# COL331: Operating Systems
# Assignment 2 Report


- Shreya Arora (2019CS10401)
- Shreejeet Golhait (2019CS10351)


# Implementation Methodology of Real Time Scheduling Policies

## 1. Earliest Deadline First (EDF) Scheduling Algorithm

This scheduling algorithm ensures that the process with the earliest absolute deadline is always scheduled next. The scheduler function continuously keeps on going in a loop in order to select the next process to be executed from the list of processes that have the state RUNNABLE. We begin by acquiring the lock on the process table (ptable). We then iterate and traverse over all the processes in the process table to select the next process to be executed. For each process, we check if the process is in the RUNNABLE state. If it is not RUNNABLE, then we can skip this process from the scheduler algorithm.

If the sched_policy of the function is 0, then that means that the scheduling policy is EDF. We evaluate the absolute deadline of the current process by adding the start_time (or arrival time) of the process to the relative deadline of the process (relative to the arrival time of the process) and then we compare the absolute deadline of this process with the deadline of the current scheduled process. If the deadline of this process is earlier then we schedule it, else we continue to loop.

Once the next process to be scheduled is selected, we update the CPU's current process as the process selected based on the algorithm above. We consequently perform the context switch to save the state of the currently RUNNING process and then get the selected process to run. This is done by updating the state of the selected process as RUNNING and switching its virtual memory. Once the selected process has been completed or if it yields the CPU, then we switch back to the scheduler context and release process table lock.

```
else if (p->sched_policy == 0){
  if (pr == 0 || ((p->start_time + p->deadline) < (pr->start_time + pr->deadline))){
    pr = p;
  }
}
```

## 2. Rate Monotonic (RM) Scheduling Algorithm

In this scheduling algorithm, we assign priorities to different tasks based on their frequency or rate. In our context, we define the weight field (lying between 1 to 3) value of each process based on its rate field (lying between 1 to 30) using the expression given in the problem statement. This weight is symbolic of the priority of the process. We follow the priority order such that lower the weight, higher is the

priority. Thus, in this manner, we ensure that the process with a higher priority (or lower weight) always executes before any other process with a lower priority (or higher weight).

The scheduler function continuously keeps on going in a loop in order to select the next process to be executed from the list of processes that have the state RUNNABLE. We begin by acquiring the lock on the process table (ptable). We then iterate and traverse over all the processes in the process table to select the next process to be executed. For each process, we check if the process is in the RUNNABLE state. If it is not RUNNABLE, then we can skip this process from the scheduler algorithm.

If the sched_policy of the function is 1, then that means that the scheduling policy is RM. We compare the weight field value of the current process with the weight field value of the current scheduled process. If the weight of this process is lesser than than the current scheduled process i.e. if its priority is higher than the current scheduled process then we select this process to be scheduled for execution next. In case of a tie between the weight values of the current process and the current scheduled process, then the process with the smaller pid between the two is selected to be scheduled for execution next in order to break the tie.

Once the next process to be scheduled is selected, we update the CPU's current process as the process selected based on the algorithm above. We consequently perform the context switch to save the state of the currently RUNNING process and then get the selected process to run. This is done by updating the state of the selected process as RUNNING and switching its virtual memory. Once the selected process has been completed or if it yields the CPU, then we switch back to the scheduler context and release process table lock.

```
else if (p->sched_policy == 1){
  if (pr == 0 || p->weight < pr->weight){
    pr = p;
  }
  else if (p->weight == pr->weight && p->pid < pr->pid){
    pr = p;
  }
}
```

The default scheduling algorithm continues to be the Robin Round scheduling algorithm and is mapped with the sched_policy field value equal to -1.

# System Calls and Helper Functions

We have updated the per process state struct by adding a few additional custom fields. These are displayed as follows:

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int deadline;                // Relative deadline of process
  int exec_time;               // Execution time of process
  int sched_policy;            // Policy → 0 = EDF, 1 = RM, -1 = RR
  int elapsed_time;            // Elapsed Time of the process
  int start_time;              // Start Time of the process
  int rate;                    // Rate of the process
  int weight;                  // Weight of the process
};
```

The custom field values defined above have been initialised in the userinit function and the fork functions as follows:

```
p->state = RUNNABLE;
p->deadline = 0;
p->exec_time = 0;
p->sched_policy = -1;
p->elapsed_time = 0;
p->start_time = ticks;
p->rate = 0;
p->weight = 0;
```

userinit () initialisation

```
np->state = RUNNABLE;
np->deadline = 0;
np->exec_time = 0;
np->sched_policy = -1;
np->elapsed_time = 0;
np->start_time = ticks;
np->rate = 0;
np->weight = 0;
```

fork () initialisation

1. **sys_exec_time(int pid, int exec_time)**

   This system call implements a function where given a pid of a process, it updates the exec_time field with the given value. We validate if the process pid values lie in the required range from 0 to NPROC and if the exec_time value is greater than 0. This system call function calls a helper function exec_time_helper(int pid, int exec_time). This helper function first acquires the lock on the process table (ptable.lock). This helps to ensure that no other process modifies it at the same time. After this, we

iterate and traverse through the process table to find the process that has the given pid value. Once the process has been found, the exec_time field of this process is updated with the given value and then we release the lock on the process table and return 0 if the system call becomes successful. If the process with the given pid value is not found in the process table, then we simply release the lock on the process table and return -22 to imply a failed system call.

```
int
sys_exec_time(void)
{
  int pid, exec_time;
  if(argint(0, &pid) < 0 || pid < 0 || pid ⩾ NPROC) {return -22;}
  if(argint(1, &exec_time) < 0 || exec_time < 0) {return -22;}
  return exec_time_helper(pid, exec_time);
}
```

```
int
exec_time_helper(int pid, int exec_time)
{
  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;
}
```

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

This system call implements a function where given a pid of a process, it updates the deadline field with the given value. We validate if the process pid values lie in the required range from 0 to NPROC and if the deadline value is greater than 0. This system call function calls a helper function deadline_helper(int pid, int deadline). This helper function first acquires the lock on the process table (ptable.lock). This helps to ensure that no other process modifies it at the same time. After this, we iterate and traverse through the process table to find the process that has the given pid value. Once the process has been found, the deadline field of this process is updated with the given value and then we release the lock on the process table and return 0 if the system call becomes successful. If the process with the given pid value is not found in the process table, then we simply release the lock on the process table and return -22 to imply a failed system call.

```
int
sys_deadline(void)
{
  int pid, deadline;
  if(argint(0, &pid) < 0 || pid < 0 || pid ≥ NPROC) {return -22;}
  if(argint(1, &deadline) < 0 || deadline < 0) {return -22;}
  return deadline_helper(pid, deadline);
}
```

```
int
deadline_helper(int pid, int deadline)
{
  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;
}
```

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

This system call implements a function where given a pid of a process, it updates the rate field with the given value. We validate if the process pid values lie in the required range from 0 to NPROC and if the rate value lies between 1 to 30. This system call function calls a helper function rate_helper(int pid, int rate). This helper function first acquires the lock on the process table (ptable.lock). This helps to ensure that no other process modifies it at the same time. After this, we iterate and traverse through the process table to find the process that has the given pid value. Once the process has been found, the rate field of this process is updated with the given value.

After this, we evaluate the weight field of this process using the updated rate field value using the following expression. We find the numerator variable as (30 - rate) * 3 + 28. The val variable is then updated as the numerator / 29 value. Finally, we check which out of 1 and val is greater and update the weight field with the greater value.

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

After this, we release the lock on the process table and return 0 if the system call becomes successful. If the process with the given pid value is not found in the process table, then we simply release the lock on the process table and return -22 to imply a failed system call.

```
int
sys_rate(void)
{
  int pid, rate;
  if(argint(0, &pid) < 0 || pid < 0 || pid ≥ NPROC) {return -22;}
  if(argint(1, &rate) < 0 || rate < 1 || rate > 30) {return -22;}
  return rate_helper(pid, rate);
}
```

```
int
rate_helper(int pid, int rate)
{
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p→pid == pid){
      p→rate = rate;
      int numerator = ((30 - rate) * 3) + 28;
      int val = (int)(numerator / 29);
      int weight = 0;
      if (val > 1){
        weight = val;
      }
      else{
        weight = 1;
      }
      p→weight = weight;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -22;
}
```

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

This system call implements a function where given a pid of a process, it updates the sched_policy field with the given value. We validate if the process pid values lie in the required range from 0 to NPROC and if the sched_policy value is either 0 or 1. This system call function calls a helper function sched_policy_helper(int pid, int policy). This helper function first acquires the lock on the process table (ptable.lock). This helps to ensure that no other process modifies it at the same time. After this, we iterate and traverse through the process table to find the process that has the given pid value. Once the process has been found, the sched_policy field of this process is updated with the given value.

We then check if the scheduling policy is schedulable by calling the schedulability (int policy) function, which returns 1 to indicate that the set of processes as per the

current policy are schedulable, and 0 to indicate that the given set of processes as per the current policy are not schedulable. If the sched_check returns 1, then we update the start_time field (arrival time) of the process with the current time of the system or ticks value.

If the sched_check returns the value 0, i.e. when it is not schedulable, then we set sets the state of the process to ZOMBIE, wake up any other process that might be waiting for the current process to complete using the (wakeup1()) function, kill the process using the kill() system call, and return -22. This ensures that when the task is not schedulable, then we kill it from the kernel space.

Finally after the schedulability function has been executed, then we release the lock on the process table and return 0 if the system call becomes successful. If the system call is not successful, then we simply release the lock on the process table and return -22 to imply a failed system call in addition to killing the process from the kernel space if the task fails the schedulability check.

The schedulability function function checks whether the current set of processes is schedulable as per the given sched_policy. We have implemented schedulability checks for both scheduling policies as follows:

- **Policy = 0: Earliest Deadline First (EDF) scheduling policy:** In this function, we iterate and traverse through the processes in the process table (ptable) and evaluate the sum of the exec_time / deadline value of each of the processes. Here, we consider the deadline value to be the period of the process as mentioned in the assignment requirements. In order to handle the floating point values in xv6, we multiply these exec_time / deadline value of each of the processes by a scaling factor and to check for schedulablity we verify if the sum of the utilisation values is less than or equal to the scaling factor. We have also maintained a proc_status array to maintain the current utilisation values of all the processes and after evaluating the current total utilisation, we update this proc_status array. If this condition is satisfied, then we return 1 implying that the set of tasks is schedulable and if this condition is not met, then we return 0 implying that the set of tasks is not schedulable.

- **Policy == 1: Rate Monotonic (RM) scheduling policy:** In this function, we iterate and traverse through the processes in the process table (ptable) and evaluate the sum of the exec_time / period value of each of the processes. We evaluate the period of the process by using its rate field value as (100 / rate) considering that 1 tick = 1 ms, execution time is in the unit of ticks and rate is in the unit of instance per second. In order to handle the floating point values in xv6, we multiply these exec_time / deadline value of each of the processes by a scaling factor and to check for schedulablity we verify if the sum of the utilisation values is less than or equal to the max values predefined in the maxvals array based on the formula for RM scheduling policy. The total utilisation value should be less than or equal to the value = n * $(2^{(1/n)} - 1)$ where n is the number of processes. We have also maintained a proc_status array to maintain the current utilisation values of all the processes and after evaluating the current total utilisation, we update this proc_status

array. If this condition is satisfied, then we return 1 implying that the set of tasks is schedulable and if this condition is not met, then we return 0 implying that the set of tasks is not schedulable.

```
int
sys_sched_policy(void)
{
  int pid, policy;
  if(argint(0, &pid) < 0 || pid < 0 || pid ≥ NPROC) {return -22;}
  if(argint(1, &policy) < 0 || policy < 0 || policy > 1) {return -22;}
  if (policy ≠ 0 && policy ≠ 1){return -22;}
  return sched_policy_helper(pid, policy);
}
```

```
int
sched_policy_helper(int pid, int policy)
{
  struct proc *p;
  int sched_check;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p→pid == pid) {
      release(&ptable.lock);
      p→sched_policy = policy;
      sched_check = schedulability(policy);
      p→sched_policy = -1;
      acquire(&ptable.lock);
      if (sched_check == 1){
        p→sched_policy = policy;
        p→start_time = ticks;
        release(&ptable.lock);
        return 0;
      }
      else{
        p→state = ZOMBIE;
        wakeup1(p);
        release(&ptable.lock);
        kill(p→pid);
        return -22;
      }
    }
  }
  release(&ptable.lock);
  return -22;
}
```

```c
int
schedulability (int policy)
{
  struct proc *p;
  int totalsum = 0;
  int scale_fac = 1000;
  int numprocs = 0;

  if (policy == 0){
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if (p->sched_policy == 0){
        if (p->state == RUNNABLE || p->state == RUNNING){
          totalsum += (scale_fac * p->exec_time)/p->deadline;
        }
      }
    }
    for (int i = 0; i < proc_num; i++){
      totalsum = totalsum + proc_status[i];
    }
    if (totalsum <= scale_fac){
      return 1;
    }
    else{
      return 0;
    }
  }
```

```c
  else if (policy == 1){
    int maxvals[64] = {1000, 828, 779, 756, 743, 734, 728, 724, 720, 717, 715, 713, 711, 710, 709, 708, 707, 706, 705, 705, 704, 704, 703, 703, 702, 702, 702, 701, 701, 701, 700, 700, 700, 700, 700, 699, 699, 699, 699, 699, 699, 698, 698, 698, 698, 698, 698, 698, 697, 697, 697, 697, 697, 697, 697, 697, 697, 697, 697, 697, 696};
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if (p->sched_policy == 1){
        if (p->state == RUNNABLE || p->state == RUNNING){
          int period = (int)(100 / p->rate);
          totalsum += (scale_fac * p->exec_time) / period;
        }
      }
    }
    for (int i = 0; i < proc_num; i++){
      totalsum = totalsum + proc_status[i];
    }
    numprocs = numprocs + proc_num;
    if (totalsum <= maxvals[numprocs]){
      return 1;
    }
    else{
      return 0;
    }
  }
  return -1;
}
```

## Additional Implementation Details

In order to implement the above mentioned system calls, we have made changes to the following files:

- **user.h** - Implemented the function prototype
- **usys.S** - Declared SYSCALL(syscall_name)
- **syscall.c** - Declared [SYS_syscall_name] syscall_name and extern int syscall_name (void)
- **sycall.h** - Declared the syscall number for the new system calls
- **sysproc.c** - Implemented the function for carrying out the system call
- **proc.c** - Implemented helper functions for carrying out the system call

## Testing Details

In order to test the above scheduling algorithms we have made the following changes to the trap.c file:

```
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
{
  myproc()->elapsed_time++;
  if((myproc()->sched_policy >= 0) &&(myproc()->elapsed_time >= myproc()->exec_time))
  {
    cprintf("The arrival time and pid value of the completed process is %d %d\n", myproc()->start_time, myproc()->pid);
    if (myproc()->sched_policy == 0){
      proc_status[proc_num++]=(myproc()->exec_time * 1000)/myproc()->deadline;
    }
    else if (myproc()->sched_policy == 1){
      int period = 100 / myproc()->rate;
      proc_status[proc_num++]=(myproc()->exec_time * 1000)/period;
    }
    exit();
  }
  else
    yield();
}
```

The proc_num variable is initialised to 0 and it defines the number of completed processes so far. The proc_status array is of the size NPROC = 64 and it maintains the utilisation values of different processes. These variables have been defined in the trap.c file and they are updated in both the trap.c file as well as the schedulability function in the proc.c file.