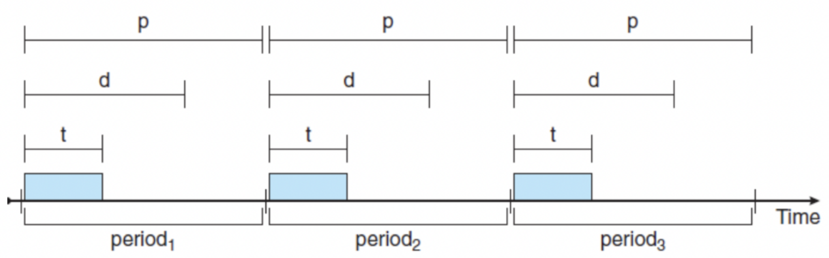
COL331 Assignment 2 - Easy

Real Time Scheduling

Divyanshu Pabia and Aryan Adlakha

RA2111003011373 & RA2111003011372



Report & Logistics

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.

```c
int deadline(int pid, int exec_t){
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ //traverse the ptable
    if(p->pid == pid){ //process with the requisite pid found!
      p->exec_time = exec_t; //exec-time of the relevant process set
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -22; //process could not be found, return -22
}
```

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.

```c
int deadline(int pid, int deadline){
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ //traverse the ptable
    if(p->pid == pid){ //process with the requisite pid found!
      p->deadline = deadline; //deadline of the relevant process set
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -22; //process could not be found, return -22
}
```

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.

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.

```
int rate(int pid, int rate){
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ //traverse the ptable
    if(p->pid == pid){ //process with the requisite pid found!
      p->rate = rate; //rate of the relevant process set
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -22; //process could not be found, return -22
}
```

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:-

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

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

$$\Sigma \frac{exec\_time_i}{deadline_i} \leq 1$$

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

```
int num = 0;
  int denim = 1;
  struct proc *p;
```

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

$$\Sigma_n \frac{exec\_time_i}{period_i} \leq n(2^{1/n} - 1)$$

. 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:-

```
int scheds[14] = {1000000,828427,779763,756828,743492,\
734772,728627,724062,720538,717735,715452,713557,711959,710593};
  int num = 0; //number of processes
  int tot = 0; //sum of 10000*(execution time * rate)
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
         if(p->policy == 1 && p->killed == 0){ //valid proc, add to utilisation
          num+=1;
          tot += 10000*(p->exec_time)*(p->rate);
          }
  }
  release(&ptable.lock);

  //if num is too high, assume n = infinity, use const rhs
  if(num > 14){
  if (tot > 693147){pmaybe->killed = 1;pmaybe->state==RUNNABLE;return -22;} else{ return 0;}}

  else if(tot > scheds[num-1]){ //compare with corresponding n
    pmaybe->killed = 1;
    pmaybe->state = RUNNABLE;
    return -22;}
  else{return 0;} //schedulable
```

1.7 allocproc

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

```
p->policy = -1;  //Set to default
p->elapsed_time = 0;
p->arrival_time = 0;
p->ticksproc = 0;
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 it's 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.

```
int sched_policy(int pid, int policy){
  struct proc *p;
  int i = 0;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){ //relevant process found
        release(&ptable.lock);
        p->policy = policy;
        p->arrival_time = (int)ticks;

        if(policy==0){p->deadline += p->arrival_time;i = isSchedEDF(p);}
       //schedulability check for EDF & deadline set
        if(policy==1){i = isSchedRM(p);} //schedulability check for RM
        return i;
       }
    }
    release(&ptable.lock);
    return -22; //process could not be located
}
```

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

```
void scheduler(void){
  struct proc *minP = 0;  //the proc that will be picked by the scheduler

  for(;;){
    sti();
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE){continue;} //not runnable anyways

      if(p->policy == -1){minP = p;}//execute on the spot
      else if(p->policy == 0){//EDF proc exists, traverse to find min deadline
                int minDeadline = 214783647;
                struct proc *q;
            for(q = ptable.proc; q<&ptable.proc[NPROC]; q++){
                if(q->state != RUNNABLE){continue;}
                if(q->policy != 0){continue;}

                if(q->deadline < minDeadline){minDeadline = q->deadline;minP = q;}
                else if(q->deadline == minDeadline){if((q->pid) < (minP->pid)){minP = q;}}
                }
        } //Found the minPid minDeadline task amongst the list
        else if(p->policy == 1){//RM process , traverse to find max weight
                int minWeight = 4;
                struct proc *q;
            for(q = ptable.proc; q<&ptable.proc[NPROC]; q++){
                if(q->state != RUNNABLE){continue;}
                if(q->policy != 1){continue;}

                if(rateToWeight(q->rate) < minWeight)
                {minP = q; minWeight = rateToWeight(q->rate);}
            else if(rateToWeight(q->rate) == minWeight && ((q->pid) < (minP->pid))){minP = q;}

                }
        }//found the minPid minWeight(max priority) task amongst list
        if(minP->state!= ZOMBIE){//Don't run this, something went wrong, reloop
          c->proc = minP;
          switchuvm(minP);
          minP->state = RUNNING;
          swtch(&(c->scheduler), minP->context);
          switchkvm();
          c->proc = 0;
          }

   }//inner for loop
   release(&ptable.lock);
  }//infinite for loop
}//function ki last bracket
```

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

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

2.2 Terminating the process after it is past it's 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

```
if(myproc() && myproc()->state == RUNNING && (tf->trapno == T_IRQ0+IRQ_TIMER))
     {
     if((myproc()->policy != -1) && (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 {
        //if process had zombied proc()->killed = 1, need to run it, exit() will
    take remove it later
      if(myproc()->state == ZOMBIE){myproc()->state = RUNNABLE;}
        yield(); //give up the hold on CPU
        }
     }
```

3 Changes in proc.h

Setting up new attributes for the struct proc

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

4 Change in exec.c

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

5 Implementing the Sys calls

5.1 Changes in usys.S

Defining the sys calls

```
SYSCALL(sched_policy)
SYSCALL(exec_time)
SYSCALL(deadline)
SYSCALL(rate)
```

5.2 Changes in syscall.h

Defining the sys calls

```
#define SYS_sched_policy 22
#define SYS_exec_time   23
#define SYS_deadline    24
#define SYS_rate        25
```

5.3 Changes in syscall.c

Making the syscalls accessible to other files

```
extern int sys_sched_policy(void);
extern int sys_exec_time(void);
extern int sys_deadline(void);
extern int sys_rate(void);

static int (*syscalls[])(void) = {

...
[SYS_exec_time]   sys_exec_time,
[SYS_deadline]    sys_deadline,
[SYS_rate]        sys_rate,
};
```

5.4 Changes in sysproc.c

The bodies of the syscalls

```
int sys_sched_policy(void){
  int pid, policy;
    if(argint(0, &pid) < 0){return -1;} //argument not integer
    if(argint(1, &policy) < 0){return -1;} //argument not integer
  return sched_policy(pid, policy);
}

int sys_deadline(void){
  int pid, deadlin;
    if(argint(0, &pid) < 0){return -1;} //argument not integer
    if(argint(1, &deadlin) < 0){return -1;} //argument not integer
  return deadline(pid, deadlin);
```

```
}

int sys_rate(void){
  int pid, rte;
    if(argint(0, &pid) < 0){return -1;} //argument not integer
    if(argint(1, &rte) < 0){return -1;} //argument not integer
  return rate(pid, rte);
}

int sys_exec_time(void){
  int pid, exect;
    if(argint(0, &pid) < 0){return -1;} //argument not integer
    if(argint(1, &exect) < 0){return -1;} //argument not integer
  return exec_time(pid, exect);
}
```