Assignment 1 - Submission Questions

SID: 500687806

1.4.2

The Round Robin (RR) scheduler that was implemented focuses on task fairness in the sense that it evenly distributes CPU time amongst all active processes within the system. The RR scheduler achieves this fairness by setting up a queue for all tasks based on a FIFO principle. Each process is given a fixed time slice, and if the process is still executing after the depletion of its allocated time it is sent to the back of the queue. The scheduler keeps track of the progress of a tasks time slice by updating it every time the scheduler is triggered by a system timer interrupt. On the ARM architecture the tick rate is 100Hz per second (Love, 2005) which is equivalent to 100 interrupts per second. It is important to choose a time slice value that is not too short as there is significant overhead if the kernel has to perform many context switches. It is also important to not have the time slices too long as it will reduce response time and make for a bad user experience. A time slice of 100ms was chosen as a happy medium, and was chosen based on the linux real-time scheduler.

A doubly linked list was used to store all active tasks utilising the inbuilt list data structure available from *list.h*.

The following functions were implemented in the comp3520 scheduler class to make it operational.

`init_comp3520_rq` - This function is used to initialise the run queue. As the RR only uses a single double linked list, it is as simple as initialising a linked list head element, which is to be used as the entry point into the RR queue.

`enqueue_task_comp3520` - This function is called whenever the core scheduler is ready to add a new task to the run queue. It adds tasks to the back of the queue using the macro *list_add_tail* provided by list.h. Variables holding the total number of running processes are incremented, and the time slice of 10 is allocated to the task. It is important to note that 10 is used to represent 100ms, as there are 10 interrupts per 100ms on an ARM machine.

`dequeue_task_comp3520` - This function is called whenever the core scheduler is ready to remove a task to the run queue. A check is made to ensure that it is in fact on the

run queue before attempting to remove it. It simply uses the list macro *list_del* to achieve this. The total number of running processes is decremented.

`pick_next_task_comp3520` - This function is called when the core scheduler is ready to pick the next task to run. It first check to ensure that there are active processes to select from, if there is not then this function returns NULL indicating that there are no tasks to run and the system can idle. If there are tasks, then the function simply returns the next task that is at the front of the queue.

`task_tick_comp3520` - This is the core function which is used to determine whether a task should be swapped out. On each call to this function the task's time slice is decremented by 1.

- If the time slice has not reached 0 then the function returns.
- If the time slice has been completely used (has hit 0) then the task is moved from to the back of the queue using the macro *list_move_tail.* The *resched_curr* function is then invoked to inform the core scheduler that it should swap tasks at its earliest convenience.

The following shows important structures of the .h files. Each schedule entity contains a node, run_list, which is used for positioning in the queue, a flag to say if it is on the run-queue and its time_slice. The run queue has the queue itself and a count of the total active processes.

```
struct comp3520_sched_entity {
  struct list_head run_list;
  bool on_rq;
  unsigned int time_slice;

  // Don't worry about this
  struct sched_statistics statistics;
};

struct comp3520_rq {
    struct list_head queue;
    unsigned int nr_running;
};
```

1.5.2

1. Design Decisions

The improved design implements a multi level feedback queue (MLFQ). It uses a total of 10 queues to store active tasks. When a task is first enqueue it enters the top level queue. Once a task uses its time slice up then it is moved down a queue level. This repeats until it is at the last queue. The schedule entity and run-queue of the new design can be seen below.

On top of the previous design, each schedule entity contains an extra field to keep track of which queue they are currently on. The run-queue has an array of queues which is aligned to another array which keeps track of how many active jobs are in each queue.

```
struct comp3520_sched_entity {
  struct list_head run_list;
  bool on_rq;
  unsigned int time_slice;
  unsigned int queue_no;

  // Don't worry about this
  struct sched_statistics statistics;
};

struct comp3520_rq {
    struct list_head queue[10];
    unsigned int nr_running[10];
};
```

The main differences to the scheduler class are as follows:

`enqueue_task_comp3520` - This function is hardly modified from the RR scheduler. All new tasks are put onto the tail of the lowest level priority queue.

`dequeue_task_comp3520` - The task is removed from the current queue it is on, and the running number of active jobs is decremented from that queue

`pick_next_task_comp3520` - The queues are looped from highest priority to lowest priority. If there are currently active tasks in a priority queue, then the first element of this queue is selected to be the next running task. If there are no tasks in any of the queues then NULL is returned.

`task_tick_comp3520` - Once a queues time slice has been used up, then the task is moved to the tail end of the priority queue that is a level lower. If the task is at the lowest level then it is moved to the tail of that queue (as it can’t move down any

further). The number of running tasks is incremented and decremented from the effected queues.

2. **Critique**

- At the moment the MLFQ does not refresh priority. This can lead to some batch style processes being target of starvation if there is a large number of interactive tasks. To improve this in the future I could add a refresh at some interval, that either boosts the priority of certain long lasting tasks, or moves all tasks to the top of the queue. This notion has been implemented in other MLFQ style systems, like the staircase scheduler (Ishkov, February 2015).
- The CFS scheduler is superior to the scheduler I have implemented in many ways.
    - CFS uses a load balancing system to keep track of virtual run-time, this is far more sophisticated than simply moving up and down queues, and allows for a much fairer allocation of scheduled time.
    - The CFS scheduler also makes use of a time based Red Black tree, which is significantly faster than traversing through a linked-list system.

Works Cited

Love, R. (2005). *Linux Kernel Development Second Edition.* Sams Publishing. Retrieved from http://books.gigatux.nl/mirror/kerneldevelopment/0672327201/ch10lev1sec2.html

Ishkov, N. (February 2015). *A complete guide to Linux process scheduling.* Tampere: University of Tampere School of Information Sciences Computer Science.