CSE6010 Mini-Project

Jipeng Wu

College of Computing
Georgia Institute of Technology

Project Presentation, 2015

Outline

1. Introduction
2. Problem Specification
3. Simulation of RTOS Runtime Context
4. Analysis

2 Concepts

1. Real-time Operating System
2. Task Scheduling

The Multi-level Feedback Queue

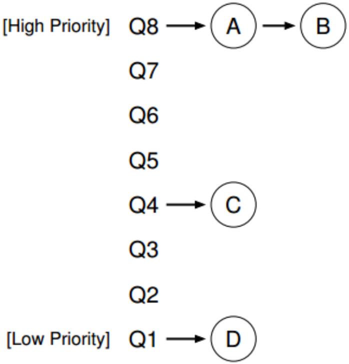

Figure: The Multi-level Feedback Queue

We Need Simple Alogrithms

1. Sometimes RTOS must guarantee events in serveral nanoseconds.
2. Even an AND gate needs about 500 picosecond! An instruction usually needs 5ns!
3. INTEGRITY-178B uses RMA(Rate Monotonic Analysis) to reduce syscall cost.

We Need Simple Alogrithms

1. Sometimes RTOS must guarantee events in serveral nanoseconds.
2. Even an AND gate needs about 500 picosecond! An instruction usually needs 5ns!
3. INTEGRITY-178B uses RMA(Rate Monotonic Analysis) to reduce syscall cost.

We Need Simple Alogrithms

1. Sometimes RTOS must guarantee events in serveral nanoseconds.
2. Even an AND gate needs about 500 picosecond! An instruction usually needs 5ns!
3. INTEGRITY-178B uses RMA(Rate Monotonic Analysis) to reduce syscall cost.

We Need Simple Alogrithms

1. Sometimes RTOS must guarantee events in serveral nanoseconds.
2. Even an AND gate needs about 500 picosecond! An instruction usually needs 5ns!
3. INTEGRITY-178B uses RMA(Rate Monotonic Analysis) to reduce syscall cost.

Periodic Hard-real-time Job Sets

- In this simulation, we have three or four periodic hard-real-time job sets: $S_i = J_{i1}, J_{i2}, \ldots, J_{in}$ Each job in S_i shares the same period p and the same instruction sequence length l.
  - J_{ik} arrives at the p(k-1)-th unit of time and must be finished before the pk-th unit of time.
- We need to simulate different runs under different real-time schedulers and evaluate their performance.

Periodic Hard-real-time Job Sets

- In this simulation, we have three or four periodic hard-real-time job sets:
  $S_i = J_{i1}, J_{i2}, \ldots, J_{in}$ Each job in S_i shares the same period p and the same instruction sequence length l.
  - J_{ik} arrives at the p(k-1)-th unit of time and must be finished before the pk-th unit of time.
- We need to simulate different runs under different real-time schedulers and evaluate their performance.

Periodic Hard-real-time Job Sets

- In this simulation, we have three or four periodic hard-real-time job sets:
  $S_i = J_{i1}, J_{i2}, \ldots, J_{in}$ Each job in S_i shares the same period p and the same instruction sequence length l.
  - J_{ik} arrives at the p(k-1)-th unit of time and must be finished before the pk-th unit of time.
- We need to simulate different runs under different real-time schedulers and evaluate their performance.

Periodic Hard-real-time Job Sets

- In this simulation, we have three or four periodic hard-real-time job sets:
  $S_i = J_{i1}, J_{i2}, \ldots, J_{in}$ Each job in S_i shares the same period p and the same instruction sequence length l.
  - J_{ik} arrives at the p(k-1)-th unit of time and must be finished before the pk-th unit of time.
- We need to simulate different runs under different real-time schedulers and evaluate their performance.

Periodic Hard-real-time Job Sets

- In this simulation, we have three or four periodic hard-real-time job sets:
  $S_i = J_{i1}, J_{i2}, \ldots, J_{in}$ Each job in S_i shares the same period p and the same instruction sequence length l.
  - J_{ik} arrives at the p(k-1)-th unit of time and must be finished before the pk-th unit of time.
- We need to simulate different runs under different real-time schedulers and evaluate their performance.

RTOS Schedulers And Periodic Jobs

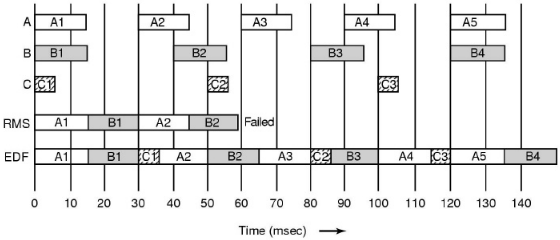

Figure: EDF and RMS

RT OS Simulation

1. Define tasks as sequences of instructions and a pointer to where it is executing. Each instruction costs the same constant units of time.
2. Use a paralleled thread to maintain time_tick, continuously incrementing it. Schedulers can read it.
3. Simulate periodic job events: every p units of time, add a new job into the task list, and record the execution time of last job.
4. For simplicity, there is only one interruption in our RTOS, the clock interruption.
5. The architecture allows both preemptive and non-preemptive scheduling.
6. Global resources: tick, task list, current task pointer, PC.

RT OS Simulation

1. Define tasks as sequences of instructions and a pointer to where it is executing. Each instruction costs the same constant units of time.
2. Use a paralleled thread to maintain time_tick, continuously incrementing it. Schedulers can read it.
3. Simulate periodic job events: every p units of time, add a new job into the task list, and record the execution time of last job.
4. For simplicity, there is only one interruption in our RTOS, the clock interruption.
5. The architecture allows both preemptive and non-preemptive scheduling.
6. Global resources: tick, task list, current task pointer, PC.

RT OS Simulation

1. Define tasks as sequences of instructions and a pointer to where it is executing. Each instruction costs the same constant units of time.
2. Use a paralleled thread to maintain time_tick, continuously incrementing it. Schedulers can read it.
3. Simulate periodic job events: every p units of time, add a new job into the task list, and record the execution time of last job.
4. For simplicity, there is only one interruption in our RTOS, the clock interruption.
5. The architecture allows both preemptive and non-preemptive scheduling.
6. Global resources: tick, task list, current task pointer, PC.

RT OS Simulation

1. Define tasks as sequences of instructions and a pointer to where it is executing. Each instruction costs the same constant units of time.
2. Use a paralleled thread to maintain time_tick, continuously incrementing it. Schedulers can read it.
3. Simulate periodic job events: every p units of time, add a new job into the task list, and record the execution time of last job.
4. For simplicity, there is only one interruption in our RTOS, the clock interruption.
5. The architecture allows both preemptive and non-preemptive scheduling.
6. Global resources: tick, task list, current task pointer, PC.

RT OS Simulation

1. Define tasks as sequences of instructions and a pointer to where it is executing. Each instruction costs the same constant units of time.
2. Use a paralleled thread to maintain time_tick, continuously incrementing it. Schedulers can read it.
3. Simulate periodic job events: every p units of time, add a new job into the task list, and record the execution time of last job.
4. For simplicity, there is only one interruption in our RTOS, the clock interruption.
5. The architecture allows both preemptive and non-preemptive scheduling.
6. Global resources: tick, task list, current task pointer, PC.

RT OS Simulation

1. Define tasks as sequences of instructions and a pointer to where it is executing. Each instruction costs the same constant units of time.
2. Use a paralleled thread to maintain time_tick, continuously incrementing it. Schedulers can read it.
3. Simulate periodic job events: every p units of time, add a new job into the task list, and record the execution time of last job.
4. For simplicity, there is only one interruption in our RTOS, the clock interruption.
5. The architecture allows both preemptive and non-preemptive scheduling.
6. Global resources: tick, task list, current task pointer, PC.

Clock Interrupt Simulation Thread

Listing 1: clock interruption ISR simulation

```
while(true){ //ISR(Interruption Service Routine)
  tick++;
  delay(CLOCK_CYCLE); //long enough to print
      something
}
```

Kernel And User Code Simulation Thread

Listing 2: main simulation thread

```
while(true){
  if(interrupted){ //simulate kernel-mode code
    store_context_of(current_task); //store
        context
    if(is_preemptive){
      schedule();
    }
    else{
      time_update();
    }
  }
  else{ // runtime strictly < 1 clock cycle
    exec(PC);//spurious exec function, print sth.
  }
}
```

Tick Updation Routine

Listing 3: main simulation thread

```
void timeupdate(){
    current_task->inst[PC].tick--;
    if(!current_task->inst[PC].tick){
      PC++;
    }
    if(PC==current_task->nr_inst){
      schedule();
    }
}
```

Performance Analysis

The analysis will include:

1. The most important performance metric is the Successive Ratio and it is defined as,
   $SR = \frac{\text{Number of jobs successfully scheduled}}{\text{Total number of jobs arrived}}$
2. Execution Time E_i.
   $E_i = \begin{cases} \text{execution time of the job} & \text{if deadline met} \\ 0 & \text{if deadline not met} \end{cases}$
3. ECU(Effective CPU Utilization) gives information about how efficiently the processor is used and it is defined as,
   $ECU = \sum_{i \in R} \frac{E_i}{T}$. $T =$ Total time of scheduling

Performance Analysis

The analysis will include:

1. The most important performance metric is the Successive Ratio and it is defined as,
   $SR = \frac{\text{Number of jobs successfully scheduled}}{\text{Total number of jobs arrived}}$
2. Execution Time E_i.
   $E_i = \begin{cases} \text{execution time of the job} & \text{if deadline met} \\ 0 & \text{if deadline not met} \end{cases}$
3. ECU(Effective CPU Utilization) gives information about how efficiently the processor is used and it is defined as,
   $ECU = \sum_{i \in R} \frac{E_i}{T}$. $T =$ Total time of scheduling

Performance Analysis

The analysis will include:

1. The most important performance metric is the Successive Ratio and it is defined as,
   $SR = \frac{\text{Number of jobs successfully scheduled}}{\text{Total number of jobs arrived}}$
2. Execution Time E_i.
   $$E_i = \begin{cases} \text{execution time of the job} & \text{if deadline met} \\ 0 & \text{if deadline not met} \end{cases}$$
3. ECU(Effective CPU Utilization) gives information about how efficiently the processor is used and it is defined as,
   $ECU = \sum_{i \in R} \frac{E_i}{T}$. $T =$ Total time of scheduling

Performance Analysis

The analysis will include:

1. The most important performance metric is the Successive Ratio and it is defined as,
$SR = \frac{\text{Number of jobs successfully scheduled}}{\text{Total number of jobs arrived}}$
2. Execution Time E_i.
$$E_i = \begin{cases} \text{execution time of the job} & \text{if deadline met} \\ 0 & \text{if deadline not met} \end{cases}$$
3. ECU(Effective CPU Utilization) gives information about how efficiently the processor is used and it is defined as,
$ECU = \sum_{i \in R} \frac{E_i}{T}$. $T =$ Total time of scheduling

Q & A