COL 331: Operating Systems

ASSIGNMENT 2: REPORT

2023-04-05

REEDAM DHAKE — 2020CS10372
NIKHIL UNAVEKAR — 2020CS10363

Contents

1 Changes made

- CPU'S set to 1 in make file and changes mentioned in assignment in trap.c
- Implemented required system calls, EDF and RMS scheduler and their check scripts (Also the default RR is implemented again for simplicity)
- Added int start_ticks, int elapsed_ticks, int deadline, int execution_time, int rate, int weight, int sched_policy, in proc.h

2 System Calls

All the system calls are implemented in the sysproc.c file that calls the functions from proc.c file. All function first acquires the ptable.lock on the process table to prevent other threads from modifying, changing or interrupting the process table concurrently. The sysproc.c following code for all 4 system calls is provided below.

```
int sys_sched_policy(void){
  ..
  int ret_val = set_sched_policy(pid, policy);
  ..
  return ret_val;
}
int sys_exec_time(void){
  ..
  return set_exec_time(pid, time);
  ..
}
int sys_deadline(void){
  ..
  return set_deadline(pid, deadline);
  ..
}
int sys_rate(void){
  ..
  return set_rate(pid, rate);
  ..
}
```

2.1 sys_sched_policy

2.1.1 Code Snippet

```
    int set_sched_policy(int pid, int policy){
      struct proc *p;
      acquire(&ptable.lock);
      if (policy!=1 && policy!=2 && policy!=0){
        release(&ptable.lock);
        return -22;
```

```
  }
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->start_ticks = ticks;
      if(policy==0){
        if(edf_schedule_check(pid) == -1){
          release(&ptable.lock);
          return -22;
        }
      }
      else if(policy==1){
        if(rms_schedule_check(pid) == -1){
          release(&ptable.lock);
          return -22;
        }
      }
      p-> sched_policy = policy;
      release(&ptable.lock);
      return 0;
    }
  }
  release (&ptable.lock);
  return -22;
}
```

2.1.2 Explanation

- This is a function in an operating system scheduler that sets the scheduling policy for a given process identified by its process ID (pid).It checks if the policy provided is valid. If the policy is not 0(EDF), 1(RMS), or 2(RR), the function returns -22 (indicating invalid policy).
- The function then iterates through the process table using for loop to find the process with the input pid. If the process is found, the function updates the process's start_ticks field to the current value of the system timer, indicating the start of a new scheduling interval.
- If the policy is 0, the function checks if the process satisfies the Earliest Deadline First (EDF) scheduling policy by calling the edf_schedule_check() function. Similarly if the policy is 1, the function checks if the process satisfies the Rate Monotonic Scheduling (RMS) policy by calling the rms_schedule_check() function. If the check fails, in both cases the function returns -22. Then the function sets the process's scheduling policy to the given policy and releases the lock on the process table before returning 0 to indicate scheduling was successful. If the process with the given pid is not found, the function returns -22.

2.2 sys_exec_time

2.2.1 Code Snippet

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

2.2.2 Explanation

- This function sets the execution time of the process with the given pid to the specified exec_time.It first checks if exec_time is non-negative using the condition if (exec_time<0) and returns -22 if it is. The function then iterates over ptable to find a process with the given pid, it then sets the execution time of that process to the specified exec_time using p→ execution_time = exec_time. Then lock is released and 0 returned, indicating success.If the function does not find a process with the given pid, it releases the lock and returns -22, shows failure.

2.3 sys_deadline

2.3.1 Code Snippet

```
  int set_deadline(int pid, int deadline)
  {
    struct proc *p;
    acquire(&ptable.lock);
    if (deadline<0)
    {
      release(&ptable.lock);
      return -22;
    }
    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;
  }
```

2.3.2 Explanation

- This function named set_deadline that sets the deadline of the specified process identified by pid. It first checks if the deadline provided is non-negative, and returns an error code of -22 if it is not. It then iterates through the process table using for loop to find the process with the specified pid. If it finds the process, it sets the deadline field of the process to the specified value using p→ deadline = deadline, releases the lock of the process table, and returns 0, indicating system call was successful. If it does not find the process, it releases the lock of the process table and returns an error code of -22.

2.4 sys_rate

2.4.1 Code Snippet

```
    int set_rate(int pid, int rate)
    {
      struct  proc*p;
      acquire(&ptable.lock);
      if (rate <= 0 || rate >30)
      {
        release(&ptable.lock);
        return -22;
      }
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
      {
        if(p->pid == pid)
        {
          p-> rate = rate;
          int rr  = (int) ((90-3*rate) + 28)/29;
          if(rr<1)
            rr=1;
          int temp = rr;
          p-> weight = temp ;
          release(&ptable.lock);
          return 0;
        }
      }
      release(&ptable.lock);
      return -22;
    }
```

2.4.2 Explanation

- This is a system call in an operating system kernel that sets the rate of a process given its PID (process ID). The function takes two arguments: the PID of the process and the desired rate. It first checks if (rate less than equal to 0 or rate greater 30)i.e provided rate is within the acceptable range (1 to 30).
- It then iterates through the process table using for loop to find the process with the specified pid. If the current process's PID matches the provided PID it sets the process's rate to the provided rate.
- It then calculates the weight (priority) of the process using int rr = (int) ((90-3*rate) + 28)/29. If rr

< 1, rr = 1 , this ensures that the weight is within the bound of (1,3). It then sets the process's weight to this computed value by p $\rightarrow$ weight = temp command. Finally it returns 0 indicating that the system call was successful. In all other cases (PID not found and rate out of bound it returns -22.

3 EDF Scheduling Policy

3.1 Code Snippet

```
void edf_scheduler(struct proc *p,struct cpu *c){
  for(;;){
    sti();
    acquire(&ptable.lock);
    struct proc *p_temp;
    int tim = 1000007;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      if(p->sched_policy==0){
        if(tim > (p->deadline+p->start_ticks)){
          tim = p->deadline+p->start_ticks;
          p_temp = p;
        }
        else if(tim == (p->deadline+p->start_ticks)){
          if(p_temp->pid > p->pid){
            p_temp = p;
          }
        }
      }
    }
    if(tim == 1000007){
      release(&ptable.lock);
      return;
    }
    p = p_temp;
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    switchkvm();
    c->proc = 0;
    release(&ptable.lock);
  }
}
```

3.2 Explanation

- Above function implements an Earliest Deadline First (EDF) scheduling algorithm in an operating system kernel. The function takes two arguments, a pointer to the currently running process (struct proc *p) and a pointer to the CPU currently executing the process (struct cpu *c).The function runs an infinite loop, with interrupts enabled (sti()) to enable preemption and acquires the process table lock.It initializes a temporary variable p_temp of type struct proc and a variable tim to a large value 1000007 which would later store earliest deadline.

- The function begins by iterating through the entire process table using a for loop, checking the state of each process. If the state of a process is not RUNNABLE, it continues to the next process. Else if the process if RUNNABLE and the scheduling policy of the process is EDF, it calculates the time until the deadline of the process (p $\rightarrow$ deadline + p $\rightarrow$ start_ticks)(Actual deadline is computed by adding start_ticks to relative deadline) and checks if it is less than the current value of tim. If it is, then it updates tim and sets p_temp to point to the current process p. If the time until the deadline of the current process is equal to tim, it breaks the tie by selecting the process with the smaller pid as described in the 17-18 lines of the above code.In this way we compute the process having the earliest deadline in go of outer for loop.
- Now if after iterating through all the processes in the process table, if tim is still set to its initial value(1000007), it means that there are no RUNNABLE processes with the EDF policy. In that case, the function releases the process table lock, returns nothing and exits from the function.
- If there is a RUNNABLE process with the EDF scheduling policy, the function sets p to point to p_temp, sets the CPU's current process to the selected process (c$\rightarrow$proc = p), and switches to the process's virtual memory (switchuvm(p)). It then sets the state of p to RUNNING and switches the CPU to the selected process's context (swtch(&(c$\rightarrow$scheduler), p$\rightarrow$context)). After the process has finished executing, it switches the CPU back to kernel memory (switchkvm()), sets the CPU's current process to 0 (c$\rightarrow$proc = 0), and releases the process table lock (release(&ptable.lock)).
- The function then loops back to the beginning and repeats this process of selecting the process with the earliest deadline until there are no more RUNNABLE processes with the EDF scheduling policy.

4 RMS Scheduling Policy

4.1 Code Snippet

```
void rms_scheduler(struct proc *p,struct cpu *c){
  for(;;){
    sti();
    acquire(&ptable.lock);
    struct proc *p_temp;
    int wt = 5;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      if(p->sched_policy==1){
        if(wt > (p->weight)){
          wt = p->weight;
          p_temp = p;
        }
        else if(wt == (p->weight)){
          if(p_temp->pid > p->pid){
            p_temp = p;
          }
        }
      }
    }
    if(wt == 5){
      release(&ptable.lock);
      return;
    }
    p = p_temp;
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    switchkvm();
    c->proc = 0;
    release(&ptable.lock);
  }
}
```

4.2 Explanation

- The above CPU scheduler function implements the Rate-Monotonic Scheduling (RMS) algorithm. The function takes two arguments: a pointer to the currently running process (p) and a pointer to the CPU structure (c). The function runs an infinite loop where it first enables hardware interrupts using sti() and then acquires the process table lock using acquire(&ptable.lock) to prevent other threads to access and modify or change the process table.

- Inside the infinite loop, it initializes a temporary variable p_temp of type struct proc and a minimum weight variable (wt) with a initial value of 5. Then silimar to EDF it then iterates through all the processes in the process table using a for loop. Again if a process is not in the RUNNABLE state, it skips to the next process using the continue statement. If a process is in the RUNNABLE state and follows the RMS scheduling policy (identified by sched_policy=1), the function compares the weight of the current process with the minimum weight found so far.If the weight of the current process is less than the minimum weight, it updates the minimum weight and sets the temporary process pointer to the current process.If the weight of the current process is the same as the minimum weight, it breaks the tie by comparing the PIDs of the current process and the temporary process pointer, and sets the temporary process pointer to the process with the lower PID.

- After iterating through all the processes in the process table, the function checks if the minimum weight is still the initial value of 5. If so, it means there are no runnable processes with RM policy, and the function releases the process table lock and returns.

- If found a RM process to be scheduled as like EDF set the currently running process pointer (p) to the process pointed to by the temporary process pointer (p_temp).

5 EDF and RMS Schedulability Checks

5.1 Code Snippet

```
1   int edf_schedule_check(int pid){
2     struct proc *p;
3     int utility = 0;
4     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
5       if(((p->sched_policy == 0) && (p->killed==0)) || (p->pid == pid) ){
6         utility = utility + ((10000* p->execution_time) / (p->deadline));
7       }
8     }
9     if( utility > 10000 ){
10      return -1;
11    }
12    return 1;
13  }
14  int rms_values[64] = {
15    1000000, 828427, .... ,696974, 696914 //64 values of possible rms values
        in liu layland.
16  };
17  int rms_schedule_check(int pid){
18    struct proc *p = ptable.proc;
19    int utility = 0;
20    int numprocs = 0;
21    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
22      if(((p->sched_policy == 0) && (p->killed==0)) || (p->pid == pid) ){
23        numprocs++;
24        utility = utility + (p->execution_time * p->rate);
25      }
26    }
27    utility = utility*10000;
28    if( utility > 1000000 || (numprocs > 0 && utility > rms_values[numprocs-1]
          )){
29      return -1;
30    }
31    return 1;
32  }
```

5.2 Explanation

- First 14 lines of code check if adding the process with the given pid to the current EDF scheduling queue would cause the system's total utility to exceed 10000, which would violate the schedulability condition for EDF scheduling. It iterates over the ptable to check all process that having EDF policy assigned and not killed. Then we add the process utility to utility variable calculated as (10000 * p→execution_time) / p→deadline. The function adds up the utilities of all processes (including input pid process) to get the total utility of the system. If the total utility is greater than 10000, the function returns -1, schedulability is not possible. Otherwise, it returns 1, indicating that the process can be added to the EDF scheduling queue without violating the schedulability condition.

- Rest lines of Code compute if the function is RMS schedulable. rms_values is an predefined array that gives the maximum limit of utility for depending on number of processes that are rms scheduled. The function first initializes p to point to the head of the ptable.proc array, which is an array of process descriptors. It also initializes utility and numprocs to 0. Now if a process comes which is to be rms scheduled, it iterates over all the processes in ptable and checks for processes having rms scheduling policy and are runnable or currently running. It calculates the utility of the function using the equation (p $\rightarrow$ execution_time * p$\rightarrow$rate) and add this value to total utilities. Note it also adds running process if its RMS scheduled. Utility is then multiplied by 10000. If total utility is greater than maximum utility allowed according to rms_arr, process cannot be scheduled and return -1 Otherwise return 1.

6 Scheduler Function

6.1 Code Snippet

```
  void scheduler(void){
      ** inputs **
    acquire(&ptable.lock);
    for(;;){
      c->proc = 0;
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != RUNNABLE)
          continue;
        if(p->sched_policy==0){
          **use edf_schedular**
        }
        else if(p->sched_policy==1){
          **use rms_schedular**
        }
      }
      **else use rms_schedular**
    }
    release(&ptable.lock);
  }
```

6.2 Explanation

- The scheduler() function is the heart of the xv6 operating system's process scheduling mechanism. It continuously loops through all processes in the process table and selects the next process to run based on its scheduling policy. The scheduler() function is called by the kernel's main loop, and it is responsible for assigning a processor to a process when it is available. It then iterates through each process in the table, and for each process that is runnable, it selects a scheduler based on the process's scheduling policy. If the policy is EDF, it calls the edf_scheduler() function, if the policy is RMS, it calls the rms_scheduler() function, and if neither, it falls back to a Round Robin scheduler by calling the rr_scheduler() function.