Linux DMS Scheduling Class with PCP

Operating Systems Assignment 2

Dakshit Babbar | 2020EE30163

April 2024

1 Objective

Given the Linux source code, we have to implement some functionalities that will be added to the Linux Kernel. Following are the objectives:

- To modify the source code to include a new scheduling class that implements the DMS (Deadline Monotonic Scheduling) algorithm.
- To implement the Priority Ceiling Protocol on top of the DMS algorithm.

2 System Setup

The system setup is completed using the *Guide to Hacking the Linux Kernel* by Abhishek Safui. Following is the overview of the steps involved in the setup:

1. Install the source code of the *Yocto Project*(*poky*). This is an embedded Linux platform tool which helps us create a minimal configuration for our kernel to reduce the compilation and testing time.
2. Build *poky* to get the following:
    - Virtual Disk Image
    - Minimal .config file
3. Install the Linux source code from *kernel.org*
4. Build the Linux source code using the minimal .config file obtained from *poky*
5. Boot the compiled kernel on a VM with the Virtual Disk Image obtained from *poky*.

3 Scheduling Class

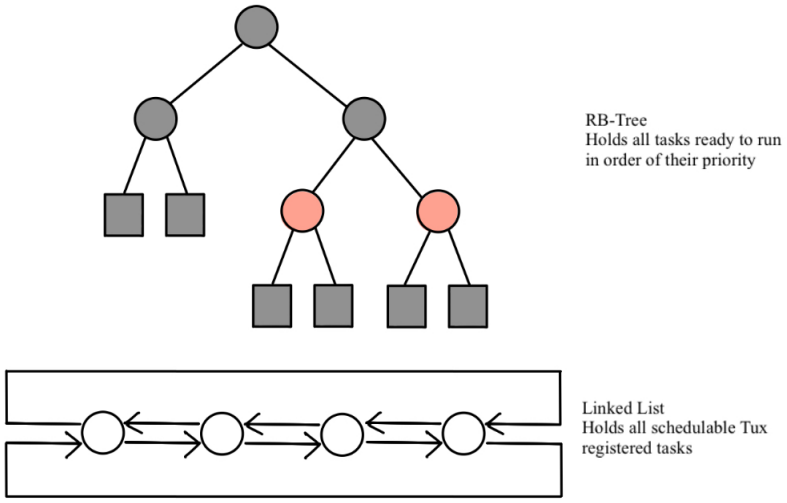
First we describe the implementation of the scheduling class that implements the DMS algorithm. Note that we have made the assumption that our scheduler will run on a uniprocessor system.

3.1 Directory Structure

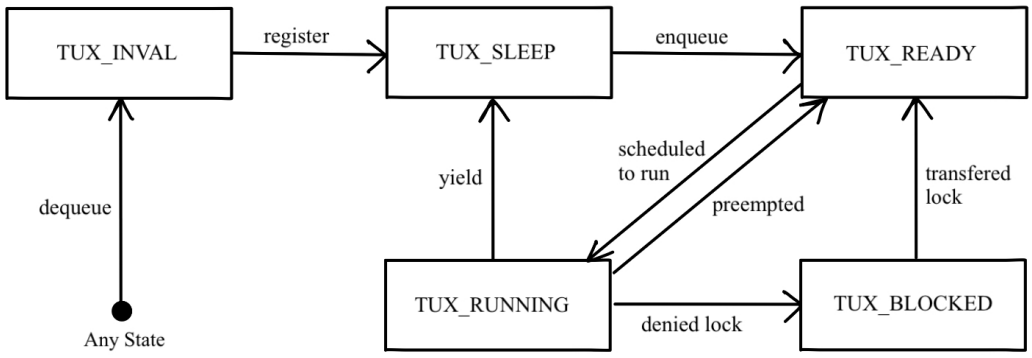
The name of the scheduler that we have implemented is Tux, for which we have created a separate file by the name of *tux.c* in the *kernel/sched/* directory. All the necessary system calls and other kernel functions are defined in this file itself.

3.2 Scheduler Description

The Tux scheduler maintains two data structures to manage the tasks, one is a Doubly Linked List that holds all the tasks that are registered as Tux Schedulable and a Red-Black Tree that holds all the tasks that are ready to run on the CPU, sorted with respect to their deadlines.



There are a total of 5 possible states that a task could be in, with respect to the Tux sheduler. All tasks in the system before they are registered as Tux Schedulable are in the TUX_INVAL state. If any task wants to register itself it will make the **sys_register_dm** system call which will take it into the TUX_SLEEP state and then eventually in the TUX_READY state, only if it passes the schedulability test. Note that once the task is registered it will always be in the Linked List unless it is deregistered using the **sys_remove** system call. Also a task will be in TUX_READY state if and only if it is in the RB-Tree.



As soon as the scheduler selects a task from the RB-Tree to run, it will transition to the TUX_RUNNING state. Once the task is running there are three possible events that it can experience,

- If it gets preempted involuntarily by some higher priority task in the system, it will transition back to the TUX_READY state.
- If its execution time is over and it voluntarily gives up the CPU (by making the **sys_yield** system call), then it will move to the TUX_SLEEP state and set a timer which will interrupt the system at the time period of the task and move the task back to the TUX_READY state.
- If it wanted a lock but a lower priority resource has already held it then it will move to the TUX_BLOCKED state, added to the wait queue of the lock and once the lock gets free this task will be transferred the lock accroding to its priority and will transition to the TUX_READY state.

3.3 Implementation Details

Here we specify the details of the changes made to the kernel as well as the functions implemented.

- **New Structures:**
  We define a new struct to manage the information related to Tux Registered tasks. Taskstruct of the task will also contain a pointer to this struct.

```
//include/linux/sched.h
struct sched_tux_entity{
    struct task_struct *task;
    struct timer_list timer;
    ...
    int state;
    int period;
    int deadline;
    int exec_time;
    int prio;
    ...
    struct rb_node ready_node;
    struct list_head sleep_node;
};
```

  We also define a new struct to manage the information related to Tux runqueue. The runqueue struct of the CPU will also contain a pointer to this struct.

```
//kernel/sched/sched.h
struct tux_rq {
   struct rb_root ready_tasks;
   struct list_head sleeping_tasks;
   atomic_t nr_ready;
};
```

- **Changes to Kernel Functions:**
  We have also modified some of the kernel functions as follows:

```
//kernel/sched/sched.h
static inline int rt_policy(int policy){
   return policy == SCHED_FIFO || policy == SCHED_RR || policy == SCHED_TUX;
}
```

```
//include/asm-generic/vmlinux.lds.h
#define SCHED_DATA         \
   ...
   *(__stop_sched_class)      \
   *(__tux_sched_class)    \
   *(__dl_sched_class)     \
    ...

//kernel/sched/core.c
static void __setscheduler_prio(struct task_struct *p, int prio){
...
   else if (rt_prio(prio))
      if(p->policy == SCHED_TUX){
         p->sched_class = &tux_sched_class;
      } else {
         p->sched_class = &rt_sched_class;
      }
...
}

//kernel/sched/core.c
void __init sched_init(void){
...
    BUG_ON(... &dl_sched_class != &tux_sched_class + 1);
#ifdef CONFIG_SMP
   BUG_ON(&tux_sched_class != &stop_sched_class + 1);
#endif
...
    for_each_possible_cpu(i) {
      ...
      init_tux_rq(&rq->tux); // initialises the tux scheduler datastructures
    }
}
```

- **Scheduling Class Functions**
  Each scheduling class in linux needs to implement some basic functions, the pseudocode for which is given below:

```
//called when a new task registers as tux schedulable
static void enqueue_task_tux(struct rq *rq, struct task_struct *p, int flags){
    //insert task in the rbtree
    //change state to TUX_READY
    //increase the running task counter of the runqueue
}

// called when a tux registered task is being removed
static void dequeue_task_tux(struct rq *rq, struct task_struct *p, int flags){
    //if task is in TUX_READY state:
        //remove task from the rbtree
        //decrease the running count of the runqueue
    //remove the task from the linked list
    //change state to TUX_INVAL
}
```

```
//called when new task enteres the runqueue to check if it preempts the current task
static void check_preempt_curr_tux(struct rq *rq, struct task_struct *p, int flags){
    //if the currently running task is not tux registred and the rbtree is not empty
        //reched()
    //else if current task is of higher sched_class
        //do nothing
    //else
        //find the highest prio task from the rbtree
        //if this is higher in prio than the current task
            //resched()
}

//called whenever the task is being set as the next on the cpu
static void set_next_task_tux(struct rq *rq, struct task_struct *p, bool first){
    //remove task from rbtree
    //DO NOT decrease the running counter
    //change state to TUX_RUNNING
}

//called whenever the task is brought down from the cpu
static void put_prev_task_tux(struct rq *rq, struct task_struct *p){
    //if task state is not TUX_RUNNING
        //return
    //insert task into rbtree
    //change state to TUX_READY
}

//picks the task with the smallest deadline
static struct task_struct* pick_next_task_tux(struct rq *rq){
    //get the task with smallest deadline
    //if it exists
        //call set_next_task()
        //return that task
    //return NULL
}

//called when the task is voluntarily gives up the cpu
static void yield_task_tux(struct rq *rq){
    //change state to TUX_SLEEP
    //decrease the running counter
    //start the timer
}

//will be called when the yield timer goes off
static void yield_timer_callback(struct timer_list *t){
    //if task is in TUX_INVAL state then return

    //insert task into rbtree
    //change state to TUX_READY
    //increase the running counter of the runqueue

    //call check_preempt_curr()
}
```

- **System Calls:**
  Following are the system calls implemented to allow the user tasks to interact with the Tux scheduler.

```
//register syscall
static int tux_register(pid_t pid, ul period, ul deadline, ul exec_time, bool rm){
    //if rm=true
        //deadline=period
    //if task schedulable
        //initialise the sched_tux_entity struct for this task
        //call the sched_setschduler() function
            //this changes the policy of the task from SCHED_NORMAL to SCHED_TUX
            //and sched_class from sched_class_fair to sched_class_tux
            //return 0
    //else
        return -EINVAL
}

//yiled syscall
static int tux_yield(void){
    //call the yiled_task_tux() function
    //schedule()
}

//remove syscall
static int tux_remove(pid_t pid){
    //call sched_setscheduler() function
        //this internally calls the dequeue function
        //changes the policy from SCHED_TUX to SCHED_NORMAL
}

//list syscall
SYSCALL_DEFINE0(list){
    //iterate through the linked list of the Tux scheduler
        //print the task info for each task in kernel logs
}
```

- **Schedulability Test:**
  The following function checks weather the input task will maintain the schedulability of the taskset. Here, for every task T_i with period P_i, execution time E_i and deadline D_i:

$$ctime = \sum_{j=1}^{i} P_j$$

$$interference(trial\_time) = \sum_{j=1}^{i-1} \left\lceil \frac{trial\_time}{T_j} \right\rceil C_j$$

```
static bool tux_is_schedulable(struct rq *rq, ul period, ul deadline, ul exec_time){
    //sort all the tasks in ascending order of deadlines (increasing prio)
    //for each task ti
        //try_time=ctime
        //while(1)
            //if intereference(trial_time) + ti->exec_time <= trial_time
                //break
            //else
                //trial_time=intereference(trial_time) + ti->exec_time
            //if trial_time > ti->deadline
                //return false
    //return true
}
```

4 Priority Ceiling Protocol

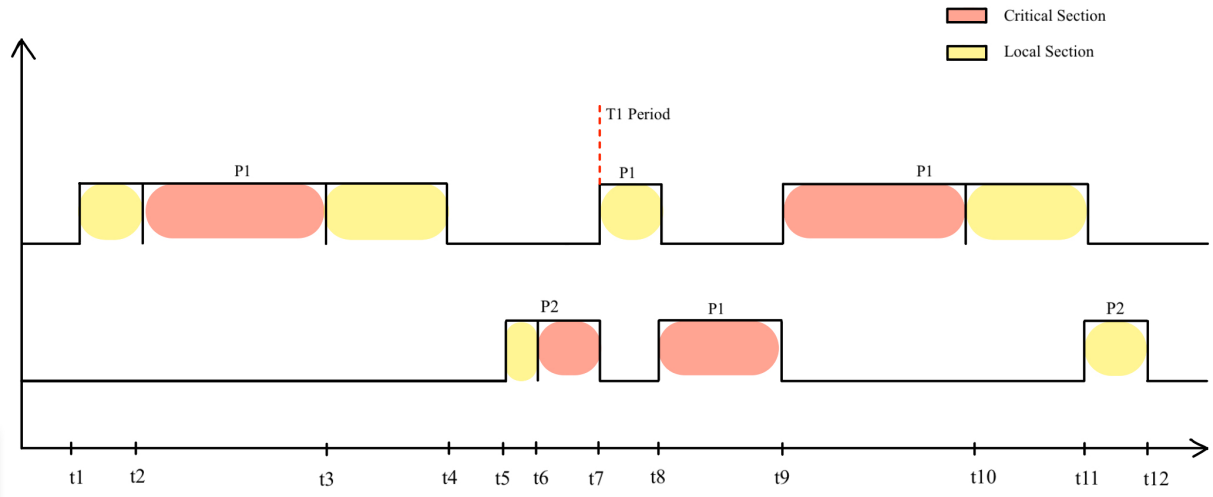
We now describe the priority ceiling protocol implemented over the DMS algorithm. Our implementation assumes that the system consists of a single resource and at at most three Tux schedulable tasks accessing that resource all of whose deadlines are known prior to execution. Nevertheless the following pseudo code describes a general case algorithm of the priority ceiling protocol.

```
struct tux_lock{
   struct task_struct *holder;
   struct task_struct *wait_queue[];
   unsigned long cieling; //max of priority of all the resources accessing this resource
};

//locker syscall
SYSCALL_DEFINE1(locker, pid_t, pid){
    //disable preemption
    //ceil=max(ceiling of resrouces that are being used by tasks other than pid)
    //if(prio(pid)<ceil)
        //do not grant lock
        //let T be the task that holds the resource that set the ceil
        //make T inherit the priority of pid
        //add pid to wait queue of the lock
        //change pid state to TUX_BLOCKED
        //decrease running count of runqueue
        //resched()
    //else just grant the lock
    //enable preemption
}

//unlocker syscall
SYSCALL_DEFINE1(unlocker, pid_t, pid){
    //disable preemption
    //if the wait queue of the lock is not empty
        //set pid prio back to the previous prio
        //remove the next highest prio task from the quueue
        //transfer lock to next
        //insert next back into the rbtree
        //change next's state to TUX_READY
        //increase running count of the runqueue
        //resched()
    //else just free the lock
    //enable preemption
}
```

Following is an example case of the priority ceiling protocol implemented. We have 2 tasks T_1 and T_2 where higher subscript denotes lower priority of the task. Initially in the system we just have T_1 which runs for its first period, where it successfully acquires the lock as there are no locks being held by tasks other than T_1. Once its critical section is over it releases the lock and goes to sleep.



While T_1 is asleep, T_2 enters the system and acquires the lock at time t_6 and starts its critical section, during which T_1 wakes up and preempts T_2 at time t_7 as it has a higher priority, and executes its own local section. Now at time t_8, T_1 tries to acquire the lock but as its priority is not greater than the maximum ceiling of the locks already in use by tasks other than T_1 hence it gets blocked on the task T_2, which in turn inherits the priority of T_1. The next highest priority task that is ready to run, that is T_2, continues executing in its critical section. At time t_9, T_2 releases the lock, transfers lock to the highest prio task in the wait queue of the lock, which is T_1, unblocks it and comes back to its original priority. Now the highest priority task in the system that is ready to run is T_1. It completes its critical section, releases the lock and then eventually goes back to sleep again. Thereafter, T_2 completes its local section and sleeps.

5 Questions

Following are the answers to the in text questions of the assignment.

Q. Explain the circumstances wherein the deadline monotonic algorithm might be preferred over the rate monotonic scheduling algorithm and vice-versa with suitable examples.
A.The DMS algorithm is where we have specific deadlines to complete the execution of the real time task before it occurs again. Hence it will be useful in cases where the deadlines are not very strict and missing some deadlines will not cause a great harm. For example, soft real time systems like multimedia including video and audio. Here we may miss some deadlines which may cause playback glitches which is tolerable as long as it is infrequent.

The RMS algorithm on the other hand strictly imposes the deadlines to be the period end of the task, and if these deadlines are not met then it can cause great harm. It can hence be used in hard real time systems like aircrafts and automobiles

Q. In case the Linux kernel utilizes the priority inversion protocol instead of the priority ceiling protocol. What will be the advantages and disadvantages of it?
A.The priority inversion protocol was developed to avoid unbounded priority inversions but it was not able to tackle the issues of deadlocks and chain blocking. To solve these two problems, PCP was developed which is a more conservative protocol and avoids any deadlocks and chain blocking. Hence if PI is used instead of PCP then the system may experience deadlocks and chain blocking.