COL331 Assignment 2 Report

Submitted by:

Dharmeswar Basumatary (2020CS50423)

Mohd Bilal Bin Rehmat (2020CS50431)

Attributes added in struct proc in the proc.h file are:

```
int exec_time;
 int arrival_time;
  int deadline;
  int elapsed_time;
  int sched_policy;
  //For RMS
  int rate;
  int weight;
```

System calls:

We are required to add the signature of system calls in user.h,defs.h

In usys.S SYSCALL(sched_policy), SYSCALL(exec_time), SYSCALL(deadline), SYSCALL(rate)

1) Sys_sched_policy();

It is implemented in sysproc.c file.

It parses the arguments pid and policy using argint(int, int*) function and calls the _sched_policy(int pid,int policy) defined in proc.c

In _sched_policy() function:

Scaninng the ptable to find the structure of the process p using pid.

If the policy is 0: (EDF)

Checking for the schedulability test U <= 1, where U = Sum of (exec_time/deadline) of all the process to be scheduled.

To implement this, a global variable U is used which store sum of exec_time/deadline for all the process with sched_policy =0.

If U(global variable) + p->exec_time/p->deadline <=1 then p->sched_policy is set to 0 else the the process is killed.

If the policy is 1:(RMS)

Checking for schedulability test for RMS:

U<= n*(2^(1/n)-1)

N is the no of process. A global variable n_rms_procs stores the no of process with sched_policy = 1.

U = SUM (p->exec_time*p->rate/100) (here)

Because rate is ticks/second and 1 tick = 10 milliseconds.

Helper function powr(base,n):

It returns base^(1/n) -1;

Global variable RMS_U is used which stores Utility of all the processes with

sched_policy = 1.

If RMS_scedulability for the process fails, then the process is killed.

Code Snippet:

```
/// Setting sched_policy
int _sched_policy(int pid, int policy)
{
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      break;
    }
  }
  if (policy == 0)
  {
    double temp = U + ((double)p->exec_time / (double)p->deadline);
    if (temp > 1.0)
    {
      release(&ptable.lock);
      kill(pid);
      return -22;
    }
    p->sched_policy = policy;
    p->arrival_time = ticks;
    U = temp;
    release(&ptable.lock);
    return 0;
  }
  else
  {
    // rms_policy = policy;
    double t1 = RMS_U + ((double)(p->exec_time * p->rate) / 100.0);
    double bound = powr((double)2,n_rms_proc+1);
    bound = bound*(double)(n_rms_proc+1);
    if(t1 > bound){
      release(&ptable.lock);
      kill(pid);
      return -22;
    }
   // cprintf("%s %d\n","setpolicy: ",n_rms_proc);
    p->sched_policy = policy;
    p->arrival_time = ticks;
    n_rms_proc++;
    RMS_U = t1;
    release(&ptable.lock);
    return 0;
  }
}

```

2) Sys_exec_time();

It is implemented in sysproc.c file.

It parses the arguments pid and exec_time using argint(int, int*) function

and calls the _exec_time(int pid,int exec_time) defined in proc.c

In _exec_time():

Scaninng the ptable to find the structure of the process p using pid.

Then, p->exec_time = exec_time is set(if not set earlier).

Code Snippet:

```
// Setting exec_time
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)
    {
      if (p->exec_time != -1)
      {
        release(&ptable.lock);
        return -22;
      }
      break;
    }
  }
  p->exec_time = exec_time;
  release(&ptable.lock);
  return 0;
}
```

3) Sys_deadline();

It is implemented in sysproc.c file.

It parses the arguments pid and deadline using argint(int, int*) function

and calls the _deadline(int pid,int deadline) defined in proc.c

In _deadline():

Scaninng the ptable to find the structure of the process p using pid.

Then, p->deadline = deadline is set (if not set earlier).

Code Snippet:

```
171
172 // Setting deadline
173 int _deadline(int pid, int deadline)
174 {
175   struct proc *p;
176   acquire(&ptable.lock);
177   for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
178   {
179     if (p->pid == pid)
180     {
181       if (p->deadline != -1)
182       {
183         release(&ptable.lock);
184         return -22;
185       }
186       break;
187     }
188   }
189   p->deadline = deadline;
190   release(&ptable.lock);
191   return 0;
192 }
193
```

4) Sys_rate();

It is implemented in sysproc.c file.

It parses the arguments pid and rate using argint(int, int*) function

and calls the _rate(int pid,int rate) defined in proc.c

Scaninng the ptable to find the structure of the process p using pid.

P->rate = rate is set. (if not set earlier).

This function also sets the weight of the process.

P->weight = Ceiling((30-rate) *3/29)).

Code Snippet:

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

Implementation of Scheduling Algorithms:

Both the scheduling algorithms **EDF (Earliest Deadline First)** and **RMS (Rate Monotonic Scheduling**) are implemented in the **scheduler()** function in the proc.c along with the original xv6 round robin algorithm.

Schedulability test both EDF And RMS is done during the sys_sched_policy() system call.

If incoming process is not schedulable it is killed in the system call.

EDF (Earliest Deadline First):

Implementation Idea:

Find a EDF schedulable process with minimum deadline+ arrival time from the ptable. If two or more processes have the same value of deadline + arrival time , pick the one with minimum pid and schedule it.

Implementation:

Helper functions:

int get_policy();

It scans the process table and returns an integer 0 or 1 or –1;

If there is a EDF schedulable process it returns 0. If there is a RMS schedulable process it returns 1. If there is both EDF and RMS schedulable processes in the ptable it returns either 0 or 1.(no priority set between EDF and RMS).

If there is no RMS or EDF schedulable process it returns –1;

Code snippet:

```
int get_policy()
{
  for (int i = 0; i < NPROC; i++)
  {
    struct proc temp = ptable.proc[i];
    if (temp.state != RUNNABLE)
      continue;
    if (temp.sched_policy != -1)
    {
      return temp.sched_policy;
    }
  }
  return -1;
}
```

int Find_edf_ind();

It scans the process table and returns the index of a runnable process with minimum deadline+arrival time and minimum pid(if more than one process has same deadline + arrival time) in the process table.

Code snippet:

```
for (int i = 0; i < NPROC; i++)
{
  struct proc temp = ptable.proc[i];
  if (temp.state == RUNNABLE && temp.pid != 0 && temp.sched_policy == 0)
  {
    if (start == 0)
    {
      start = 1;
      index = i;
      min = temp.deadline + temp.arrival_time;
      continue;
    }
    if (temp.deadline + temp.arrival_time < min || (temp.deadline + temp.arrival_time == min && ptable.proc[index].pid > temp.pid))
    {
      min = temp.deadline + temp.arrival_time;
      index = i;
    }
  }
}
return index;
}
```

In the Scheduler () function in proc.c

Schedule_policy = Get_policy() function is called.

If its return value is 0 then, then find_edf_ind() function is called to get the index of the process in the ptable to be scheduled using EDF. Then context switch is done. At the time of context switch elapsed_time of the process is incremented by one (in ticks);

Code Snippet for edf part in scheduler:

```
  else if (schedule_policy == 0)
  {
    int ind = find_edf_ind();
    p = &ptable.proc[ind];
    p->elapsed_time++;
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;

    swtch(&(c->scheduler), p->context);
    switchkvm();

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    c->proc = 0;
  }
```

RMS (Rate Monotonic Scheduling)

Implementation Idea:

Find a RMS schedulable process from the ptable with minimum weight (where weight =

Ceiling((30-rate) *3/29)) .If two or more processes have same weight pick the process with smallest pid).

Implementaion:

Weight of each process is stored in weight attribute of the process when sys_rate (pid,rate) system call is called.

Helper functions:

Int find_rms_ind();

It scans the process table and returns the index of a runnable process with minimum weight and minimum pid(if more than one process has same minimum weight) from the process table.

Code Snippet:

```
for (int i = 0; i < NPROC; i++)
{
  struct proc temp = ptable.proc[i];
  if (temp.state == RUNNABLE && temp.pid != 0 && temp.sched_policy == 1)
  {
    if (start == 0)
    {
      start = 1;
      index = i;
      weight = temp.weight;
      continue;
    }
    if (temp.weight < weight || (temp.weight == weight && ptable.proc[index].pid > temp.pid))
    {
      weight = temp.weight;
      index = i;
    }
  }
}
return index;
```

In the Scheduler () function in proc.c

Schedule_policy = Get_policy() function is called.

If the return value of get_policy is 1 then, then find_rms_ind() function is called to get the index of the process in the ptable to be scheduled using RMS. Then context switch is done. At the time of context switch elapsed_time of the process is incremented by one (in ticks);

Code Snippet for RMS in scheduler function:

```
    }
    else if (schedule_policy == 1)
    {
      int rm_ind = find_rms_ind();
      p = &ptable.proc[rm_ind];
      p->elapsed_time++;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();
      c->proc = 0;
    }
    release(&ptable.lock);
  }
```