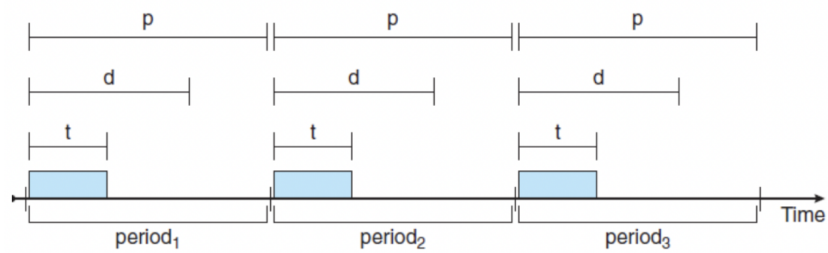


COL331 Assignment 2 - Easy

Real Time Scheduling

Gurarmaan S. Panjeta & Aryan Dua

2020CS50426 & 2020CS50475



Report & Logistics

April 4, 2023



April 4, 2023

Changes to the Code

1 Changes in proc.c

The functions for the system calls are implemented as enlisted below. The system calls themselves are defined further down.

1.1 exec_time

The pseudo-code for the exec_time function is listed below. The control flow and an over-all view of the implementation has been presented with appropriate comments.

This system call finds the process with the given pid , and sets it's exec_time parameter. If the process cannot be located, it returns -22.

```
1 int deadline(int pid, int exec_t){
2     acquire(&ptable.lock);
3     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ //traverse the ptable
4         if(p->pid == pid){ //process with the requisite pid found!
5             p->exec_time = exec_t; //exec-time of the relevant process set
6             release(&ptable.lock);
7             return 0;
8         }
9     }
10    release(&ptable.lock);
11    return -22; //process could not be found, return -22
12 }
```

1.2 deadline

The pseudo-code for the deadline function is listed below. The control flow and an over-all view of the implementation has been presented with appropriate comments.

This system call finds the process with the given pid , and sets it's deadline parameter. If the process cannot be located, it returns -22.

```
1 int deadline(int pid, int deadline){
2     acquire(&ptable.lock);
3     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ //traverse the ptable
4         if(p->pid == pid){ //process with the requisite pid found!
5             p->deadline = deadline; //deadline of the relevant process set
6             release(&ptable.lock);
7             return 0;
8         }
9     }
10    release(&ptable.lock);
11    return -22; //process could not be found, return -22
12 }
```

1.3 rate

The pseudo-code for the rate function is listed below. The control flow and an over-all view of the implementation has been presented with appropriate comments.



April 4, 2023

This system call finds the process with the given pid , and sets it's rate parameter. If the process cannot be located, it returns -22.

```
1 int rate(int pid, int rate){
2     acquire(&ptable.lock);
3     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ //traverse the ptable
4         if(p->pid == pid){ //process with the requisite pid found!
5             p->rate = rate; //rate of the relevant process set
6             release(&ptable.lock);
7             return 0;
8         }
9     }
10    release(&ptable.lock);
11    return -22; //process could not be found, return -22
12 }
```

1.4 rateToWeight

RM scheduling in this assignment uses a weight parameter to set relative priorities of tasks. The function is :=

$$w = \max\left(1, \left\lceil \left(\frac{30-r}{29}\right) * 3 \right\rceil\right)$$

Since rate can belong from [1,30], and weights from [1,3], this function is evaluated for this range and the piecewise definition is listed below:-

```
1 int rateToWeight(int rate){
2     if(rate < 1){return -1;} //input outside allowed range
3     else if(rate < 11){return 3;}
4     else if(rate < 21){return 2;}
5     else if(rate < 31){return 1;}
6     else{return -1;} //input outside allowed range
7 }
```

1.5 isSchedEDF

A subroutine to figure out whether this process is schedulable or not using EDF. Kills the process if not.

A set of processes is EDF schedulable if

$$\sum \frac{exec_time_i}{deadline_i} \leq 1$$

. Since the execution times and deadlines given to us are integers, we can evaluate this fraction and get integer numerator and denominator. Then, *numerator* ≤ *denominator* enforces the same condition! This is implemented below:-

```
1 int num = 0;
2 int denom = 1;
3 struct proc *p;
```



April 4, 2023

```
4  acquire(&ptable.lock);
5  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
6      if(p->policy == 0 && p->pid!=0 &&p->killed ==0){ //process has to be
          included in utilisation!
7          num = num*((p->deadline)-(p->arrival_time)) + denom*(p->exec_time);
8          denom = denom*((p->deadline)-(p->arrival_time));
9      }
10 } // num/den is sigma(ei/pi)
11 release(&ptable.lock);
12 if(num > denom){ //not schedulable, kill the process
13     pmaybe->killed = 1;
14     pmaybe->state = RUNNABLE; //so that exit() can be called on it later
15     return -22;}
16 else{return 0;}
```

1.6 isSchedRM

A subroutine to figure out whether this process is schedulable or not using RM. Kills the process if not.

A set of processes is RM schedulable if

$$\sum_n \frac{exec_time_i}{period_i} \leq n(2^{1/n} - 1)$$

. I evaluate the RHS for $n = 1$ to 14 using a simple cpp code(submitted as "a.cpp").

Since $1/period$ is rate, and given to us, I keep track of the summation of products. Some bookkeeping of time units and for higher accuracy, more digits, asks us to multiply this number by 10000 for comparison. This is implemented below:-

```
1  int scheds[14] = {1000000,828427,779763,756828,743492,\
2  734772,728627,724062,720538,717735,715452,713557,711959,710593};
3  int num = 0; //number of processes
4  int tot = 0; //sum of 10000*(execution time * rate)
5  struct proc *p;
6  acquire(&ptable.lock);
7  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
8      if(p->policy == 1 && p->killed == 0){ //valid proc, add to utilisation
9          num+=1;
10         tot += 10000*(p->exec_time)*(p->rate);
11     }
12 }
13 release(&ptable.lock);
14
15 //if num is too high, assume n = infinity, use const rhs
16 if(num > 14){
17     if (tot > 693147){pmaybe->killed = 1;pmaybe->state==RUNNABLE;return -22;} else{
18         return 0;}}
19
20 else if(tot > scheds[num-1]){ //compare with corresponding n
21     pmaybe->killed = 1;
22     pmaybe->state = RUNNABLE;
23     return -22;}
24 else{return 0;} //schedulable
```



April 4, 2023

1.7 allocproc

Some changes and initialisations needed to be made when a process is first birthed

```
1 p->policy = -1; //Set to default
2 p->elapsed_time = 0;
3 p->arrival_time = 0;
4 p->ticksproc = 0;
5 p->deadline = 0;
```

1.8 sched_policy

The pseudo-code for the sched_policy function is listed below. The control flow and an over-all view of the implementation has been presented with appropriate comments.

This system call finds the process with the given pid, and sets its policy parameter. The arrival time is also set to the number of ticks at this point. The relative deadline (in case of EDF) is decided at this point, and appropriately set. If the process cannot be located, it returns -22.

```
1 int sched_policy(int pid, int policy){
2     struct proc *p;
3     int i = 0;
4     acquire(&ptable.lock);
5     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
6         if(p->pid == pid){ //relevant process found
7             release(&ptable.lock);
8             p->policy = policy;
9             p->arrival_time = (int)ticks;
10
11             if(policy==0){p->deadline += p->arrival_time; i = isSchedEDF(p);}
12             //schedulability check for EDF & deadline set
13             if(policy==1){i = isSchedRM(p);} //schedulability check for RM
14             return i;
15         }
16     }
17     release(&ptable.lock);
18     return -22; //process could not be located
19 }
```

1.9 scheduler

This is the function that does all the scheduling. It has the following basic structure-

An infinite loop that keeps finding the next process to execute. The body of the above loop has a nested loop for traversing the ptable. In the traversal of the ptable, if the process is a default policy proc, it is scheduled immediately. If it follows the EDF policy, we form a third nested loop, and pick the process with the most recent deadline and minimum pid. If it follows the RM policy, we form a third nested loop, and pick the process with the least weight and minimum pid.

The following is the pseudo code for the scheduler



April 4, 2023

```
1 void scheduler(void){
2     struct proc *minP = 0; //the proc that will be picked by the scheduler
3
4     for(;;){
5         sti();
6         acquire(&ptable.lock);
7         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
8             if(p->state != RUNNABLE){continue;} //not runnable anyways
9
10            if(p->policy == -1){minP = p;} //execute on the spot
11            else if(p->policy == 0){ //EDF proc exists, traverse to find min deadline
12                int minDeadline = 214783647;
13                struct proc *q;
14                for(q = ptable.proc; q < &ptable.proc[NPROC]; q++){
15                    if(q->state != RUNNABLE){continue;}
16                    if(q->policy != 0){continue;}
17
18                    if(q->deadline < minDeadline){minDeadline = q->deadline; minP = q;}
19                    else if(q->deadline == minDeadline){if((q->pid) < (minP->pid)){
20                        minP = q;}}
21                }
22                //Found the minPid minDeadline task amongst the list
23            else if(p->policy == 1){ //RM process, traverse to find max weight
24                int minWeight = 4;
25                struct proc *q;
26                for(q = ptable.proc; q < &ptable.proc[NPROC]; q++){
27                    if(q->state != RUNNABLE){continue;}
28                    if(q->policy != 1){continue;}
29
30                    if(rateToWeight(q->rate) < minWeight)
31                        {minP = q; minWeight = rateToWeight(q->rate);}
32                    else if(rateToWeight(q->rate) == minWeight && ((q->pid) < (minP->pid))
33                        ){minP = q;}
34                }
35                //found the minPid minWeight(max priority) task amongst list
36            if(minP->state != ZOMBIE){ //Don't run this, something went wrong, reloop
37                c->proc = minP;
38                switchvm(minP);
39                minP->state = RUNNING;
40                swtch(&(c->scheduler), minP->context);
41                switchkvm();
42                c->proc = 0;
43            }
44            //inner for loop
45            release(&ptable.lock);
46        } //infinite for loop
47    } //function ki last bracket
```



April 4, 2023

2 Changes in trap.c

2.1 Elapsed_time per process

Maintained an attribute `elapsed_time` for each process, incrementing it for every tick the process is running

```
1  if(myproc() != 0 && (tf->cs &3) == DPL_USER && myproc()->policy != -1){
2      myproc()->elapsed_time += 1;
3  }
```

2.2 Terminating the process after it is past its `exec_time`

After every tick, the process was checked if it was EDF/RM and whether it had completed `exec_time` as its `elapsed_time`

```
1  if(myproc() && myproc()->state == RUNNING && (tf->trapno == T_IRQ0+IRQ_TIMER))
2  {
3      if((myproc()->policy != -1) && (myproc()->elapsed_time >= myproc()->exec_time))
4      {
5          cprintf("The arrival time and pid value of the completed process is %d %d\n", myproc()->arrival_time, myproc()->pid);
6          exit();
7      }
8      else {
9          //if process had zombied proc()->killed = 1, need to run it, exit() will take remove it later
10         if(myproc()->state == ZOMBIE){myproc()->state = RUNNABLE;}
11         yield(); //give up the hold on CPU
12     }
13 }
```

3 Changes in proc.h

Setting up new attributes for the struct `proc`

```
1  struct proc {
2      ...
3      int policy;           //policy, default, EDF or RM
4      int deadline;         //deadline of the edf if possible
5      int exec_time;        //exec_time of the edf
6      int rate;             //rate of the rm process
7      int elapsed_time;     //no. of ticks the proc has run
8      int ticksproc;        //not used anywhere though
9      int arrival_time;     //arrival time of the proc, set when policy updated
10 };
```

4 Change in exec.c

```
1  curproc->policy = -1; //for default policy
```



April 4, 2023

5 Implementing the Sys calls

5.1 Changes in usys.S

Defining the sys calls

```
1 SYSCALL(sched_policy)
2 SYSCALL(exec_time)
3 SYSCALL(deadline)
4 SYSCALL(rate)
```

5.2 Changes in syscall.h

Defining the sys calls

```
1 #define SYS_sched_policy 22
2 #define SYS_exec_time 23
3 #define SYS_deadline 24
4 #define SYS_rate 25
```

5.3 Changes in syscall.c

Making the syscalls accessible to other files

```
1 extern int sys_sched_policy(void);
2 extern int sys_exec_time(void);
3 extern int sys_deadline(void);
4 extern int sys_rate(void);
5
6 static int (*syscalls[])(void) = {
7
8 ...
9 [SYS_exec_time] sys_exec_time,
10 [SYS_deadline] sys_deadline,
11 [SYS_rate] sys_rate,
12 };
```

5.4 Changes in sysproc.c

The bodies of the syscalls

```
1 int sys_sched_policy(void){
2     int pid, policy;
3     if(argint(0, &pid) < 0){return -1;} //argument not integer
4     if(argint(1, &policy) < 0){return -1;} //argument not integer
5     return sched_policy(pid, policy);
6 }
7
8 int sys_deadline(void){
9     int pid, deadlin;
10    if(argint(0, &pid) < 0){return -1;} //argument not integer
11    if(argint(1, &deadlin) < 0){return -1;} //argument not integer
12    return deadline(pid, deadlin);
```




April 4, 2023

```
13 }
14
15 int sys_rate(void){
16     int pid, rte;
17     if(argint(0, &pid) < 0){return -1;} //argument not integer
18     if(argint(1, &rte) < 0){return -1;} //argument not integer
19     return rate(pid, rte);
20 }
21
22 int sys_exec_time(void){
23     int pid, exect;
24     if(argint(0, &pid) < 0){return -1;} //argument not integer
25     if(argint(1, &exect) < 0){return -1;} //argument not integer
26     return exec_time(pid, exect);
27 }
```