Indian Institute of Technology Delhi

Operating System

ELL 783

Assignment 2

Name: Ranjan Kumar Singha
Entry No: 2023EET2192

Name: Atul Kumar Rana
Entry No: 2023EET2195

Course Instructor: Dr. Smruti R. Sarangi

1 Additional Process Attributes

Added the following fields to `proc.h` to keep track of various attributes needed to implement the required scheduling algorithms and system calls.

- `int sched_policy;`: Scheduling policy of the process (-1: XV6 default policy or 0: EDF or 1: RMS or 4: non-schedulable processes). It is set by a user process using the `sched_policy(pid, value)` system call.
- `int elapsed_time;`: Elapsed time of the process. It counts the number of ticks for which the process was in `RUNNING` state.
- `int execution_time;`: Total allowed execution time of the process. It is set by a user process using the `execution_time(pid, value)` system call.
- `int deadline;`: Hard Deadline of the process. Any new process that fails the schedulability check is killed so that no deadlines for accepted processes are sacrificed. It is set using the `deadline(pid, value)` system call.
- `int rate;`: Rate of the assumed periodic processes. It is set by a user process using the `rate(pid, value)` system call.
- `int priority;`: Current priority level of each process (1-3) (higher value represents lower priority). It is calculated using the rate of the process.
- `unsigned int arrival_time;`: Start time of the process. It records the tick value when the process's `sched_policy` was set.

```
  int sched_policy;              // Schedule policy
  int elapsed_time;              // elapsed time
  int execution_time;            // execution time
  int deadline;                  // deadline
  int rate;                      // rate
  int priority;                  // priority
  unsigned int arrival_time;     // arrival time
```

Code Snippet in proc.c

Initialized the count of EDF and RMS

```
int edf_count=0;  // For how many process I have to run EDF
int rms_count=0;  // For how many process I have to run RMS
```

This code snippet is responsible for decrementing the counts of processes remaining to be executed under the EDF and RMS scheduling algorithms after the completion of a process execution.

```
// After complete execution of a process, decrement the time
   count of the algorithm that has to be implemented.
  if(curproc->sched_policy==0){
    edf_count = edf_count - 1;
  }
  else if(curproc->sched_policy==1){
    rms_count = rms_count - 1;
  }
```

In the proc.c file, the userinit() and fork() functions initialize critical attributes of the process structure (proc) to ensure proper initialization for every process, whether created during system boot or through forking.

```
void userinit(void) {
    // Other initialization code...

    // Necessary variables needed for Parent Process
    p->sched_policy = -1;
    p->execution_time = -1;
    p->deadline = -1;
    p->elapsed_time = 0;

    // Other initialization code...
}
```

```
int fork(void) {
    // Other initialization code...

    // Necessary variables for the newly created Child
       processes
    np->sched_policy = -1;
    np->execution_time = -1;
    np->elapsed_time = 0;

    // Other initialization code...
}
```

Code Snippet 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))
    {
        cprintf("The arrival time and pid value of the
           completed process is %d %d \n", myproc()->
           arrival_time, myproc()->pid);
        exit();
    }
    else
        yield();
}
```

In the modified `trap.c` file, the above code snippet is used to handle processes. If the current process (`myproc()`) is in the `RUNNING` state and a timer interrupt occurs, the code checks if the process has become unwanted or completed. If the process has become unwanted (i.e., its `sched_policy` is not equal to `-1` and its `elapsed_time` exceeds its `exec_time`), the process is terminated and its details (arrival time and PID) are printed. Otherwise, the process yields the CPU.

2 sys_sched_policy

EDF Code Snippet

```
if (policy == 0) { // EDF
    int edf_check = handle_edf_policy(pid);
    if (edf_check != 0) {
        // Terminate process if EDF policy check fails
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if (p->pid == pid) {
                p->state = ZOMBIE;
                break;
            }
        }
        release(&ptable.lock);
        return edf_check; // Return error code
    }
}
```

- The code snippet checks if the specified scheduling policy is EDF (policy == 0).
- It calls the `handle_edf_policy` function to perform policy-specific checks.
- The `handle_edf_policy` function calculates the CPU utilization and checks if it exceeds 100

- If the check fails (indicating a violation of scheduling constraints), the process with the specified PID is terminated by setting its state to ZOMBIE.

RMS Code Snippet

```
else if (policy == 1) { // RMS
    // Count RMS processes
    int count = 0;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->sched_policy == 1 || p->pid == pid) {
            count++;
        }
    }
    // Check RMS policy constraints
    int rms_check = handle_rms_policy(pid, count);
    if (rms_check != 0) {
        // Terminate process if RMS policy check fails
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if (p->pid == pid) {
                p->state = ZOMBIE;
                break;
            }
        }
        release(&ptable.lock);
        return rms_check;
    }
}
```

- The code snippet checks if the specified scheduling policy is RMS (policy == 1).
- It counts the number of processes currently using the RMS policy or having the specified process ID.
- It calculates the total resource consumed by RMS processes and checks if it exceeds the threshold value.
- If the check fails (indicating a violation of resource limits), the process with the specified PID is terminated by setting its state to ZOMBIE.

`handle_edf_policy` Function

The `handle_edf_policy` function calculates the CPU utilization for processes adhering to the Earliest Deadline First (EDF) policy.

- CPU Utilization Calculation:
  - It iterates through all processes in the process table.

- For each process with the EDF policy or the specified process ID, it calculates CPU utilization as the ratio of execution time to deadline.

- Violation Check:
  - If the total CPU utilization exceeds 100

```c
int handle_edf_policy(int pid) {
    float cpu_utilization = 0.0;
    struct proc *p;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if ((p->sched_policy == 0 && p->state != UNUSED) || p->pid == pid) {
             cpu_utilization += (float)(p->execution_time) / (float)(p->deadline);
        }
    }
    if (cpu_utilization > 1.0) {
        return -22; // Return error code indicating Not Schedulable
    }
    return 0;
}
```

`handle_rms_policy` Function

The `handle_rms_policy` function calculates the total resource consumed by processes adhering to the Rate-Monotonic Scheduling (RMS) policy.

- Resource Calculation:
  - It iterates through all processes in the process table.
  - For each process with the RMS policy or the specified process ID, it calculates resource consumption as the product of execution time and rate.
- Resource Limit Check:
  - If the total resource consumption exceeds the threshold value determined by the rate-monotonic bound, the function returns an error code (-22) to indicate that the resource limit has been exceeded.

```c
int handle_rms_policy(int pid, int count) {
    float total_resource = 0.0;
    struct proc *p;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->sched_policy == 1 || p->pid == pid) {
```

```
            total_resource += ((float)(p->execution_time) * (float)(p->rate));
        }
    }
    if (total_resource > (threshold_array[count - 1] * 100.0)) {
        return -22; // Return error code indicating resource limit exceeded
    }
    return 0;
}
```

RMS Threshold Array

The `threshold_array` stores Luiland bound values used in Rate-Monotonic Scheduling (RMS). Each element of the array corresponds to a specific count of processes, representing the threshold value beyond which the system may become overloaded.

- **Threshold Values**:
  - The `threshold_array` contains predefined threshold values based on Luiland's bound.
  - Each element in the array corresponds to a count of processes, starting from 1.
  - These values are derived from the rate-monotonic scheduling theory and are used to determine the maximum total resource consumption allowed for a given number of processes.

RMS Threshold Array

```
// Array to store Luiland bound values
float threshold_array[64] = {
    1.0000, 0.8284, 0.7798, 0.7568, 0.7435, 0.7348, 0.7286, 0.7241,
    ...
    0.6974, 0.6973, 0.6972, 0.6972, 0.6971, 0.6970, 0.6970, 0.6969
};
```

3 sys_exec_time Function

- This function sets the execution time for a specific process identified by its PID.
- It takes two arguments: the PID of the process and the execution time to be set.

- If successful, it updates the execution time of the process and returns 0. If unsuccessful, it returns -1.

```c
int
sys_exec_time(void) {
    int pid, exec_time;
    if(argint(0, &pid) < 0 || argint(1, &exec_time) < 0){
        return -1;
    }
    else{
        struct proc *p;
        acquire(&ptable.lock);
        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 -1;
    }
}
```

4 sys_deadline Function

- This function sets the deadline for a specific process identified by its PID.
- It takes two arguments: the PID of the process and the deadline to be set.
- If successful, it updates the deadline of the process and returns 0. If unsuccessful, it returns -1.

```c
int
sys_deadline(void){
    int pid, deadline;
    if(argint(0, &pid) < 0 || argint(1, &deadline) < 0){
        return -1;
    }
    else{
        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 -1;
    }
}
```

5 sys_rate Function

- This function sets the rate for a specific process identified by its PID.
- It calculates the priority of the process based on the rate and sets it accordingly.
- If successful, it updates the rate and priority of the process and returns 0. If unsuccessful, it returns -1.

```
int
sys_rate(void){
    int pid,rate;
    if(argint(0, &pid) < 0 || argint(1, &rate) < 0){
        return -1;
    }
    else{
        struct proc *p;
        acquire(&ptable.lock);
        for(p=ptable.proc;p<&ptable.proc[NPROC];p++){
            if(p->pid==pid){
                p->rate=rate;
                p->priority=max_value(1,ceil_value(((30.0-rate)
                    /29.0)*3.0));
                release(&ptable.lock);
                return 0;
            }
        }
    release(&ptable.lock);
    return -1;
    }
}
```

Formula for the weight (w) for a process with rate (r):

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

```c
// Functions for necessary operations
int
max_value(int num1,int num2){
    return num1>num2 ? num1 : num2;
}

int
ceil_value(float num){
    return (int)num+1;
}
```

6 Screenshot