Lecture 18: Real-Time Scheduling I

Seyed-Hosein Attarzadeh-Niaki

Based on the slide by Edward Lee and Rodolfo Pelizzoni

Review

- Multitasking
- Threads
- Mutual exclusion
- Deadlock

Elements of Computation

Instruction

It is the elementary entity of a programming language.

Examples in ASM x86:

```
MOV AX, 5
MOV BX, 7
ADD AX, BX
```

Examples in C:

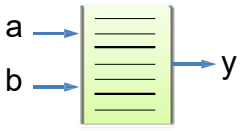
```
int x;
x = a + b;
if (x > threshold) y = 1;
else y = 0;
```

7

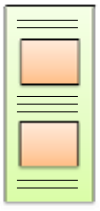
Elements of Computation

Function

It is a container for a set of instructions. It may take multiple input arguments and produces a single output.



A function can call other functions:



8

Elements of Computation

Task

- It is a function performing a given computational activity in a system (e.g., sensory processing, motor control, filtering).
- It is the elementary entity managed by an operating system.
- It may have specific constraints (e.g., activation time, period, deadline, precedence relations with other tasks).
- It can communicate with other tasks by shared resources.

Resource

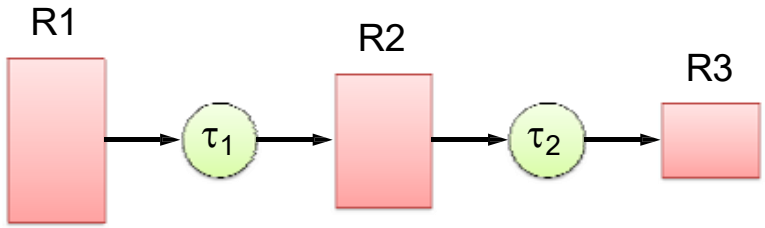
It is a set of variables that can be used by tasks to store data or temporary results.

9

Elements of Computation

Application

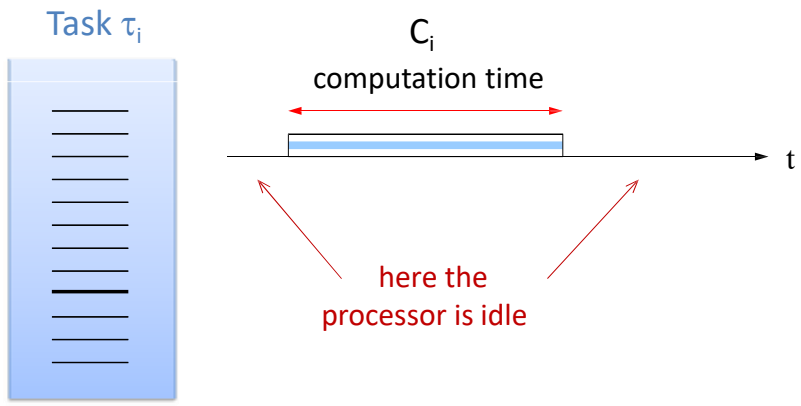
It consists of a set of tasks interacting through a set of shared resources.



10

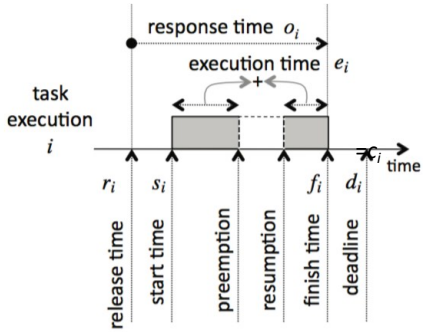
Task Execution

- The execution of a task on a processor is represented by a bar on a timeline.



Task Model

- Task: a real-time computation unit
  - Think of it as an execution thread with additional timing parameters (in particular: service time is known).
  - The job of the real-time scheduler is to *schedule tasks*.
- Three main task models
  - Aperiodic
  - Periodic
  - Sporadic
- Scheduler may (also) use *priority*
  - Fixed priority: const. over all execs
  - Dynamic priority: change allowed



$$s_i \geq r_i \qquad f_i \geq s_i$$

$$o_i = f_i - r_i$$

Aperiodic Tasks

- Event-triggered computation.
- Task is activated by an external event.
- Task runs once to respond to the event.
- Relative deadline D: available time to respond to the event.
- Ex: event = loss of power. Task = drop control rods into nuclear reactor (this actually happened at Fukushima)

Periodic Task

- Time-triggered computation
- Task is activated periodically every T time units.
- Each periodic instance of the task is called a job.
- Each job has the same relative deadline (usually = to period).
- Ex: most digital controllers.

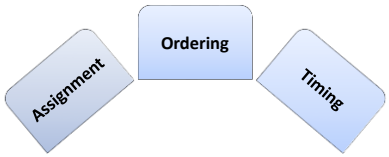
- **Sporadic task**: same as periodic task, but the task is activated at most every T time units (minimum interarrival time).
- Ex: processing network packets.

Scheduling Concurrent Tasks

- When
  - there are fewer processors than tasks,
  - tasks must be performed at a particular time

  a *scheduler* must be intervene.
- Multiprocessor scheduler also performs *processor assignment*.
- *Real-time systems*: a collection of tasks where
  - in addition to the precedence constraints,
  - there are *timing constraints*.
- Timing constraints: deadlines, earliest start time, periodicity, …

Scheduling Decisions



Fully-static scheduler: all three decisions at design time

- No need for semaphores or locks.
- Difficult to realize: execution times hard to predict

Static order scheduler (*off-line scheduler*)

- task assignment and ordering at design time

Static assignment scheduler: assignment at design time

- A *run-time scheduler* decides what task to execute next online

Fully-dynamic scheduler (*on-line schedulers*):

- all decisions at run time

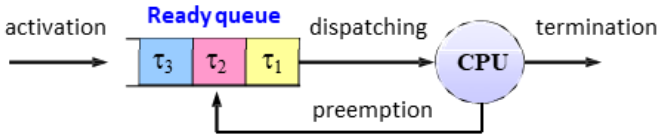
Preemptive Scheduling

Assumptions

1. All threads have priorities
   - either statically assigned (constant for the duration of the thread)
   - or dynamically assigned (can vary).
2. Kernel keeps track of which threads are "enabled" (able to execute, e.g. not blocked waiting for a semaphore or a mutex or for a time to expire).

Preemptive scheduling

- At any instant, the enabled thread with the highest priority is executing.
- Whenever any thread changes priority or enabled status, the kernel can dispatch a new thread.



When Can a New Thread Be Dispatched?

- *Under Non-Preemptive scheduling*
  - When the current thread completes.
- *Under Preemptive scheduling*
  - Upon a timer interrupt
  - Upon an I/O interrupt (possibly)
  - When a new thread is created, or one completes.
  - When the current thread blocks on or releases a mutex
  - When the current thread blocks on a semaphore
  - When a semaphore state is changed
  - When the current thread makes any OS call
    - file system access
    - network access
    - …

How to decide which thread to schedule?

Considerations

- Preemptive vs. non-preemptive scheduling
- Periodic vs. aperiodic tasks
- Fixed priority vs. dynamic priority
- Priority inversion anomalies
- Other scheduling anomalies

Comparing Schedulers

- *Feasible schedule*: all task executions meet their deadlines ($f_i \leq d_i$)
  - A scheduler that yields a feasible schedule for any task set (that conforms to its task model) for which there is a feasible schedule is said to be *optimal with respect to feasibility*.

1. *Utilization*: percentage of time that the processor spends executing tasks.
2. *Maximum **lateness***: $L_{\max} = \max_{i \in T}(f_i - d_i)$
   - For feasible schedules, zero or negative
3. *Total completion time* (*makespan*): $M = \max_{i \in T} f_i - \min_{i \in T} r_i$
   - A performance goal

Utilization & Schedulability Analysis

- Task Utilization for a periodic/sporadic task: $U_i = e_i / p_i$.
  - Percentage of processor time required by the task.
- System Utilization: $U = U_1 + U_2 + \ldots + U_N$.
  - Percentage of processor time required by all tasks.

- Base uniprocessor scheduling result: task set is clearly not schedulable if: $U > 1$.
- For many scheduling algorithms, we can define a utilization bound U_b such that the task set if schedulable if: $U <= U_b$.

17

Outline of a Microkernel Scheduler

- Scheduler may be part of a compiler or code generator, part of an OS or microkernel, or both.
- Main Scheduler Thread (Task)
  - set up periodic timer interrupts;
  - create default thread data structures;
  - dispatch a thread (procedure call);
  - execute main thread (idle or power save, for example).
- Thread data structure
  - copy of all state (machine registers)
  - address at which to resume executing the thread
  - status of the thread (e.g. blocked on mutex)
  - priority, WCET (worst case execution time), and other info to assist the scheduler

Outline of a Microkernel Scheduler

- Timer interrupt service routine
  - dispatch a thread.
- Dispatching a thread
  - *disable interrupts;*
  - determine which thread should execute (scheduling);
  - if the same one, enable interrupts and return;
  - save state (registers) into current thread data structure;
  - save return address from the stack for current thread;
  - copy new thread state into machine registers;
  - replace program counter on the stack for the new thread;
  - *enable interrupts;*
  - return.

Classical Scheduling Policies

First Come First Served

- Assigns the CPU to tasks based on their arrival times (intrinsically non preemptive).
- Very unpredictable: response times strongly depend on task arrivals

Shortest Job First

- selects the ready task with the shortest computation time
- Not optimal in the sense of feasibility

Priority Scheduling

- task with the highest priority is selected for execution
- Problem: starvation

Round Robin

- The ready queue is served with FCFS, but Each task cannot execute for more than a time limit
- No response-time guarantee

Not suited for real-time systems

Rate Monotonic Scheduling

Assume *n* tasks invoked periodically with
- periods $p_1, \ldots, p_n$ (impose real-time constraints)
- worst-case execution times (WCET) $e_1, \ldots, e_n$
  - assumes no mutexes, semaphores, or blocking I/O
- no precedence constraints
- fixed priorities
- preemptive scheduling

Rate Monotonic Scheduling: Priorities ordered by period (smallest period has the highest priority)

Rate Monotonic Scheduling

Assume *n* tasks invoked periodically with
- periods $p_1, \ldots, p_n$ (impose real-time constraints)
- worst-case execution times (WCET) $e_1, \ldots, e_n$
  - assumes no mutexes, semaphores, or blocking I/O
- no precedence constraints
- fixed priorities
- preemptive scheduling

Theorem: If any (fixed) priority assignment yields a feasible schedule, then RMS (smallest period has the highest priority) also yields a feasible schedule.

RMS is optimal in the sense of feasibility.

Liu and Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," J. ACM, 20(1), 1973.

Liu and Layland, *JACM*, Jan. 1973

Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment

C. L. LIU

Project MAC, Massachusetts Institute of Technology

AND

JAMES W. LAYLAND

Jet Propulsion Laboratory, California Institute of Technology

ABSTRACT. The problem of multiprogram scheduling on a single processor is studied from the viewpoint of the characteristics peculiar to the program functions that need guaranteed service. It is shown that an optimum fixed priority scheduler possesses an upper bound to processor utilization which may be as low as 70 percent for large task sets. It is also shown that full processor utilization can be achieved by dynamically assigning priorities on the basis of their current deadlines. A combination of these two scheduling techniques is also discussed.

KEY WORDS AND PHRASES: real-time multiprogramming, scheduling, multiprogram scheduling, dynamic scheduling, priority assignment, processor utilization, deadline driven scheduling

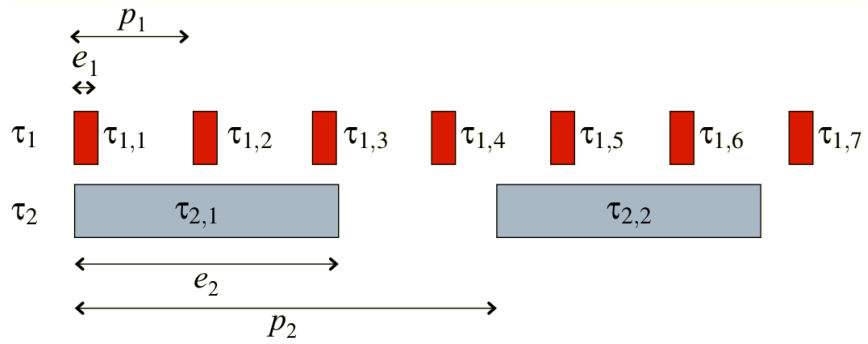
CR CATEGORIES: 3.80, 3.82, 3.83, 4.32

Feasibility for RMS

Feasibility is defined for RMS to mean that

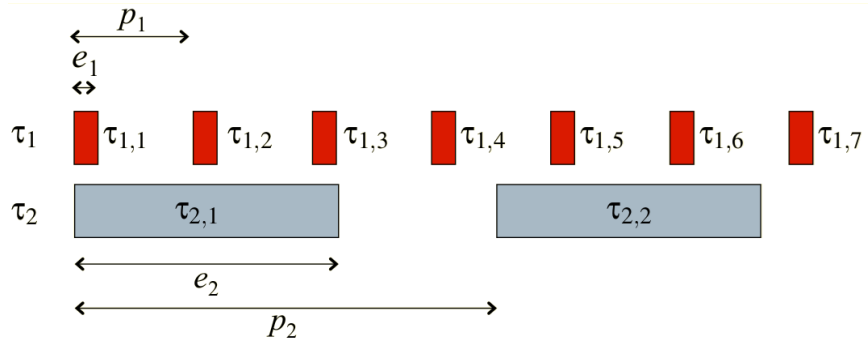
- every task executes to completion once within its designated period.

Showing Optimality of RMS: Two tasks with different periods



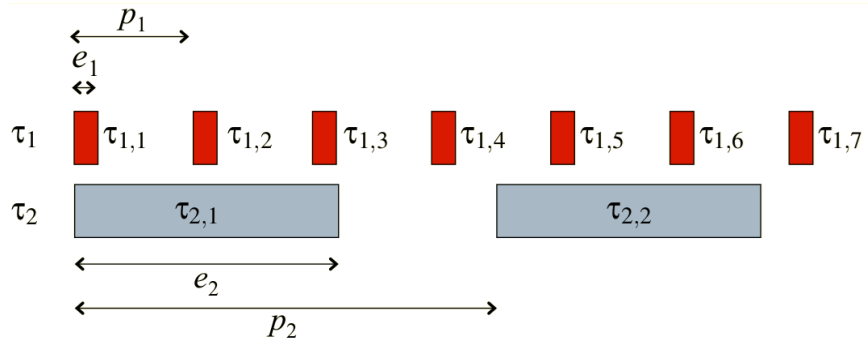
Is a non-preemptive schedule feasible?

Showing Optimality of RMS: Two tasks with different periods



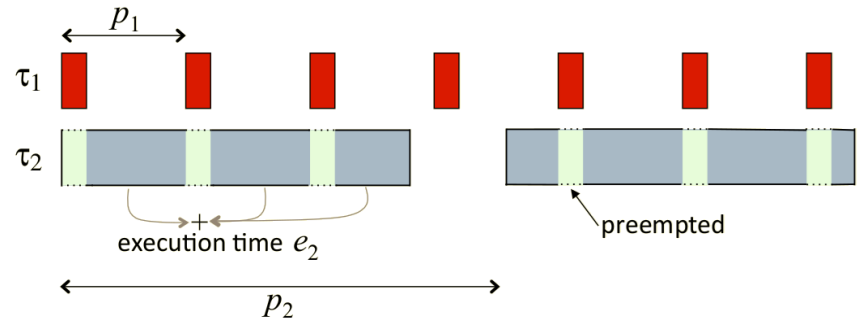
Non-preemptive schedule is not feasible. Some instance of the Red Task (τ_1) will not finish within its period if we do non-preemptive scheduling.

Showing Optimality of RMS: Two tasks with different periods



What if we had a preemptive scheduling with higher priority for the red task?

Showing Optimality of RMS: Two tasks with different periods



Preemptive schedule with the red task having higher priority is feasible. Note that preemption of the blue task extends its completion time.

Any Implicit Assumptions in Our Reasoning?

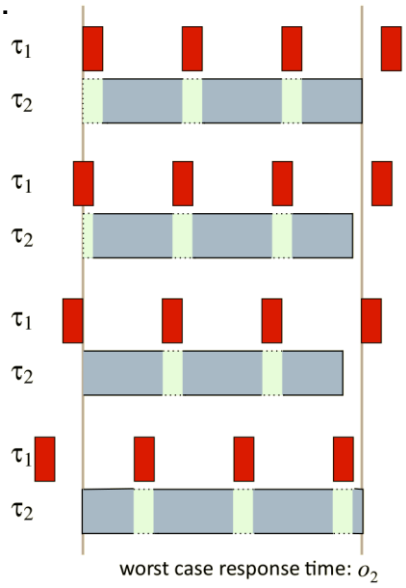
- Assume zero time needed to switch between tasks.
- Sources of context switch time?

Showing Optimality of RMS for 2 Tasks: Key Proof Ideas

1. Worst case: when both tasks arrive together.
2. For this case, show if non-RMS is feasible, then RMS is feasible.

Showing Optimality of RMS: Alignment of tasks

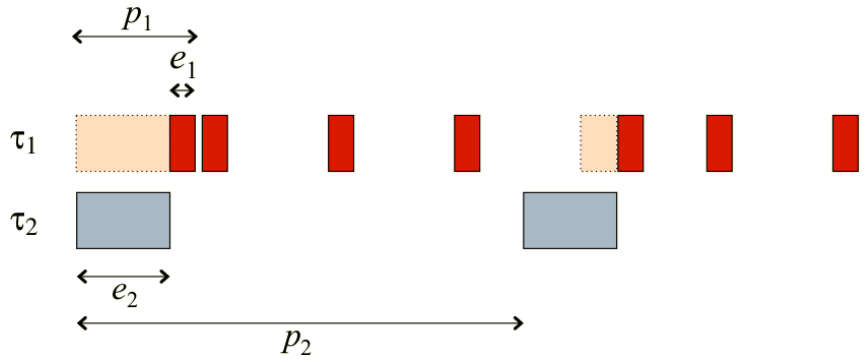
- Completion time of the lower priority task is worst (latest) when
  - its *starting phase* matches that of higher priority tasks.
- Thus, when checking schedule feasibility, it is sufficient to consider only the worst case:
  - All tasks start their cycles at the same time.



Showing Optimality of RMS: (for two tasks)

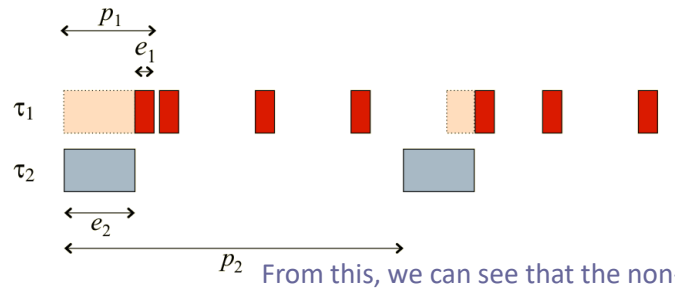
It is sufficient to show that if a non-RMS schedule is feasible, then the RMS schedule is feasible.

Consider two tasks as follows:



Showing Optimality of RMS: (for two tasks)

The non-RMS, fixed priority schedule looks like this:



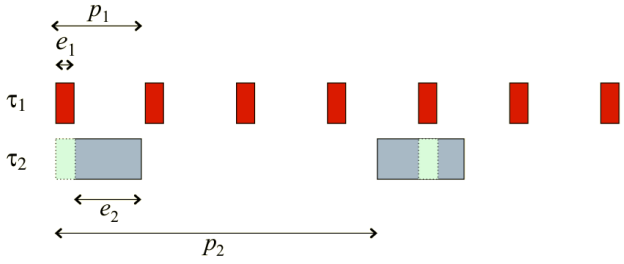
From this, we can see that the non-RMS schedule is feasible if and only if

$$e_1 + e_2 \leq p_1$$ ← Actually: min(p_1,p_2)

We can then show that this condition implies that the RMS schedule is feasible.

Showing Optimality of RMS: (for two tasks)

The RMS schedule looks like this: (task with smaller period moves earlier)



The condition for the non-RMS schedule feasibility:

$$e_1 + e_2 \leq p_2$$

is clearly sufficient (though not necessary) for feasibility of the RMS schedule. QED.

Rate-Monotonic Scheduling

- Liu & Layland schedulability analysis: task set is schedulable if:
  $U <= U_b(N) = N(2^{1/N} - 1)$
- If n = 2, then $n(2^{1/n} - 1) \approx 0.828$
- Note that
  $\lim_{N->+inf} N(2^{1/N}-1) = \log 2 \sim= 0.693$
- What happens if $U_b(N) < U <= 1$?
  - Nothing can be said according to the analysis
  - Task set might or might not be schedulable

35

Comments

- This proof can be extended to an arbitrary number of tasks (though it gets much more tedious).
- This proof gives optimality only w.r.t. feasibility. It says nothing about other optimality criteria.
- Practical implementation:
  - Timer interrupt at greatest common divisor of the periods.
  - Multiple timers

Next Lecture

- Earliest Due Date (EDD) and Earliest Deadline First (EDF) scheduling
  - Optimality
- Precedence Constraints
  - Latest Deadline First (LDF) scheduling
  - EDF* scheduling