COL331 Assignment 2

DMS/RMS scheduling and PCP

Pranay Shah

Compiled on: August 19, 2023

1 Introduction

2 Deadline Monotonic Scheduling

In DMS, the task τ_i arrives periodically with time period P_i, execution time T_i and deadline D_i. The dealine monotonic scheduling algorithm is a static scheduling algorithm, in which the priorities are assigned to tasks based on their relative deadlines.

2.1 Implementation

We simulate the DMS scheduling in the following way: First add all the registered processes to the real time scheduling class. This will prevent other CFS user processes to intervene. Now, keep all the processes except the one with the highest priority in the "TASK_STOPPED" state. Now, the kernel scheduler wont switch out this process assuming that there are no processes which are not registered for DMS in a higher kernel scheduling class.

We maintain two main queues implemented using kernel `rb_tree` structs: one for all tasks registered under dms scheduling and another for all the tasks which are now in the "ready" state. The ready queue is a subset of the registered queue which corresponds to the list of tasks which have arrived and waiting to execute. All the processes in the registered queue are in the "TASK_STOPPED" state except for one (which is the currently running process).

2.2 The PCB

We create a Process control block for each task registered under dms. The fields of which are as shown in the following snippet.

```
1  struct sched_dms_entity{
2      struct rb_node ready_node;
3      struct rb_node registered_node;
4      pid_t task_pid;
5
6      unsigned int dms_exec_time;
7      unsigned int dms_deadline;
8      unsigned int dms_period;
9      bool ready; // false -> yielded, true -> ready
10     bool isfirst;
11
12     struct timer_list timer;
13 };
```

Where `ready_node` and `registered_node` are `rb_node`s in our regsitered and ready runqueues.

2.3 Timer Callbacks

Each sched entity has a timer struct associated with it. We can use them to execute a call back function periodically - each execution corresponds to the arrival of the next instance of a task. The call back is as shown in the following snippet:

```
void timer_callback(struct timer_list *data)
{
    struct sched_dms_entity *sdmse;
    unsigned long flags;
    sdmse = container_of(data, struct sched_dms_entity, timer);
    if(!find_task_by_vpid(sdmse->task_pid))
        return;

    spin_lock_irqsave(&dms_rq.rq_lock, flags);
    if(!sdmse->ready){ // if true, not completed last task; handle this
        dms_enqueue_ready(sdmse);
        sdmse->ready = true;
    }
    spin_unlock_irqrestore(&dms_rq.rq_lock, flags);

    dms_schedule();
    mod_timer(data, jiffies + msecs_to_jiffies(sdmse->dms_period));
}
```

Here, we change the state of the process to the ready state and insert it to the ready queue. Then we call the `dms_schedule` method which context switches (if necessary) to the new highest priority process. We also set the next time when the call back will be run using `mod_timer`.

2.4 The Scheduling function

The `dms_schedule` method is as shown in the following snippet:

```
void dms_schedule(void)
{
   struct rb_node *left;
    struct sched_dms_entity *nxt_se;
    struct task_struct *nxt_task, *cur_task;
    unsigned long flags;

    spin_lock_irqsave(&dms_rq.rq_lock, flags);
    left = rb_first_cached(&dms_rq.ready_tree);
    if(!left)
        goto unlock;
    nxt_se = container_of(left, struct sched_dms_entity, ready_node);
    if(nxt_se->task_pid == dms_rq.curr_pid)
        goto unlock;
```

```
    nxt_task = find_task_by_vpid(nxt_se->task_pid);

    cur_task = find_task_by_vpid(dms_rq.curr_pid);
    if(cur_task)
        send_sig(SIGSTOP, cur_task, 1); // check if task has been killed

    dms_rq.curr_pid = nxt_se->task_pid;
    send_sig(SIGCONT, nxt_task, 1);

unlock:
    spin_unlock_irqrestore(&dms_rq.rq_lock, flags);
    /* printk("Done Scheduling\n"); */
}
```

There are multiple things going on here, but the main gist is that we fetch the highest priority ready task from the ready queue, check if its the currently running process. If not, we stop the running process and start the next running process. The stopping and starting is done by sending the SIGSTOP and SIGCONT signals respectively.

2.5 Yielding

When a process is done with its task, it calls the sys_yield syscall, the Implementation of which is as shown:

```
SYSCALL_DEFINE1(yield, pid_t, pid)
{
    ...
    send_sig(SIGSTOP, cur_task, 1);

    if(sdmse->isfirst){
        mod_timer(&sdmse->timer, jiffies + msecs_to_jiffies(sdmse->dms_period));
        sdmse->isfirst = false;
        return 0;
    }
```

When a process yields, we stop it and dequeue it from the ready queue (if it was present in it). If a dequeue occured then rescheduleling is required and hence we call `dms_schedule` in the end.

Thus the cycle is complete. Tasks arrive using timer callbacks, get schduled according to their Deadlines, finish their execution and yield.

2.6 Schedulability test

We perform the Schedulability test according to the algorithm provided by Audsley. For this, we go through all the items in the registered runqueue in the sorted order. The following snippet shows the implementation.

```
    t = 0;
    for(i = 0; i < dms_rq.nr_registered; i++){
        t += exec_times[i];
        go = true;
        while(go){
            interference = 0;
            for(j = 0; j < i; j++)
                interference += ((t + periods[j] - 1) / periods[j]) *
                    exec_times[j];
            if(interference + exec_times[i] <= t)
                go = false;
            else
                t = interference + exec_times[i];

            if(t > deadlines[i]){
                return false;
            }
        }
    }
```

2.7 RMS

We can clearly see that RMS is a subset of DMS and hence, we route sys_rm_register to sys_dm_register which takes care of all processes scheduled in the same way.

3 Priocity Ceiling Protocol

3.1 Global variables

We maintain a global list of all resources that have been mapped by at least one DMS registered task. Corresponding to each resource we store some information regarding which processes have mapped to it its ceiling priority.

3.2 Integrating with DMS

To integrate this protocol with our DMS Implementation, we first notice that the priority of a task may change over time (inheritance) and there can be two tasks with the same priorty. To maintain the ready and registered runqueues in DMS, we need to modify the comparision functions given to the corresponging `rb_tree`s. To this extent, we introduce a new struct `sched_prio` as shown in the snippet below.

```
struct sdmse_prio{
#ifdef CONFIG_PRIO_CEIL_PROTOCOL_DMS
    struct list_head inherited_priorities_entry;
    // history stuff
```

```
    bool inherited;
    unsigned int inherited_while_holding_rid;
#endif
    pid_t pid;
    unsigned int dms_deadline;
};
```

When the boolean value inherited is false, the comparision of two prio structs works as in the DMS case: compare deadlines and use pids to break ties. The problem arises when both pid and dealine are equal for the two prio structs. This impilies that one of them must be inherited and that prio has higher priority.

We also add, the following new fields in our PCB for dms:

```
#ifdef CONFIG_PRIO_CEIL_PROTOCOL_DMS
    struct list_head locked_resources_head;
    struct list_head inherited_priorities_head;
#endif
```

The inherited_priorities_head corresponds to a stack of inherited priorities. each entry has a corresponging resource id that the task held when performing the inheritance. This is used later on when releasing a resource. The sched prio of a sched dms entity is found in the following way:

1. Check if the `inherited_priorities_head` list is empty or not.
2. If empty then set inherited false and return sdmse prio with original pid and deadline.
3. otherwise, The current prio is the top entry in the list. return that.

Corresponding function is:

```
inline struct sdmse_prio get_sdmse_prio(struct sched_dms_entity *a){
    struct sdmse_prio cur_prio = {.inherited = false, .pid = a->task_pid,
        .dms_deadline = a->dms_deadline};
    if(!list_empty(&a->inherited_priorities_head)){
        cur_prio = *list_first_entry(&a->inherited_priorities_head, struct
            sdmse_prio, inherited_priorities_entry);
        return cur_prio;
    }
    return cur_prio;
}
```

Using these methods and making small changes in other functions, we can continue using our dms system and build PCP on top of it.

3.3 PCP lock resource

We maintain a global variable system_ceiling_setter which is the pid corresponding to the current system ceiling setter. We also maintaint a sdmse prio struct which corresponds to

the system ceiling priority. The following snippet then follows the PCP.

```
SYSCALL_DEFINE2(pcp_lock, pid_t, pid, unsigned int, rid)
{
    ...
    if(pid == system_ceiling_setter){
        // I am the ceiling setter
        goto giveresource;
    }
    struct sdmse_prio cur_prio = get_sdmse_prio(sdmse);
    if(__sdmse_prio_less(cur_prio, system_ceiling_prio)){
        // higher priority than ceiling
        // make this the system ceiling setter and give it the resource.
        system_ceiling_setter = pid;
        goto giveresource;
    }
    else{
        struct sched_dms_entity *ceil_setter_sdmse;
        ceil_setter_sdmse = get_sdmse_from_pid_registered(system_ceiling_setter);
        struct sdmse_prio csprio = get_sdmse_prio(ceil_setter_sdmse);
        if(__sdmse_prio_less(csprio, cur_prio)){
            goto rejectresource;
        }
        // inherit priority. dequeue, push inherited prio, enqueue. reschedule.
        spin_lock(&dms_rq.rq_lock);
        dms_dequeue_ready(ceil_setter_sdmse);
        inherit_prio(ceil_setter_sdmse, cur_prio);
        dms_enqueue_ready(ceil_setter_sdmse);
        spin_unlock(&dms_rq.rq_lock);

        dms_schedule();

        goto rejectresource;
    }

giveresource:
    pcpr->locked = true;
    calculate_system_ceiling();
    return 0;

rejectresource:
    return -22;
}
```

The first if condition corresponds to the first clause: when the requesting task is the system ceiling setter. The second condtion corresponds to the second clause: when the priority of requesting task is higher than the system ceiling. In inherit prio, we push the priority of higher priority task to the stack of the system ceiling setter. This is then used when

comparing it to other tasks in the ready queue.

3.4 PCP Unlock Resource

When unlocking a resource, We need to set the priority back to the one which we had when getting the resource. This is done by popping the priority inheritance stack and comparing the resource ids that the task had locked when making the corresponding inheritance. Once all such entries have been popped, we can then use the `get_sdmse_prio` function to get the priority of this task. This follows from the assumption that all the resource accquisitions are nested and perfectly balanced. The following snippet shows the implementation of the above idea.

```
SYSCALL_DEFINE2(pcp_unlock, pid_t, pid, unsigned int, rid){
    ...
    dms_dequeue_ready(sdmse);
    struct sdmse_prio *cur_prio;
    while(!list_empty(&sdmse->inherited_priorities_head)){
        cur_prio = list_pop_entry(&sdmse->inherited_priorities_head, struct
            sdmse_prio, inherited_priorities_entry);
        if(cur_prio->inherited_while_holding_rid != rid){
            list_push_entry(cur_prio, &sdmse->inherited_priorities_head,
                inherited_priorities_entry);
            break;
        }
        kfree(cur_prio);
    }
    dms_enqueue_ready(sdmse);

    struct pcp_resource *pcpr = get_pcpr_from_rid(rid);
    pcpr->locked = false;

    calculate_system_ceiling();

    return 0;
}
```

4 Novelties

4.1 Runqueue Cleaning

Some tasks might get killed before removing themselves using the `sys_remove` syscall. They will interfere with our Schedulability checks and also cause runtime errors if not cleaned properly. Also, memory leaking can occur. I have gracefully handled such situations by placing safety checks everywhere and cleaning the runqueue whenever possible - Every time a process registers itself, whenever `sys_list` has been called etc.

4.2 System Call to clear pcp global variables

I create a new system call pcp_clear, that clears out all the global variables used during pcp. This is useful as we may need to reuse the pcp structures and this requires a way to clear them.

4.3 KConfig Options

Both the implementations are made completely modular such that they can be turned on or off by changing the .config file. The pcp config option depends on the dms config option. prambal