Real-Time Scheduling

Formal Model

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

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

Real-Time Scheduling – Formal Model

- Introduce an abstract model of real-time systems
    - abstracts away unessential details
    - sets up consistent terminology

- Three components of the model
    - A workload model that describes applications supported by the system

      i.e. jobs, tasks, ...
    - A resource model that describes the system resources available to applications

      i.e. processors, passive resources, ...
    - Algorithms that define how the application uses the resources at all times

      i.e. scheduling and resource access protocols

Basic Notions

- A *job* is a unit of work that is scheduled and executed by a system

  compute a control law, transform sensor data, etc.

- A *task* is a set of related jobs which jointly provide some system function

  check temperature periodically, keep a steady flow of water

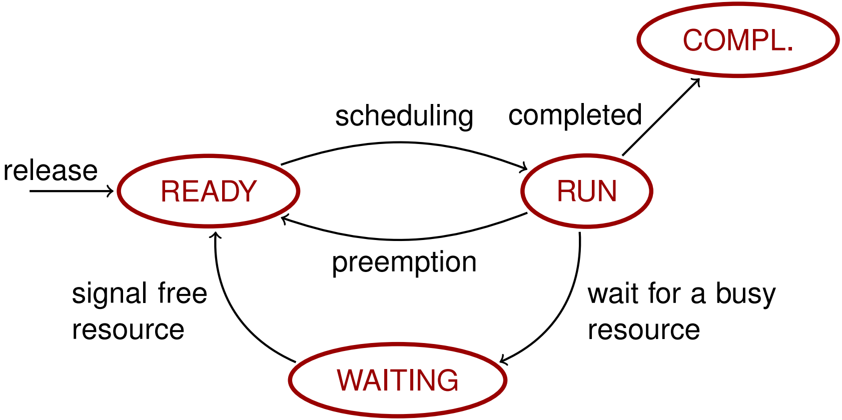
- A job executes on a *processor*

  CPU, transmission link in a network, database server, etc.

- A job may use some (shared) passive *resources*

  file, database lock, shared variable etc.

Life Cycle of a Job



Jobs – Parameters

We consider finite, or countably infinte number of jobs $J_1, J_2, \ldots$

Each job has several parameters.

There are four types of job parameters:

- temporal
  - release time, execution time, deadlines
- functional
  - Laxity type: hard and soft real-time
  - preemptability, (criticality)
- interconnection
  - precedence constraints
- resource
  - usage of processors and passive resources

Job Parameters – Execution Time

Execution time e_i of a job J_i – the amount of time required to complete the execution of J_i when it executes alone and has all necessary resources

- Value of e_i depends upon complexity of the job and speed of the processor on which it executes; may change for various reasons:
  - Conditional branches
  - Caches, pipelines, etc.
  - ...
- **Execution times fall into an interval** $[e_i^-, e_i^+]$; we assume that we know this interval (WCET analysis) but not necessarily e_i

We usually validate the system using only e_i^+ for each job
i.e. assume $e_i = e_i^+$

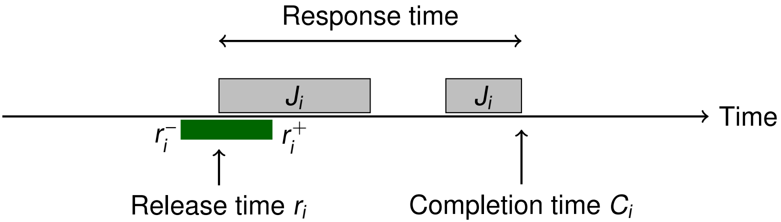
Job Parameters – Release and Response Time

Release time r_i – the instant in time when a job J_i becomes available for execution

- Release time may *jitter*, only an interval $[r_i^-, r_i^+]$ is known
- A job can be executed at any time at, or after, its release time, provided its processor and resource demands are met

Completion time C_i – the instant in time when a job completes its execution

Response time – the difference $C_i - r_i$ between the completion time and the release time

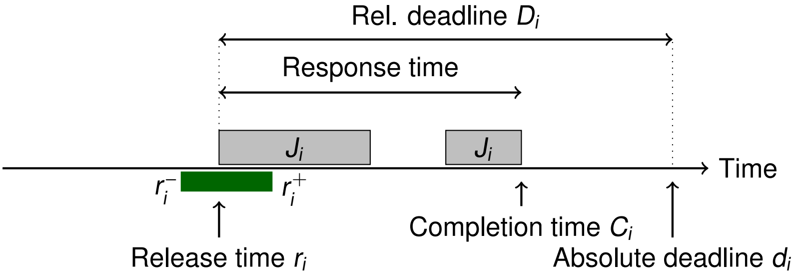


Job Parameters – Deadlines

Absolute deadline d_i – the instant in time by which a job must be completed

Relative deadline D_i – the maximum allowable response time i.e. $D_i = d_i - r_i$

Feasible interval is the interval $(r_i, d_i]$



A *timing constraint* of a job is specified using release time together with relative and absolute deadlines.

Laxity Type – Hard Real-Time

A **hard real-time constraint** specifies that a job should never miss its deadline.

Examples: Flight control, railway signaling, anti-lock brakes, etc.

Several more precise definitions occur in literature:

- A timing constraint is hard if the failure to meet it is considered a fatal error

  e.g. a bomb is dropped too late and hits civilians

- A timing constraint is hard if the usefulness of the results falls off abruptly (may even become negative) at the deadline

  Here the nature of abruptness allows to soften the constraint

Definition 1

A *timing constraint is hard* if the user requires *formal validation* that the job meets its timing constraint.

Laxity Type – Soft Real-Time

A **soft real-time constraint** specifies that a job could occasionally miss its deadline

Examples: stock trading, multimedia, etc.

Several more precise definitions occur in literature:

- A timing constraint is soft if the failure to meet it is undesirable but acceptable if the probability is low
- A timing constraint is soft if the usefulness of the results decreases at a slower rate with *tardiness* of the job

  e.g. the probability that a response time exceeds 50 ms is less than 0.2

Definition 2

A *timing constraint is soft* if either validation is not required, or only a demonstration that a *statistical constraint* is met suffices.

Jobs – Preemptability

Jobs may be interrupted by higher priority jobs

- A job is *preemptable* if its execution can be interrupted
- A job is *non-preemptable* if it must run to completion once started

  (Some preemptable jobs have periods during which they cannot be preempted)
- The *context switch time* is the time to switch between jobs

  (Most of the time we assume that this time is negligible)

Reasons for preemptability:

- Jobs may have different levels of criticality

  e.g. brakes vs radio tunning
- Priorities may make part of scheduling algorithm

  e.g. resource access control algorithms

Jobs – Precedence Constraints

Jobs may be constrained to execute in a particular order

- This is known as a *precedence constraint*
- A job J_i is a *predecessor* of another job J_k and J_k a *successor* of J_i (denoted by $J_i < J_k$) if J_k cannot begin execution until the execution of J_i completes
- J_i is an *immediate predecessor* of J_k if $J_i < J_k$ and there is no other job J_j such that $J_i < J_j < J_k$
- J_i and J_k are *independent* when neither $J_i < J_k$ nor $J_k < J_i$

A job with a precedence constraint becomes ready for execution when its release time has passed and when all predecessors have completed.

Example: authentication before retrieving an information, a signal processing task in radar surveillance system precedes a tracker task

Tasks – Modeling Reactive Systems

Reactive systems – run for unlimited amount of time

A system parameter: number of tasks

- may be known in advance (flight control)
- may change during computation (air traffic control)

We consider three types of tasks

- Periodic – jobs executed at regular intervals, hard deadlines
- Aperiodic – jobs executed in random intervals, soft deadlines
- Sporadic – jobs executed in random intervals, hard deadlines

... precise definitions later.

Processors

A processor, P, is an *active* component on which jobs are scheduled

The general case considered in literature:

m processors $P_1, \ldots, P_m$, each P_i has its *type* and *speed*.

We mostly concentrate on *single processor* scheduling

- Efficient scheduling algorithms
- In a sense subsumes multiprocessor scheduling where tasks are assigned *statically* to individual processors

  i.e. all jobs of every task are assigned to a single processor

Multi-processor scheduling is a rich area of current research, we touch it only lightly (later).

Resources

A resource, R, is a *passive* entity upon which jobs may depend

In general, **we consider n resources $R_1, \ldots, R_n$ of distinct types**

Each R_i is used in a mutually exclusive manner

- A job that acquires a free resource locks the resource
- Jobs that need a busy resource have to wait until the resource is released
- Once released, the resource may be used by another job

  (i.e. it is not consumed)

(More generally, each resource may be used by k jobs concurrently, i.e., there are k units of the resource)

Resource requirements of a job specify

- which resources are used by the job
- the time interval(s) during which each resource is required

  (precise definitions later)

Scheduling

Schedule assigns, in every time instant, processors and resources to jobs.

More formally, a schedule is a function

$$\sigma : \{J_1, \ldots\} \times \mathbb{R}_0^+ \to \mathcal{P}(\{P_1, \ldots, P_m, R_1, \ldots, R_n\})$$

so that for every $t \in \mathbb{R}_0^+$ there are rational $0 \leq t_1 \leq t \leq t_2$ such that $\sigma(J_i, \cdot)$ is constant on $[t_1, t_2)$.

(We also assume that there is the least time quantum in which scheduler does not change its decisions, i.e. each of the intervals $[t_1, t_2)$ is larger than a fixed $\varepsilon > 0$.)

Valid and Feasible Schedule

A schedule is *valid* if it satisfies the following conditions:

- Every processor is assigned to at most one job at any time
- Every job is assigned to at most one processor at any time
- No job is scheduled before its release time
- The total amount of processor time assigned to a given job is equal to its actual execution time
- *All the precedence and resource usage constraints are satisfied*

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

A set of jobs is *schedulable* if there is a feasible schedule for the set.

Scheduling – Algorithms

Scheduling algorithm computes a schedule for a set of jobs

A set of jobs is *schedulable according to a scheduling algorithm* if the algorithm produces a feasible schedule

Sometimes efficiency of scheduling algorithms is measured using a *cost function*:

- the maximum/average response time
- the maximum/average lateness – the difference between the completion time and the absolute deadline, i.e. $C_i - d_i$
- miss rate – the percentage of jobs that are executed but completed too late
- ...

Definition 3

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.

Real-Time Scheduling

Individual Jobs

Scheduling of Individual Jobs

We start with scheduling of finite sets of jobs $\{J_1, \ldots, J_m\}$ for execution on **single processor** systems

Each J_i has a release time r_i, an execution time e_i and a relative deadline D_i.

We assume hard real-time constraints

The question: Is there an optimal scheduling algorithm?

We proceed in the direction of growing generality:

1. No resources, independent, synchronized (i.e. $r_i = 0$ for all i)
2. No resources, independent but not synchronized
3. No resources but possibly dependent
4. The general case

No resources, Independent, Synchronized

	J_1	J_2	J_3	J_4	J_5
e_i	1	1	1	3	2
d_i	3	10	7	8	5

Is there a feasible schedule? Minimize maximal lateness.

Note: Preemption does not help in synchronized case

Theorem 4

If there are no resource contentions, then executing independent jobs in the order of non-decreasing deadline (EDD) produces a feasible schedule (if it exists) and minimizes the maximal lateness (always).

Proof.

Any feasible schedule σ can be transformed in finitely many steps to EDD schedule whose lateness is $\leq$ the lateness of σ (whiteboard). □

Is there any simple schedulability test?

$\{J_1, \ldots, J_n\}$ where $d_1 \leq \cdots \leq d_n$ is schedulable iff
$\forall i \in \{1, \ldots, n\} \;:\; \sum_{k=1}^{i} e_k \leq d_i$

No resources, Independent (No Synchro)

	J_1	J_2	J_3
r_i	0	0	2
e_i	1	2	2
d_i	2	5	4

- find a (feasible) schedule (with and without preemption)
- determine response time of each job in your schedule
- determine lateness of each job in your schedule (is the maximal lateness minimized?)

  Recall that lateness of J_i is equal to $C_i - d_i$

Preemption makes a difference

No resources, Independent (No Synchro)

Earliest Deadline First (EDF) scheduling:
At any time instant, a job with the earliest absolute deadline is executed

Here EDF works in the preemptive case but not in the non-preemptive one.

	J_1	J_2
r_i	0	1
e_i	4	2
d_i	7	5

No Resources, Dependent (No Synchro)

Theorem 5

If there are no resource contentions, jobs are independent and preemption is allowed, the EDF algorithm finds a feasible schedule (if it exists) and minimizes the maximal lateness.

Proof.

Any feasible schedule σ can be transformed in finitely many steps to EDF schedule which is feasible and whose lateness is $\leq$ the lateness of σ (whiteboard). $\square$

No resources, Independent (No Synchro)

The **non-preemptive** case is NP-hard.

Heuristics are needed, such as the **Spring algorithm**, that usually work in much more general setting (with resources etc.)

Use the notion of *partial schedule* where only a subset of tasks has been scheduled.

Exhaustive search through partial schedules

- start with an empty schedule
- in every step either
  - add a job which maximizes a *heuristic function H* among jobs that have not yet been tried in this partial schedule
  - or backtrack if there is no such a job
- After failure, backtrack to previous partial schedule

Heuristic function identifies plausible jobs to be scheduled (earliest release, earliest deadline, etc.)

No resources, Dependent (No Synchro)

Theorem 6

Assume that there are no resource contentions and jobs are preemptable. There is a polynomial time algorithm which decides whether a feasible schedule exists and if yes, then computes one.

Idea: Reduce to independent jobs by changing release times and deadlines. Then use EDF.

Observe that if $J_i < J_k$ then replacing

- r_k with $\max\{r_k, r_i + e_i\}$
- d_i with $\min\{d_i, d_k - e_k\}$

does not change feasibility.

Replace systematically according to the precedence relation.

No Resources, Dependent (No Synchro)

Replace r_k and d_k systematically as follows:

- Pick J_k whose all predecessors have been processed and replace r_k with $\max\{r_k, \max_{J_i < J_k} r_i + e_i\}$. Repeat for all jobs.
- Pick J_k whose all successors have been processed and replace d_k with $\min\{d_k, \min_{J_k < J_i} d_i - e_i\}$. Repeat for all jobs.

This gives a new set of jobs $J_1^*, \ldots, J_m^*$ where J_k^* has the release time r_k^* and the absolute deadline d_k^*.

We impose **no precedence constraints** on $J_1^*, \ldots, J_m^*$.

Lemma 7

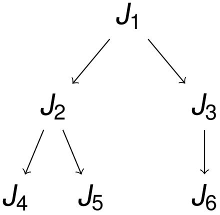
$\{J_1, \ldots, J_m\}$ is feasible iff $\{J_1^*, \ldots, J_m^*\}$ is feasible. If EDF schedule is feasible on $\{J_1^*, \ldots, J_m^*\}$, then the same schedule is feasible on $\{J_1, \ldots, J_m\}$.

The same schedule means that whenever J_i^* is scheduled at time k, then J_i is scheduled at time k.

No Resources, Dependent (No Synchro)

	J_1	J_2	J_3	J_4	J_5	J_6
e_i	1	1	1	1	1	1
d_i	2	5	4	3	5	6

Dependencies:



Find a feasible schedule.

Resources, Dependent, Not Synchronized

Even the preemptive case is NP-hard

- reduce the non-preemptive case without resources to the preemptive with resources
- Use a common resource *R*, which every job acquires when its execution starts and releases *R* when execution is complete – causes no preemption

Could be solved using heuristics, e.g. the Spring algorithm.