used to block threads until they are supposed to get the CPU.

Resources Provided

The links below direct you to various files/man pages that are vital for this implementation.

Boiler-plate source files

You are provided with a file called **project1.c** which creates threads using the `pthread` library and invokes the above four functions. You are **NOT** to make any edits to project1.c - all your code goes in to scheduler.c. The application (the **`main`** function in 'project1.c') will take an integer command line argument specifying the type of scheduler to be used ([0-3]: 0-FCFS, 1-SRTF, 2-PBS, 3-MLFQ as mentioned above) which will be passed to the **`init_scheduler`** function. The file 'project1.c' parses an input file which indicates the thread arrivals and their execution times (burst length) and issues corresponding **`schedule_me()`** calls. 'project1.c' also spits out an output file which indicates the actual schedule order (Gantt chart) of the threads.
The file, **project1.c** can be downloaded by clicking on the link.

Compiling and Running

To compile your code (project1.c and scheduler.c) we provide a sample Makefile here. Download and run the Makefile script using the command 'make'. This will result in the generation of the binary 'out'.
To run the binary use the command: `out [scheduler_type] [input_file]`
where [scheduler_type] is 0,1,2 or 3 indicating FCFS, SRTF, PBS and MLFQ respectively.
[input_file] is the path to the input file.

Debugging your code with test cases

For testing/debugging your work, a set of input files and their corresponding output files are provided **here**.

Input files are available in the directory 'TestInputs' which can be used by all schedulers. Each line in the input file is represented in the following format,

[arrival time][tab][thread id][tab][burst length][tab][priority]

The output is divided into two parts:
a. **Gantt chart** where each line is represented in the following format,

[start time] - [end time]: [thread id]

followed by b. **Thread wait and preemption stats** where each line is represented in the following format,

[thread id][tab][number of preemptions][tab][total wait time]

The output files corresponding to the input files are available in the directory 'TestOutputs' for each scheduler type.

The test cases are NOT limited to these files. There will be additional hidden test cases on which your code will be evaluated.

Details on the scheduling policies

Below is a brief description of the four kind of schedulers that you will implement:

1. **FCFS:** This scheduling scheme is non-preemption based and schedules the threads based on who arrives first at the scheduler. The threads are scheduled for the entire `remainingTime` duration without preemption in between. Assume no two threads arrive at the same time.
2. **SRTF:** This scheduling scheme is preemption based and schedules the threads based on who has the least `remainingTime` value. If two or more threads have the same `remainingTime` value, then the thread with least `currentTime` is chosen for scheduling. Further note that when a thread with a shorter execution/remaining time arrives in the system at a non-integral time value, it still waits for the next integral boundary before pre-empting the currently running thread.
3. **PBS:** This scheduling scheme is preemption based and schedules the threads based on who has the highest priority. If two or more threads have the same `tprio` value, then the thread with least `currentTime` is chosen for scheduling.
4. **MLFQ:** This scheduling scheme is preemption based with multi-level feedback queues. The READY queue should be implemented as a multi-level queue with 5 levels. The time quantum for the 5 levels (starting from the highest level) are 5, 10, 15, 20, and 25 time units respectively. Threads initially join the READY queue at the highest level. When a thread completes its time quantum on a particular level, it will be moved to the next lower level. Your scheduler should select threads from a lower level queue only when there are no threads in any of the higher levels. A thread entering the READY queue at the highest level will preempt threads at the lower levels at the next integral time unit. Within a level, you should use FCFS to select the threads. You will use Round-robin at the last level.

Note that `tprio` is only used in PBS, and is a don't care value for the other 3 schemes.

Some useful pthread calls:

You are allowed to use the pthread (**pthread.h**) thread library to implement the scheduler functionality. You may find the following pthread functions useful when implementing the scheduler. You can click on the functions to view their man pages.

int pthread_cond_init(pthread_cond_t *, const pthread_condattr_t *);
int pthread_cond_signal(pthread_cond_t *);
int pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *);
int pthread_cond_destroy(pthread_cond_t *);
pthread_t pthread_self(void);
int pthread_mutex_init(pthread_mutex_t*, const pthread_mutexattr_t *);
int pthread_mutex_lock(pthread_mutex_t*) ;
int pthread_mutex_unlock(pthread_mutex_t*);
int pthread_mutex_destroy(pthread_mutex_t*);

Additional Information

An illustrative figure to describe the operation of the PBS scheduler is provided here.

NOTE: You are NOT supposed to modify project1.c, or modify any of the input/output formatting mechanisms. The inputs and sample outputs are given just for illustrative purposes to test your

code. You can create more extensive input files for further testing. The TAs may surprise you with several other test inputs during the demo and your routines should still work.

Please stay tuned constantly to the angel page for the exact and latest interface functions you need to implement, their arguments, test programs, examples, documentation/manuals and announcements.

The projects (including a **brief** report) is due by 11:59PM on the designated day and the reports+programs should be turned in using the Submission Guidelines. NO EXTENSIONS WILL BE ENTERTAINED.

You need to set up an appointment with your TAs to demonstrate your implementation. You can work in teams (at most 2 per team - they could be across sections), or individually, for the project. You are free to choose your partner but if either of you choose to drop out of your team for any reason at any time, each of you will be individually responsible for implementing and demonstrating the entire project and writing the project report by the designated deadline.
If you have difficulty finding a partner, give your name to the TAs who will maintain an online list of students without partners to pair you up. Even though you will work in pairs, each of you should be fully familiar with the entire code and be prepared to answer a range of questions related to the entire project. It will be a good idea to work together at least during the initial design of the code. You should also ensure that there is a fair division of labor between the members.

No form of academic dishonesty of any kind will be tolerated. All projects should be individually undertaken by each team to the best of that team's ability, without seeking any external help. No exchange of code is permitted between teams. Severe penalties will be imposed not just in the project grade, but the dishonestry will be reported to the University's Office of Student Conduct for possible further academic sanctions.

Test files

Full tarball

Submission Guidelines

Piazza