Task Scheduling

Definitions

A schedule σ is said to be **feasible** if it satisfies a set of constraints.

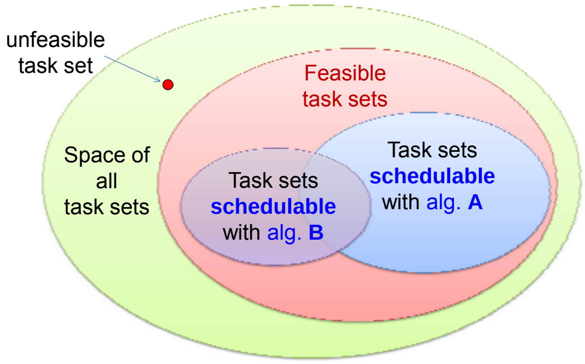
A task set Γ is said to be **feasible**, if there exists an algorithm that generates a feasible schedule for Γ.

A task set Γ is said to be **schedulable** with an algorithm A, if A generates a feasible schedule.

Examples of constraints

- Timing constraints: activation, period, deadline, jitter.
- Precedence: order of execution between tasks.
- Resources: synchronization for mutual exclusion.

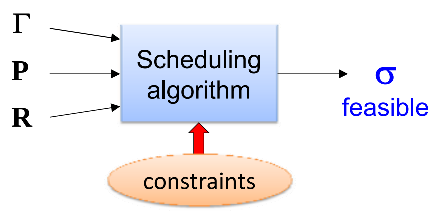
Feasibility vs. schedulability



3

The scheduling problem

Given a set Γ of n tasks, a set **P** of p processors, and a set **R** of r resources, find an assignment of **P** and **R** to Γ that produces a feasible schedule under a set of constraints.



Complexity

- In 1975, Garey and Johnson showed that the general scheduling problem is **NP hard**.

  In practice, it means that the time for finding a feasible schedule grows exponentially with the number of tasks.

Fortunately, polynomial time algorithms can be found under particular conditions.

Why do we care about complexity?

- Let's consider an application with **n = 30** tasks on a processor in which the elementary step takes **1 μs**
- Consider 3 algorithms with the following complexity:

A_1: ***O(n)***	A_2: ***$O(n^8)$***	A_3: ***$O(8^n)$***
⬇	⬇	⬇
30 μs	**182 hours**	**40.000 billion years**

Simplifying assumptions

- Single processor
- Homogeneous task sets
- Fully preemptive tasks
- Simultaneous activations
- No precedence constraints
- No resource constraints

Task set assumptions

We consider algorithms for different types of tasks:

- Single-job tasks (one shot)
  tasks with a single activation (not recurrent)
- Periodic tasks
  recurrent tasks regularly activated by a timer (each task potentially generates infinite jobs)
- Aperiodic/Sporadic tasks
  recurrent tasks irregularly activated by events (each task potentially generates infinite jobs)
- Mixed task sets

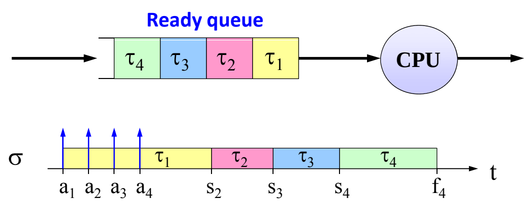
Classical scheduling policies

- First Come First Served
- Shortest Job First
- Priority Scheduling
- Round Robin

Not suited for real-time systems

First Come First Served

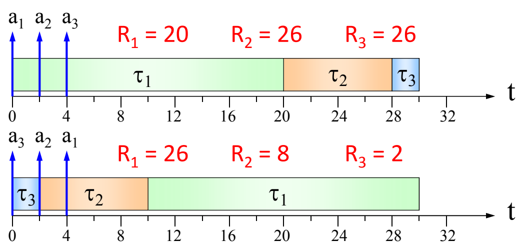
It assigns the CPU to tasks based on their arrival times (intrinsically non preemptive):



First Come First Served

- Very unpredictable

response times strongly depend on task arrivals:



11

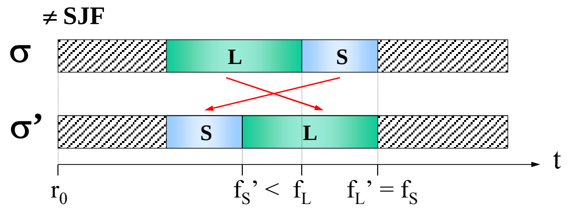
Shortest Job First (SJF)

It selects the ready task with the shortest computation time.

- Static (C_i is a constant parameter)
- It can be used on line or off-line
- Can be preemptive or non preemptive
- It minimizes the average response time

12

SJF - Optimality



$f_S' + f_L' \le f_S + f_L$

$$\overline{R}(\sigma') = \frac{1}{n}\sum_{i=1}^{n}(f'_i - r_i) \le \frac{1}{n}\sum_{i=1}^{n}(f_i - r_i) = \overline{R}(\sigma)$$

SJF - Optimality

$$\sigma \rightarrow \sigma' \rightarrow \sigma'' \rightarrow \ldots \rightarrow \sigma^*$$

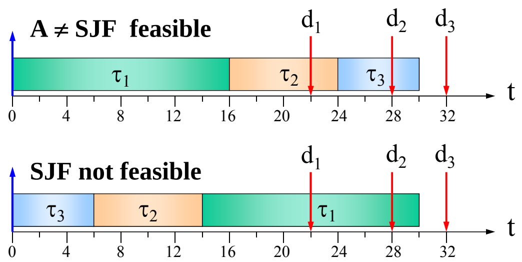
$$\overline{R}(\sigma) \ge \overline{R}(\sigma') \ge \overline{R}(\sigma'') \;\ldots\; \ge \overline{R}(\sigma^*)$$

$\sigma^* = \sigma_{SJF}$

$\overline{R}(\sigma_{SJF})$ is the minimum response time achievable by any algorithm

Is SJF suited for Real-Time?

- It is not optimal in the sense of feasibility



Priority Scheduling

- Each task has a priority P_i, typically $P_i \in [0, 255]$
- The task with the highest priority is selected for execution.
- Tasks with the same priority are served FCFS

NOTE:

$p_i \propto 1/C_i$	$\Rightarrow$	SJF
$p_i \propto 1/a_i$	$\Rightarrow$	FCFS

Priority Scheduling

- **Problem: starvation**

  low priority tasks may experience long delays due to the preemption of high priority tasks.

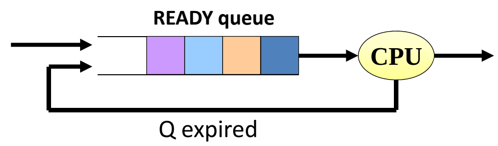
- **A possible solution: aging**

  priority increases with waiting time

Round Robin

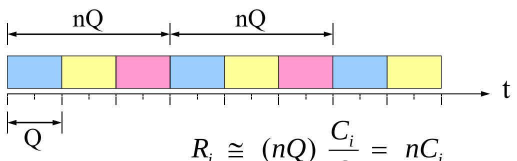
The ready queue is served with FCFS, but ...

- Each task τ_i cannot execute for more than Q time units (Q = time quantum).
- When Q expires, τ_i is put back in the queue.



Round Robin

n = number of task in the system



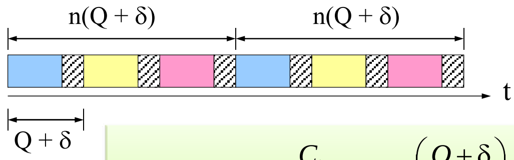
$$R_i \cong (nQ)\,\frac{C_i}{Q} = nC_i$$

Time sharing

Each task runs as it was executing alone on a virtual processor n times slower than the real one.

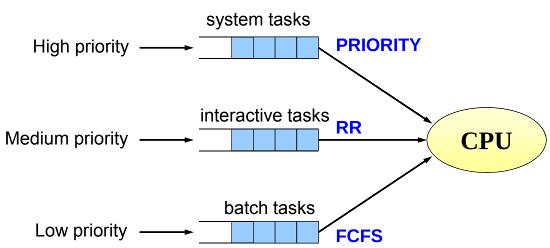
Round Robin

- **if** Q > max(C_i) **then** **RR ≡ FCFS**
- **if** Q ≅ context switch time (δ) **then**

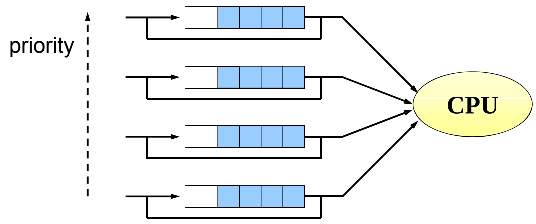


$$R_i \cong n(Q+\delta)\,\frac{C_i}{Q} = nC_i\left(\frac{Q+\delta}{Q}\right)$$

Multi-Level Scheduling



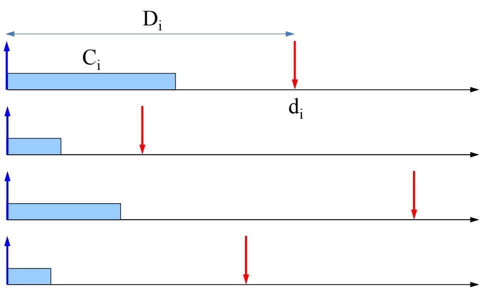
Multi-Level Scheduling



Real-Time Scheduling Algorithms

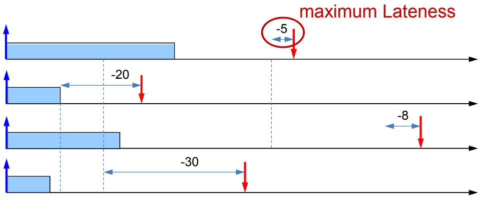
How to schedule RT tasks?

How to schedule RT tasks to maximize feasibility?



Earliest Due Date [Jackson 55]

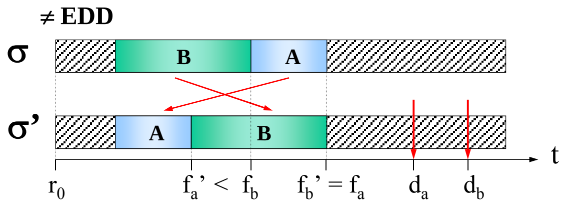
Given a set of real-time tasks arrived simultaneously, executing them by increasing deadline will minimize the maximum lateness (L_{max}).



NOTE:
- No other scheduler can decrease L_{max}
- Preemption is not required

25

EDD - Optimality



$L_{max} = L_a = f_a - d_a$

$L_a' = f_a' - d_a < f_a - d_a$

$L_b' = f_b' - d_b < f_a - d_a$

$L'_{max} < L_{max}$

26

EDD - Optimality

$$\sigma \rightarrow \sigma' \rightarrow \sigma'' \rightarrow \ldots \rightarrow \sigma^*$$

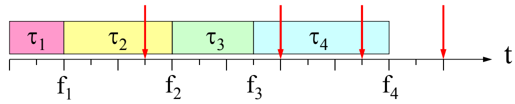
$$L_{max}(\sigma) \geq L_{max}(\sigma') \geq L_{max}(\sigma'') \ldots \geq L_{max}(\sigma^*)$$

$$\sigma^* = \sigma_{EDD}$$

$L_{max}(\sigma_{EDD})$ is the minimum value achievable by any algorithm

27

EDD guarantee test (off line)



A task set Γ is feasible iff $\forall i \quad f_i \leq d_i$

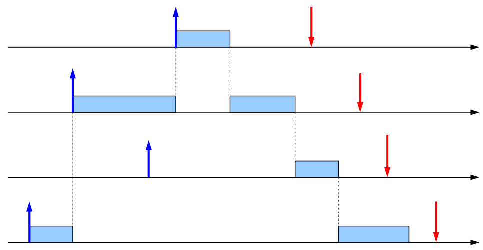
$$f_i = \sum_{k=1}^{i} C_k$$

$$\forall i \quad \sum_{k=1}^{i} C_k \leq D_i$$

28

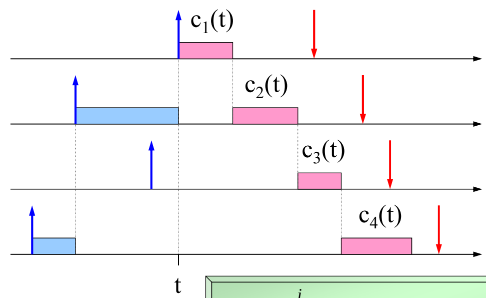
Earliest Deadline First

If tasks arrive dynamically, the maximum lateness can be minimized executing them by increasing absolute deadline, but preemption must be enabled.



29

EDF Guarantee test (on line)



$$\forall i \quad \sum_{k=1}^{i} c_k(t) \leq d_i - t$$

30

Complexity

EDD

Scheduler (queue ordering): $O(n \log n)$

Feasibility Test (guarantee test): $O(n)$

EDF

Scheduler (insertion in the queue): $O(n)$

Feasibility Test (guarantee single task): $O(n)$

31

EDF optimality

⇒ In the sense of feasibility [Dertouzos 1974]

An algorithm A is **optimal** in the sense of feasibility if it generates a feasible schedule, if there exists one.

Demonstration method

It is sufficient to prove that, given an arbitrary feasible schedule, the schedule generated by EDF is also feasible.

32

A property of optimal algorithms

If a task set Γ is not schedulable by an optimal algorithm, then Γ cannot be scheduled by any other algorithm.

If an algorithm A minimizes L_{max} then A is also optimal in the sense of feasibility. The opposite is not true.

33

Periodic Task Scheduling

Problem formulation

- We consider a computing system that has to execute a set Γ of n periodic real-time tasks:

$$\Gamma = \{ \tau_1, \tau_2, \dots \tau_n \}$$

- Each task τ_i is characterized by:

C_i worst-case computation time

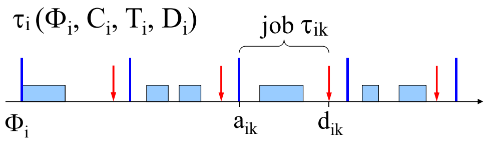
T_i activation period

D_i relative deadline

Φ_i initial arrival time (phase)

35

Problem formulation



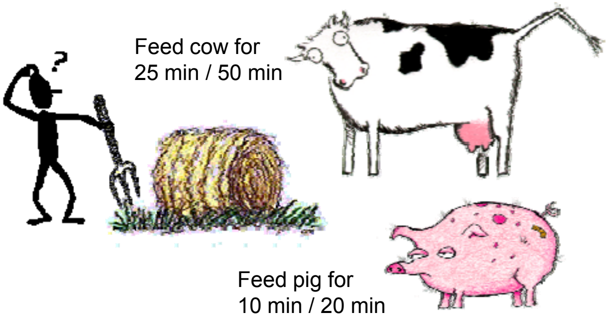
For each periodic task τ_i we must guarantee that:

- each job τ_{ik} is activated at $a_{ik} = \Phi_i + (k-1)T_i$
- each job τ_{ik} completes within $d_{ik} = a_{ik} + D_i$

There are several wrong ways to achieve this goal.

36

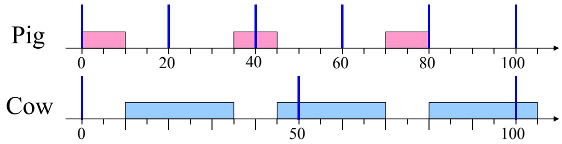
A farm scheduling problem



37

First try

Alternate pig with cow



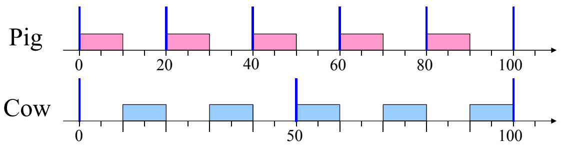
Evaluation:

Pig gets hungry
Cow gets fat

38

Second try

Feed pig and cow 10 min each



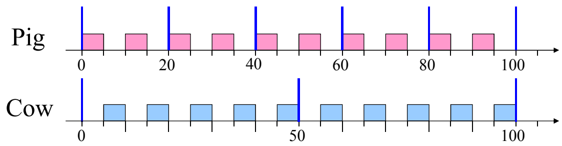
Evaluation:

Pig is OK
Cow is not happy

39

Third try

Feed pig and cow 5 min each



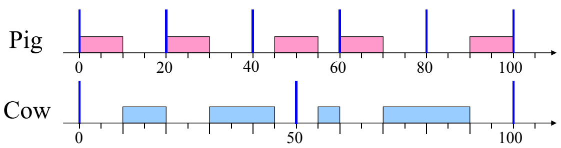
Evaluation:

Pig is OK, Cow is OK
but the farmer is tired

40

Optimal algorithm

Feed the most starving animal (≡ EDF)



Evaluation: Everybody is happy

41

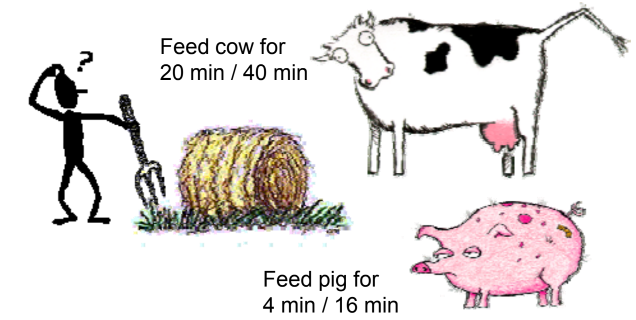
What do we learn?

- Reducing the execution time window, we get closer to a feasible solution.
- The time is split proportionally between the animals.

In the example, each animal required food for 50% of the time, but how can we generalize the solution if the animals require different fraction of time?

42

A new scheduling problem

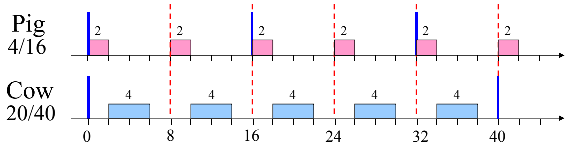


Proportional share algorithm

Basic idea

- Divide the timeline into slots of equal length.
- Within each slot serve each task for a time proportional to its utilization:

Pig utilization factor	=	4/16	= 1/4
Cow utilization factor	=	20/40	= 1/2



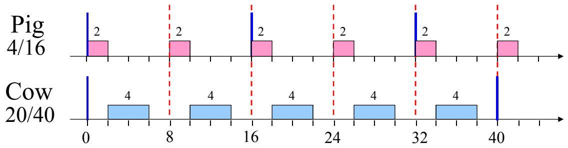
Proportional share algorithm

In general

Let: U_i = required feeding fraction

$\Delta = GCD(T_1, T_2) = 8$

execute each task for $\delta_i = U_i\Delta$ in each slot Δ



NOTE: $U_i\Delta$ ensures C_i in T_i, in fact: $\delta_i(T_i/\Delta) = C_i$

Feasibility test: $\Sigma\delta_i \leq \Delta$ i.e. $\Sigma U_i \leq 1$

Proportional share algorithm

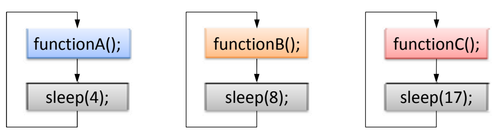
- This method approximates a fluid system, where execution progresses proportionally to U_i
- The major problem is that if periods are not harmonic, $\Delta = GCD(T_1, \ldots, T_n)$ is small and a task is fragmented into many chunks: T_i/Δ of small duration $\delta_i = U_i\Delta$.

too much overhead

Work and Sleep

According to this method, a task executes for C_i units and then suspends for $T_i - C_i$ units:

task	C_i	T_i	Sleep time
A	1	5	4
B	2	10	8
C	3	20	17

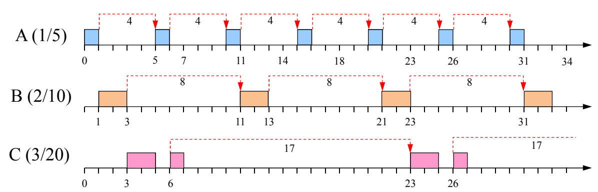


Work and Sleep

Example 1:

task	C_i	T_i	Sleep
A	1	5	4
B	2	10	8
C	3	20	17

It works well for small computation times

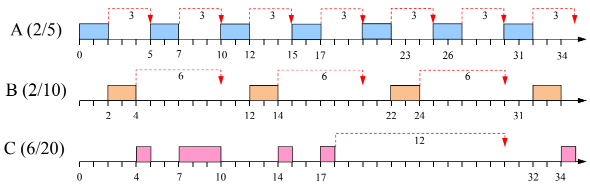


Work and Sleep

Example 2:

task	C_i	T_i	Sleep
A	2	5	3
B	2	8	6
C	6	20	12

Problem
Low priority tasks experience long delays



49

Loop Scheduling

It is a simple trick to schedule periodic activities at different rates using a single loop (often used in Arduino):

```
int   count = 0;           // relative time
int   T1 = 20;             // period 1 in ms
int   T2 = 50;             // period 2 in ms
int   T3 = 80;             // period 3 in ms

      while (1) {
            if (count%T1 == 0) function1();
            if (count%T2 == 0) function2();
            if (count%T3 == 0) function3();

            count++;
            if (count == T1*T2*T3) count = 0;

            delay(1);      // wait for 1 ms
      }
```

50

Loop Scheduling

Note that the counter must be reset at the least common multiple of the periods, called the hyperperiod (H):

```
      count++;
      if (count == T1*T2*T3) count = 0;
```

Q: How many bits are needed to represent the hyperperiod?

T1 = 10
T2 = 40
T3 = 50
T4 = 100
T5 = 500
T6 = 1000

$H = 10^{12}$

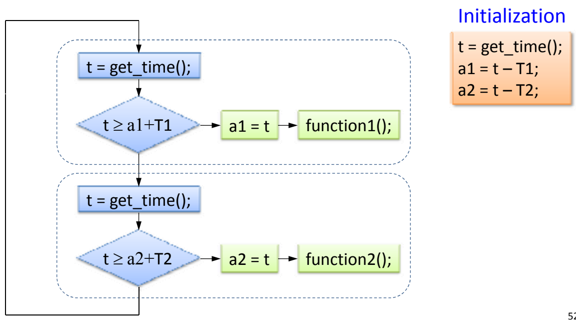
$$\text{bits} = \lceil \log_2 10^{12} \rceil = \lceil 39.86 \rceil = 40$$

It does not fit to a long integer.
We are in trouble!

51

Loop Scheduling

A better way is to rely on a system call that returns the system time:



52

Loop Scheduling

Implementation:

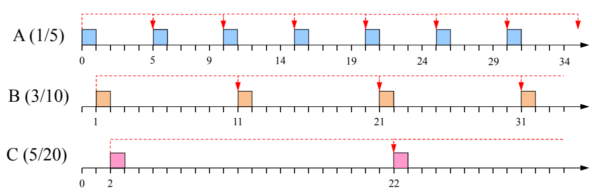
```
#define    N     5                  // number of tasks
time       t;                       // current time
time       a[N], T[N];              // act. times, periods

      initialize_periods(T);        // e.g., read from file

      t = get_time();
      for (i=0; i<N; i++) a[i] = t – T[i];

      while (1) {
            for (i=0; i<N; i++) {
                  t = get_time();
                  if (t >= a[i] + T[i]) {
                        a[i] = t;
                        function(i);
                  }
            }
      }
```

53

Loop Scheduling

Example 1:

task	C_i	T_i
A	1	5
B	1	10
C	1	20



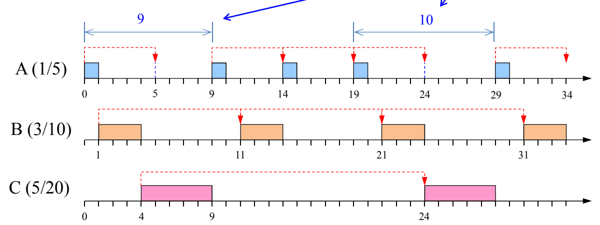
54

Loop Scheduling

Example 2:

task	C_i	T_i
A	1	5
B	3	10
C	5	20

Problem
Tasks experience delays from the other tasks



55

Loop Scheduling

If the scheduler is not the only thread, a sleep must be inserted:

```
#define    N          5           // number of tasks
#define    DELTA      3           // milliseconds
time       t;                     // current time
time       a[N], T[N];            // act. times, periods

      initialize_periods(T);      // e.g., read from file

      t = get_time();
      for (i=0; i<N; i++) a[i] = t - T[i];
      while (1) {
            for (i=0; i<N; i++) {
                  t = get_time();
                  if (t-a[i] >= T[i]) {
                        a[i] = t;
                        function(i);
                  }
            }
            sleep(DELTA);         // suspend for 3 ms
      }
```

56

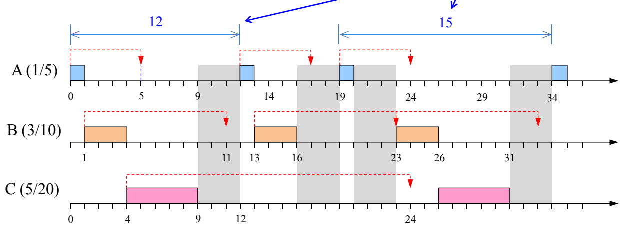
Loop Scheduling

Example 3:

DELTA = 3

task	C_i	T_i
A	1	5
B	3	10
C	5	20

Problem
The experienced delay is given by other tasks + suspension



NOTE: Suspension time can be higher due to other tasks

57

Timeline Scheduling

Also known as cyclic scheduling, it has been used for 30 years in military systems, navigation, and monitoring systems.

Examples

- Air traffic control systems
- Space Shuttle
- Boeing 777
- Airbus navigation system

58

Timeline Scheduling

Method

- The time axis is divided in intervals of equal length (*time slots*).
- Each task is statically allocated in a slot in order to meet the desired request rate.
- The execution in each slot is activated by a timer.

59

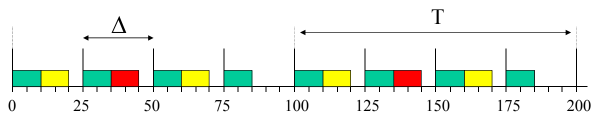
Timeline Scheduling

Example:

task	C_i	T_i
A	10 ms	25 ms
B	10 ms	50 ms
C	10 ms	100 ms

Δ = GCD (minor cycle)

T = lcm (major cycle)

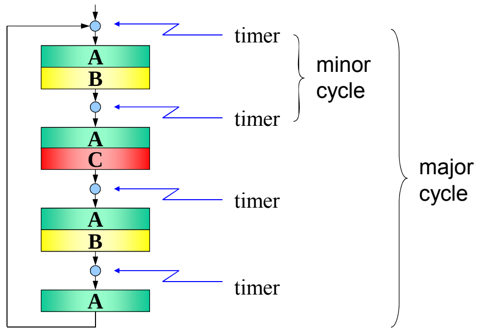


Guarantee: $C_A + C_B \leq \Delta$, $C_A + C_C \leq \Delta$

60

Timeline Scheduling

Implementation:



61

Cycling Scheduling

Coding:

```
#define    MINOR      25          // minor cycle = 25 ms

    initialize_timer(MINOR);      // interrupt every 25 ms

    while (1) {
        sync();                   // block until interrupt
        function_A();
        function_B();
        sync();                   // block until interrupt
        function_A();
        function_C();
        sync();                   // block until interrupt
        function_A();
        function_B();
        sync();                   // block until interrupt
        function_A();
    }
```

62

Timeline scheduling

Advantages

- Simple implementation (no RTOS is required).
- Low run-time overhead.
- All tasks run with very low jitter.

Disadvantages

- It is not robust during overloads.
- It is difficult to expand the schedule.
- It is not easy to handle aperiodic activities.

63

Problems during overloads

What do we do during task overruns?

➢ **Let the task continue**

– we can have a ***domino effect*** on all the other tasks (timeline break)

➢ **Abort the task**

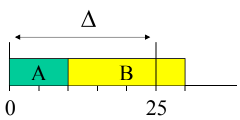
– the system can remain in inconsistent states.

64

Expandibility

If one or more tasks need to be upgraded, we may have to re-design the whole schedule again.

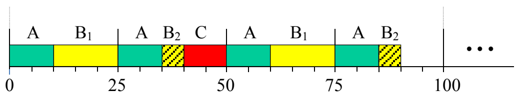
Example: B is updated so that $C_B = 20$ ms
now $C_A + C_B > \Delta$



65

Expandibility

- We have to split task B in two subtasks (B_1, B_2) and re-build the schedule:



Guarantee: $\begin{cases} C_A + C_{B1} \le \Delta \\ C_A + C_{B2} + C_C \le \Delta \end{cases}$

66

Expandibility

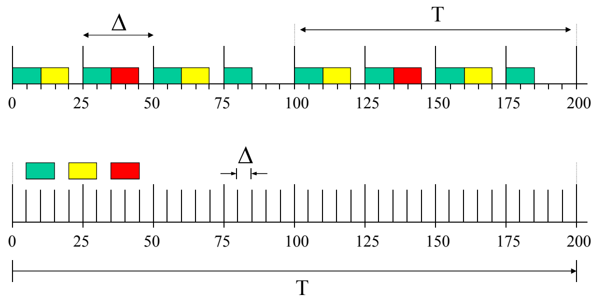
If the frequency of some task is changed, the impact can be even more significant:

task	T_{old}	T_{new}
A	25 ms	25 ms
B	50 ms	**40 ms**
C	100 ms	100 ms

minor cycle:	$\Delta = 25$	$\Delta = 5$	40 sync. per cycle!
major cycle:	$T = 100$	$T = 200$	

67

Example



68

Priority Scheduling

Priority Scheduling

Method

1. Assign priorities to each task based on its timing constraints.
2. Verify the feasibility of the schedule using analytical techniques.
3. Execute tasks on a priority-based kernel.

70

How to assign priorities?

- Typically, task priorities are assigned based on the their relative importance.
- However, different priority assignments can lead to different processor utilization bounds.

71

Priority vs. importance

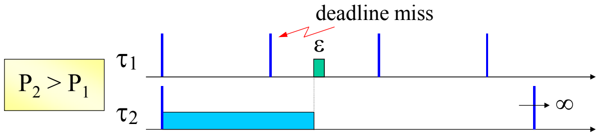
If τ_2 is more important than τ_1 and is assigned higher priority, the schedule may not be feasible:

72

Priority vs. importance

If priority are not properly assigned, the utilization bound can be arbitrarily small:

An application can be unfeasible even when the processor is almost empty!



$$U = \frac{\varepsilon}{T_1} + \frac{C_2}{\infty} \rightarrow 0$$

Optimal priority assignments

- **Rate Monotonic (RM):** (optimal among FP algs for T = D)
  $P_i \propto 1/T_i$ (static)
- **Deadline Monotonic (DM):** (optimal among FP algs for $D \leq T$)
  $P_i \propto 1/D_i$ (static)
- **Earliest Deadline First (EDF):** (optimal among all algs)
  $P_i \propto 1/d_{ik}$ (dynamic)

$$d_{i,k} = r_{i,k} + D_i$$

Rate Monotonic is optimal

RM is **optimal** among all fixed priority algorithms (if $D_i = T_i$):

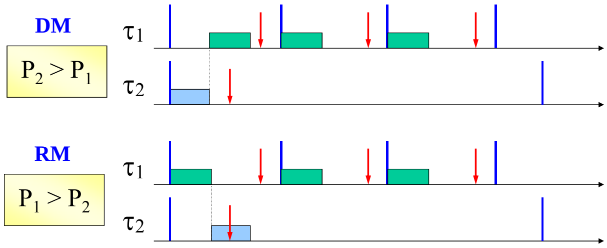
If there exists a fixed priority assignment which leads to a feasible schedule, then the RM schedule is feasible.

⇕

If a task set is not schedulable by RM, then it cannot be scheduled by any fixed priority assignment.

Deadline Monotonic is optimal

If $D_i \leq T_i$ then the **optimal** priority assignment is given by **Deadline Monotonic** (**DM**):



EDF Optimality

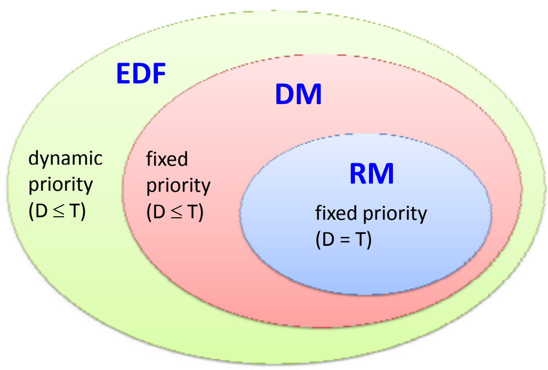
EDF is **optimal** among all algorithms:

If there exists a feasible schedule for a task set, then EDF will generate a feasible schedule.

⇕

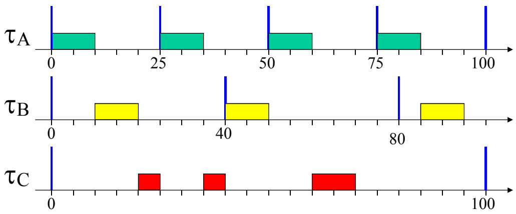
If a task set is not schedulable by EDF, then it cannot be scheduled by any algorithm.

Optimality



Rate Monotonic (RM)

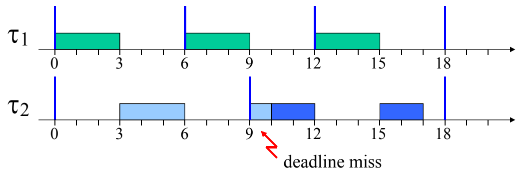
- Each task is assigned a fixed priority proportional to its rate [Liu & Layland '73].



Note that small parameter variations are automatically handled by the scheduler without any intervention.

An unfeasible RM schedule

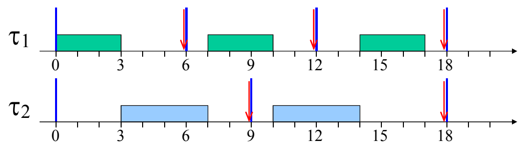
$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$



EDF Schedule

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$

$D_i = T_i$



How can we verify feasibility?

- Each task uses the processor for a fraction of time:

$$U_i = \frac{C_i}{T_i}$$

- Hence the total **processor utilization** is:

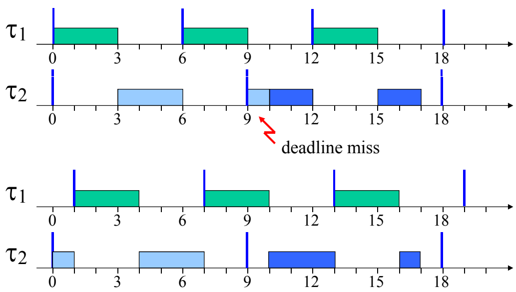
$$U_p = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

- U_p is a measure of the **processor load**.

Identifying the worst case

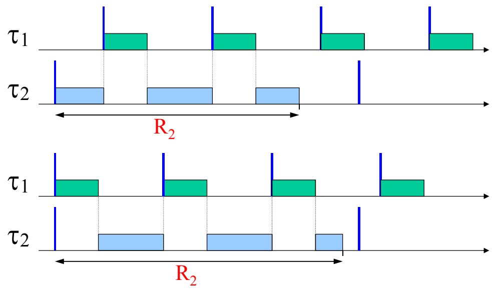
Feasibility may depend on the initial activations (phases):

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$



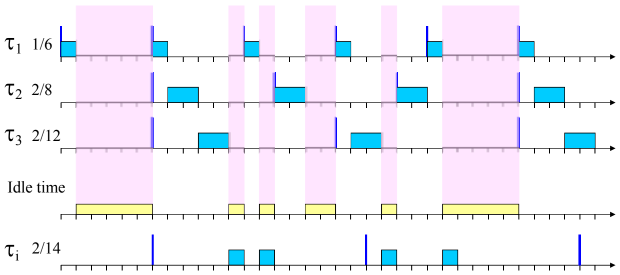
Critical Instant

For any task τ_i, the longest response time occurs when it arrives together with all higher priority tasks.



Critical Instant

For independent preemptive tasks under fixed priorities, the critical instant of τ_i, occurs when it arrives together with all higher priority tasks.



A necessary condition

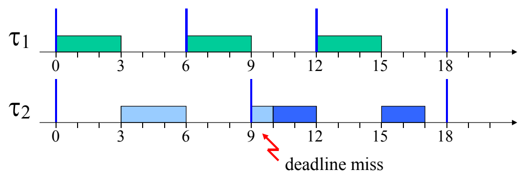
A necessary condition for having a feasible schedule is that $\mathbf{U_p \leq 1}$.

In fact, if $\mathbf{U_p > 1}$ the processor is overloaded hence the task set cannot be schedulable.

However, there are cases in which $\mathbf{U_p \leq 1}$ but the task set is not schedulable by RM.

An unfeasible RM schedule

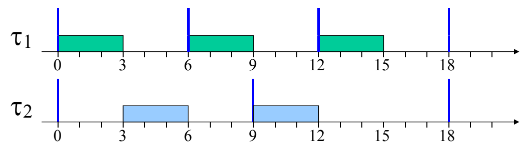
$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$



Given this task set (period configuration), what is the higher utilization that guarantees feasibility?

Utilization upper bound

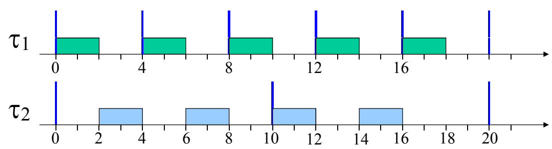
$$U_p = \frac{3}{6} + \frac{3}{9} = 0.833$$



NOTE: If C_1 or C_2 is increased, τ_2 will miss its deadline!

A different upper bound

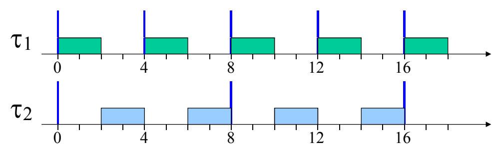
$$U_{ub} = \frac{2}{4} + \frac{4}{10} = 0.9$$



NOTE: The upper bound U_{ub} depends on the specific task set.

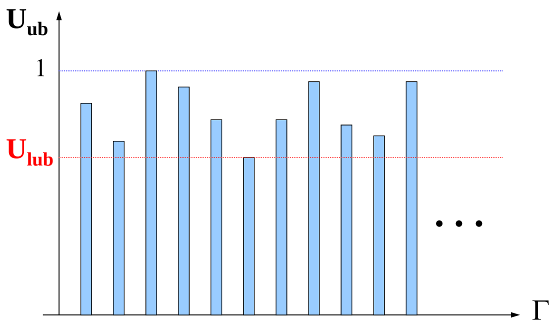
A different upper bound

$$U_p = \frac{2}{4} + \frac{4}{8} = 1$$



NOTE: The upper bound U_{ub} depends on the specific task set.

The least upper bound



A sufficient condition

If $U_p \leq U_{lub}$ the task set is certainly schedulable with the RM algorithm.

NOTE

If $U_{lub} < U_p \leq 1$ we cannot say anything about the feasibility of that task set.

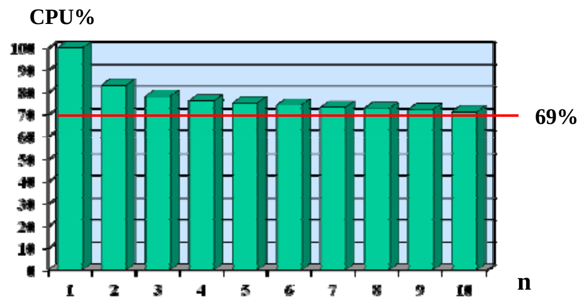
U_{lub} for RM

- In 1973, **Liu and Layland** proved that for a set of n periodic tasks:

$$U_{lub}^{RM} = n\left(2^{1/n} - 1\right)$$

for $n \to \infty$ $U_{lub} \to \ln 2$

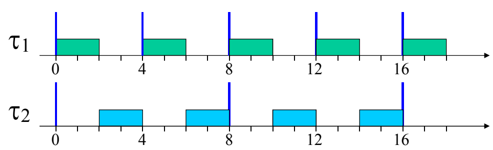
RM Least Upper Bound



A special case

If tasks have harmonic periods U_{lub} = 1.

$$U_p = \frac{2}{4} + \frac{4}{8} = 1$$



RM Guarantee Test

- We compute the processor utilization as:

$$U_p = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

- Guarantee Test (only sufficient):

$$U_p \leq n\left(2^{1/n} - 1\right)$$

Basic Assumptions

A1. C_i is constant for every job of τ_i

A2. T_i is constant for every job of τ_i

A3. For each task, $D_i = T_i$

A4. Tasks are independent:

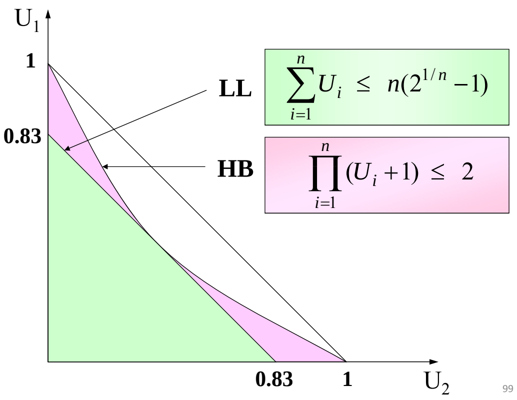
- no precedence relations
- no resource constraints
- no blocking on I/O operations

The Hyperbolic Bound

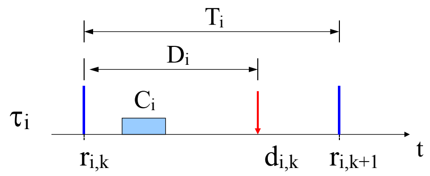
- In 2000, **Bini et al.** proved that a set of *n* periodic tasks is schedulable by RM if:

$$\prod_{i=1}^{n} (U_i + 1) \leq 2$$

HB vs. LL



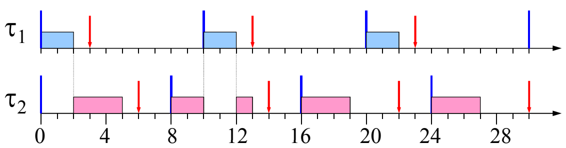
Extension to tasks with D < T



Scheduling algorithms

- Deadline Monotonic: **$p_i \propto 1/D_i$** (static)
- Earliest Deadline First: **$p_i \propto 1/d_i$** (dynamic)

Deadline Monotonic



Problem with the Utilization Bound

$$U_p = \sum_{i=1}^{n} \frac{C_i}{D_i} = \frac{2}{3} + \frac{3}{6} = 1.16 > 1$$

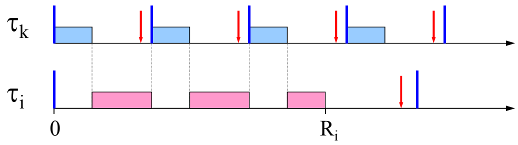
but the task set is schedulable.

Response Time Analysis

[Audsley '90]

- For each task τ_i compute the interference due to higher priority tasks: $I_i = \sum_{D_k < D_i} C_k$
- compute its response time as $R_i = C_i + I_i$
- verify that $R_i \leq D_i$

Computing Interference



Interference of τ_k on τ_i in the interval $[0, R_i]$: $I_{ik} = \left\lceil \frac{R_i}{T_k} \right\rceil C_k$

Interference on τ_i by high-priority tasks: $I_i = \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$

103

Computing Response Times

$$R_i = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$$

Iterative solution:

$$\begin{cases} R_i^0 = C_i \\ R_i^s = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i^{(s-1)}}{T_k} \right\rceil C_k \end{cases}$$

iterate while $R_i^s > R_i^{(s-1)}$

104

Dynamic Priority Scheduling

Earliest Deadline First (EDF)

- Each job receives an absolute deadline:

$$d_{i,k} = r_{i,k} + D_i$$

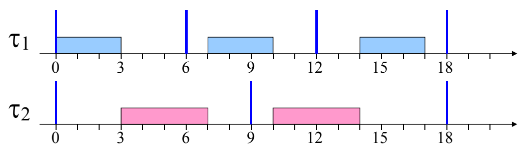
- At any time, the processor is assigned to the job with the earliest absolute deadline.
- Under EDF, any task set can utilize the processor up to 100%.

106

EDF Example

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$

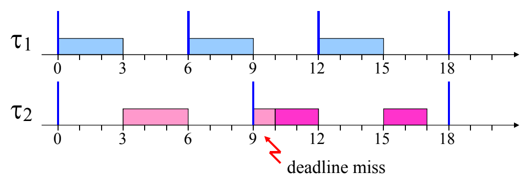
$D_i = T_i$



107

Unfeasible under RM

$$U_p = \frac{3}{6} + \frac{4}{9} = 0.944$$



108

EDF Optimality

EDF is **optimal** among all algorithms:

If there exists a feasible schedule for a task set Γ, then EDF will generate a feasible schedule.

⇕

If Γ is not schedulable by EDF, then it cannot be scheduled by any algorithm.

109

EDF schedulability

- In 1973, **Liu and Layland** proved that for a set of n periodic tasks:

$$U_{lub}^{EDF} = 1$$

- This means that a task set Γ is schedulable by EDF **if and only if**

$$U_p \leq 1$$

110

EDF with D ≤ T

Schedulability Analysis

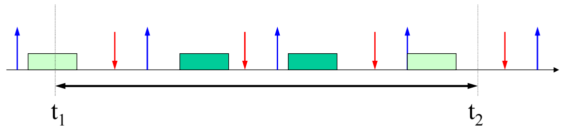
Processor Demand Criterion [Baruah '90]

In any interval of length L, the computational demand $g(0,L)$ of the task set must be no greater than the available time in that interval.

$$\forall L > 0, \quad g(0,L) \leq L$$

111

Processor Demand

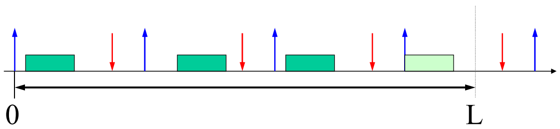


The demand in [t_1, t_2] is the computation time of those tasks started at or after t_1 with deadline less than or equal to t_2:

$$g(t_1,t_2) = \sum_{r_i \geq t_1}^{d_i \leq t_2} C_i$$

112

Demand of a periodic task

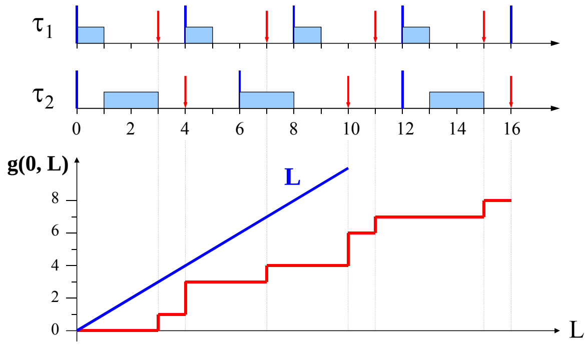


$$g_i(0,L) = \left\lfloor \frac{L - D_i + T_i}{T_i} \right\rfloor C_i$$

$$g(0,L) = \sum_{i=1}^{n} \left\lfloor \frac{L - D_i + T_i}{T_i} \right\rfloor C_i$$

113

Example



114

Bounding complexity

- Since $g(0,L)$ is a step function, we can check feasibility only at deadline points.
- If tasks are synchronous and $U_p < 1$, we can check feasibility up to the hyperperiod H:

$$H = lcm(T_1, \dots, T_n)$$

Bounding complexity

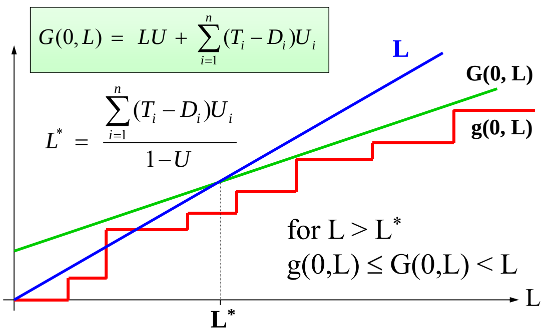
- Moreover we note that: $g(0, L) \leq G(0, L)$

$$G(0,L) = \sum_{i=1}^{n} \left(\frac{L + T_i - D_i}{T_i} \right) C_i$$

$$= \sum_{i=1}^{n} L \frac{C_i}{T_i} + \sum_{i=1}^{n} (T_i - D_i) \frac{C_i}{T_i}$$

$$= LU + \sum_{i=1}^{n} (T_i - D_i) U_i$$

Limiting L



Processor Demand Test

$$\forall L \in D, \quad g(0,L) \leq L$$

$$D = \{d_k \mid d_k \leq \min(H, L^*)\}$$

$$\begin{cases} H = lcm(T_1, \dots, T_n) \\ L^* = \dfrac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U} \end{cases}$$

Summary

- **Three scheduling approaches:**
  - Off-line construction (Timeline)
  - Fixed priority (RM, DM)
  - Dynamic priority (EDF)
- **Three analysis techniques:**
  - Processor Utilization Bound $U \leq U_{lub}$
  - Response Time Analysis $\forall i \;\; R_i \leq D_i$
  - Processor Demand Criterion $\forall L \;\; g(0,L) \leq L$

Schedulability Analysis

	$D_i = T_i$	$D_i \leq T_i$
RM	***Suff.:*** *polynomial* $O(n)$ LL: $\Sigma U_i \leq n(2^{1/n} - 1)$ HB: $\Pi(U_i+1) \leq 2$ ***Exact*** *pseudo-polynomial* RTA	***pseudo-polynomial*** Response Time Analysis $\forall i \;\; R_i \leq D_i$ $R_i = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$
EDF	***polynomial:*** $O(n)$ $\Sigma U_i \leq 1$	***pseudo-polynomial*** Processor Demand Analysis $\forall L > 0, \quad g(0,L) \leq L$