COL331 Assignment 2 (EASY) Report

Rajat Bhardwaj : 2020CS50436, Ananya Mathur : 2020CS50416

April 2023

1 Real Time Scheduling

1.1 System Calls

1.1.1 Setting up schedule policy of a process

```
int sys_sched_policy(int pid, int policy)
```

This function is implemented in the 'sysproc.c'. This function extracts the PID and policy from the input and calls the 'setschedpolicy(int pid,int policy)' function implemented in the 'procs.c' file and defined in the header (defs.h) to make it visible to all the files.

```
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {if (p->pid == pid)
    {
      p->sched_policy = policy;
      p->arrival_time = ticks;
      break;
    }
  }
   release(&ptable.lock);
```

The 'setschedpolicy(int pid,int policy)' function (which is explained under 'Schedulability Check' heading) first locks the ptable and traverses it, on finding the required process (with the same PID) it sets the 'sched_policy' equal to the policy from the input.

```
check = schedulability_check(policy);
  if (check == -22)
  {
    p->sched_policy = -1;
    kill(pid);
    return -22;
  }
```

Now we call the 'schedulability_check(int policy)' function to ensure the processes are still schedulable after adding the new process. If they are not schedulable then we set the sched_policy of the new process to -1 (so that it does not interfere in the future schedulability checks) and kill it.

1.1.2 Setting up the execution time of a process

```
int sys_exec_time(int pid, int exec_time)
```

This function is implemented in the 'sysproc.c' file. This function extracts the pid and execution time from the input and calls the 'setexectime(int pid,int exec_time)' function which is implemented in the 'proc.c' file and also defined in the header(defs.h) to make it visible to all the files.

```
struct proc *p;
 acquire(&ptable.lock);
 for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
 {
   if (p->pid == pid)
   {
     p->execution_time = exec_time;
     break;
   }
 }
 release(&ptable.lock);
 if (p->sched_policy != -1)
   return -22;
 return 0;
```

This function traverse the ptable after locking it. As soon as it finds the required process(with the same pid as input) it sets up the 'exec_time' of that process equal the execution time given in the input. It also makes a trivial if the 'sched_policy' of that process was already defined then changing the execution time again will cause problems in the schedulability. Thus we return -22 (as told in the assignment pdf).

1.1.3 Setting up the deadline of a process

```
int sys_deadline(int pid, int deadline)
```

This function is implemented in the 'sysproc.c'. This function extracts the PID and deadline from the input and calls the ' setdeadline(int pid,int deadline)' function implemented in the 'procs.c' file and defined in the header (defs.h) to make it visible to all the files.

```
 struct proc *p;
 acquire(&ptable.lock);
 for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
 {
   if (p->pid == pid)
   {
     p->deadline = deadline;
     break;
   }
 }
 release(&ptable.lock);
 if (p->sched_policy != -1)
   return -22;
 return 0;
```

After locking the ptable, this function iterates it until it finds the required process with the PID equal to the given PID. Then it just sets the deadline of the pid equal to the given deadline. This function also makes a trivial check. If we have already assigned a 'sched_policy' for this process then it might cause a problem in the schedulability which can be handled. Since we are told to return -22 in the assignment pdf, we are doing that.

1.1.4 Setting up the rate of a process

```
int sys_rate(int pid, int rate)
```

This function is implemented in the 'sysproc.c'. This function extracts the PID and rate from the input and calls the ' setrate(int pid,int rate)' function implemented in the 'procs.c' file and defined in the header (defs.h) to make it visible to all the files.

```
struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid == pid)
    {
      p->rate = rate;
      int w = (30 - rate) * 3 + 28;
      w = w / 29;
      if (w < 1)
        w = 1;
      p->weight = w;
      break;
    }
  }
  release(&ptable.lock);
  if (p->sched_policy != -1)
    return -22;
  return 0;
```

We iterate in the ptable after locking it. After finding the required process with the given pid. We set the weight parameter of the process according to the given formula

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

To handle the 'ceil' we add 28 in the numerator (after multiplying it by 3) so that if the numerator is not divisible by the denominator(29) then the output is always 1 more than the normally what we would have gotten without adding 28. But if the numerator was divisible then we do not get an incremented output.

1.2 Schedulability Check

As mentioned previously, 'schedulability_check(int policy)' function is called in the 'setschedpolicy(int pid, int policy)' after assigning the sched_policy of the process. This function is also implemented in the **proc.c** file

```
if (policy == 0)
  {
    float sum = 0;
    struct proc *p;
    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->sched_policy == 0)
        sum += ((float)p->execution_time) / p->deadline;
    }
    release(&ptable.lock);
    if (sum > 1)
      return -22;
    return 0;
  }
```

The above check is for the EDF scheduling policy. We calculated the utilization by taking the sum of the fraction of the execution time and the deadline of all the processes for which the **sched_policy** was equal to 0(EDF) Now we check if the sum of the utilization is strictly greater than 1 then the processes are not schedulable thus we return -22 and also kill the process (in the parent function) else we return 0 (it is schedulable)

```
else
  {
    float sum = 0;
    int num_runnable = 0;
    struct proc *p;
    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->sched_policy == 1)
      {
        num_runnable += 1;
        sum += (float)p->execution_time * p->rate;
      }
    }
    release(&ptable.lock);
    if (sum > 100 * lielay[num_runnable])
      return -22;
    return 0;
  }
```

The above check is for the RMA scheduling policy. We have precomputed the values for the Liu and Layland's upper-bound as $n(2^{\frac{1}{n}} - 1)$ for n = 0 till 64 and stored it in array **lielay**[] We calculated the utilization as follows. First, we have to scale the rate (from per second to per 10 milliseconds) and take the reciprocal to find the period (in ticks) then divide the execution time by the period we found. Instead, we just multiply the execution time with the rate and take the sum for all processes and make the comparison with 100 times the liu and layland's number calculated as $n(2^{\frac{1}{n}} - 1)$ and stored in **lielay**[]the processes are not schedulable if the utilization is less than or equal to the upperbound, then we return 0 else we return -22 and also kill the process (in the parent function)

1.3 Scheduler

```
void scheduler(void)
{

  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for (;;)
  {

    sti();
    int policy = -1;
    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->state == RUNNABLE && (p->sched_policy == 0 || p->sched_policy == 1))
      {
        policy = p->sched_policy;
        break;
      }
    }
    release(&ptable.lock);
```

To implement EDF and RM scheduling policies we modified the function scheduler in proc.c. We added code in the scheduler after interrupts are enabled to determine the current scheduling policy. This was done by iterating through the ptable and as soon as a runnable process is found with policy as 0 (EDF) or 1(RM) we break out of the iterating for loop. In case there is no such process, policy variable holds default value -1. (usual Round Robin scheduler of xv6)

1.3.1 EDF Scheduling (policy = 0)

```
    if (policy == 0)
    {
      int earliest_dl = 0, earliest_dl_pid = 0;
      struct proc *hp_process = myproc();
      acquire(&ptable.lock);

      for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
      {
        if (p->state == RUNNABLE)
        {
          earliest_dl = p->deadline + p->arrival_time;

          earliest_dl_pid = p->pid;
          hp_process = p;
          break;
        }
      }
      release(&ptable.lock);
      acquire(&ptable.lock);

      for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
      {

        if (p->state != RUNNABLE)
          continue;

        if (p->deadline <= earliest_dl)
        {
          if ((p->deadline + p->arrival_time < earliest_dl) || (p->deadline + p->
    arrival_time == earliest_dl && p->pid < earliest_dl_pid))
          {
            earliest_dl = p->deadline + p->arrival_time;
            earliest_dl_pid = p->pid;
            hp_process = p;
          }
        }
      }

      c->proc = hp_process;
      switchuvm(hp_process);
      hp_process->state = RUNNING;

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

      c->proc = 0;
      release(&ptable.lock);
    }
```

We declare integer variables earliest_dl, earliest_dl_pid to hold the values of the earliest deadline of any runnable process and the pid of the process corresponding to it respectively. We also declare hp_process, which is a pointer to a struc proc. It points towards the struct proc of the process with earliest deadline among all runnable processes.

Initially these hold respective values for the first runnable process in ptable. Then, we iterate the ptable once again and check whether there exists a runnable process with lower deadline than the initialised values. If yes, we update the declared variables.

Finally, the cpu is alloted this process and it is scheduled.

1.3.2 RM Scheduling (policy = 1)

```
else if (policy == 1)
    {

      int min_w = 0, min_w_pid = 0;

      struct proc *hp_process = myproc();
      acquire(&ptable.lock);

      for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
      {
        if (p->state == RUNNABLE)
        {
          min_w = p->weight;
          min_w_pid = p->pid;
          hp_process = p;
          break;
        }
      }

      for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
      {

        if (p->state != RUNNABLE)
          continue;

        if (p->weight <= min_w)
        {
          if (p->weight < min_w || (p->weight == min_w && p->pid < min_w_pid))
          {
            min_w = p->weight;
            min_w_pid = p->pid;
            hp_process = p;
          }
        }
      }

      c->proc = hp_process;
      switchuvm(hp_process);
      hp_process->state = RUNNING;

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

      c->proc = 0;
      release(&ptable.lock);
    }
    }
```

We declare integer variables min_w, min_w_pid to hold the values of the lowest weight (highest priority) of any runnable process and the pid of the process corresponding to it respectively. We also declare hp_process, which is a pointer to a struc proc. It points towards the struct proc of the process with lowest weight (highest priority) among all runnable processes.

Initially these hold respective values for the first runnable process in ptable. Then, we iterate the ptable once again and check whether there exists a runnable process with lower weight than the initialised values. If yes, we update the declared variables.

Finally, the cpu is alloted this process and it is scheduled.

1.3.3 Round Robin Scheduling (default) (policy = -1)

```
   else
    {

      acquire(&ptable.lock);
      for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
      {
        if (p->state != RUNNABLE)
          continue;

        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;

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

        c->proc = 0;
      }
      release(&ptable.lock);
    }
  }
}
```

In case the policy variable holds -1, the default xv6 round robin scheduler is run.