李彦廷
B07902088

## I. Design

The general design for the scheduler, in terms of the control of time passed (timer), is such that it is handled by the child processes (tasks); this will be elaborated in this section. Some important assumptions this program makes are that the system is 64-bit with at least two cores.

First, in anticipation of a large number of input tasks (that I later learned is capped at 20), I divided the tasks into separate buckets based on their arrival time, chaining them together with the linked list implementation in linux kernel (simply because I like the modularity of its design). Scheduler makes necessary adjustments, such as signal masks, CPU affinity, and the likes, before fork()-ing all the tasks and putting them on suspend with sigsuspend(). Then the appropriate scheduler for the appointed policy is called.

In general, the timer of the scheduler is either controlled by the currently executing child process (task), or idling in the scheduler itself. Thus when one is executing the time unit loop given in the specification, the other would be suspended. As such nearly all timer control is on the running task, the scheduler only truly runs the time unit loop when it has no tasks to execute (with tasks arriving later, of course); otherwise, the scheduler is suspended with sigsuspend() while the task runs its assigned number of time unit loop. This means all tasks are always "awake" running for a predetermined amount of time, so in the case of preemptive shortest job first, pipes are used to communicate the 'time slice' (so to speak) given to the task.

FIFO: sequentially, in order of arrival, awake child processes up to execute for their full duration of execution time.

RR: continuously looping through the linked list of buckets containing unfinished processes, each given one RR time slice (t = 500); completed tasks and empty buckets are removed from the appropriate list for the validity of the loop check.

SJF: O(N) sweep through input tasks, up until buckets whose tasks have arrived, to seek the task with minimum execution time and run it for its full duration; queue is used for tasks that are currently eligible for CPU time

PSJF: similar to SJF; the scheduling events to check for in the scheduler is when new tasks have arrived. As such, the child currently selected will execute for either its full execution time (or the remaining execution time) or until the next time point where new tasks shall arrive (and the smallest out of all eligible tasks is again checked), whichever one occurs first. Pipes are used to tell the tasks how long to execute for before it needs to return control back to the scheduler.

## II. Kernel Version

4.14.25 (same as HW1)

## III. Theoretical and Experimental Results

There are both cases of the experimental results being faster (less units run than predicted) or slower (more units run than predicted) than the theoretical schedule. Some potential causes may be: a) the use of virtual machine, and as such the processor needs to upkeep both ends; b) overhead in my implementation of the scheduler, such as the maintenance of certain data structures (like the aforementioned buckets) and some minor uncertainties in the arrival of signals (a great majority of my implementation depends much on - at times frequent - signaling between processes, which may incur some uncertainties depending on when the signals are sent); c) the inherent scheduling policy in Linux, given that what we are doing are still based on a working model (Linux and its CFS) and not working directly with the actual scheduling of processes, the inherent CFS may still affect the scheduling of our simulation-processes.

<u>Side note</u>: the format of the output files differs slightly in the addition of a single whitespace (my sincerest apology) in "[Project 1]" instead of "[Project1]". This whitespace has been removed in the system call .c file included, but please note that the output .txt files are "[Project 1]" (I ran short of time to recompile the kernel for this whitespace inconsistency).