COL331 Assignment 2: Hard Track

Aniruddha Deb
2020CS10869

April 2023

1 Deadline Monotonic and Rate Monotonic Scheduling

Implementing a sched class from scratch for deadline monotonic scheduling was infeasible, and hence we use a kthread-based scheduling algorithm implemented on top of the linux real-time fifo scheduler. The scheduler uses a linked list of tasks sorted in decreasing order of priority along with a dispatcher thread to schedule tasks whenever they are registered/yielded.

The main process control block structure that we use is built upon the linux task_struct, with additional fields.

```
struct dm_pcb {
    struct list_head list_node;
    struct list_head wait_queue_node;
    struct task_struct* linux_task;
    struct hrtimer timer;
    unsigned int period;
    unsigned int deadline;
    unsigned int duration;
    unsigned int state;

    // negative of deadline
    int priority;
    // used to tiebreak between tasks with the same priority
    int priority_epsilon;

    struct sched_param old_sched_params;
};
```

Here:

- period, deadline and duration store the task's period, deadline and duration as specified in the syscall
- state stores the current state of the task: one of running, sleeping or ready
- priority and priority_epsilon are used to determine the task's priority. priority_epsilon is zero for all tasks who have not allocated resources, and is only used while tiebreaking between tasks by the scheduler when priority inheritance happens.

1.1 registering and deregistering

One common function, `__register_dm` is used to register both deadline monotonic and rate monotonic tasks. Rate monotonic tasks are built upon Deadline monotonic tasks, and use the same scheduler and task queue, with the only difference being that the deadline and period are the same for these tasks.

For the schedulability check, we implement the algorithm in [1]: this is a neccessary and sufficient check for checking whether a given set of tasks is deadline schedulable, and this was tested with testcases where tasks had both same and different periods.

To register a task, it's PCB is initialized and it is inserted into the list at an appropriate position. The schedulability check is then run: if it fails, the task is removed. If not, a timer is registered with the task which triggers a scheduling callback with a frequency equal to the period of the task. This scheduling callback sets the task state to ready, and wakes up the dispatcher kernel thread to update the currently running task.

To deregister a task, the task is removed from the task list, and it's original CFS priority is restored. It's PCB is deleted, and any resources that it acquired are relinquished.

1.2 yield

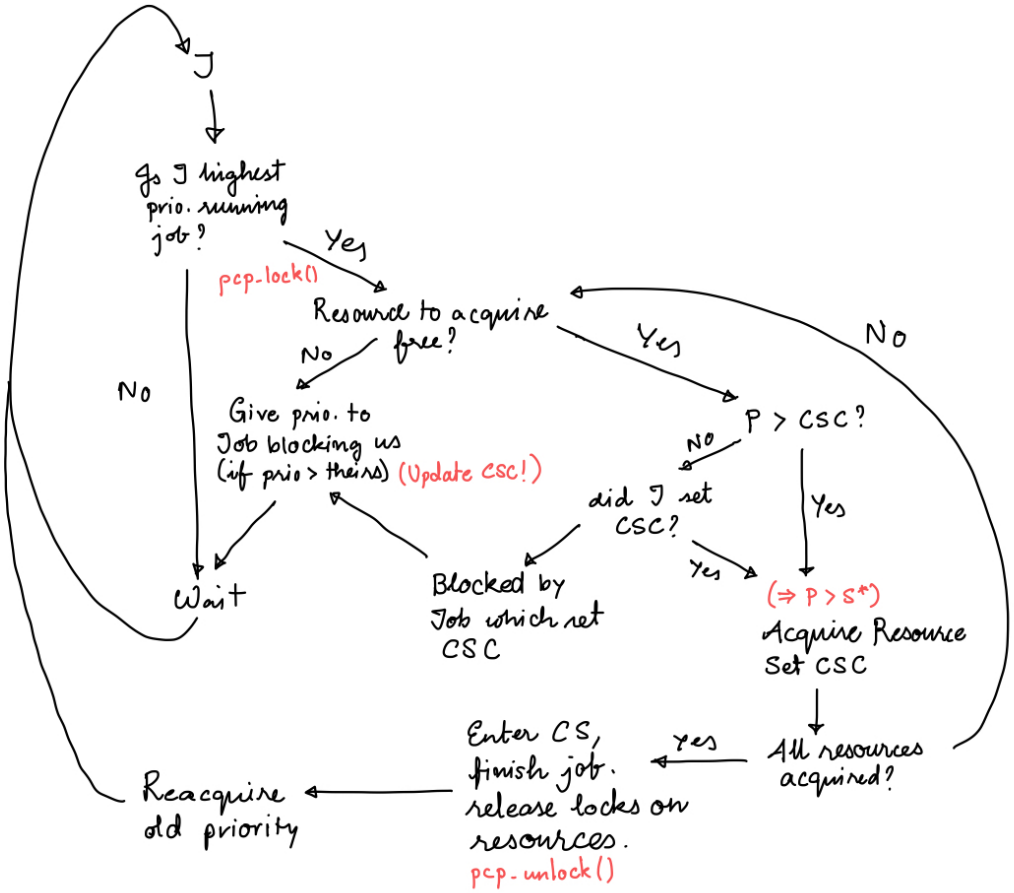
The yield call sets the state of the process to sleeping and the currently running task to NULL. It then wakes up the dispatcher thread to reschedule the tasks.

1.3 Additional implementation details

Poor accuracy was observed with using a `timer_list`, so a `hrtimer` was used instead.

2 Priority Ceiling Protocol

The priority ceiling protocol was described in [2], and is implemented according to the following flowchart:



We assume a fixed set of resources (10), all of which have the following structure:

```
struct resource_t {
    struct mutex mtx;
    int wait_queue_len;
    struct list_head wait_queue;
    int rid;
    int ceiling;

    struct dm_pcb* owner;
    int owner_priority_before_acquisition;
    int owner_priority_epsilon_before_acquisition;
};
```

The fields are self-explanatory. The `owner_priority_before_acquisition` fields store the priority the task had before acquiring the resource, so that we can invert the priority once the task is done using the resource and leaves it's critical section.

2.1 Changes to vanilla DM scheduling

Priority inheritance required adding the epsilon term to the priority class: since we're relying on the dispatcher to schedule our threads, if a task A with a high deadline has acquired a resource and a task B with a low deadline is waiting on it, then A acquires B's priority (deadline). Now, the task scheduler needs a way to schedule A before B everytime, even if A is after B in the queue. We cannot add an epsilon to A's priority, as it is an integer and may clash with some other priority. Hence, we use another field for storing the epsilon, and compare this value if the two priorities of tasks are equal while scheduling.

A simpler way to do this would have been to set priority = -deadline*64, and this would support an epsilon of 0 to 63 (basically treat priority as a fixed point number. However, having a separate field supports a larger number of overlapping tasks.

2.2 Resource mapping and starting PCP

The resource map syscall is used to determine the priority ceiling of a resource. The priority ceiling is defined as the maximum priority of any job that may acquire the resource while it is running. To compute this, we need to know the set of jobs that will acquire that resource beforehand. This information is provided using this method.

The start PCP syscall is used to initialize the resources' locks and lists and set the timers of all the tasks that take part in PCP.

2.3 Locking and Unlocking

Most of the logic of PCP lies in locking and unlocking, and these two algorithms are implemented according to the above flowchart.

We may be blocked when we try to lock (acquire) a resource in two cases:

1. The resource is in use by another job J. We are then blocked by job J, and we transfer our priority to J if it's greater than J's current priority.
2. The resource is free, but we are under the current system ceiling (and neither have we set the current system ceiling). In this case, we are blocked by the job that set the current system ceiling. We can't transfer our priority as it would anyway be less than (or equal to) the job that set the current system ceiling, so we transfer one epsilon to it (if our priority is equal to it) so that it may be scheduled always.

If we are blocked, we register ourselves in the waitqueue in order, go to sleep by sending a SIGSTOP and calling the scheduler to reschedule. If not, we can acquire the resource (after setting the old priority fields in the resource appropriately) and continue executing.

For unlocking a resource, we first update the resource statistics, get back to our old priority before acquiring the resource and then unlock the mutex. We then choose the first element in the wait queue (this has the highest priority among the waiting processes), and assign the resource to it. Since only one task can run at any time and we have just stopped running after releasing this resource, either we hold more resources and will continue to run or this new task will run. We let the scheduler decide this: the new task will be scheduled after being given this resource. If it fails to acquire any other resource it needs which we hold, it would be blocked and we will be rescheduled by the scheduler in the other program's PCP lock syscall.

3 References

1. Audsley, Neil C., et al. "Deadline monotonic scheduling." (1990).
2. Sha, L., et al. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization." IEEE Transactions on Computers, vol. 39, no. 9, Sept. 1990, pp. 1175–85. DOI.org (Crossref), https://doi.org/10.1109/12.57058.