OPERATING SYSTEMS: ASSIGNMENT 2

Real Time Scheduling

SHIVAM JAIN (2020CS50626)

SHIVAM VERMA (2020CS50442)

1. SYSTEM CALLS

Following system calls were implemented that assisted to change the scheduling policy of a process to the real-time scheduling policy `(EDF or RM)` in `xv6`.

- **sys_sched_policy (int pid, int policy)**

  It took the `pid` of the process and `scheduling algorithm` as argument changed the policy of corresponding processes. Following is the table of policy and corresponding scheduling algorithm.

-1	**Round Robin Scheduling**
0	**Earliest Deadline First (EDF)**
1	**Rate Monotonic Scheduling (RMS)**

- **sys_exec_time (int pid, int exec_time)**

  This system call is used to define the duration of a process's execution. After the execution time of a process is elapsed it is terminated by the kernel. This working is explained later in the report. If the system call was successful, it returns 0; otherwise, it returns -22.

- **sys_deadline (int pid, int deadline)**

  This system call is used to set the relative deadline of any task with respective pid. This deadline is relative and the actual deadline will the sum of this value and the process' arrival time. If the system call was successful, it returns 0; otherwise, it returns -22.

- **sys_rate (int pid, int rate)**

  This system call is used to specify the rate (reciprocal of the period) for a process in the RM scheduling policy. The unit of rate is instances per second. If the system call was successful, it returns 0; otherwise, it returns -22. The value of rate is in range [1, 30] and the weight of a process is in range [1, 3] which can be derived using following equation:

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

  The lower the weight of any process, the higher will be its priority while scheduling.

2. IMPLEMENTATION DETAILS

1. Changes were made in the `proc struct`, so that information regarding each process could be properly stored and retrieved whenever required. This is how `proc struct` looked after adding new attributes. The unit of `time` in the following attributes is `ticks` which is equal to `10 ms` in xv6.

```c
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 sched_policy;            // current scheduling policy of the process
  int weight;                  // current weight level of each process
  int rate;                    // rate of a process
  int execution_time;          // kernel will terminate process after this time is elapsed
  int elapsed_time;            // time elapsed while process is running
  int wait_time;               // time spent in sleeping
  int deadline;                // strict global deadline for a process
  int deadline_relative;       // deadline relative to its arrival time
  int arrival;                 // time at which a process arrives
};
```

This is process control block (PCB) data structure used in `XV6 operating systems`. The struct contains various fields that hold information about a process, such as its memory size, page table, kernel stack, state, process ID, parent process, trap frame, context, open files, current directory, and process name.

In addition to the standard fields, this particular PCB includes fields related to process scheduling. These fields include the scheduling policy, weight level, rate, execution time, elapsed time, wait time, deadline, deadline relative to its arrival time, and arrival time. These fields are used by the scheduler to determine which process to run next and for how long.

Overall, this PCB data structure provides a way for the operating system to manage and keep track of processes, allowing it to switch between processes efficiently and ensure that each process gets a fair amount of resources.

2. A function is added to the kernel that updates various time-related fields for each process in the process table. The function loops through all the processes in the process table and updates the appropriate fields based on the current state of the process.

   The function begins by acquiring a lock on the process table to ensure that it is not modified by another thread while the function is executing. It then loops through each process in the process table using a pointer to the proc struct.

   For each process, the function uses a switch statement to determine the current state of the process. If the process is in the `RUNNABLE` state, the function increments the process's wait time field. This field represents the amount of time the process has spent waiting in the `RUNNABLE` state to be scheduled to run by the scheduler.

   If the process is in the `RUNNING` state, the function checks if the process has a deadline set. If it does, the function increments the elapsed time field, which represents the amount of time the process has been executing since it was last scheduled.

   If the process is in any other state, the function does nothing.

   Finally, the function releases the lock on the process table, allowing other threads to access it.

   This function is called periodically by the operating system's timer interrupt handler (inside `trap.c`) to update the time-related fields for each process. The updated fields can be used by the scheduler to make scheduling decisions, such as determining which process to run next based on its wait time or ensuring that a process is terminated if it exceeds its deadline.

```c
void time_ticks_update() {
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    switch(p->state) {
      case RUNNABLE:
        p->wait_time++;
        break;
      case RUNNING:
        if (p->deadline != INF)
        p->elapsed_time++;
        break;
      default:
        ;
    }
  }
  release(&ptable.lock);
}
```

3. The kernel terminates a process if its elapsed time crosses the execution time. It is done through following piece of code, which is added in `trap` function.

```c
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
{
 if((myproc()->sched_policy >= 0) && (myproc()->elapsed_time >= myproc()->execution_time))
 {
  cprintf("The completed process has pid: %d\n", myproc()->pid);
  exit();
 }
 else
  yield();
}
```

This code snippet checks if the current process is running and if a timer interrupt has occurred. If the process has a valid scheduling policy and has completed its execution time, it prints a message indicating that the process has completed and exits. Otherwise, it yields the CPU to allow other processes to run.

4. While we are setting policy for a particular process, we are also checking that can the augmented set of processes be scheduled with the given policy. If this process pass the check we add it to global process table or else we kill the process and return -22. The implementation of this algorithm can be seen below. (To avoid floating point arithmetic, we are multiplying numerator by 1000).

```c
int utility[] = {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, 698, 697, 697, 697, 697, 697, 697, 697, 697, 697, 697, 697, 697, 697, 696, 696, 696, 696, 696, 696, 696};

int __policy(int pid, int policy) {
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->sched_policy = policy;
      p->arrival = ticks;
      p->deadline = p->arrival + p->deadline_relative;

      // now let us see if its schedulable or not
      int U_edf = 0;
      int U_rms = 0;
      for (int i = 0; i < idx; i++) {
        if (policy == 0) U_edf += (all_processes[i]->execution_time * 1000) / (all_processes[i]->deadline_relative);
        if (policy == 1) U_rms += (10 * (all_processes[i]->rate) * (all_processes[i]->execution_time )) ;
      }
      if (policy == 0) U_edf += (p->execution_time * 1000) / (p->deadline_relative);
      if (policy == 1) U_rms += (10 * (p->rate) * (p->execution_time));

      if (policy == 0 && U_edf <= 1000) {
        all_processes[idx++] = p;
      } else if (policy == 1 && U_rms <= utility[idx]) {
        all_processes[idx++] = p;
      } else {
        p->killed = 1;
        release(&ptable.lock);
        return -22;
      }
      break;
    }
  }
  release(&ptable.lock);
  return 0;
}
```

This code implements a function named `__policy` which sets the scheduling policy of a process identified by `pid` to a given `policy` value. The function first acquires the `ptable.lock` to access the process table and iterates over all processes to find the one with the given `pid`. Once the process is found, its scheduling policy, arrival time, and deadline are updated based on the given `policy` value.

Then, the function checks if the process is schedulable under the current scheduling policy. For this purpose, it calculates the total utilization of CPU time (`U_edf` for `EDF` and `U_rms` for `RMS`) by summing up the utilization of all currently running processes as well as the new process, and compares it with the maximum allowed CPU utilization (1000 for `EDF` and the utility of the highest priority process for `RMS`).

If the new process is schedulable, it is added to the `all_processes` array and the function returns `0` indicating success. Otherwise, the process is marked as killed and the function returns `-22` (which is a Unix error code for `EINVAL` meaning invalid argument).

If the scheduling policy is `EDF`, the function prints a message indicating that the process is killed due to the violation of the CPU utilization bound and provides the value of the total CPU utilization. If the policy is `RMS`, the message includes the value of the highest priority process's utility as well.

3. SCHEDULING POLICIES & ALGORITHM

EDF

`EDF` stands for `Earliest Deadline First`. In `EDF` scheduling, processes are scheduled based on their deadlines, where the process with the earliest deadline is given priority to be executed first. This means that if there is a process with a deadline that is approaching quickly, it will be executed before other processes. `EDF` is often used in systems where tasks have hard deadlines that must be met, as it ensures that the tasks with the earliest deadlines are completed first.

RMS

`RMS` stands for `Rate Monotonic Scheduling`. In `RMS` scheduling, processes are scheduled based on their period, where the process with the shortest period is given priority to be executed first. This means that if there is a process with a short period, it will be executed before other processes. `RMS` is often used in systems where tasks have periodic deadlines and where the tasks have similar execution times, as it ensures that the tasks with the shortest periods are completed first.

IMPLEMENTATION:

The implementation of scheduling is in `scheduler` function inside `proc.c` file. The `scheduler` function is responsible for selecting the next process to be executed on the CPU.

The function starts by disabling interrupts on the processor to avoid race conditions while accessing shared data structures. It then tries to acquire a lock on the process table to access the information about the processes.

The function uses two scheduling policies, `EDF` and `RMS`. `EDF` (Earliest Deadline First) policy selects the process with the earliest deadline, and `RMS` (Rate Monotonic Scheduling) policy selects the process with the shortest period. The function also supports a third scheduling policy, Round Robin `(RR)`, which selects processes in a cyclic manner.

The function first searches for processes with `EDF` policy and selects the one with the earliest deadline. If no `EDF` process is found, it searches for processes with `RMS` policy and selects the one with the shortest period. If neither `EDF nor RMS` processes are found, it selects a process with `RR` policy.

The loop uses different criteria to select the next process depending on the scheduling policy:

- `Round Robin Policy (RR):` It simply selects the next `RUNNABLE` process in the process table.`
- `Earliest Deadline First (EDF) Policy:` It selects the process with the earliest deadline. If there is a tie, it selects the process with the `smallest PID`.
- `Rate Monotonic Scheduling (RMS) Policy:` It selects the process with the smallest weight. If there is a tie, it selects the process with the `smallest PID`.

Once a process is selected, the function switches to the selected process's address space by calling `switchuvm()`, sets its state to `RUNNING`, and switches to its context by calling `swtch()`. When the process returns, the function switches back to the kernel's address space by calling `switchkvm()`, and sets the current process to null. If no `RUNNABLE` process is found, the function simply releases the lock on the process table and loops back to the beginning to try again.

Overall, this implementation of the scheduler is using a preemptive scheduling approach, where the scheduler can interrupt a running process to switch to another process. It also supports multiple scheduling policies, allowing the `xv6` to adapt to different application requirements.

CODE SNIPPETS FOR SCHEDULING FUNCTION:

- SELECTING APPROPRIATE PROCESS TO BE SCHEDULED NEXT

```
acquire(&ptable.lock);
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
  if (p->state != RUNNABLE)
    continue;

  if (p->sched_policy == 0)
  { // edf scheduling
    if ((p->deadline < minD) || (p->deadline == minD && p->pid < pid_1))
    {
      minD = p->deadline;
      pid_1 = p->pid;
      new = p;
      found = 1;
    }
  }
  if (p->sched_policy == 1)
  { // rms scheduling
    if ((p->weight < minP) || (p->weight == minP && p->pid < pid_2))
    {
      minP = p->weight;
      pid_2 = p->pid;
      new = p;
      found = 1;
    }
  }
}
```

- SCHEDULING THE SELECTED PROCESS

```
if (found == 1) {
  c->proc = new;
  switchuvm(new);
  new->state = RUNNING;
  swtch(&(c->scheduler), new->context);
  switchkvm();
  c->proc = 0;
  release(&ptable.lock);
  continue;
}
```

4: EXTRA

- **CAN THERE BE NON-DETERMINISM IN ORDER OF PROCESSES EXITING ?**

  `YES`. The order of execution of processes in a preemptive scheduling algorithm like `EDF` can depend on factors such as the current state of the system, the timing of process creation and termination, and the specific implementation of the scheduler.

  Consider the following scenario:

  Suppose parent `P` has to fork `3` children `P1`,`P2`, and `P3`. Assume all processes are `EDF schedulable`, and the order of deadlines is `P3 < P2 < P1 < P`. Now one would expect that the final schedule would be something like `P3, P2, P1, P`. (`EDF` algorithm).

  But suppose, `P` is currently scheduled on the processor, and it forks `P1 and P2`. A timer interrupt occurs at this stage and `P` is displaced by `P2` as it has an earlier deadline than `P`. Note that `P3` has yet not been forked, but a timer interrupt has occurred. So only `P, P1, and P2` are the `3` existing processes currently. Among all these `P2` has the earliest deadline, hence it will be scheduled. Now `P2` will finish its job and exit. After this `P` will get an opportunity to run and it will finally fork `P3`. The order `P3, P1, and P` will now be followed as per the deadline values.

  Hence we got the output as `P2, P3, P1, P` which is different from what we expected.

  In the scenario we described, the order of execution of `P, P1, P2, and P3` can be non-deterministic due to the timing of process creation and the occurrence of a timer interrupt. Therefore, it can be challenging to predict the exact sequence of execution of processes in such scenarios. However, it is still possible to analyse the worst-case behaviour of the scheduler and determine if it meets certain performance guarantees, such as meeting all task deadlines in the case of `EDF scheduling`.