COL331 - Assignment 2

Rishi Jain (2020CS10373), Vidushi Agarwal (2020CS10410)

April 05, 2023

1 System calls

1.1 deadline

Arguments: int pid, int deadline
Iterate over the ptable and select the process with the given pid, assign the deadline provided in the arguments to the deadline parameter of the proc struct. In the proc.c file, the following function is defined:

```c
int deadline(int pid, int deadline_st){
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->deadline = deadline_st;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -22;
}
```

1.2 rate

Arguments: int pid, int rate_val
Iterate over the ptable and select the process with the given pid, assign the rate provided in the arguments to the rate parameter of the proc struct. For RMS scheduling the scheduler accounts for the weights of the processes while scheduling. The weights are calculated using the following function:
$weight, w = max(1, ceil(3 * (30 - rate)/29))$
In the proc.c file, the following function is defined:

```c
int rate(int pid, int rate_val){
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->rate = rate_val;
      int temp = (30-rate_val)*3;
      if(temp%29 == 0)
        temp = temp/29;
      else
        temp = temp/29 + 1;
      if(temp < 1)
        temp = 1;
      p->weight = temp;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -22;
}
```

1.3 exec_time

Arguments: int pid, int exec_time
Iterate over the ptable and select the process with the given pid, assign the exec_time provided in the arguments to the exec_time parameter of the proc struct. In the proc.c file, the following function is defined:

```c
int exec_time(int pid, int exec_time){
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->exec_time = exec_time;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -22;
}
```

1.4 sched_policy

Arguments: int pid, int policy
Iterate over the ptable and select the process with the given pid, assign the policy provided in the arguments to the sched_policy parameter of the proc struct. Check for the feasibility of scheduling the set of processes, if the utilization bound is violated kill the process and return -22 else change the system policy to the policy specified. The function is defined in *proc.c* file. The code snippet is the same as in the schedulability section.

2 Initialization

All the parameters in the proc struct needed for a scheduler to schedule a process are initialized in the *allocproc* function of *proc.c* as:

```
  p->exec_time = -1;
  p->sched_policy =-1;
  p->elapsed_time =0;
  p->deadline = 1;
  p->rate = 20;
  p->weight = 1;
```

3 Execution constraints

For the real time scheduling policies, EDF and RMS, Elapsed time for the process is updated at every timer interrupt and subsequently the Execution and Elapsed time constraints are checked in the *trap* function of *trap.c* program file. If the process's elapsed time exceeds its execution time the execution is completed and the process exits.

```
struct proc *min_proc = 0;
      if(myproc() && myproc()->state == RUNNING &&
  tf->trapno == T_IRQ0+IRQ_TIMER )
  {
  myproc()->elapsed_time++;
  if((myproc()->sched_policy >= 0) && myproc()->exec_time>=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();
  }
```

4 EDF Scheduling Policy

In EDF Scheduling policy, the priorities are decided on the basis of the deadline. Process with earliest deadline is given the highest priority.

Implementation:

1. We need to ensure that at every timer interrupt, the process with smallest deadline is scheduled first (with pid number as tie breaker).
2. In the *sched* function in *proc.c*, if the current system policy is 0 i.e. EDF, iterate over the process list and schedule the process with smallest deadline which is in *RUNNABLE* or *RUNNING* state.
3. In the *trap* function in *trap.c* file, at every timer interrupt, update the current process's elapsed time. If the elapsed time of the process exceeds the execution time, exit from the process.

```
struct proc *min_proc = 0;
   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
     continue;
    if(p->sched_policy != 0)
     continue;
    if(min_proc == 0){
     min_proc = p;
    }
    else{
     if((p->deadline + p->arrival_time  < min_proc->deadline + min_proc->arrival_time  ) || ((p->deadline + p->arrival_time == min_proc->deadline + min_proc->arrival_time) && (p->pid<min_proc->pid)){
      min_proc = p;
     }}}
   if(min_proc == 0){
    system_sched_policy =-1;
    release(&ptable.lock);
    continue;
   }
```

5 RMS Scheduling Policy

In RMS Scheduling policy, the priorities are decided on the basis of the rates of the processes. Higher the rate, higher is the priority. In this approach, for every process weights are calculated as decreasing function of rates. The processes with smallest weight is scheduled first.

Implementation:

1. We need to ensure that at every timer interrupt, the process with smallest weight is scheduled

first (with pid number as tie breaker).

2. In the *sched* function in *proc.c*, if the current system policy is 1 i.e. RMS, iterate over the process list and schedule the process with smallest weight which is in $RUNNABLE$ or $RUNNING$ state.

3. In the *trap* function in *trap.c* file, at every timer interrupt, update the current process's elapsed time. If the elapsed time of the process exceeds the execution time, exit from the process.

```
struct proc *min_proc = 0;
      struct proc *min_proc = 0;
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
          continue;
        if(p->sched_policy != 1)
          continue;
        if(min_proc == 0){
          min_proc = p;
        }
        else{
          if((p->weight < min_proc-> weight) || (p->weight == min_proc-> weight
    && p->pid < min_proc->pid)){
            min_proc = p;
          }}}
      if(min_proc == 0){
        system_sched_policy =-1;
        release(&ptable.lock);
        continue;
      }
```

6 EDF Schedulability

While changing the current process scheduling policy, in the $sched_policy$ system call, we need to check whether the process is schedulable with EDF scheduling policy. For that the total utilization bound is checked.
$\because U_{total} \leq 1$
Also since $U_{total} = \sum_{i=1}^{n} \frac{d_i}{p_i}$
To calculate U_{total}, we need to calculate the utilization of each process. Since each process is periodic, we have added the utilization of each process to UEDF when sched_policy is called.
$\implies UEDF + (d/p) * 100 \leq 100$, where UEDF is total utilization of all the processes, d is duration of the given process, p is period of the given process which is equal to its deadline.

```
    if(policy==0){
        int curtot = UEDF + ((p->exec_time)*100)/p->deadline;
        if(curtot>100){
          release(&ptable.lock);
          p->sched_policy =-1;
          //kill the process
          kill(pid);
          return -22;
        }
        else{
          UEDF = curtot;
          system_sched_policy = policy;
          p->arrival_time = ticks;
          p->elapsed_time=0;
        }
      }
```

7 RMS Schedulability

While changing the current process scheduling policy, in the $sched_policy$ system call, we need to check whether the process is schedulable with RMS scheduling policy. For that the total utilization bound is checked.
$\because U_{total} \leq n(2^{1/n} - 1)$
$\implies URMS + d/p \leq Liu\_laybound(n)$, where URMS is total utilization of all the processes, d is duration of the given process, p is period of the given process which is 1/rate.
ALL the Liu-laybound values are precomputed for $n = 1 - 64$ and stored in a global array (to avoid floats, the values are multiplied by a factor of 10000 before storing).

```
    else if(policy==1){
        int U =URMS + p->exec_time*p->rate ;
        int curNP = NPRMS+1;
        if(U*100>liu_layBound[curNP]){
          release(&ptable.lock);
          p->sched_policy =-1;
          //kill the process
          kill(pid);
          return -22;
        }
        else{
          URMS = U;
          NPRMS=curNP;
          system_sched_policy = policy;
          p->arrival_time = ticks;
          p->elapsed_time =0;
        }
      }
```