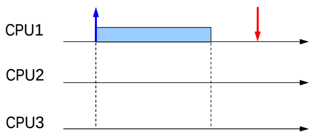
Real-time scheduling for multiprocessor systems

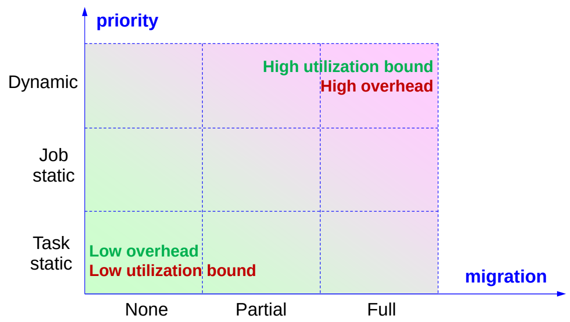
MP scheduling is difficult

"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" [Liu 1969]



Classification

Multiprocessor scheduling algorithms can be classified according to two orthogonal criteria:



Classification (by migration)

Algorithms can be distinguished by migration constraints:

- **No migration**
  Tasks are statically allocated to processors and never migrate (**Partitioned scheduling**).
- **Partial migration**
  Tasks can only perform a limited number of migrations or can migrate on a subset of processors (**Semi-partitioned scheduling**).
- **Full migration**
  Tasks are dynamically allocated to processors and can migrate at any time on any processor (**Global scheduling**).

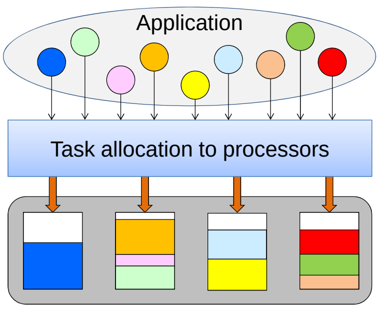
Classification (by priority)

Algorithms can be also distinguished by the way priorities are assigned to tasks:

- **Fixed**
  priority is statically assigned to tasks and is fixed for all the jobs of a task (e.g., Rate Monotonic, Deadline Monotonic).
- **Job-static**
  different jobs can have different priority, which is fixed for the entire job execution (e.g., EDF).
- **Dynamic**
  priority can change during job execution (e.g., Least Laxity First).

Partitioned Scheduling

Once tasks are allocated to processors, they can be handled by uniprocessor scheduling algorithms:



Partitioned Scheduling

Partitioned scheduling reduces to:

Bin Packing + **Uniprocessor scheduling**

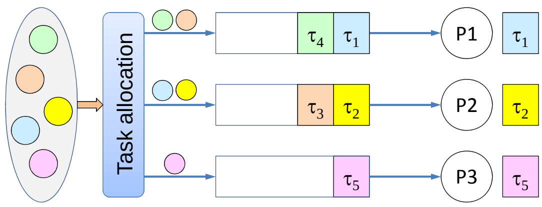
Bin Packing: NP-hard in the strong sense → Various heuristics used: FF, NF, BF, FFDU, BFDD, etc.

Uniprocessor scheduling: Well known

Since migration is forbidden, processors may be underutilized.

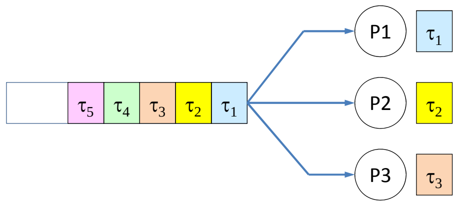
Partitioned Scheduling

- Each processor manages its own ready queue
- The processor for each task is determined off-line
- The processor cannot be changed at run time



Global scheduling

- The system manages a single queue of ready tasks
- The processor is determined at run time
- During execution a task can migrate to another processor



Global scheduling

Example (Global Rate Monotonic)

- Consider the following task set:

	C_i	T_i
τ_1	3	6
τ_2	7	10
τ_3	8	12
τ_4	6	15
τ_5	3	18

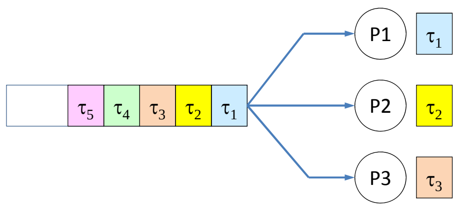
- The task set has to be scheduled on 3 identical processors (m = 3)
- Priority are assigned according to Rate Monotonic

→ $P_1 > P_2 > P_3 > P_4 > P_5$

Global scheduling

Work conserving scheduler

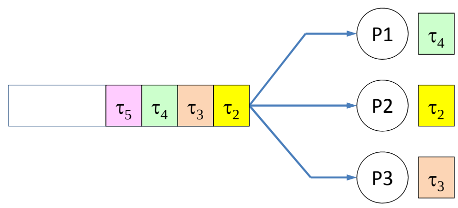
- The m highest priority tasks are always those executing.
- No processor is ever idle when a task is ready to execute.



Global scheduling

Example (Global-RM)

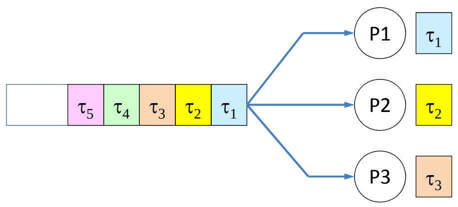
When a task finishes its execution (e.g., τ_1), the next one in the queue (τ_4) is scheduled on the available CPU:



Global scheduling

Example (Global-RM)

When a higher priority task arrives (e.g., τ_1), it preempts the task with lowest priority among the executing ones (τ_4):

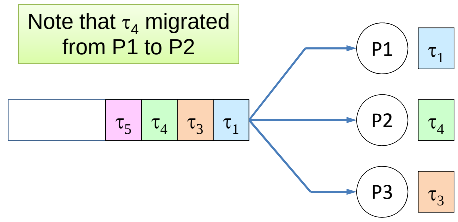


Global scheduling

Example (Global-RM)

When another task ends its execution (e.g., τ_2), the preempted task (τ_4) can resume its execution.

Note that τ_4 migrated from P1 to P2

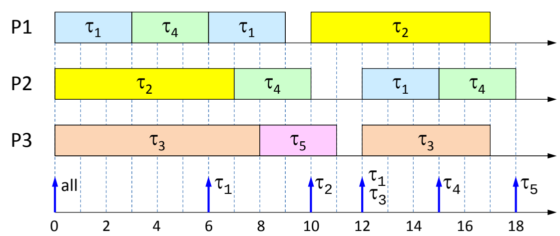


Global scheduling

Processor-level representation

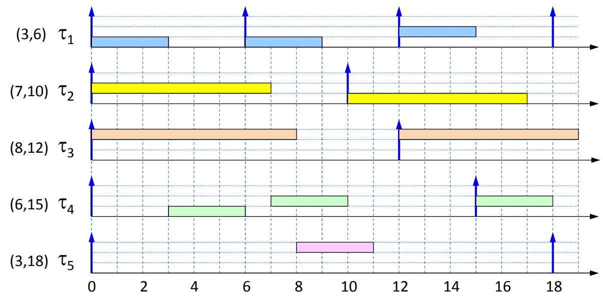
τ_1 (3, 6) τ_2 (7, 10) τ_3 (8, 12)

τ_4 (6, 15) τ_5 (3, 18)



Global scheduling

Task-level representation



Hybrid approaches

Different restrictions can be imposed on task migration:

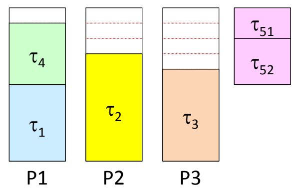
- **Job migration**
  Tasks are allowed to migrate, but only at jobs boundaries.
- **Semi-partitioned scheduling**
  Some tasks are statically allocated to processors, others are split into chunks (subtasks) that are allocated to different processors.
- **Clustered scheduling**
  A task can only migrate within a predefined subset of processors (cluster).

Semi-partitioned scheduling

- Tasks are statically allocated to processors, if possible.
- Remaining tasks are split into chunks (subtasks), which are allocated to different processors.

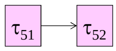
	C_i	T_i	u_i
τ_1	3	6	0.5
τ_2	7	10	0.7
τ_3	9	15	0.6
τ_4	8	20	0.4
τ_5	15	30	0.5

U = 2.7

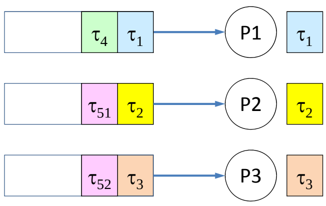


Semi-partitioned scheduling

- Note that subtasks are not independents, but are subject to a precedence constraint:

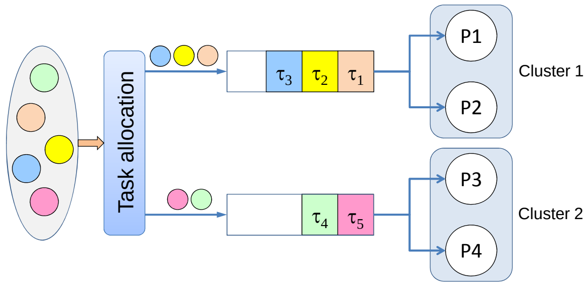


This precedence must be managed!



Clustered scheduling

- A task can only migrate within a predefined subset of processors (cluster).



Schedulability bound

Given a set Γ of *n* periodic tasks with total utilization U to be scheduled by an algorithms A on a set of *m* identical processors, find a bound $U_A(n,m)$ such that,

if $U \leq U_A(n,m)$, then Γ is schedulable by A.

A necessary condition

A task set can be schedulable only if $U \leq m$.

In fact, it is clear that if U > m, the total demand in the hyperperiod H will certainly exceed the total available time (that is UH > mH), hence some task will miss its deadline.

An algorithm A is optimal in the sense of schedulability iff $U_A(n,m) = m$.

A negative result

The schedulability bound of global-EDF and global-RM is equal to 1, independently of the number m of available processors.

This means that given a platform of m identical processors, there exist applications with U > 1 that are not schedulable by global-EDF and global-RM.

To prove this result it suffices to identify an application Γ with utilization U = 1+ε (ε is a constant arbitrarily small) that is not schedulable by global-EDF and global-RM.

Dhall's effect

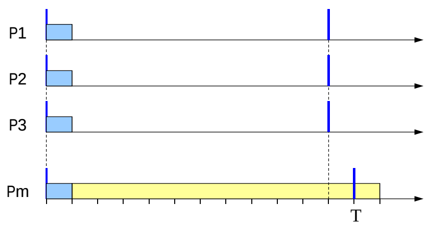
m processors

m+1 tasks

global schedule

	C_i	T_i	U_i
τ_1	1	T–1	ε
τ_2	1	T–1	ε
⋮			
τ_m	1	T–1	ε
τ_{m+1}	T	T	1

EDF and RM produce an unfeasible schedule with a total utilization arbitrarily close to 1



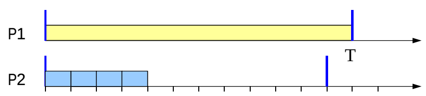
Partitioned

m processors

m+1 tasks

	C_i	T_i	U_i
τ_1	1	T–1	ε
τ_2	1	T–1	ε
⋮			
τ_m	1	T–1	ε
τ_{m+1}	T	T	1

Note that a feasible partitioned schedule exists on just 2 processors



Dhall's effect implications

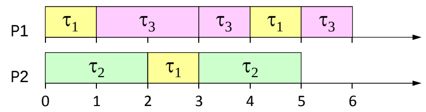
- Dhall's Effect shows the limitation of global EDF and RM: both utilization bounds tend to 1, independently of the value of m.
- Researchers lost interest in global scheduling for ~25 years, since late 1990s.
- Such a limitation is related to EDF and RM, not to global scheduling in general.

Global vs. partitioned

On the other hand, there are task sets that are schedulable only with a **global** scheduler.

Example:

	C_i	T_i
τ_1	1	2
τ_2	2	3
τ_3	2	3

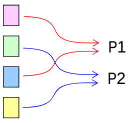


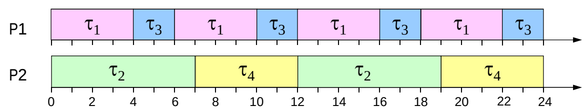
Global vs. partitioned

But there are also task sets that are schedulable only with a **partitioned** scheduler.

Example:

	C_i	T_i
τ_1	4	6
τ_2	7	12
τ_3	4	12
τ_4	10	24



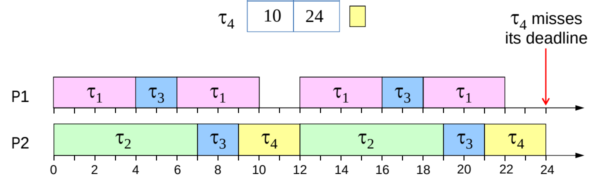


All 4! = 24 global priority assignments lead to deadline miss.

Global vs. partitioned

Example of unfeasible schedule with priorities: $P_1 > P_2 > P_3 > P_4$

	C_i	T_i
τ_1	4	6
τ_2	7	12
τ_3	4	12
τ_4	10	24



Global scheduling: pros & cons

- ✓ Automatic load balance among processors
- ✓ Can better manage dynamic workloads
- ✓ Lower *average* response time (see queueing theory)
- ✓ More efficient reclaiming of unused processors
- ✓ More efficient overload management
- ✓ Lower number of preemptions

- ✘ High migration cost: can be mitigated by proper HW (e.g., MPCore's Direct Data Intervention)
- ✘ Less schedulability results → Further research needed

Evaluation metrics

- Percentage of schedulable task sets
  - Over a randomly generated load
  - Depends on the task generation method
- Processor speedup factor S

  An algorithm A has a speedup factor S if any task set feasible on a given platform can be scheduled by A on a platform in which all processors are S times faster.
- Run-time complexity
- Sustainability and predictability properties

  Schedulability is preserved for more relaxed constraints

Sustainability

A scheduling algorithm is **sustainable** iff schedulability of a task set is preserved when

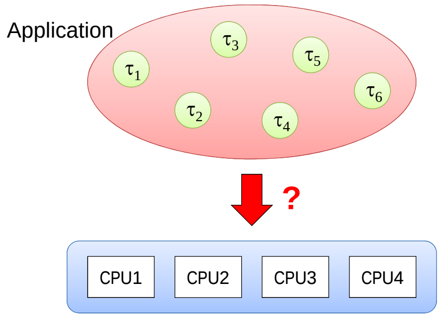
1. decreasing execution requirements
2. increasing periods of inter-arrival times
3. increasing relative deadlines

- Baker and Baruah [ECRTS, 2009] showed that: Global EDF for sporadic tasks is sustainable with respect to points 1 and 2.

Task Allocation Algorithms

Task allocation

How tasks are executed on the various processors?

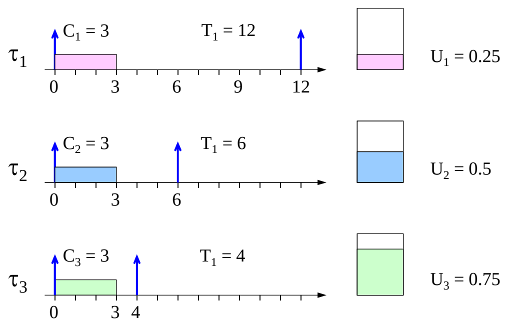


Task allocation

- **Static partitioning**
  The processor where a task has to be executed is determined off-line and cannot be changed at run time.
- **Dynamic allocation**
  The processor where a task has to be executed is determined at runtime and can be changed during execution (task migration).
- **Hybrid approaches**
  Clustered: a task can dynamically be assigned only in a subset of processors (cluster).
  Semi-partitioned: some tasks can be split in parts allocated to different processors.

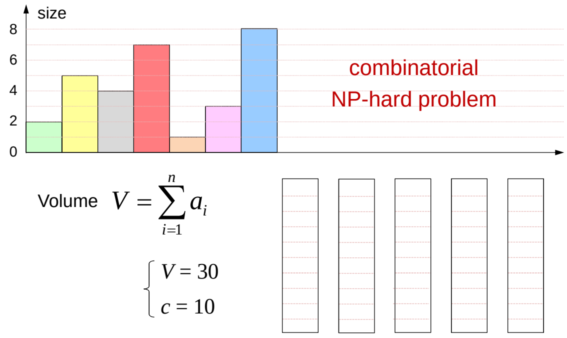
How to allocate tasks?

Tasks can be allocated based on their utilization.



The Bin Packing problem

Pack n objects of different size $a_1, a_2, \ldots, a_n$ into the minimum number of bins (containers) of fixed capacity c.

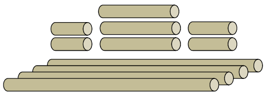


Practical examples

- How to fit vehicles into railcars
- How to store files into CDs

- How to fill minibuses with groups of people that must stay together.

- How to cut pieces of pipes from pipes of given length to minimize wastes.



Bin Packing algorithms

They can be distinguished into

Online

- Items arrive one at a time (in unknown order);
- Each item must be put in a bin before considering the next item.

Off line

- All items are given upfront, so they can be put into bins in any order.

Definitions

M_A number of bins used by an algorithm A

M_0 minimum number of bins used by the optimal algorithm

Performance ratio $\rho = \frac{M_A}{M_0}$

M_{lb} (**Lower bound**) Number of bins required for sure by any algorithm

M_{ub} (**Upper bound**) Number of bins that cannot be exceeded for sure by any algorithm

$$M_{lb} \le M_0 \le M_A \le M_{ub}$$

An easy lower bound

Given a set of n items of volume $V = \sum_{i=1}^{n} a_i$

No algorithm can use less than M_{lb} bins, where $M_{lb} = \left\lceil \frac{V}{c} \right\rceil$

In fact,

- if V is a multiple of c, that is $V = kc$ for some integer $k > 0$, then M cannot be less than $k = V/c$.
- if V is not a multiple of c, that is $kc < V < (k+1)c$, then M cannot be less than $k+1 = ceiling(V/c)$.

An easy upper bound

Given a set of n items of volume $V = \sum_{i=1}^{n} a_i$

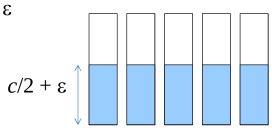
No algorithm can use more than M_{ub} bins, where $M_{ub} = \left\lceil \frac{2V}{c} \right\rceil$

Proof

The worst-case sequence that maximizes waste is a sequence of n items of size $c/2 + \varepsilon$

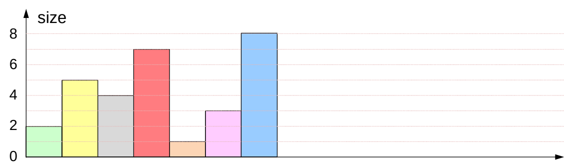
$V = n(c/2 + \varepsilon)$

$$M = n = \frac{2V}{c+2\varepsilon} \le \frac{2V}{c} \le \left\lceil \frac{2V}{c} \right\rceil$$



Optimal algorithm

Note that optimality implies clairvoyance for online sequences.

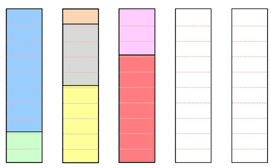


$V = 30$

$c = 10$

$M_{lb} = 3$

$M_0 = 3$



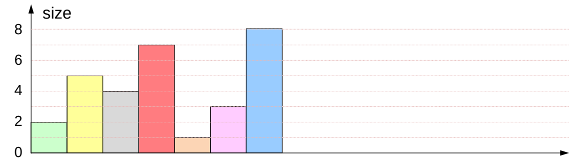
Bin Packing algorithms

Since the optimal solution is NP-hard, several heuristic algorithms have been proposed:

- **Next Fit (NF)**
  Place each item in the same bin as the last item. If it does not fit, start a new bin.
- **First Fit (FF)**
  Place each item in the first bin that can contain it.
- **Best Fit (BF)**
  Places each item in the bin with the smallest empty space.
- **Worst Fit (WF)**
  Places each item in the used bin with the largest empty space, otherwise start a new bin.

Next Fit

Place each item in the same bin as the last item. If it does not fit, start a new bin.

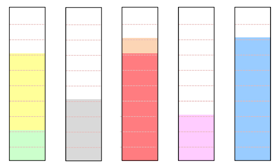


$V = 30$

$c = 10$

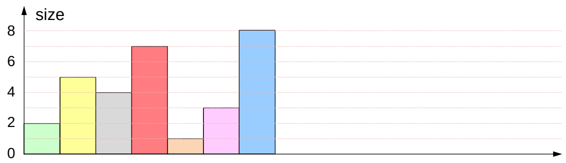
$M_0 = 3$

$M_{NF} = 5$



First Fit

Place each item in the first bin that can contain it.

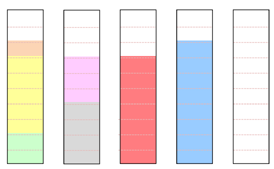


$V = 30$

$c = 10$

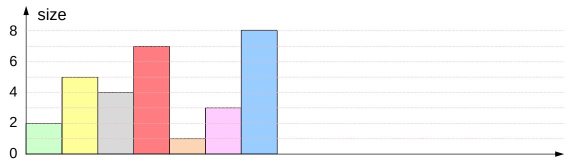
$M_0 = 3$

$M_{FF} = 4$



Best Fit

Places each item in the bin with the smallest empty space.

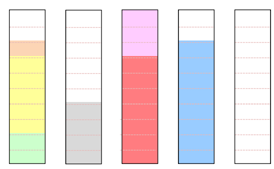


$V = 30$

$c = 10$

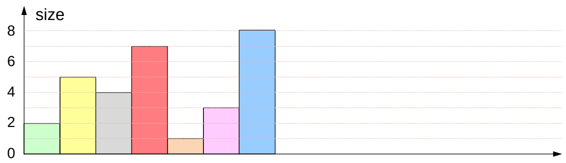
$M_0 = 3$

$M_{BF} = 4$



Worst Fit

Places each item in the used bin with the largest empty space, otherwise start a new bin.

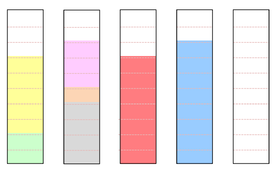


$V = 30$

$c = 10$

$M_0 = 3$

$M_{WF} = 4$



Comparison

Suppose the current situation is represented in blue and a new item of size 2 arrives:

new item

Observations

The performance of each algorithm strongly depends on the input sequence

however:

NF has a poor performance since it does not exploit the empty space in the previous bins

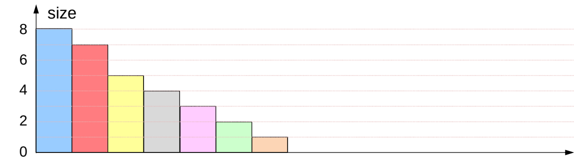
FF improves the performance by exploiting the empty space available in all the used bins.

BF tends to fill the used bins as much as possible.

WF tends to balance the load among the used bins.

First Fit Decreasing

If all items are known off-line, sort the items in decreasing order, then use First Fit.

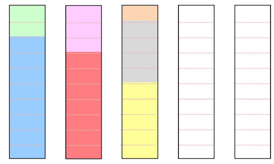


$V = 30$

$c = 10$

$M_0 = 3$

$M_{FFD} = 3$

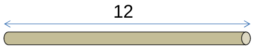


Another example

We need a set of pipes of the following lengths:

Length (m)	Number
2	2
3	4
4	3
6	1
7	2

But on the market we can only buy pipes of 12 meters:

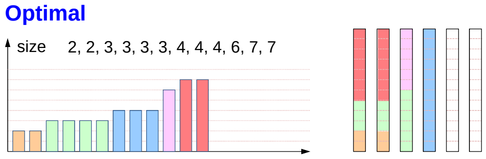


How can we cut the pipes to minimize the wasted material?

$$\begin{cases} V = 48 \\ c = 12 \end{cases} \Rightarrow M_0 = \left\lceil \frac{V}{c} \right\rceil = 4$$

Optimal solution

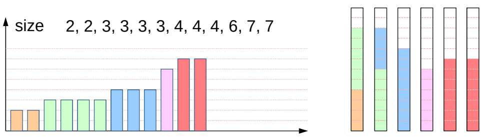
Optimal



$M_0 = 4$

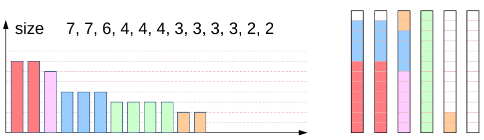
Heuristic solutions

First Fit



$M_{FF} = 6$

First Fit Decreasing



$M_{FFD} = 5$

Performance evaluation

The worst-case performance of an algorithm A with respect to the optimal algorithm and for any possible sequence can be measured by the

Competitive ratio

If σ is a sequence of items, the competitive ratio of a bin packing algorithm A is defined as

$$\varphi_A = \max_{\sigma} \left\{ \frac{M_A(\sigma)}{M_0(\sigma)} \right\}$$

Some theoretical result

Any online algorithm uses at least 4/3 times the optimal number of bins:

$$M_{on} \geq \frac{4}{3} M_0$$

NF and **WF** never use more than $2\ M_0$ bins.

FF and **BF** never use more than $(1.7\ M_0 + 1)$ bins.

FFD never uses more than $(4/3\ M_0 + 1)$ bins.

FFD never uses more than $(11/9\ M_0 + 4)$ bins.

BP for task allocation

Note that the ratio M_A/M_0 is not a good metric to compare different task allocation algorithms, because:

- since the problem is NP hard, M_0 cannot be computed in polynomial or pseudo-polynomial time;
- it does not take into account the number of tasks and the task set utilization.
- even if M_0 is known, we would not get a tight bound on the number of processors needed to schedule a task set.

In fact, for a set of $n = 10m$ tasks, each with utilization $0.5 + \varepsilon$, we would have $M_0 = 10m$, and $M_{FF} < 1.7\ M_0 + 1 = 17m + 1$.

That is, the ratio suggests to use $(17m + 1)$, when $U = 5m$. So the solution would be higher than $M_{ub} = \text{ceiling}(2U) = 10m$.

Definitions

To derive useful allocation bounds as a function of task utilizations, we need some definitions:

- Γ set of n tasks: $\Gamma = \{\tau_1, \dots, \tau_n\}$
- $\wp$ set of m processors: $\wp = \{P_1, \dots, P_m\}$
- u_i utilization of task τ_i
- U total task set utilization $U = \sum_{i=1}^{n} u_i$

- n_j number of tasks currently allocated on processor P_j
- U_j total utilization of processor P_j $U_j = \sum_{\tau_i \in P_j} u_i$

Definitions

Worst-case achievable utilization

The worst-case achievable utilization for a scheduler S and an allocation algorithm A is a real number $U_{wc}^{S\text{-}A}$ such that:

- any task set with utilization $U \leq U_{wc}^{S\text{-}A}$ is schedulable by S using A;
- it is always possible to find a task set with utilization $U > U_{wc}^{S\text{-}A}$ that is not schedulable by S using A.

First-Fit allocation algorithm

```
int first_fit_allocation(Γ,℘,S)
{
    for (i=1; i<=n; i++) { // for each task i
        j = 1;             // try from proc P1
        while (!schedulable(i,j,S) && (j < m)) j++;
        if (j < m) return(UNSCHEDULABLE);
        allocate(i,j);     // assign task i to Pj
    }
    return(SCHEDULABLE);
}
```

`schedulable(i,j,S)` returns 1 if $(u_i + U_j \leq U_{wc}^{S\text{-}FF})$, 0 otherwise

`allocate(i,j)` assigns τ_i to P_j and updates $U_j = U_j + u_i$

First-Fit decreasing algorithm

Like FF, but it initially sorts the task by decreasing utilizations:

```
int first_fit_allocation(Γ,℘,S)
{
    sort_by_decreasing_u(Γ); // u_1 >= u_2 >=...
    for (i=1; i<=n; i++) {   // for each task i
        j = 1;               // try from proc P1
        while (!schedulable(i,j,S) && (j < m)) j++;
        if (j < m) return(UNSCHEDULABLE);
        allocate(i,j);       // assign task i to Pj
    }
    return(SCHEDULABLE);
}
```

Some utilization bounds

[Lopez-Diaz-Garcia, 2000]

Any task set with total utilization $U \leq (m+1)/2$ is schedulable in a multiprocessor made up of m processors using FF allocation and EDF scheduling on each processor.

Proof

Note that $(m+1)$ periodic tasks with utilization 0.5 can be scheduled on m processors, but $(m+1)$ tasks with utilization $0.5+\varepsilon$ cannot be scheduled on m processors, independently of the allocation algorithms used.

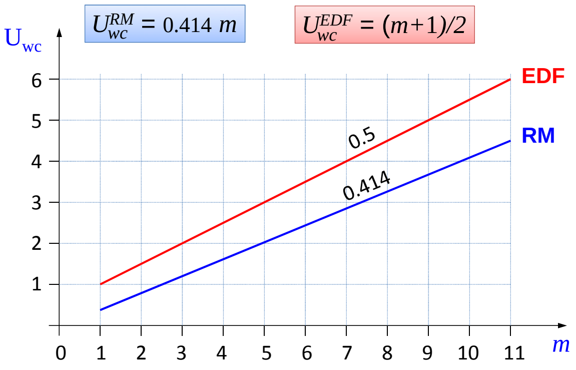
Some utilization bounds

[Oh & Baker, 1998]

Any task set with total utilization $U \leq m(2^{1/2} - 1)$ is schedulable in a multiprocessor made up of m processors using FF allocation and RM scheduling on each processor.

Comparison

$U_{wc}^{RM} = 0.414\ m$ $\quad$ $U_{wc}^{EDF} = (m+1)/2$



A better EDF bound

A better EDF bound can be found if tasks are not allowed to have arbitrary utilization $u_i \in [0,1]$, but can have a maximum utilization α, that is:

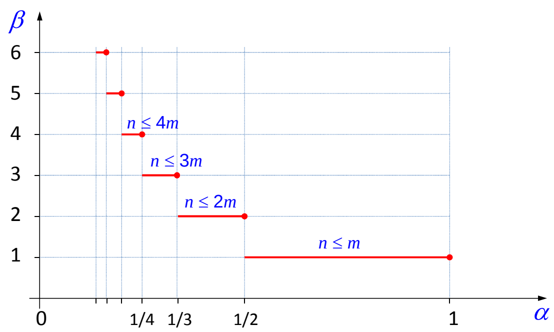
$$\forall i \quad 0 \leq u_i \leq \alpha \leq 1$$

Let β be the maximum number of tasks of utilization α that fit in one processor. Then, for the EDF schedulability it must be $\beta\alpha \leq 1$, hence $\beta \leq 1/\alpha$. But since β is an integer, it must be:

$$\beta = \left\lfloor \frac{1}{\alpha} \right\rfloor$$

EDF schedulability

Note that if $(n \leq \beta m)$, then n tasks are always schedulable on m processors.



EDF schedulability

[Lopez-Diaz-Garcia, 2000]

If $(n > \beta m)$ and $\forall i \ u_i \leq \alpha$, a task set is schedulable by EDF using FF allocation if

$$U \leq \frac{\beta m + 1}{\beta + 1}$$

Note that:

- if $\alpha = 1$, then $\beta = 1$, and $\quad U_{wc}^{EDF-FF} = \dfrac{m+1}{2}$
- if $\alpha \to 0$, then $\beta \to \infty$, and $\quad U_{wc}^{EDF-FF} \to m$

A better RM bound

A better RM bound can also be found assuming that tasks can have a maximum utilization α, that is: $\forall i \quad 0 \le u_i \le \alpha \le 1$

Let β be the maximum number of tasks of utilization α that fit in one processor. Then, for the RM schedulability it must be that $\beta\alpha \le \beta(2^{1/\beta}-1)$, that is:

$$\beta \le \frac{1}{\log_2(\alpha+1)}$$

But since β is an integer, it must be:

$$\beta = \left\lfloor \frac{1}{\log_2(\alpha+1)} \right\rfloor$$

RM schedulability

[Lopez-Diaz-Garcia, 1999]

When each task has utilization $u_i \le \alpha$, a task set is schedulable by RM using FF allocation if

$$U \le \beta(m-1)(2^{1/(\beta+1)}-1)+(n-\beta(m-1))(2^{1/(n-\beta(m-1))}-1)$$

Note that:

- if $\alpha = 1$ $(\beta = 1)$ $\Rightarrow$

$$U_{wc}^{RM-FF} = (m-1)(2^{1/2}-1)+(n-m+1)(2^{1/(n-m+1)}-1)$$

- if $\alpha \to 0$ $(\beta \to \infty)$ $\Rightarrow$ $U_{wc}^{RM-FF} \to m \ln 2$

Other utilization bounds

[Andersson-Baruah-Jonsson, 2001]

When each task has utilization $u_i \le m/(3m-2)$, the task set is feasible by global RM scheduling if

$$U \le \frac{m^2}{3m-2}$$