Department of Computer Science & Engineering

ASSIGNMENT 2 : EASY

Abhinav Barnawal *Shreyansh Singh*
: *2020CS50415* *2020CS10385*
Class: *2202-COL331*
Session: *2022-23*
Email: *cs5200415@iitd.ac.in* *cs1200385@iitd.ac.in*

Course: *Operating Systems* – Instructor: *Prof. Smruti Ranjan Sarangi*
Submission date: *April 5, 2023*

Contents

1. Additional Process Attributes

We have added the following additional attributes of the `proc` structure:

- `int sched_policy`: Scheduling policy of the process (-1: XV6 default policy or 0: EDF or 1: RMS or 4: non-schedulable processes). It is set by a user process using the `sched_policy(pid, value)` system call.
- `int elapsed_time`: Elapsed time of the process. It counts the number of ticks for which the process was in `RUNNING` state.
- `int exec_time`: Total allowed execution time of the process. It is set by a user process using the `exec_time(pid, value)` system call.
- `int rate`: Rate of the assumed periodic processes. It is set by a user process using the `rate(pid, value)` system call.
- `int priority`: Current priority level of each process (1-3) (higher value represents lower priority). It is calculated using `rate` of the process.
- `int deadline`: Hard Deadline of the process. Any new process that fails the schedulability check is killed so that no deadlines for accepted processes are sacrificed. It is set using the `deadline(pid, value)` system call.
- `int wait_time`: Wait time of the process. It records the number of ticks for which the process was in `RUNNABLE` state, i.e., waiting to be scheduled.
- `int arrival_time`: Start time of the process. It records the tick value when the process's `sched_policy` was set
- `int last_tick`: Last tick value at which the process's `elapsed_time` was incremented.

We have also modified `trap.c` to kill unwanted or completed processes. A process becomes unwanted if its `sched_policy != -1` and `elapsed_time` exceeds its `exec_time`. Below is the code for the same.

```
if (myproc() && myproc()->state == RUNNING &&
      tf->trapno == T_IRQ0 + IRQ_TIMER)
  {
    if ((myproc()->sched_policy >= 0) &&
        (myproc()->elapsed_time >= myproc()->exec_time))
    {
      cprintf("The arrival time and pid value of the completed process
              is %d %d \n", myproc()->arrival_time, myproc()->pid);
      exit();
    }
    else
      yield();
  }
```

2. EDF Scheduling Algorithm

Schedulability Test

We check whether a process is schedulable in the *sched_policy* function. We add the utility of all scheduled processes and check if the total utility is more than 1. If so, the process will be exited and not be considered for checking the schedulability of future processes. The processes which have been finished are also considered while calculating the total utility since the processes are periodic. Since the floating point is not allowed, we have multiplied the number by 1000000, and thus we have the accuracy of up to 6 decimal places.

```
int temp = 1000000 * p->exec_time / p->deadline;
for (struct proc *p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
  if (p1->sched_policy == 0 && p->pid != p1->pid
                            && p1->exec_time < 1000000){
    temp += 1000000 * p1->exec_time / p1->deadline;
  }
}
if (temp > 1000000){
    release(&ptable.lock);
    kill(pid);
    return -22;
}
```

Scheduling Algorithm

We have maintained a variable *sched_pol* which stores whether RMS or EDF scheduling has to be done. In EDF scheduling, the process with the earliest deadline is scheduled. For this, we calculate the time left ($arrival\_time + deadline - ticks$) before the deadline for all processes and schedule the process with the least time left. If the time left for two processes is the same, then the process with a lesser pid is scheduled earlier.

```
int earliest_deadline = 100000000, pid_to_sched = -10;
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->state != RUNNABLE || p->sched_policy != sched_pol)
      continue;
    int time_left = p->arrival_time + p->deadline - ticks;
    if (time_left < earliest_deadline)
    {
      earliest_deadline = time_left;
      pid_to_sched = p->pid;
    }
    else if (time_left == earliest_deadline)
    {
      if (p->pid < pid_to_sched)
        pid_to_sched = p->pid;
    }
}
if (pid_to_sched == -10)
    continue;
```

3. RMS Scheduling Algorithm

Schedulability Test

We check whether a process is schedulable in the *sched_policy* function. We add the utility of all scheduled processes and count the number of processes (n). If the total utility is more than `LIU_BOUND[n]`, then the process will be exited, and it will not be considered for checking the schedulability of future processes. `LIU_BOUND` for $n \in [1, 64]$ is precomputed and stored. So, while checking `LIU_BOUND` can be directly accessed. The processes which have been completed are also considered while calculating the total utility since the processes are periodic. Since the floating point is not allowed, we have multiplied the number by 1000000, and thus we have the accuracy of up to 6 decimal places.

```
int temp = (10000 * p->rate * p->exec_time);
int n = 1;
for (struct proc *p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
  if (p1->sched_policy == 1 && p->pid != p1->pid && p1->exec_time < 1000000){
    temp += 10000 * p1->rate * p1->exec_time;
    n++;
  }
}
if (temp > LIU_BOUND[n]){
    release(&ptable.lock);
    kill(pid);
    return -22;
}
```

Scheduling Algorithm

In RMS scheduling, the process with the highest priority is scheduled. For this, we compare the weight of all the processes and schedule the process with the least weight. If the weight for two processes is the same, then the process with a lesser pid is scheduled earlier.

```
int lowest_prio = 100, pid_to_sched = -10;
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->state != RUNNABLE || p->sched_policy != sched_pol)
      continue;
    if (p->priority < lowest_prio)
    {
      lowest_prio = p->priority;
      pid_to_sched = p->pid;
    }
    else if (p->priority == lowest_prio)
    {
      if (p->pid < pid_to_sched)
        pid_to_sched = p->pid;
    }
}
```

```
if (pid_to_sched == -10)
    continue;
```

Acknowledgement

Our sincere thanks to Prof. Smruti Ranjan Sarangi for his expertise and guidance that led to the completion of this work. We also thank the respected Teaching Assistants for their immense support and almost instant replies to our doubts, without whom it would have never been possible to present this report.