COL331: Operating Systems
Assignment 2: Implementation of Real-Time Scheduling Policies

Ishita Chaudhary
2019CS10360

1. **Implementing relevant system calls and other changes**

   We work on a new branch of xv6 (link to base code of xv6). To run the xv6 OS on QEMU virtual machine, I installed Qemu on my local machine. The following parts were implemented and tested on the local machine (Ubuntu version 20.04) as well. The number of CPUS is set to 1 in Makefile.

   To store process-specific features like the deadline, execution time, etc. We add the following member variables to struct proc in proc.h file:

   1. exec_time: stores the number of ticks needed for a process to execute.
   2. deadline: stores the number of ticks a process should complete from its arrival time.
   3. arrival_time: stores the global value of ticks when the process arrives. This is set to the global tick value when the scheduling policy of a process is updated to a real-time schedule policy (EDF/RM).
   4. elapsed_time: stores the number of ticks for which the process has run at any point.
   5. sched_policy: determines the scheduling policy type of a process: -1 for default round robin of xv6, 0 for EDF, and 1 for RM.
   6. rate: stores the rate (=1/period) of a process; unit: instances per second. This is used to calculate the weight of a process in the RM scheduling policy.

We initiate all these variables in the allocproc function of proc.c, it is called for every UNUSED process to initiate its variables, and then the state of the process is changed to EMBRYO. Hence, we initiate all variables to 0, except sched_policy, which is initiated to -1 (denoting default round-robin scheduling for xv6).

Now we must add the following system calls in xv6:

1. *sys_sched_policy:* this system call takes the pid and the policy (0 for EDF and 1 for RM) as input and sets the policy of a process with the given pid. It also performs the schedulability check for both algorithms and kills the process if it is not schedulable.
2. *sys_exec_time:* this system call takes the pid and the execution time as input and sets the execution time of a process with the given pid.
3. *sys_deadline:* this system call takes the pid and the absolute deadline as input and sets the deadline of a process with the given pid.
4. *sys_rate:* this system call takes the pid and the rate as input and sets the rate of a process with the given pid.

All of these system calls perform some basic checks (like pid >0, 1<=rate<=30, deadline/rate/exec_time are positive, and the policy is 0 or 1). If these basic checks fail, and/or the given pid is not present in the ptable (might be killed or exited), the system call returns -22. If it successfully sets the desired variable to its desired value, it returns 0.

To add a new system call of type sys_name, the following changes were made in the respective files:

- usys.S: we added the line ”SYSCALL(name)”
- user.h: add the line ”int name(signature of parameters separated by comma)”. This file contains the function signature of all system calls.
- syscall.c: add ”extern int sys name(void)” and ”[SYS name] sys name” where other system calls are defined in the syscalls[] array. This is where we give the pointer to the syscall function.
- syscall.h: add ”#define SYS name custom integer”; this maps the system call to a unique number which becomes its ID.
- sysproc.c: this is where the function of each system call is implemented. The syscall() function in syscall.c calls these functions.

For each of these above system calls, we need to iterate over the ptable, where all processes are stored. If the pid matches the given pid, we update the desired variable. Since we cannot access ptable outside of proc.c, hence we create helper functions in proc.c, which are called in the respective system call implementations of sysproc.c. We acquire the lock of the table so that it is not edited by another process while iterating over it. We release this lock once we are done. The syntax is taken from other system calls defined in xv6, which access the ptable.

Since we made a helper function in proc.c, we need to include the function signature in defs.h.

```c
int set_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){
        p->sched_policy=policy;
        p->arrival_time=ticks;
        //cprintf("arrival time for pid %d is %d\n", p->pid, p->arrival_time);
        release(&ptable.lock);
        return 0;
      }
    }
    release(&ptable.lock);
    return -22;
}
```

```c
int
sys_sched_policy(int pid, int policy)
{
  if (argint(0, &pid) < 0 || argint(1, &policy)<0) return -22;
  int check_if_pid_present= set_sched_policy(pid, policy);
  if(check_if_pid_present==-22) return -22;

  if (policy==0){ //EDF
    int schedulable = is_edf_schedulable(pid);
    //cprintf("Got Schedulability %d\n", schedulable);
    if (schedulable == -22){
        //cprintf("Killing process %d as it is not schedulable\n", pid);
        kill(pid);
      }
    return schedulable;
  }else if(policy==1){ //RMS
    int schedulable = is_rms_schedulable(pid);
    //cprintf("Got Schedulability %d\n", schedulable);
    if (schedulable == -22){
        //cprintf("Killing process %d as it is not schedulable\n", pid);
        kill(pid);
      }
    return schedulable;
  }else{
    return -22;
  }
}
```

2. **Schedulability Checks**
   For both these checks, we assume the tasks are periodic.
   - **EDF Schedulability Check**
     We assume that for EDF, the period is the same as the deadline. Hence, if the sum of utility of all runnable/running processes in ptable is less than equal to 1, the processes are schedulable. The utility of a process is defined as the fraction of its execution time by its period. To avoid floating point arithmetic operations, we multiply our numerator by 100 and check the sum against 100.

```
int is_edf_schedulable(int pid){
  int utility_sum=0;
  struct proc *p;
  acquire(&ptable.lock);
    //cprintf("Starting check\n");
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->sched_policy==0 && (p->state==RUNNABLE || p->state== RUNNING)){
        utility_sum+=100*p->exec_time/p->deadline;
        //cprintf("Current utility: %d after adding for pid: %d\n",utility_sum,p->pid);
      }

    }
    if(utility_sum>100){
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid==pid){
          p->state=UNUSED;
          break;
        }
      }
    }
    release(&ptable.lock);
    //cprintf("Starting check\n");
    if (utility_sum<=100) return 0;
    return -22;
}
```

   - **RM Schedulability Check**
     For RM Scheduling, we use Liu Layland bound, i.e., n processes are definitely schedulable if the sum of their utilities is less than or equal to $n(2^{1/n}-1)$. Here, we find the nth root of 2 using the bisection method. Since the rate is inverse of the period, the utility of a process is (rate*execution time/100) as the unit of rate is per second and that of the period was msec.

     If a process fails the schedulability check, its state in ptable is updated to UNUSED so that its utility does not get added when any other process is added.

```
int is_rms_schedulable(int pid){
  int utility_sum=0;
  int num_proc=0;
  struct proc *p;
  acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->sched_policy==1 && (p->state==RUNNABLE || p->state==RUNNING)){
        utility_sum+=p->exec_time*p->rate;
        num_proc++;
      }
    }
    int max_util = 100*num_proc*(nth_root_2(num_proc)-1);
    if(utility_sum>max_util){
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid==pid){
          p->state=UNUSED;
          break;
        }
      }
    }
    release(&ptable.lock);


    //cprintf("Got max_util %d for num proc= %d and utility_sum = %d\n", max_util, num_proc, utility_sum);
    if (utility_sum<=max_util) return 0;

    return -22;
}
```

3. Scheduling Algorithms

Changing the Scheduler in proc.c

Assuming all processes will have either of the scheduling policies, we find out the current policy by iterating over the ptable and ignoring the round-robin policy of init and sh processes. We then choose our process to schedule according to the policy. We have put the default scheduler in the choose_round_robin function (if no real-time scheduling policy exists). Otherwise, we call choose_edf_process or choose_rm_process, respectively. All other code of the scheduler involving the context switch and changing the state of the process to RUNNING remains the same.

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

// Switch to chosen process.  It is the process's job
// to release ptable.lock and then reacquire it
// before jumping back to us.

if (policy == -1){
  p = choose_round_robin();
}else if (policy == 0){
  //cprintf("curr policy is %d\n", policy);
  //cprintf("Calling choose func\n");
  p = choose_edf_process();
} else{
  //cprintf("Calling choose func\n");
  p = choose_rm_process();
}
```

1) **Earliest Deadline First Scheduling**
   EDF Scheduling is a dynamic priority scheduling algorithm, i.e., a task with the closest deadline will have the highest priority and will be scheduled first.

   choose_edf_process: we iterate over the ptable and check if the process is in runnable or running state, and its policy is EDF, then the process with the nearest (lowest) relative deadline is chosen to schedule. In the case of ties, the lower pid is scheduled first.

```
struct proc* choose_edf_process(void)
{
    //cprintf("Inside choose func\n");
    struct proc *p, *best_p=0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if((p->state == RUNNABLE || p->state == RUNNING) && p->sched_policy==0){
        //cprintf("checking proc: %d\n", p->pid);
        if(   best_p==0 ||
              best_p->deadline+best_p->arrival_time>p->deadline+p->arrival_time ||
              (best_p->deadline+best_p->arrival_time==p->deadline+p->arrival_time && best_p->pid>p->pid))
        {
          best_p=p;
        }
      }
    }
    //cprintf("Scheduling process %d\n",best_p->pid);
    return best_p;
}
```

2) **Rate Monotonic Scheduling**

RMS is a static priority scheduling algorithm, i.e., the priorities of tasks remain the same. And is calculated by the formula:

$$weight=max(1, ceil(\ 3x(30-rate)/29\ ))$$

$$priority=1/weight$$

The process with the highest priority is executed first.

choose_rm_process: we iterate over the ptable and check if the process is in runnable or running state, and its policy is RM, then the process with the lowest weight is chosen to schedule. In the case of ties, the lower pid is scheduled first.

```
struct proc* choose_rm_process(void){
    //cprintf("Inside choose func\n");
    struct proc *p, *best_p=0;
    int best_weight = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if((p->state == RUNNABLE || p->state == RUNNING) && p->sched_policy==1){
        //cprintf("checking proc: %d\n", p->pid);
        // Calculating Weight
        double val = 3*(30-(p->rate))/29;
        int val_int = val;
        if (val-val_int*1.0 > 0){
          val_int += 1;
        }
        int cur_weight = 1;
        if (val_int>cur_weight){
          cur_weight = val_int;
        }

        if (best_weight == 0){
          best_p = p;
          best_weight = cur_weight;
        }else{
          if (cur_weight< best_weight || (cur_weight==best_weight && p->pid< best_p->pid)){
              best_p = p;
              best_weight = cur_weight;
          }
        }
      }
    }
    //cprintf("Scheduling process %d\n",best_p->pid);
    return best_p;
}
```

Preempting a process

In both real-time scheduling processes, we must exit a process if it has completed its execution time. For this, we edit the trap function in the trap.c file. The elapsed_time variable is incremented each time a process is in its running state. If a process has a real-time scheduling policy and its elapsed_time is greater than or equal to its execution time, we exit from the process.

4. **Testing**

The submitted version can be tested by putting the test case in a test.c file. Compiling the xv6 using the terminal commands "make clean && make && make qemu-nox" and then on the terminal of xv6 run the system call "test". This gives us the desired output. We added _test in UPROGS and test.c in EXTRA in Makefile.

All the tests from assig2_*.c passed on the implementation, giving correct outputs and adhering to their scheduling policies, i.e., they were scheduled and completed according to their priorities. The process that failed the schedulability check were not scheduled and killed.