Indian Institute of Technology Delhi

ELL 783 - Operating Systems

Assignment 2

Real Time Scheduling

Name: Mohammad Yusuf, Madhur Sharma
Entry No.: 2023EET2757, 2023EET2193
Course Instructor: Prof. Smruti Ranjan Sarangi

Introduction

This report details the implementation methodology of real-time scheduling policies, specifically the Earliest Deadline First (EDF) and Rate-Monotonic (RM) scheduling algorithms, within the xv6 operating system. The report includes code snippets and pseudocode to demonstrate the implementation approach.

EDF Scheduling Algorithm

The EDF (Earliest Deadline First) scheduling algorithm is a dynamic priority scheduling algorithm used in real-time operating systems to schedule processes based on their deadlines. In this report, we discuss the implementation of the EDF scheduling algorithm in the xv6 operating system.

Schedulability Test

In our implementation, we perform a schedulability test to determine whether a process is schedulable under the EDF policy. The utility of all scheduled processes is added together, and if the total utility exceeds a certain threshold, the process is exited and not considered for future scheduling checks. To ensure accuracy, we perform calculations using float cpu_utilz variable which is precise up to 4 decimal places. Below is a snippet of the code illustrating the schedulability test:

```c
      // EDF
      if (policy == 0)
      {
        for (struct proc *p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
          if (p1->sched_policy == 0 || p1->pid == p-> pid){
            //calculating utilizationof all sched_policy=0 including the new process
            cpu_utilz += (float)(p1->exec_time) / p1->deadline;
          }

          if(cpu_utilz > 1){
            release(&ptable.lock);
            kill(pid);
            return -22;
          }
        }
      }
```

Scheduling Algorithm

The EDF scheduling algorithm prioritizes processes based on their deadlines, ensuring that processes with earlier deadlines are scheduled first. We calculate the time left before the deadline for each runnable process and schedule the one with the least time left. In cases where multiple processes have the same time left, we prioritize the process with the lower PID. Here is a simplified version of the scheduling algorithm code:

<u>Code</u>

```c
if (sch_policy == 0)  //EDF
{
  int closest_deadline = 100000000;
  struct proc *p_selected = NULL;

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state == RUNNABLE && p->sched_policy == 0) {
      int p_time_left = (p->arrival_time + p->deadline) - ticks;

      if (p_time_left < closest_deadline){    //find earliest deadline
        closest_deadline = p_time_left;
        p_selected = p;
      }
      else if (p_time_left == closest_deadline){
        if (p->pid < p_selected->pid)  //select lowest pid
          p_selected=p;
      }
    }
  }

  if(p_selected!=NULL){
    p=p_selected;

    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;

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

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

In conclusion, our implementation of the EDF scheduling algorithm in xv6 ensures that processes are scheduled based on their deadlines, allowing for efficient utilization of resources in a real-time operating system environment. The schedulability test and scheduling algorithm work together to prioritize processes effectively, contributing to the overall performance and responsiveness of the system.

RMS Scheduling Algorithm

The RMS (Rate-Monotonic Scheduling) algorithm is a fixed-priority scheduling algorithm used in real-time operating systems to schedule processes based on their rates or priorities. This report presents the implementation and functioning of the RMS scheduling algorithm within the xv6 operating system environment.

Schedulability Test

Our implementation includes a schedulability test to determine whether a process is schedulable under the RMS policy. The test involves calculating the total utility of all scheduled processes and comparing it with precomputed rms_bound[] values, which is the Liu Layland Bound, based on the number of processes. If the total utility exceeds the rms_bound[i] , the process is terminated and not considered for future scheduling checks. The accuracy of calculations is maintained by using float arithmetic and rms_bound[] values are out of 100, which is precise up to 4 decimal points. Here is a snippet of the code illustrating the schedulability test:

```
// RMS
else if (policy == 1)
{
  int num_rms_proc = 0;

  for (struct proc *p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
    if(p1->sched_policy == 1 || p1->pid == p->pid){
      //calculating utilizationof all sched_policy=1 including the new process
      num_rms_proc++;
      cpu_utilz += (float)(p1->rate*p1->exec_time);
    }
  }

  if (cpu_utilz > rms_bound[num_rms_proc-1]){
    release(&ptable.lock);
    kill(pid);
    return -22;
  }
}
```

Scheduling Algorithm

In RMS scheduling, processes are prioritized based on their rates or priorities, with higher priority processes scheduled first. Our scheduling algorithm calculates the weight of each process and schedules the one with the least weight. In cases where multiple processes have the same weight, the process with the lower PID is given precedence. Here is a simplified version of the scheduling algorithm code:

```
else if (sch_policy == 1) //RMS
{

  int min_priority = 5;
  struct proc *p_selected = NULL;

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state == RUNNABLE && p->sched_policy == 1){
      int priority = rms_priority(p);

      if (priority < min_priority){    //find minimum priority
        min_priority = priority;
        p_selected=p;
      }
      else if (priority == min_priority){    //select lower pid
        if (p->pid < p_selected->pid)
          p_selected=p;
      }
    }
  }

  if (p_selected != NULL){
    p = p_selected;
    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;

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

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

The implementation of the RMS scheduling algorithm in xv6 enhances the real-time capabilities of the operating system by efficiently scheduling processes based on their rates or priorities. The schedulability test ensures that processes are admitted into the scheduling queue appropriately, while the scheduling algorithm prioritizes processes effectively to meet real-time requirements. Overall, the RMS algorithm contributes to the reliability and responsiveness of the xv6 operating system in handling real-time tasks.

Modifications in the proc Structure:

- sched_policy: This attribute denotes the scheduling policy of the process. It can take values such as -1 for the default xv6 policy, 0 for EDF (Earliest Deadline First), 1 for RMS (Rate-Monotonic Scheduling), and 4 for non-schedulable processes. Users can set this attribute using the sched_policy(pid, value) system call.

- elapsed_time: It tracks the elapsed time of the process by counting the number of ticks during which the process was in the RUNNING state.

- exec_time: This attribute specifies the total allowed execution time for the process, settable by user processes using the exec_time(pid, value) system call.

- rate: Represents the rate of assumed periodic processes. Users can set this attribute using the rate(pid, value) system call.

- deadline: Denotes the hard deadline of the process. If a new process fails the schedulability check, it is killed to ensure that deadlines for accepted processes are not compromised. The deadline is set using the deadline(pid, value) system call.

- arrival_time: Represents the start time of the process, recording the tick value when the process's sched_policy was set.

```c
// 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;                // deadline of the process
  int exec_time;               // execution time of the process
  int sched_policy;            // scheduling policy of the process , -1 for round robin 0 for edf, 1 for rms
  int elapsed_time;            // elapsed time of the process
  int rate;                    // rate of the process
  int arrival_time;            // arrival time of the process

};

```

trap.c Modifications

The trap.c file has been modified to handle unwanted or completed processes efficiently. A process is considered unwanted if its sched_policy is not -1 (indicating a specific scheduling policy) and its elapsed_time exceeds its exec_time. The modified code snippet below illustrates this functionality:

```c
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER){

  if((myproc()->sched_policy >= 0) && (myproc()->elapsed_time >= myproc()->exec_time)){
    cprintf("The completed process has arrival time and pid values: %d %d\n", myproc()->arrival_time, myproc()->pid);
    exit();
  }
  else{
      yield();
  }

}
```

RMS priority

This code calculates priorities of rate monotonic scheduled processes based on rate values.

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