COL331 Assignment 2

Ankit Raushan(2020CS10324)
Sibasish Rout(2020CS10386)

Real-time scheduling policies

In operating systems scheduling policies allocate CPU time to processes and also determine which task is going to be run next. Real time scheduling policies ensure tasks with specific requirements are executed in a timely and predictable manner.

There are 2 main type of scheduling policies -

1. Hard Real-time Scheduling :

In hard real-time scheduling missing a deadline can cause catastrophic results. These policies have to make sure that tasks are completed within their deadlines. Earliest Deadline First (EDF) and Rate-Monotonic Scheduling (RMS) are two examples of hard real-time scheduling policies.

2. Soft Real-time Scheduling :

In these systems missing a deadline generally do not cause catastrophic consequences but degrade performance of the system.

Implementation methodology for all the real-time scheduling policies

The basic steps in implementation of a real-time scheduling policy is as following -

- Identify the set of tasks that require real-time scheduling having specific timing requirements like deadlines. These tasks are then characterized using deadline, execution time, rate, priority, etc. based on requirements.
- Appropriate scheduling policies are then applied on these set of tasks to determine their order of execution. CPU time and memory is allocated then to the selected task.

  Some of these real-time scheduling policies are -

i) Earliest Deadline First (EDF) -

In EDF, task with earlier deadlines are given higher priorities then the tasks with later deadlines. Hence task with earliest deadline is scheduled first. In case of draw we generally use lower pid process to declare the winner.

ii) Rate Monotonic Scheduling (RMS) -

In RM policy, task with less weight are given higher priorities then the tasks with more weight. Hence task with least weight is scheduled first. In case of draw we generally use lower pid process to declare the winner.

iii) Deadline Monotonic Scheduling

The DMS algorithm works as follows:

1. Assign a priority to each task based on its deadline. A task with a shorter deadline is assigned a higher priority.
2. If there are multiple tasks with the same deadline, then the task with a smaller period is assigned a higher priority.
3. If there are still multiple tasks with the same deadline and period, then the task with a smaller ID is assigned a higher priority.
4. The scheduler executes the task with the highest priority first.

iv) Priority Inheritance Protocol (PIP)

The steps involved in the priority inheritance protocol are:

1. When a high-priority task requests a shared resource that is currently held by a lower-priority task, the high-priority task is blocked and its priority is temporarily inherited by the lower-priority task.
2. The lower-priority task continues executing with the higher priority until it releases the shared resource.
3. Once the shared resource is released, the priority of the lower-priority task is restored to its original priority, and the high-priority task is unblocked and allowed to execute.

v) Highest Locker Protocol (HLP)

The HLP protocol works in the following manner

1. Let ceil(resource) be the priority of the highest-priority process that can possibly acquire the resource.

2.Once a process acquires a resource, raise its priority to ceil(resource).

The moment a process acquires a resource, it cannot be blocked any more by a low-priority process and thus the problem of deadlocks is solved but however intermediate priority processes may not be able to run.

vi) Priority Ceiling Protocol(PCP)

The steps involved in the priority ceiling protocol are:

1. Assign a priority ceiling to each shared resource. The priority ceiling of a resource is the highest priority of any task that can access the resource.
2. When a task needs to access a shared resource, it checks the priority ceiling of the resource. If the priority ceiling is higher than the task's priority, the task's priority is temporarily raised to the priority ceiling.
3. If a higher-priority task tries to access the same resource while it is held by a lower-priority task, the priority of the lower-priority task is temporarily raised to the priority ceiling of the resource.
4. Once the resource is released, the priority of the task holding the resource is restored to its original priority.

EDF Implementation

Implementing EDF required setting deadline, arrival time and scheduling policy of the process. So at first I defined following attributes for a process - sched_policy, deadline, arrival_time, exec_time, elapsed_time
in proc.h

```
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFI
  struct inode *cwd;
  char name[16];
  int sched_policy;
  int exec_time;
  int deadline;
  int arrival_time;
  int elapsed_time;
```

Then I implemented these system calls to set these attributes -

1. **sys_deadline(int pid, int deadline)**

   To set the deadline sys_deadline system call is made. This is implemented in sysproc.c. If setting deadline was successful then 0 is returned otherwise -22 is returned.

```
int sys_deadline(void){
  int id;
  int deadlin;
  argint(0,&id);
  argint(1,&deadlin);
  int a;
  a=deadline(id,deadlin);
  if(a==0) return 0;
  return -22;
}
```

   sys_deadline call deadline function which I have implemented in proc.c. We first acquire lock over process table and then we iterated to find the required pid and then sets it deadline.

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

  }
  release(&ptable.lock);
  return -22;
}
```

2. sys_exec_time(int pid, int deadline)

To set the exec_time exec_time system call is made. This is implemented in sysproc.c. If setting exec_time was successful then 0 is returned otherwise -22 is returned.

```
int sys_exec_time(void){
  int id;
  int exec_tim;
  argint(0,&id);
  argint(1,&exec_tim);
  int a;
  a=exec_time(id,exec_tim);
  if(a==0) return 0;
  return -22;
}
```

sys_exec_time call deadline function which I have implemented in proc.c. We first acquire lock over process table and then we iterated to find the required pid and then sets it exec_time.

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

3. sys_sched_policy(int pid, int policy)

sys_sched_policy system call is required to set the policy of a task to EDF or RM. This is implemented in sysproc.c. If setting policy was successful then 0 is returned otherwise -22 is returned and process whose policy setting was unsuccessful is killed.

```
// code for scheduling policy
int sys_sched_policy(void){
  int id;
  int pnum;
  argint(0,&id);
  argint(1,&pnum);
  int a;
  a=sched_policy(id,pnum);
  if(a==0) return 0;
  // exit();
  kill(id);
  return -22;
}
```

sys_sched_policy calls sched_policy function implemented in proc.c.

```
int sched_policy(int id,int policy){
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == id){
      if(policy==0 ){
        if(edf(p->exec_time,p->deadline)){
          p->arrival_time=ticks;
          p->sched_policy=0;
          release(&ptable.lock);
          return 0;
        }
        // p->arrival_time=ticks;
        p->sched_policy=2;
      }
      if(policy==1){
        if(rma(p->exec_time,p->rate)){
          p->arrival_time=ticks;
          p->sched_policy=1;
          release(&ptable.lock);
          return 0;
        }
        p->arrival_time=ticks;
        p->sched_policy=2;
      }
    }
  }
  release(&ptable.lock);
  return -22;
}
```

If EDF policy is to be applied then sched_policy attribute of the process is set to be 0 else if RM policy is to be applied then sched_policy attribute of the process is to be set 1 on the task otherwise the task remains in Round Robin policy.
To set the policy we first acquire lock over the process table. Then we iterate over table to find the require policy. We then check whether augmenting the process will yet result in edf schedulability or not. This was done by edf function which iterate over process table to find right pid and add to sum(global variable) execution_time/deadline. If this sum is not greater than 1 then process is schedulable.

```
int edf(int a,int b,int id){
  struct proc *p;
  // float sum=0.0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if( p->pid == id ){
      sum+=(float) p->exec_time/p->deadline;
    }
  }
  // sum+=(float) a/b;
  if(sum<=1.0){
    return 1;
  }
  return 0;
}
```

We then set the policy attribute of the process to 0. We also set arrival_time attribute here and then return 0 which is success. Otherwise the process is killed.

```
  if(edf(p->exec_time,p->deadline,p->pid)){
    p->arrival_time=ticks;
    p->sched_policy=0;
    release(&ptable.lock);
    return 0;
  }
  sum-=(float) p->exec_time/p->deadline;
  p->arrival_time=ticks;
  p->sched_policy=2;
}
```

After these system calls we are set and finally we made change to the scheduler function in proc.c to implement edf policy. Basically among tasks having sched_policy 0 we find for runnable task having earliest deadline. Since deadline was relative we add arrival_time to it. In case of draw we use lower id process to declare the winner.

```
EDF:
for(;;){
  // Enable interrupts on this processor.
  sti();
  // acquire(&ptable.lock);
  int flag=0;
  struct proc *highproc;
  // Loop over process table looking for process to run.
  acquire(&ptable.lock);
  for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
    if(p1->state==RUNNABLE && flag==0){
      highproc=p1;
      flag=1;
      break;
    }

  }
  if(flag==0){
    release(&ptable.lock);
    goto RR;
  }
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state==RUNNABLE && (highproc->deadline+highproc->arrival_time > p->deadline+p->arrival_time || (highproc->deadlir
      highproc=p;
    }
  }
  p=highproc;
  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;
  release(&ptable.lock);
```

This completes our methodology of EDF policy implementation.

RM Policy Implementation

Implementing RM scheduling policy required setting additional attributes rate and weight of the process in proc.h.

```
int rate;
int weight;
```

Then I implemented sys_rate system call to set rate of a system.

1. **sys_rate(int pid, int rate)**

   To set the rate sys_rate system call is made. This is implemented in sysproc.c. If setting deadline was successful then 0 is returned otherwise -22 is returned.

```
int sys_rate(void){
  int id;
  int rat;
  argint(0,&id);
  argint(1,&rat);
  int a;
  a=rate(id,rat);
  if(a==0) return 0;
  return -22;
}
```

sys_rate call deadline function which I have implemented in proc.c. We first acquire lock over process table and then we iterated to find the required pid and then sets it rate.

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

We also set weight attribute here. Weight is set as max(1,ceil(30-rate)*3/29) as we can see from the below code snippet.

```
  p->rate=rat;
  int k=(30-rat)*3;
  int l=k%29;
  k=k/29;
  if(l>0)k++;
  if(k<=1)k=1;
  p->weight=k;
```

2. **sys_sched_policy(int pid,int policy)**

In EDF implementation we have discussed this system call.

```
if(policy==1){
  if(rma(p->exec_time,p->rate)){
    p->arrival_time=ticks;
    p->sched_policy=1;
    release(&ptable.lock);
    return 0;
  }
  p->arrival_time=ticks;
  p->sched_policy=2;
}
```

But in place of edf schedulability we have called rma function to check whether the process will be schedulable or not after augmenting the process

to set of processes under RM policy. To check we iterate over process table to find the right policy then we add to global variable sum1 find sum1, execution_time*rate/100. We also maintained a count(n) of RUNNING/RUNNABLE processes under RM. If this sum is not greater than (2*(2^(1/n)-1) then process is schedulable. Actually we have maintained an array(arr) of 64 length having values (2*(2^(1/n)-1) from n=1 to 64 because xv6 can have at max 64 processes.

```
int rma(int a,int b,int id){
  struct proc *p;
  // float sum=0.0;
  // int npr=1;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == id ){
      npr++;
      sum1+=(float) (p->exec_time*p->rate)/100;
    }
  }
  npr++;
  // sum1+=(float) (a*b)/100;
  if(sum1<=arr[npr-1]){
    return 1;
  }
  return 0;
}
```

After these system calls we are set and finally we made change to the scheduler function in proc.c to implement edf policy. Basically among tasks having sched_policy 1 we find for runnable task having least weight. In case of draw we use lower id process to declare the winner.

```
RM:
for(;;){
  // Enable interrupts on this processor.
  sti();
  // acquire(&ptable.lock);
  int flag=0;
  struct proc *highproc;
  // Loop over process table looking for process to run.
  acquire(&ptable.lock);
  for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
    if(p1->state==RUNNABLE && flag==0){
      highproc=p1;
      flag=1;
      break;
    }

  }
  if(flag==0){
    release(&ptable.lock);
    goto RR;
  }
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state==RUNNABLE && (highproc->weight > p->weight || (highproc->weight==p->weight && highproc->pid > p->pid))){
      highproc=p;
    }
  }
  p=highproc;
  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;
  release(&ptable.lock);
```