# Real-Time Scheduling

Priority-Driven Scheduling

Fixed-Priority

[Some parts of this lecture are based on a real-time systems course of Colin Perkins

http://csperkins.org/teaching/rtes/index.html]

## Current Assumptions

- Single processor
- Fixed number, $n$, of *independent periodic* tasks

  i.e. there is no dependency relation among jobs
  - Jobs can be preempted at any time and never suspend themselves
  - No aperiodic and sporadic jobs
  - No resource contentions

Moreover, unless otherwise stated, we assume that

- **Scheduling decisions take place precisely at**
  - release of a job
  - completion of a job

  (and nowhere else)
- Context switch overhead is negligibly small

  i.e. assumed to be zero
- There is an unlimited number of priority levels

## Fixed-Priority Algorithms

Consider a set of tasks $T = \{T_1, \dots, T_n\}$

Any fixed-priority algorithm schedules tasks according to fixed priorities assigned to tasks, w.l.o.g. assume

$$T_1 \succ T_2 \succ \cdots \succ T_n$$

i.e. $T_1$ has the highest priority, $T_n$ has the lowest priority

Define

$$T_{\uparrow i} := \{T_k \mid T_k \succeq T_i\}$$

## Utilization

- *Utilization $u_i$ of a periodic task $T_i$* with period $p_i$ and execution time $e_i$ is defined by $u_i := e_i/p_i$
  - The fraction of time a periodic task with period $p_i$ and execution time $e_i$ keeps a processor busy
- *Total utilization $U^T$ of a set of tasks $T = \{T_1, \ldots, T_n\}$* is defined as the sum of utilizations of all tasks of $T$, i.e. by

$$U^T := \sum_{i=1}^{n} u_i$$

- $U$ is a *schedulable utilization* of an algorithm ALG if all sets of tasks $T$ satisfying $U^T \leq U$ are schedulable by ALG.

  *Maximum schedulable utilization $U_{ALG}$ of an algorithm ALG* is the *supremum of schedulable utilizations of ALG*.

## Critical Instant – Informally

To be able to further analyze fixed-priority algorithms we need to consider a notion of *critical instant*

Intuitively, a critical instant is the time instant in which the system is most loaded, and has its worst response time

Schedulability of a set of tasks is determined by response times of jobs released at critical instants

## Critical Instant – Formally

**Definition 1**

A **critical instant** $t_{crit}$ of a task $T_i$ is a time instant in which a job $J_{i,k}$ in $T_i$ is released so that $J_{i,k}$ either does not meet its deadline, or has the maximum response time of all jobs in $T_i$

Denote by $W_i$ the response time of such $J_{i,k}$

**Theorem 2**

*In a fixed-priority system where every job completes before the next job in the same task is released, a critical instant occurs when one of its jobs $J_{i,k}$ is released at the same time with a job from every higher-priority task.*

Note that the situation described in the theorem does not have to occur if tasks are not in phase. So we use critical instants either to study tasks in phase, or to get upper bounds on schedulability as follows:

- Set phases of all jobs in $T_{\uparrow i}$ to zero, which gives a new set of tasks $T'$
- Determine the response time $w'$ of the first job $J_{i,1}$ in $T'_i$

Then $w' \geq W_i$, the response time of a job in the original set $T$ released at the critical instant

## Optimality of RM for Simply Periodic Tasks

**Definition 3**

A set $\{T_1, \ldots, T_n\}$ is **simply periodic** if for every pair $T_i$, $T_k$ satisfying $p_i < p_k$ we have that $p_k$ is an integer multiple of $p_i$

**Example 4**

The helicopter control system from the first lecture

**Theorem 5**

*A set $T$ of $n$ simply periodic, independent, preemptable tasks with $D_i = p_i$ is schedulable on one processor according to RM* **iff** $U^T \leq 1$.

i.e. on simply periodic tasks RM is as good as EDF

## Optimality of DM (RM) among Fixed-Priority Algs.

**Theorem 6**

*A set of independent, preemptable periodic tasks with $D_i \leq p_i$ that are in phase can be feasibly scheduled on one processor according to DM if it can be feasibly scheduled by some fixed-priority algorithm.*

**Proof.**

Assume a fixed-priority feasible schedule with $T_1 \succ \cdots \succ T_n$.

Consider the least $i$ such that the relative deadline $D_i$ of $T_i$ is larger than the relative deadline $D_{i+1}$ of $T_{i+1}$.

Swap the priorities of $T_i$ and $T_{i+1}$.

The resulting schedule is still feasible.

DM is obtained by using finitely many swaps. □

Note: If the assumptions of the above theorem hold and all relative deadlines are equal to periods, then RM is optimal among all fixed-priority algorithms.

## Fixed-Priority Algorithms: Schedulability

We consider two schedulability tests:

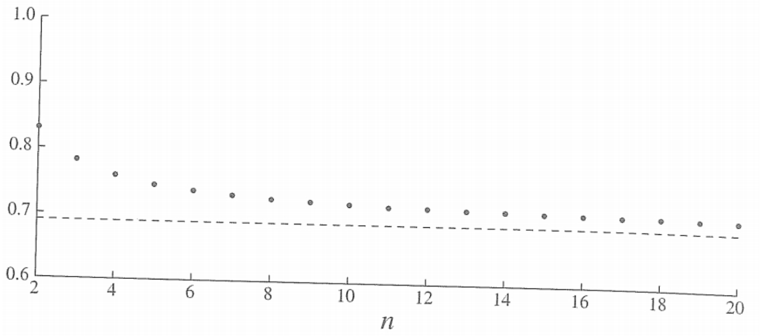
- Schedulable utilization $U_{RM}$ of the RM algorithm.
- Time-demand analysis based on response times of jobs released at critical instants

# Schedulable Utilization for RM

**Theorem 7**

*Let us fix $n \in \mathbb{N}$ and consider only independent, preemptable periodic tasks with $D_i = p_i$.*

- *If $T$ is a set of $n$ tasks satisfying $U^T \leq n(2^{1/n} - 1)$, then $U^T$ is schedulable by RM algorithm.*
- *For every $U > n(2^{1/n} - 1)$ there is a set $T$ of $n$ tasks satisfying $U^T \leq U$ that is not schedulable by RM.*



## Schedulable Utilization for RM

It follows that the maximum schedulable utilization $U_{RM}$ over independent, preemptable periodic tasks satisfies

$$U_{RM} = \inf_{n} n(2^{1/n} - 1) = \lim_{n \to \infty} n(2^{1/n} - 1) = \ln 2 \approx 0.693$$

Note that $U^{T} \leq n(2^{1/n} - 1)$ is a sufficient but not necessary condition for schedulability of $T$

## Proof Sketch of Theorem 7

For the proof we need the following notions:

- A set of tasks is *difficult-to-schedule* if it is schedulable by RM but any increase in execution time and/or decrease in period makes the set unschedulable
- A set of tasks is *most-difficult* if its total utilization is smallest among all difficult to schedule sets

**Step 1.** Show that for every difficult-to-schedule set there is another one whose utilization cannot be larger and

1. is in phase
2. satisfies $p_n \leq 2p_1$

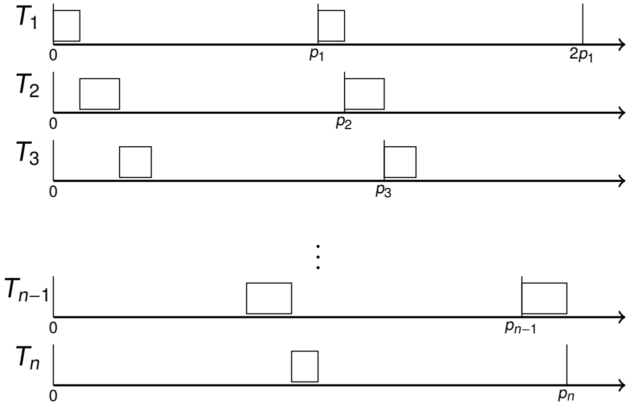
from this moment on we may concentrate on sets $T$ satisfying 1. and 2.

**Step 2.** For a fixed set of periods $p_1, \dots, p_n$, find the most-difficult set $T$ with these periods, which gives $U[p_1, \dots, p_n] := U^T$

**Step 3.** Find $T$ with the "most-difficult" values of periods $p_1, \dots, p_n$, that is minimize $U[p_1, \dots, p_n]$, and show that $U^T = n(2^{1/n} - 1)$

## Proof Sketch of Theorem 7 – Step 2.

Most-difficult in phase set with $p_n \leq 2p_1$:



$$e_k = p_{k+1} - p_k \qquad \text{for } k = 1, \dots, n-1$$

$$e_n = p_n - 2\sum_{k=1}^{n-1} e_k = 2p_1 - p_n$$

## Time-Demand Analysis

Assume that $D_i \leq p_i$ for every $i$.

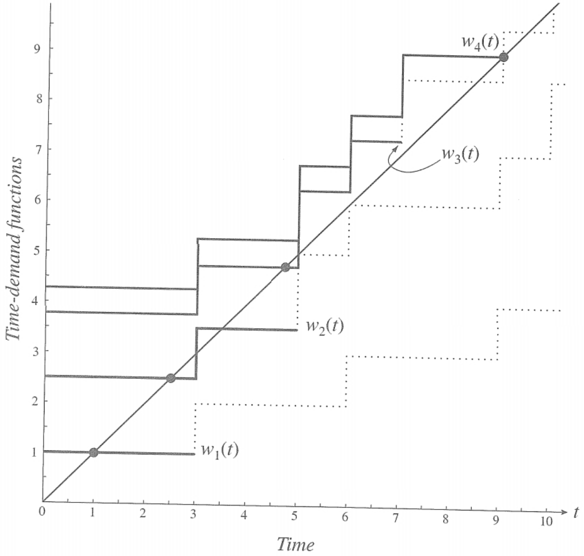
- Compute the total demand for processor time by a job released at a critical instant of a task, and by all the higher-priority tasks, as a function of time from the critical instant
- Check if this demand can be met before the deadline of the job:
  - Consider one task $T_i$ at a time, starting with highest priority and working to lowest priority
  - Focus on a job $J_{i,c}$ in $T_i$, where the release time, $t_0$, of that job is a critical instant of $T_i$
  - At time $t_0 + t$ for $t \geq 0$, the processor time demand $w_i(t)$ for this job and all higher-priority jobs released in $[t_0, t]$ is bounded by

$$w_i(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k \qquad \text{for } 0 < t \leq p_i$$

## Time-Demand Analysis

- Compare the time demand, $w_i(t)$, with the available time, $t$:
  - If $w_i(t) \leq t$ for some $t \leq D_i$, the job $J_{i,c}$ released at critical instant of $T_i$ meets its deadline, $t_0 + D_i$
  - If $w_i(t) > t$ for all $0 < t \leq D_i$, then the task probably cannot complete by its deadline; and the system likely cannot be scheduled using a fixed priority algorithm
    (Note that this condition is only sufficient as the expression for $w_i(t)$ relies on the fact that jobs of all higher priority tasks are released at the critical instant $t_0$)
- Use this method to check that all tasks are schedulable if released at their critical instants; if so conclude the entire system can be scheduled

## Time-Demand Analysis



Example: $T_1 = (3, 1)$, $T_2 = (5, 1.5)$, $T_3 = (7, 1.25)$, $T_4 = (9, 0.5)$

## Time-Demand Analysis

- The time-demand function $w_i(t)$ is a staircase function
  - Steps in the time-demand for a task occur at multiples of the period for higher-priority tasks
  - The value of $w_i(t) - t$ linearly decreases from a step until the next step
- If our interest is the schedulability of a task, it suffices to check if $w_i(t) \leq t$ at the time instants when a higher-priority job is released and at $D_i$
- Our schedulability test becomes:
  - Compute $w_i(t)$
  - Check whether $w_i(t) \leq t$ for some $t$ equal either to $D_i$, or to $j \cdot p_k$ where $k = 1, 2, \ldots, i$ and $j = 1, 2, \ldots, \lfloor D_i / p_k \rfloor$

## Time-Demand Analysis

- Time-demand analysis schedulability test is more complex than the schedulable utilization test but more general:
  - Works for *any* fixed-priority scheduling algorithm, provided the tasks have short response time ($D_i \leq p_i$)
    Can be extended to tasks with arbitrary deadlines
- Still more efficient than exhaustive simulation
- Only a sufficient test (as well as the utilization test for fixed-priority systems)

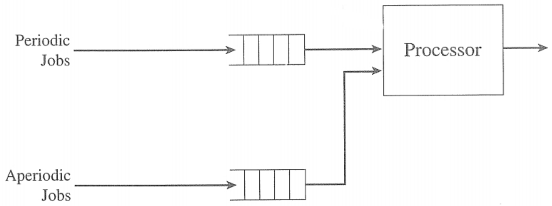
# Real-Time Scheduling

## Priority-Driven Scheduling

### Aperiodic Tasks

## Current Assumptions

- Single processor
- Fixed number, *n*, of *independent periodic* tasks
  - Jobs can be preempted at any time and never suspend themselves
  - No resource contentions
- Aperiodic jobs exist
  - They are independent of each other, and of the periodic tasks
  - They can be preempted at any time
- There are no sporadic jobs (for now)
- Jobs are scheduled using a priority driven algorithm



## Scheduling Aperiodic Jobs

Consider:

- A set $T = \{T_1, \ldots, T_n\}$ of periodic tasks
- An aperiodic task $A$

Recall that:

- A schedule is feasible if all jobs with hard real-time constraints complete before their deadlines

  $\Rightarrow$ This includes all periodic jobs
- A scheduling algorithm is optimal if it always produces a feasible schedule whenever such a schedule exists, and if a cost function is given, minimizes the cost

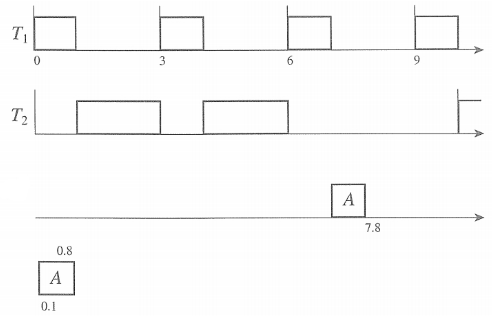
  $\Rightarrow$ We consider a cost function which to a schedule assigns the average response time of aperiodic jobs

We assume that the periodic tasks can be scheduled using a priority-driven algorithm

# Background Scheduling of Aperiodic Jobs

- Aperiodic jobs are scheduled and executed only at times when there are no periodic jobs ready for execution
- Advantages
  - Clearly produces feasible schedules
  - Extremely simple to implement
- Disadvantages
  - Not optimal since it is almost guaranteed to delay execution of aperiodic jobs in favour of periodic ones

**Example:** $T_1 = (3, 1)$, $T_2 = (10, 4)$

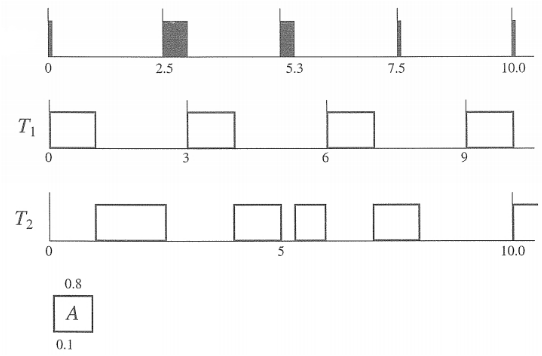


## Polled Execution of Aperiodic Jobs

- We may use a *polling server*
    - A periodic job $(p_s, e_s)$ scheduled according to the periodic algorithm, generally as the highest priority job
    - When executed, it examines the aperiodic job queue
        - If an aperiodic job is in the queue, it is executed for up to $e_s$ time units
        - If the aperiodic queue is empty, the polling server self-suspends, giving up its execution slot
        - The server does not wake-up once it has self-suspended, aperiodic jobs which become active during a period are not considered for execution until the next period begins
- Simple to prove correctness, performance less than ideal – executes aperiodic jobs in particular timeslots

# Polled Execution of Aperiodic Jobs

**Example:** $T_1 = (3, 1)$, $T_2 = (10, 4)$, *poller* $= (2.5, 0.5)$



Can we do better?

Yes, polling server is a special case of *periodic-server* for aperiodic jobs

## Periodic Severs – Terminology

*periodic server* = a task that behaves much like a periodic task, but is created for the purpose of executing aperiodic jobs

- A periodic server, $T_S = (p_S, e_S)$
  - $p_S$ is a period of the server
  - $e_S$ is the (maximal) *budget* of the server
- The budget can be *consumed* and *replenished*; the budget is *exhausted* when it reaches 0

  (Periodic servers differ in how they consume and replenish the budget)
- A periodic server is
  - *backlogged* whenever the aperiodic job queue is non-empty
  - *idle* if the queue is empty
  - *eligible* if it is backlogged and the budget is not exhausted
- When a periodic server is eligible, it is scheduled as any other periodic task with parameters $(p_S, e_S)$

## Periodic Severs

Each periodic server is thus specified by

- *consumption rules*: How the budget is consumed
- *replenishment rules*: When and how the budget is replenished

Example – polling server:

- *consumption rules*:
  - Whenever the server executes, the budget is consumed at the rate one per unit time.
  - Whenever the server becomes idle, the budget gets immediately exhausted
- *replenishment rule*: At each time instant $k \cdot p_S$ replenish the budget to $e_S$

## Deferrable Sever

- *Consumption rule*:
  - The budget is consumed at the rate of one per unit time whenever the server executes
  - Unused budget is retained throughout the period, to be used whenever there are aperiodic jobs to execute

    (i.e. instead of discarding the budget if no aperiodic job to execute at start of period, keep in the hope a job arrives)
- *Replenishment rule*:
  - The budget is set to $e_S$ at multiples of the period
    - i.e. time instants $k \cdot p_S$ for $k = 0, 1, 2, \ldots$

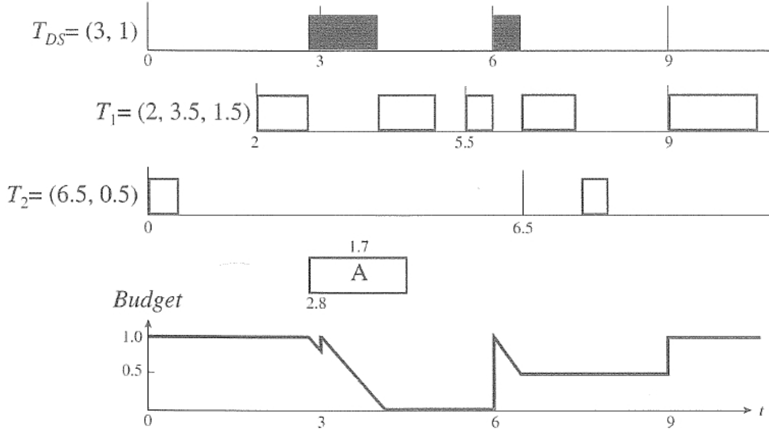
    (Note that the server is not able tu cumulate the budget over periods)

We consider both

- Fixed-priority scheduling
- Dynamic-priority scheduling (EDF)

## Deferrable Server – RM

Here the tasks are scheduled using RM.



Is it possible to increase the budget of the server to 1.5 ?