ELL 405 Operating Systems

Assignment Report

Assignment 2

Anurag Gupta 2020EE10583
Bismandeep Singh 2020EE10589

IIT DELHI, 7th April 2024

1 Modifications in proc struct

Few variables have been added to the `proc` struct in `proc.h`. They are:

1. `sched_policy`: -1 for Default (Round Robin), 0 for EDF, and 1 for RMA scheduling
2. `exec_time`: Total number of ticks for which a Real Time process would run
3. `deadline`: (For EDF) Number of ticks (relative to the arrival time) within which the Real Time Process should finish execution
4. `rtp_arrival_time`: Ticks value at which the Real Time process is first created and arrives in run_queue (The point at which its scheduling policy is set), helps calculate absolute deadline
5. `elapsed_time`: Number of ticks for which the process has completed its execution on the unicore processor (across all context switches, starting from its current period)
6. `rate`: Instances per second of the program, parameter for RMA
7. `weight`: Priority assigned based on rate parameter for RMA

2 System Call Implementation

Four new syscalls have been created to set the value of the values listed above

1. `sys_sched_policy`: Set the scheduling policy for a process (either -1, 0 or 1)
2. `sys_exec_time`: Set the execution time of a process in ticks (10ms)
3. `sys_deadline`: Set the deadline of a process in ticks
4. `sys_rate`: Set execution rate of a process (in instances / second)

3 Scheduling Policies

The `sys_set_sched_policy` checks whether the current task list (EDF or RMA) is schedulable or not. If yes, then set the scheduling policy, otherwise kill the process and return -22.

```c
int set_sched_policy(int pid, int policy){
  struct proc *p;
  // Flags for valid_pid argument and to check if the
  // parameters of a given policy are set before scheduling
  int valid_pid = 0, set_param = 0, schedulable, timer_ticks;
  // Read the timer_ticks count
  acquire(&tickslock);
  timer_ticks = ticks;
  release(&tickslock);
  // Acquire ptable lock
  acquire(&ptable.lock);
  // Traverse the ptable and set the flags
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid == pid) {
      valid_pid = 1; // Pid found, argument is valid
      // Check if the required parameters for EDF policy (exec_time and deadline
   ) and RMA (exec_time and rate) are set
      ...
    }
  }
  // If invalid pid or parameters not set, the new ptable is schedulable
  // as original was schedulable as well and no change in policy was made
  if(!valid_pid || !set_param) {
    release(&ptable.lock);
    return -1;
  }
  // Set the sched_policy field
  p->sched_policy = policy;
  // Set the rtp_arrival time
  p->rtp_arrival_time = timer_ticks;
  // Check schedulability according to the policy
  schedulable = (policy == 0) ? edf_schedulability_check(timer_ticks) :
   rma_schedulability_check(timer_ticks);
  // if schedulable then return 0
  // otherwise mark the process to be killed by the kernel
    ...
  release(&ptable.lock);
  return -22;
}
```

3.1 Earliest Deadline First (EDF)

Earliest Deadline First (EDF) scheduling assigns priority based on the nearest upcoming deadline, ensuring tasks with the closest deadlines are executed first.

3.1.1 Schedulability Condition

To check if a set of processes are schedulable under EDF schedule, we confirm that the total CPU utilization is less than or equal to 100%.

$$\sum_{i=1}^{N} U_i = \sum_{i=1}^{N} \frac{e_i}{p_i} \leq 1 \tag{1}$$

where,
N = The number of Running or Runnable processes,
U_i = CPU utilization by the i^{th} process,
e_i = Execution time (in ticks) of the i^{th} process,
p_i = Period (in ticks) of the i^{th} process

3.1.2 Implementation

```
int edf_schedulability_check(int timer_ticks){
  struct proc *p;
  // Total CPU Utilization in fraction (in simplest form), denoted by a
   numerator and denominator
  float utilization = 0;
  // Per process execution time and deadline
  int e_i, d_i;
  // Traverse the ptable to find set of edf scheduled RT processes
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    // Check if process is not in RUNNABLE or RUNNING or meant to be killed
    if(! (p->state == RUNNABLE || p->state == RUNNING) || p->killed == 1)
      continue;
    // Check if process is EDF scheduled
    if(p->sched_policy == 0) {
      // Set e_i as exec_time - elapsed_time of the process
```

```
      e_i = p->exec_time - p->elapsed_time;
      // Set d_i as absoulte deadline - current ticks
      d_i = p->rtp_arrival_time + p->deadline - timer_ticks;
      // Check if e_i > d_i or d_i < 0
      if(e_i > d_i || d_i <= 0) {
        return -22; // Deadline Missed
      }
      // Update utililization
      utilization += ((float) e_i) / d_i;
    }
  }
  // Check if deadline miss doesnt occur & total utilization <= 1, If yes then
   schedulable
  return (utilization <= 1.0) ? 0 : -22;
}
```

3.2 Rate Monotonic Scheduling (RMS)

Rate-Monotonic Analysis (RMA) scheduling prioritizes tasks based on their rates, where tasks with shorter periods are given higher priority, ensuring tasks are scheduled according to their periodicity.

3.2.1 Schedulability Condition

Liu & Layland Utilization bound

This is a sufficient but not a necessary condition for a set of processes to be RMA schedulable.

$$\sum_{i=1}^{N} e_i * r_i \leq N * (2^{\frac{1}{N}} - 1) \tag{2}$$

where,
N = The number of Running or Runnable processes,
e_i = Execution time (in ticks) of the i^{th} process,
r_i = Rate (in Instances per ticks) of the i^{th} process

Liu & Lehoczky Check

This is a sufficient and necessary condition for a set of processes to be RMA schedulable, but is more computationally expensive than Liu & Layland, hence is checked only if the former fails.

$$e_i + \sum_{j=1}^{i} \lceil \frac{p_i}{p_j} \rceil * e_j \leq p_i \; \forall, i \in [1, N] \tag{3}$$

where,
N = The number of Running or Runnable processes,
e_i = Execution time (in ticks) of the i^{th} process,
r_i = Rate (in Instances per ticks) of the i^{th} process,
$r_1 \geq r_2 \geq r_3 ... \geq r_N$

3.2.2 Implementation

```
int rma_schedulability_check(int timer_ticks){
  struct proc *p;
  int num_proc = 0; // Number of RMA Scheduled Processes
  float utilization = 0; // Total CPU Utilization
  // Traversing the ptable
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    // Check if process is not in RUNNABLE or RUNNING or meant to be killed
    if(! (p->state == RUNNABLE || p->state == RUNNING) || p->killed == 1)
      continue;
    // Check if scheduling policy is 1 (RMA)
    if (p->sched_policy == 1){
      num_proc ++; // Incremenet num_proc
      utilization += (p->exec_time * p->rate) * 0.01; // Utilization = exec_time
    * rate * 0.01(seconds / tick)
    }
  }
  // If no RMA scheduled process, the previous set of processes is schedulable
  if (num_proc == 0)
    return 0;
  // If total utilizationis > 1, the set of processes is not schedulable under
   any scheduling policy
  if (utilization > 1)
    return -22; // CPU utilization can't be greater than 1
  // Check for liu_layland condition
```

```
  if (utilization <= liu_layland_bound[num_proc - 1])
    return 0; // schedulable under RMA
  // Check for Liu-Lehoczky's condition
  if (liu_lehoczky_check(num_proc) == 0)
    return 0;
  // Set of processes not schedulable under RMA
  return -22;
}
```

3.3 Modifications to scheduler

Since, EDF tasks are given higher priority, the scheduler goes through all the tasks that are EDF schedulable, finds the one with the closest deadline (and smaller `pid`) and schedules it. If there are no EDF tasks, the scheduler goes through all tasks that are RMA schedulable and schedules the task with the least `weight` (and smaller `pid`). If there are no EDF and RMA tasks, then the default Round Robin schedule is used.

```
void scheduler(void){
  struct proc *p = 0;
  struct cpu *c = mycpu();  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    // Check for EDF scheduled set of process
    p = scheduler_edf();
    if(p == 0) // No EDF scheduled process, try RMA
      p = scheduler_rma();
    if(p != 0) {
      // EDF/RMA scheduled process found, perform context switch
      // Switch to chosen process.  It is the process's job to release ptable.
   lock
      // and then reacquire it before jumping back to us.
      ...
      // Process ran for 1 tick, Only tracked for Real Time Process
      p->elapsed_time++;
```

```
      ...
      // Continue with the infinite loop
      continue;
    }
    // No EDF/RMA scheduled process, default to ROUND-ROBIN Scheduling
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        ...
    }
    release(&ptable.lock);
  }
}
```

3.3.1 Finding next EDF scheduled process

```
struct proc* scheduler_edf(void){
  struct proc *p, *sched_p = 0;
  // Variables to track deadline
  uint earliest_deadline = UINT32_MAX, current_deadline;
  // Traverse the ptable t find the process with earliest
  // deadline, if same deadline then smallest pid
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if((p->state == RUNNABLE || p->state == RUNNING) && p->sched_policy == 0) {
      // Calculate current deadline
      current_deadline = p->rtp_arrival_time + p->deadline;
      if(current_deadline < earliest_deadline) {
          earliest_deadline = current_deadline;
          sched_p = p;
      }
      else if(current_deadline == earliest_deadline) {
        if(p->pid < sched_p->pid)
          sched_p = p;
      }
    }
  }
  return sched_p;
}
```

3.3.2 Finding next RMA scheduled process

```
struct proc* scheduler_rma(void){
  struct proc *p, *sched_p = 0;
  // Variables to track deadline
  uint min_weight = UINT32_MAX;
  // Traverse the ptable t find the process with earliest
  // deadline, if same deadline then smallest pid
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if((p->state == RUNNABLE || p->state == RUNNING) && p->sched_policy == 1) {
      if(p->weight < min_weight) {
          min_weight = p->weight;
          sched_p = p;
      }
      else if(p->weight == min_weight) {
        if(p->pid < sched_p->pid)
          sched_p = p;
      }
    }
  }
  return sched_p;
}
```

4 Extra

Implemented Liu & Lehoczky check for RMA in addition to the common Liu & Layland Bound. All provided test cases work if only Liu & Layland bound is checked, however, Liu & Lehoczky provides a sufficient and necessary condition for the cases where the former might declare a set of schedulable processes unschedulable.

```
int liu_lehoczky_check(int num_proc){
  // Store the RMA scheduled process in a proc
  struct proc* proc_array[num_proc];
  int index = 0;
  for (int i = 0; i < NPROC; i++){
    struct proc *p = &ptable.proc[i];
    // Check if a process is not in RUNNABLE or RUNNING or meant to be killed
    if(! (p->state == RUNNABLE || p->state == RUNNING) || p->killed == 1)
      continue;
    // Add the process to proc_array if scheduling policy is RMA
```

```
    if (p->sched_policy == 1)
      proc_array[index++] = p;
  }
  // Sort the array in descending order of rate
  quicksort(proc_array, 0, num_proc - 1, compare_rma);
  // Check if all processes meet their deadlines under zero_phasing (deadline = period)
  for(int i = 0; i < num_proc; i++) {
      // Calculate sum(p_i/p_j * e_j) for all interfering processes, this is the virtual execution time that has to be met within the deadline
      int virtual_exec_time = proc_array[i]->exec_time;
      for(int j = 0; j < i; j++) {
          virtual_exec_time += CEIL(proc_array[j]->rate, proc_array[i]->rate) * proc_array[j]->exec_time;
      }
      // Check if virtual exec_time (in ticks) < period * 100 , or virtual exec_time (in ticks) * rate <= 100
      if(virtual_exec_time * proc_array[i]->rate > 100)
        return -22; // Condition violated, not schedulable
  }
  // All processes meet their deadlines under zero-phasing => Schedulable under RMA
  return 0;
}
```