Real-Time Scheduling

Multiprocessor Real-Time Systems

Multiprocessor Real-time Systems

- Many embedded systems are composed of many processors (control systems in cars, aircraft, industrial systems etc.)
- Today most processors in computers have multiple cores

  The main reason is that increasing frequency of a single processor is no more feasible (mostly due to power consumption problems, growing leakage currents, memory problems etc.)

Applications must be developed specifically for multiprocessor systems.

Multiprocessor Frustration

In case of real-time systems, multiple processors bring serious difficulties concerning correctness, predictability and efficiency.

The “root of all evil” in global scheduling: (Liu, 1969)

Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors.

The Model

- A *job* is a unit of work that is scheduled and executed by a system
  (Characterised by the release time r_i, execution time e_i and deadline d_i)
- A *task* is a set of related jobs which jointly provide some system function
- Jobs execute on *processors*

In this lecture we consider m processors

- Jobs may use some (shared) passive *resources*

Schedule

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

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.

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)

We also consider *online* scheduling algorithms that do not use any knowlede about jobs that will be released in the future but are given a complete information about jobs that have been released.
(e.g. EDF is online)

Multiprocessor Taxonomy

- Identical processors: All processors identical, have the same computing power
- Uniform processors: Each processor is characterized by its own computing capacity κ, completes κt units of execution after t time units
- Unrelated processors: There is an execution rate ρ_{ij} associated with each job-processor pair (J_i, P_j) so that J_i completes $\rho_{ij} t$ units of execution by executing on P_j for t time units

In addition, cost of communication can be included etc.

Assumptions – Priority Driven Scheduling

Throughout this lecture we assume:

- Unless otherwise stated, consider *m identical* processors
- Jobs can be preempted at any time and never suspend themselves
- Context switch overhead is negligibly small

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

- For simplicity, we assume *independent* jobs that do not contend for resources

Unless otherwise stated, we assume that scheduling decisions take place only when a job is released, or completed.

Multiprocessor Scheduling Taxonomy

Multiprocessor scheduling attempts to solve two problems:

- the *allocation problem*, i.e., on which processor a given job executes
- the *priority problem*, i.e., when and in what order the jobs execute

Issues

What results from single processor scheduling remain valid in multiprocessor setting?

- Are there simple optimal scheduling algorithms?
- Are there optimal *online* scheduling algorithms (i.e. those that do not know what jobs come in future)
- Are there efficient tests for schedulability?

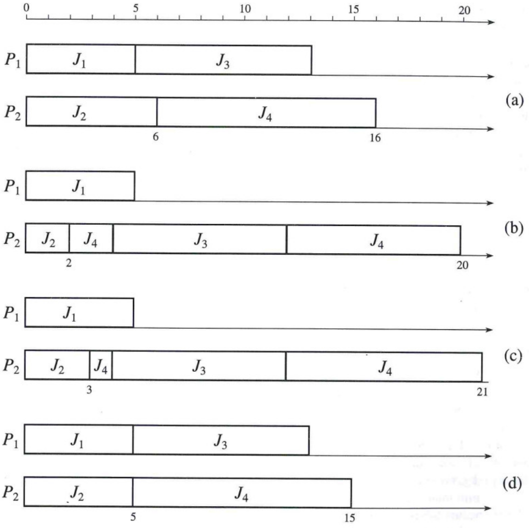
In this lecture we consider:

- Individual jobs
- Periodic tasks

Start with n individual jobs $\{J_1, \ldots, J_n\}$

Individual Jobs – Timing Anomalies

Priority order: $J_1 \sqsupset \cdots \sqsupset J_4$



Individual Jobs – EDF

EDF on m identical processors: At any time instant, jobs with the earliest absolute deadlines are executed on available processors.
(Recall: no job can be executed on more than one processor at a given time!)

Is this optimal? NO!

Example:
J_1, J_2, J_3 where

- $r_i = 0$ for $i \in \{1, 2, 3\}$
- $e_1 = e_2 = 1$ and $e_3 = 5$
- $d_1 = 1$, $d_2 = 2$, $d_3 = 5$

Individual Jobs – Online Scheduling

Theorem 33

No optimal **on-line** *scheduler can exist for a set of jobs with two or more distinct deadlines on any $m > 1$ processor system.*

Proof.

Assume $m = 2$ and consider three jobs J_1, J_2, J_3 are released at time 0 with the following parameters:

- $e_1 = e_2 = 2$ and $e_3 = 4$
- $d_1 = d_2 = 4$ and $d_3 = 8$

Depending on scheduling in $[0, 2]$, new tasks T_4, T_5 are released at 2:

- If J_3 is executed in $[0, 2]$, then at 2 release J_4, J_5 with $d_4 = d_5 = 4$ and $e_4 = e_5 = 2$.
- If J_3 is not executed in $[0, 2]$, then at 4 release J_4, J_5 with $d_4 = d_5 = 8$ and $e_4 = e_5 = 4$.

In either case the schedule produced is not feasible. However, if the scheduler is given either of the sets $\{J_1, \ldots, J_5\}$ at the beginning, then there is a feasible schedule. □

Individual Jobs – Speedup Helps(?)

Theorem 34

If a set of jobs is feasible on m identical processors, then the same set of jobs will be scheduled to meet all deadlines by EDF on identical processors in which the individual processors are $(2 - \frac{1}{m})$ *times as fast as in the original system.*

The result is tight for EDF (assuming dynamic job priority):

Theorem 35

There are sets of jobs that can be feasibly scheduled on m identical processors but EDF cannot schedule them on m processors that are only $(2 - \frac{1}{m} - \varepsilon)$ *faster for every* $\varepsilon > 0$.

... there are also general lower bounds for online algorithms:

Theorem 36

There are sets of jobs that can be feasibly scheduled on m (here m is even) identical processors but **no online** *algorithm can schedule them on m processors that are only* $(1 + \varepsilon)$ *faster for every* $\varepsilon < \frac{1}{5}$.

[Optimal Time-Critical Scheduling Via Resource Augmentation, Phillips et al, STOC 1997]

Reactive Systems

Consider fixed number, n, of *independent periodic* tasks
$\mathcal{T} = \{T_1, \ldots, T_n\}$
i.e. there is no dependency relation among jobs

- Unless otherwise stated, assume no phase and deadlines equal to periods
- Ignore aperiodic tasks
- No sporadic tasks unless otherwise stated

Utilization u_i of a periodic task T_i with period p_i and execution time e_i is defined by $u_i := e_i/p_i$
u_i is the fraction of time a periodic task with period p_i and execution time e_i keeps a processor busy

Total utilization $U^{\mathcal{T}}$ of a set of tasks $\mathcal{T} = \{T_1, \ldots, T_n\}$ is defined as the sum of utilizations of all tasks of $\mathcal{T}$, i.e. by $U^{\mathcal{T}} := \sum_{i=1}^{n} u_i$

Given a scheduling algorithm *ALG*, the schedulable utilization U_{ALG} of *ALG* is the maximum number U such that for all $\mathcal{T}$: $U_{\mathcal{T}} \leq U$ implies $\mathcal{T}$ is schedulable by *ALG*.

Multiprocessor Scheduling Taxonomy

Allocation (migration type)

- No migration: each **task** is allocated to a processor
- (Task-level migration: **jobs** of a task may execute on different processors; however, each job is assigned to a single processor)
- Job-level migration: A single job can migrate and execute on different processors

  (however, parallel execution of one job is not permitted and migration takes place only when the job is rescheduled)

Priority type

- Fixed task-level priority (e.g. EDF)
- Fixed job-level priority (e.g. RM)
- (Dynamic job-level priority)

Partitioned scheduling = No migration

Global scheduling = job-level migration

Fundamental Limit – Fixed Job-Level Priority

Consider m processors and $m+1$ tasks $\mathcal{T} = \{T_1, \ldots, T_{m+1}\}$, each $T_i = (L, 2L-1)$.

Then $U_{\mathcal{T}} = \sum_{i=1}^{m+1} L/(2L-1) = (m+1)\,(L/(2L-1))$

For very large L, this number is close to $(m+1)/2$.

The set $\mathcal{T}$ is not schedulable using any *fixed job-level* priority algorithm.

In other words, the schedulable utilization of fixed job-level priority algorithms is at most $(m+1)/2$, i.e., half of the processors capacity.

There are variants of EDF achieving this bound (see later slides).

Partitioned vs Global Scheduling

Most algorithms up to the end of 1990s based on *partitioned scheduling*

- no migration

From the end of 1990s, many results concerning *global scheduling*

- job-level migration

The task-level migration has not been much studied, so it is not covered in this lecture.

We consider fixed job-level priority (e.g. EDF) and fixed task-level priority (e.g. RM).

As before, we ignore dynamic job-level priority.

Partitioned Scheduling & Fixed Job-Level Priority

The algorithm proceeds in two phases:

1. Allocate tasks to processors, i.e., partition the set of tasks into m possibly empty *modules* $M_1, \dots, M_m$
2. Schedule tasks of each M_i on the processor i according to a given single processor algorithm

The quality of task assignment is determined by the number of assigned processors

- Use EDF to schedule modules
- Suffices to test whether the total utilization of each module is ≤ 1
  (or, possibly, $\leq \hat{U}$ where $\hat{U} < 1$ in order to accomodate aperiodic jobs ...)

Finding an optimal schedule is equivalent to a simple *uniform-size bin-packing problem* (and hence is NP-complete)

Similarly, we may use RM for fixed task-level priorities (total utilization in modules $\leq \log 2$, etc.)

Partitioned Scheduling & Fixed Job-Level Priority

Assume that tasks are assigned to modules using the First Fit (FF) algorithm and that EDF is used in modules (the algorithm EDF-FF).

Theorem 37

Given a set of tasks $\mathcal{T}$, denote by β the number $\lfloor 1/\max_i u_i \rfloor$ where $\max_i u_i$ is the maximum utilization of tasks in $\mathcal{T}$.

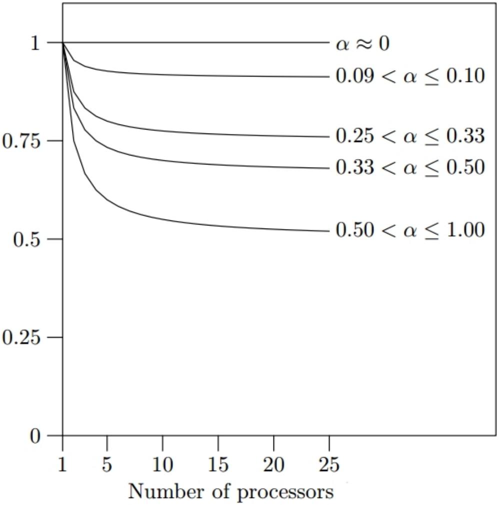
1. *Assume $n > \beta m$. If $U_{\mathcal{T}} \leq \frac{\beta m+1}{\beta+1}$, then $\mathcal{T}$ is schedulable using any EDF-FF algorithm.*
2. *For every $\varepsilon > 0$ there is a set of $n > \beta m$ tasks $\mathcal{T}$ such that $U_{\mathcal{T}} = \frac{\beta m+1}{\beta+1} + \varepsilon$ and $\mathcal{T}$ is not schedulable by any EDF-FF.*

The theorem holds also for other allocation heuristics (+ EDF) such as First Fit Ordered, Best Fit, Best Fit Ordered.

No *reasonable* allocation algorithm can give a scheduling algorithm with better schedulable utilization than $\frac{\beta m+1}{\beta+1}$.

There is an analogous result (with different bounds) for fixed task-level priority systems, where RM-FF is used.

Partitioned Scheduling – EDF-FF



The value $\left(\frac{\beta m+1}{\beta+1} \,/\, m\right)$ (vertical axis) w.r.t. the number of processors m (horizontal axis), here $\alpha = \max_i u_i$ is the maximum utilization

Global Scheduling – Fixed Job-Level Priority

Dhall's effect:

- Consider $m > 1$ processors
- Let $\varepsilon > 0$
- Consider a set of tasks $\mathcal{T} = \{T_1, \ldots, T_m, T_{m+1}\}$ such that
  - $T_i = (2\varepsilon, 1)$ for $1 \leq i \leq m$
  - $T_{m+1} = (1, 1 + \varepsilon)$
- $\mathcal{T}$ is schedulable
- Stadnard EDF and RM schedules are not feasible (whiteb.)

However,

$$U_{\mathcal{T}} \quad = \quad m\frac{2\varepsilon}{1} + \frac{1}{1+\varepsilon}$$

which means that for small ε the utilization $U_{\mathcal{T}}$ is close to 1
(i.e., $U_{\mathcal{T}}/m$ is very small for $m >> 0$ processors)

How to avoid Dhall's effect?

- Note that RM and EDF only account for task periods and ignore the execution time!
- (Partial) Solution: Dhall's effect can be avoided by giving high priority to tasks with high utilization

  Then in the previous example, T_{m+1} is executed whenever it comes and the other tasks are assigned to the remaining processors – produces a feasible schedule

Global Scheduling – Fixed Job-Level Priority

Theorem 38

A set of periodic tasks $\mathcal{T}$ with deadlines equal to periods can be EDF-scheduled upon m unit-speed identical processors, provided its cumulative utilization is bounded from above as follows:

$$U_{\mathcal{T}} \leq m - (m-1)\max_i u_i$$

(This holds also for systems with relative deadlines bounded by periods – just substitute utilizations with densities e_i/D_i.)

The above bound on EDF is tight:

Theorem 39

Let $m > 1$. For every $0 < u_{\max} < 1$ and small $0 < \varepsilon << u_{\max}$ there is a set of tasks $\mathcal{T}$ such that

- *maximum utilization in $\mathcal{T}$ is $u_{\max}$,*
- $U_{\mathcal{T}} = U_{\mathcal{T}} \leq m - (m-1)u_{\max} + \varepsilon$,
- *$\mathcal{T}$ is not schedulable by EDF.*

[Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors, Goossens et al, R-T S., 2003]

Global Scheduling – Fixed Job-Level Priority

Apparently there is a problem with long jobs due to Dhall's effect.

There is an improved version of EDF called EDF-US(1/2) which

- assigns the highest priority to tasks with $u_i \geq 1/2$
- assigns priorities to the rest according to deadlines

which reaches the generic schedulable utilization bound $(m + 1)/2$.

Global Scheduling – Fixed Task-Level Priority

RM algorithm – always execute the jobs with highest rate

Lemma 40

If $u_i \leq m/(3m-2)$ for all $1 \leq i \leq n$ and $U_{\mathcal{T}} \leq m^2/(3m-2)$, then $\mathcal{T}$ is schedulabe by RM.

There is a problem with long jobs due to Dhall's effect.

Solution: Deal with long jobs separately which gives RM-US:

- Assign the same maximum priority to all T_i with $u_i > m/(3m-2)$, break ties arbitrarily.
- If $u_i \leq m/(3m-2)$ assign rate-monotonic priority.

Theorem 41

If $U_{\mathcal{T}} \leq m^2/(3m-2)$, then $\mathcal{T}$ is schedulabe by RM-US.

Note that for large m this bound is close to $m/3$ (i.e., the utilization is 33%).

Partitioned vs Global

Advantages of the global scheduling:

- Load is automatically balanced
- Better average response time (follows from queueing theory)

Disadvantages of the global scheduling:

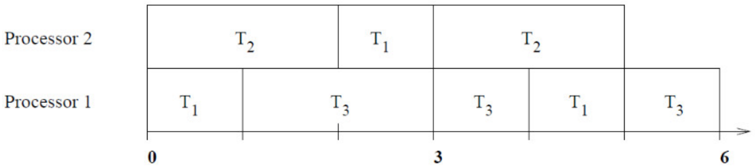
- Problems caused by migration (e.g. increased cache misses)
- Schedulability tests more difficult (active area of research)

Is either of the approaches better from the schedulability standpoint?

Global Beats Partitioned

There are sets of tasks schedulable only with global scheduler:

- $\mathcal{T} = \{T_1, T_2, T_3\}$ where $T_1 = (1,2), T_2 = (2,3), T_3 = (2,3)$, can be scheduled using a global scheduler:

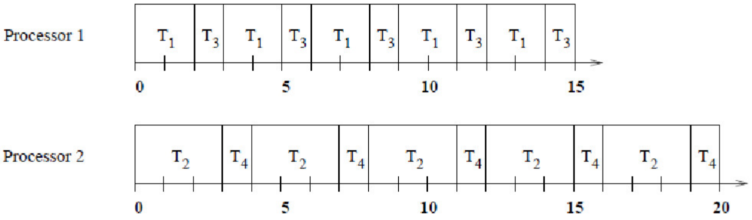


- No feasible partitioned schedule exists, always at least one processor gets tasks with total utilization higher than 1.

Partitioned Beats Global

There are task sets that can be scheduled only with partitioned scheduler (assuming fixed task-level priority assignment):

- $\mathcal{T} = \{T_1, \ldots, T_4\}$ where $T_1 = (2, 3)$, $T_2 = (3, 4)$, $T_3 = (5, 15)$, $T_4 = (5, 20)$, can be scheduled using a fixed task-level priority partitioned schedule:



- Global scheduling (fixed job-level priority): There are 9 jobs released in the interval $[0, 12)$. Any of the 9! possible priority assignments leads to a deadline miss.

Optimal Algorithm?

There IS an optimal algorithm in the case of job-level migration & dynamic job-level priority. However, the algorithm is *time driven*.

The *priority fair* (PFair) algorithm is optimal for periodic systems with deadlines equal to periods

Idea (of PFair): In any interval $(0, t]$ jobs of a task T_i with utilization u_i execute for amount of time W so that $u_i t - 1 < W < u_i t + 1$
(Here every parameter is assumed to be a natural number)

This is achieved by cutting time into small quanta and scheduling jobs in these quanta so that the execution times are always (more or less) in proportion.

There are other optimal algorithms, all of them suffer from a large number of preemptions/migrations.

No optimal algorithms are known for more general settings: deadlines bounded by periods, arbitrary deadlines.

Recall, that no optimal *on-line* scheduling possible