# COL331 Operating Systems Assignment 2 (Easy)

Pratik Nimbalkar (2020CS10607)
Dhananjay Sapawat (2019CS10345)

## 1 Real Time Scheduling

We have added 4 system calls namely sys_sched_policy, sys_exec_time, sys_deadline, sys_rate. Also we have implemented EDF and RMS Scheduling policies and checked the schedulability of a given process depending on its scheduling policy.
Here is the explanation of all our work in detail.

1. **int sys_sched_policy(int pid, int policy)**
   We have declared the system call in sysproc.c as shown in the image below. It makes a call to Process_sched_policy function which is defined in proc.c

```c
int sys_sched_policy(void)
{
  return Process_sched_policy();
}
```

**Process_sched_policy function:**

This function takes two integer arguments: the PID (process identifier) of the process to be scheduled and the policy number that corresponds to the scheduling policy to be assigned to the process.

The function begins by checking if the arguments have been properly passed by calling the argint function. If either of the arguments is not passed correctly, the function returns -1 indicating an error.

If the arguments are properly passed, the function acquires the lock on the process table to prevent other threads from modifying it while it is being processed. It then iterates through the process table looking for a process with a matching PID. If it finds the process, it retrieves its deadline, execution time, and rate from the process control block. It then sets the scheduling policy for the process as specified by the second argument and sets the arrival time of the process to the current system time (measured in ticks). The function then releases the lock on the process table and proceeds to apply the scheduling policy.

If the policy is 0 (which corresponds to the EDF scheduling policy), the function calculates the utilization factor (u) of the process using the formula u = u + (10000 * e / d), where e is the execution time of the process and d is its deadline. If the calculated utilization factor exceeds 10000 (which is the maximum allowable value), the function terminates the process using the kill function, returns an error code of -22, and restores the previous value of u. If the utilization factor is within the allowable range, the function returns 0 indicating success.

If the policy is not 0 i.e. is 1 (which corresponds to the RMS scheduling policy), the function calculates the utilization factor using the formula u = u + 100 * e * r, where e is the execution time of the process and r is its rate. It also increments the RMS number (rms_no) to keep track of the number of processes that have been assigned the RMS scheduling policy. If the calculated utilization factor exceeds the threshold specified by the current RMS number (rms_limit[rms_no - 1]), the function terminates the process using the kill function, returns an error code of -22, and restores the previous value of u and rms_no. If the utilization factor is within the allowable range, the function returns 0 indicating success.

Finally, if the function fails to find a process with the specified PID, it returns an error code of -22 indicating an invalid PID.

```
static struct proc *initproc;
int u = 0;
int nextpid = 1;
extern void forkret(void);
extern void trapret(void);

int rms_limit[64] =
{10000,8284,7797,7568,7434,7347,7286,7240,7205,7177,7154,7135,7119,7105,
int rms_no = 0;
```

We declared and defined u, nextpid, rms_limit array and rms_no here in proc.c
The rms_limit array is a 64 member array which gives the value of utilisation U for n = 1 to n= 64 where n is the number of processes. This will help us to write the code for checking the schedulability of RMS process when number of process reached till now is from 1 to 64.

```
int Process_sched_policy(void){
  int pid, policy;
  if (argint(0, &pid) < 0 || argint(1, &policy) < 0) {
      return -1;
  }
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (p->pid == pid) {
          int d = p->deadline;
          int e = p->exec_time;
          int r = p->rate;
          p->sched_policy = policy;
          p->arrival_time = ticks;
          release(&ptable.lock);
          if(policy == 0){
            u = u + (10000*e/d);
            if(u > 10000){
              kill(pid);
              u = u - (10000*e/d);
              return -22;
            }
            else{
              return 0;
            }
          }
          else{
            u = u + 100*e*r;
            rms_no++;
            if(u > rms_limit[rms_no - 1]){
              kill(pid);
              u = u - 100*e*r;
              rms_no--;
              return -22;
            }
            else{
              return 0;
            }
          }
      }
  }
  release(&ptable.lock);
  return -22; // Invalid PID
}
```

2. **int sys_exec_time(int pid, int exec_time)**
   We have declared the system call in sysproc.c as shown in the image below. It makes a call to Process_exec_time function which is defined in proc.c

```
int sys_exec_time(void)
{
    return Process_exec_time();
}
```

### Process_exec_time function:

```c
int Process_exec_time(void){
  int pid, exec_time;
  if (argint(0, &pid) < 0 || argint(1, &exec_time) < 0) {
      return -1;
  }
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (p->pid == pid) {
          p->exec_time = exec_time;
          release(&ptable.lock);
          return 0;
      }
  }
  release(&ptable.lock);
  return -22; // Invalid PID
}
```

It takes two integer arguments: the PID (process identifier) of the process and the new execution time to be assigned to the process.

The function begins by checking if the arguments have been properly passed by calling the argint function. If either of the arguments is not passed correctly, the function returns -1 indicating an error.

If the arguments are properly passed, the function acquires the lock on the process table to prevent other threads from modifying it while it is being processed. It then iterates through the process table looking for a process with a matching PID. If it finds the process, it updates its execution time with the new value passed in the second argument. The function then releases the lock on the process table and returns 0 indicating success.

If the function fails to find a process with the specified PID, it returns an error code of -22 indicating an invalid PID.

### 3. int sys_deadline(int pid, int deadline)

We have declared the system call in sysproc.c as shown in the image below. It makes a call to Process_deadline function which is defined in proc.c

```c
int sys_deadline(void)
{
    return Process_deadline();
}
```

**Process_deadline function:**

```c
int Process_deadline(void){
  int pid, deadline;
  if (argint(0, &pid) < 0 || argint(1, &deadline) < 0) {
      return -1;
  }
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (p->pid == pid) {
          p->deadline = deadline;
          release(&ptable.lock);
          return 0;
      }
  }
  release(&ptable.lock);
  return -22; // Invalid PID
}
```

It works in exact same way as Process_exec_time function, only difference being that instead of execution time, the process's deadline is updated with the 2nd argument passed in the function i.e. int deadline.

## 4. int sys_rate(int pid, intrate)

```c
int sys_rate(void)
{
    return Process_rate();
}
```

We have declared the system call in sysproc.c as shown in the image below. It makes a call to Process_rate function which is defined in proc.c

Process_rate function works in exact same way as Process_exec_time and Process_deadline function, only difference being that instead of execution time and deadline, the process's rate is updated with the 2nd argument passed in the function i.e. int rate.

```
int Process_rate(void){
  int pid, rate;
  if (argint(0, &pid) < 0 || argint(1, &rate) < 0) {
      return -1;
  }
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if (p->pid == pid) {
          p->rate = rate;
          release(&ptable.lock);
          return 0;
      }
  }
  release(&ptable.lock);
  return -22; // Invalid PID
}
```

## RMS and EDF scheduling:

```
void
scheduler(void)
{
  struct proc *p;
  struct proc *p2 = NULL;
  struct cpu *c = mycpu();
  c->proc = 0;
  for(;;){
    // Enable interrupts on this processor.
    sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->sched_policy == 0){
            if(p->state == RUNNABLE && (p2 == NULL || edf_priority(p2) > edf_priority(p) ) ){
              p2 = p;
            }
        }
        else{
            if(p->state == RUNNABLE && (p2 == NULL || rms_priority(p2) > rms_priority(p) ) ){
              p2 = p;
            }
        }
      }
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      if (p2 != NULL) { // add a check to make sure p2 is not NULL
        c->proc = p2;
        switchuvm(p2);
        p2->state = RUNNING;
        if(p2->start_time == 0){
          p2->start_time = ticks;
        }
        p2->elapsed_time++;
        swtch(&(c->scheduler), p2->context);
        switchkvm();

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

    p2 = NULL; // reset p2 for the next loop
    release(&ptable.lock);

  }
}
```

```c
int edf_priority(struct proc *p){
  return abs(p->deadline + p->start_time);
}

int rms_priority(struct proc *p){
  int numerator = (30 - p->rate)*3;
  int m = numerator % 29;
  int a = numerator / 29;
  if(a<1){
    return 1;
  }
  else if(m == 0){
    return a;
  }
  else{
    return a + 1;
  }
}
```

The scheduler function is responsible for deciding which should be the next process that should run on the CPU. It uses the scheduling policy assigned to each process to determine the order in which to execute the processes.
We have initialised two pointers to a process structure, p and p2, to null, and a pointer to a cpu structure, c, to the current cpu. And set the proc field of c to null.
Inside the 2nd for loop in the function (which is inside the 1st for loop), we are traversing through the ptable.

Next, it checks the scheduling policy assigned to the process. If the policy is EDF (Earliest Deadline First), it checks if the process is in the RUNNABLE state, if p2 is null, or if the priority of the current process p is higher than that of p2. If all these conditions are met, it sets the p2 pointer to p.
The edf_priority function simply states that a process with more deadline is given a higher edf_priority value. Hence if the deadline of p2 is more i.e. edf_priority more then we will schedule p (with less deadline). Hence we have p2 as p because always p2 is getting scheduled and after making p2 as p, p would get scheduled.

If the scheduling policy is RMS (Rate-Monotonic Scheduling), it checks if the process is in the RUNNABLE state, if p2 is null, or if the priority of the current process p is higher than that of p2. If all these conditions are met, it sets the p2 pointer to p. In the rms_priority function, instead of deadline, w i.e. weight is used to give priorities to processes.
Here w is

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

We set the rms_priority value of p2 more than that of p if its w value is more and hence we schedule p in such a case where rms_priority of p2 is more by assigning p2 to p.

Please note that  the numbers edf_priority and rms_priority do not represent that if the value of them is more then priority of process is more or we need to schedule it first. It only gives indication that the deadline of that process is more and w is more respectively.

## Change in trap.c

```
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER){

      if((myproc()->sched_policy == 0) && (myproc()->elapsed_time >= myproc()->exec_time) && (myproc()->pid > 2)){
        cprintf("The arrival time and pid value of the completed process is %d %d\n", myproc()->arrival_time, myproc()->pid);
        exit();
       }
       else if((myproc()->sched_policy == 1) && (myproc()->elapsed_time >= myproc()->exec_time) && (myproc()->pid > 2)){
        cprintf("The arrival time and pid value of the completed process is %d %d\n", myproc()->arrival_time, myproc()->pid);
        exit();
       }
       else{
          yield();
       }

    }
```

We have added the given code by TA on piazza in trap.c file.