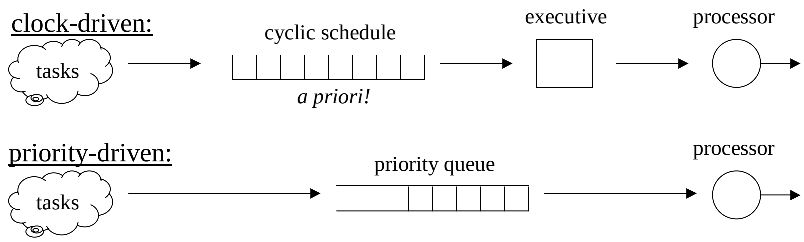
Priority-Driven Scheduling of Periodic Tasks

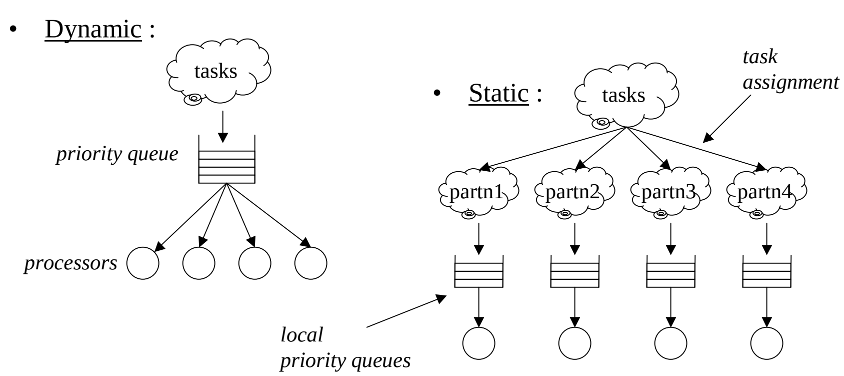
- Priority-driven *vs.* clock-driven scheduling:



- Assumptions:
  - tasks are periodic
  - jobs are ready as soon as they are released
  - preemption is allowed
  - tasks are independend
  - no aperiodic or sporadic tasks
- We will later:
  - integrate aperiodic and sporadic tasks
  - integrate resources
  - *etc.*

Why Focus on Uniprocessor Scheduling?

- Dynamic *vs.* static multiprocessor scheduling:



- Poor worst-case performance of priority-driven algorithms in dynamic environments.
- Difficulty in validating timing constraints.

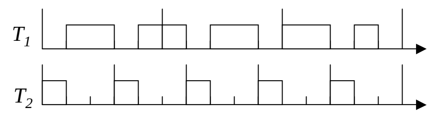
Static-Priority *vs*. Dynamic Priority

- **Static-Priority**: All jobs in task have same priority.
- example:

  Rate-Monotonic: "The shorter the period, the higher the priority."

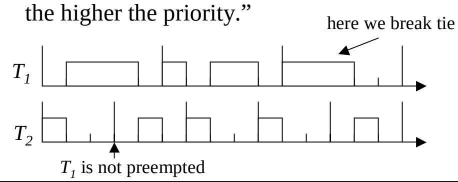
$T_1 = (5, 3, 5)$

$T_2 = (3, 1, 3)$



- **Dynamic-Priority**: May assign different priorities to individual jobs.
- example:

  Earliest-Deadline-First: "The nearer the absolute deadline, the higher the priority."



Example Algorithms

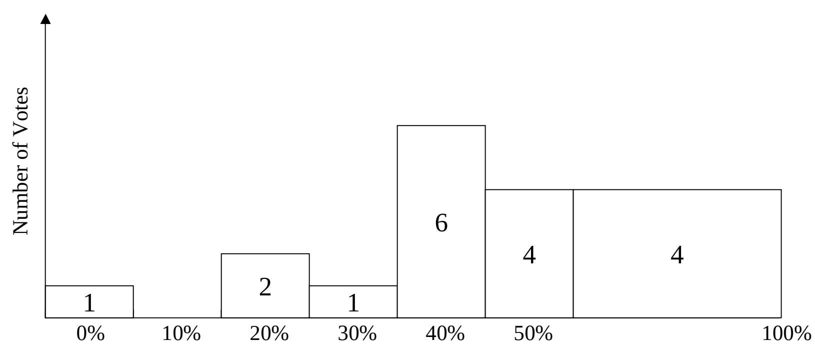
- Static-Priority:
  - **Rate-Monotonic** (**RM**): "The shorter the period, the higher the priority." [Liu+Layland '73]
  - **Deadline-Monotonic** (**DM**): "The shorter the relative deadline, the higher the priority." [Leung+Whitehead '82]
- For arbitrary relative deadlines, DM outperforms RM.
- Dynamic-Priority:
  - **EDF**: Earliest-Deadline-First.
  - **LST**: Least-Slack-Time-First.
  - **FIFO/LIFO**
  - *etc.*

Considerations about Priority-Driven Scheduling

- FIFO/LIFO do not take into account urgency of jobs.
- Static-priority assignments based on functional criticality are typically non-optimal.
- We confine our attention to algorithms that assign priorities based on temporal parameters.
- Def: [**Schedulable Utilization**] Every set of periodic tasks with total utilization less or equal than the schedulable utilization of an algorithm can be feasibly scheduled by that algorithm.
- The higher the schedulable utilization, the better the algorithm.
- Schedulable utilization is always less or equal 1.0!

Schedulable Utilization of FIFO

- Result of Opinion Poll in CPSC-663 of Fall 2001:



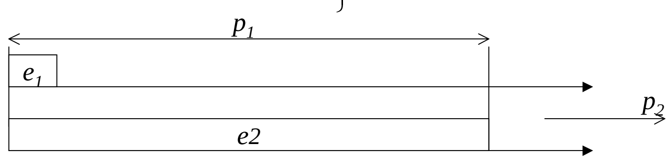
Schedulable Utilization of FIFO (II)

- Theorem: $U_{FIFO} = 0$
- Proof:

  Given any utilization level $\varepsilon > 0$, we can find a task set, with utilization ε, which may not be feasibly scheduled according to FIFO.

  Example task set:

$$\left.\begin{array}{lll} T_1 & : & e_1 = \frac{\varepsilon}{2} p_1 \\ T_2 & : & p_2 = \frac{2}{\varepsilon} p_1 \\ & & e_2 = p_1 \end{array}\right\} \Rightarrow U = \varepsilon$$



Optimality of EDF for Periodic Systems

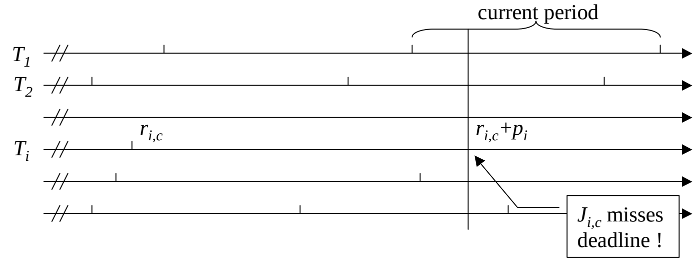
- Theorem: A system of independent preemptable tasks with relative deadlines equal to their periods is feasible *iff* their total utilization is less or equal 1 .
- Proof: *only-if*: obvious

  if : find algorithm that produces feasible schedule of any system with total utilization not exceeding 1.

  Try EDF.
- We show: If EDF fails to find feasible schedule, then the total utilization must exceed 1.
- Assumptions:
  - At some time t, Job $J_{i,c}$ of Task T_i misses its deadline.
  - *WLOG*: if more than one job have deadline t, break tie for $J_{i,c}$.

Optimality of EDF (cont)

- Case 1: Current period of every task begins at or after $r_{i,c}$.
- Case 2: Current period of some task my start before $r_{i,c}$.
- Case 1:



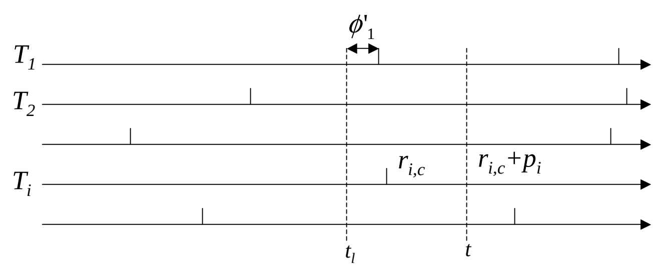
- Current jobs other than $J_{i,c}$ do not execute before time t.

$$\begin{aligned} t &< \frac{(t-\phi_i)e_i}{p_i} + \sum_{k \neq i} \left\lfloor \frac{t-\phi_k}{p_k} \right\rfloor e_k \\ &\leq t \cdot \frac{e_i}{p_i} + t \cdot \sum_{k \neq i} \frac{e_k}{p_k} \\ &= t \cdot U \end{aligned}$$

$$\Rightarrow U > 1$$

Optimality of EDF (cont 2)

- Case 2: Some current periods start before $r_{i,c}$.
- Notation: T: Set of all tasks.
  T': Set of tasks where current period starts before $r_{i,c}$.
  T-T': Set of tasks where current period start at or after $r_{i,c}$.



- t_l : Last point in time before t when some current job in T' is executed.
- No current job is executed immediately after time t_l.
- Why? 1. All jobs in T' are done.
  2. Jobs in T-T' not yet ready.

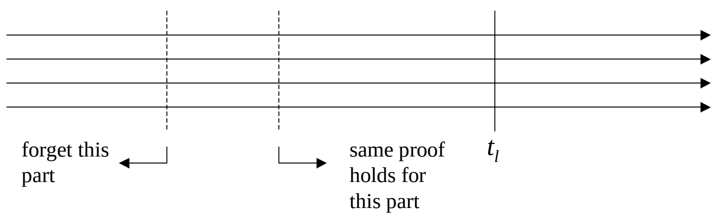
Case 2 (cont)

$$t - t_l < \frac{(t - t_l - \phi'_i)e_i}{p_i} + \sum_{T_k \in T - T'} \left\lfloor \frac{t - t_l - \phi'_k}{p_k} \right\rfloor e_k$$

$$\leq (t - t_l)\frac{ei}{pi} + (t - t_l)\sum_{T_k \in T - T'} \frac{e_k}{p_k} \leq (t - t_l)U$$

$$\Rightarrow U > 1$$

- What about assumption that processor never idle?

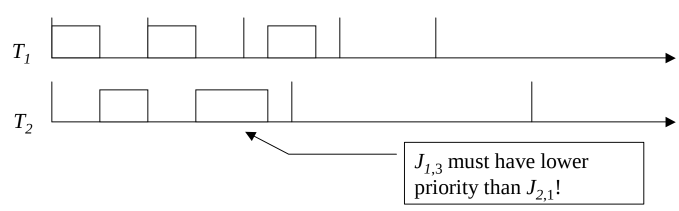


Q.E.D.

What about Static Priority?

- Static-Priority is not optimal!
- Example:

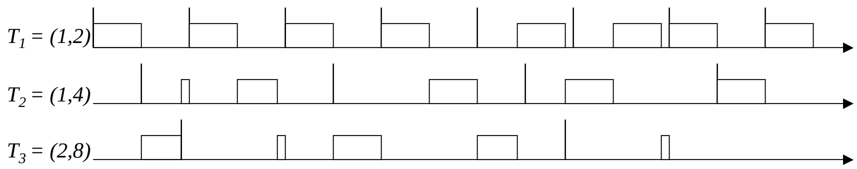
$$\left.\begin{array}{l} T_1 = (\ 2,\ 1,\ 2\ ) \\ T_2 = (\ 5,\ 2.5,\ 5\ ) \end{array}\right\} \quad U = \frac{e_1}{p_1} + \frac{e_2}{p_2} = 1 \leq 100\%$$



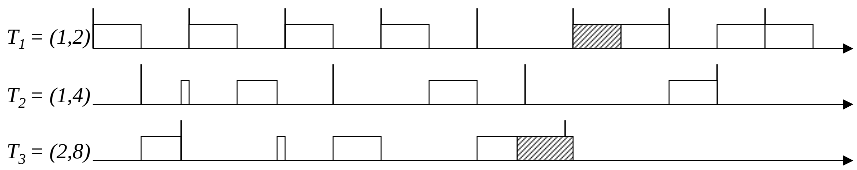
- So: Why bother with static-priority?
  - simplicity
  - predictability

Unpredictability of EDF Scheduling

- Over-running jobs hold on to their priorities
- Example:

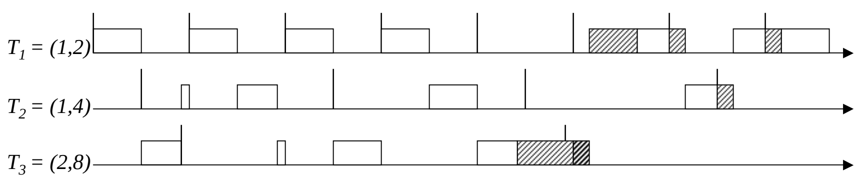


Normal Operation

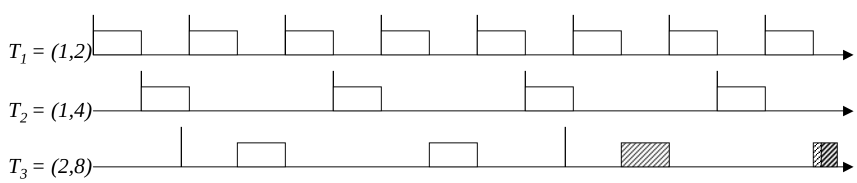


T3 over-runs by a bit more than one time unit

Unpredictability of EDF Scheduling (II)



T3 over-runs for a bit longer....



The same situation using Rate-Monotonic Scheduling:
high-priority tasks are protected

Schedulability Bounds for Static-Priority

- Simply-Periodic Workloads:

  Simply-Periodic: A set of tasks is simply periodic if, for every pair of tasks, one period is multiple of other period.

- Theorem: A system of simply periodic, independent, preemptable tasks whose relative deadlines are equal to their periods is schedulable according to RM *iff* their total utilization does not exceed 100%.

- Proof: Assume T_i misses deadline at time t.
  t is integer multiple of p_i.
  t is also integer multiple of $p_k, \forall p_k < p_i$.

  => total time to complete jobs with deadline t:

  Utilization due to i highest-priority tasks

  If job misses deadline, then $$\sum_{k=1}^{i} \frac{t \cdot e_k}{p_k} = t \cdot U_i = t \cdot \sum_{k=1}^{i} \frac{e_k}{p_k}$$

  $$U_i > 1 \Rightarrow U > 1.$$

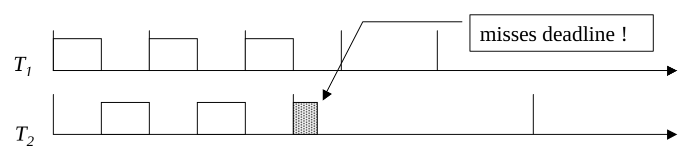
  Q.E.D.

Schedulable Utilization of Tasks with $D_i = p_i$, Using Rate-Monotonic Algorithm

- Theorem: [Liu&Layland '73] A system of n independend, preemptable periodic tasks with $D_i = p_i$ can be feasibly scheduled by the RM algorithm if its total utilization U is less or equal to

  $$U_{RM}(n) = n(2^{1/n} - 1)$$

- Why not 1.0? $T_1 = (2, 1, 2)$
  $T_2 = (5, 2.5, 5)$



- Proof: First, show that theorem is correct for special case where longest period $p_n < 2p_1$ (p_1 = shortest period).
  Will remove this restriction later.

Proof of Liu&Layland

- General idea: Find the <u>most-difficult-to-schedule</u> system of n tasks among all <u>difficult-to-schedule</u> systems of n tasks.

- **Difficult-to-schedule**: Fully utilizes processor for some time interval. Any increase in execution time would make system unschedulable.
- **Most-difficult-to-schedule**: system with lowest utilization among difficult-to-schedule systems.

- Each of the following <u>4 steps</u> brings us closer to this system.

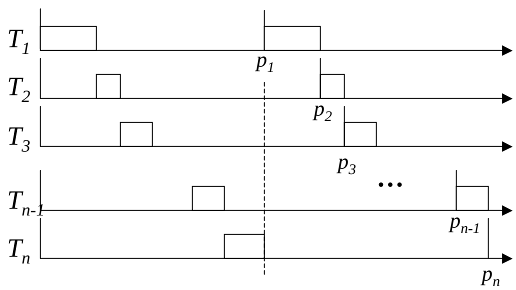
- <u>Step 1:</u> Identify phases of tasks in most-difficult-to-schedule system.

  System must be in-phase. (talk about this later)

Proof of Liu&Layland (cont)

- Step 2: Choose relationship between periods and execution times. Hypothesize that parameters of MDTS system are thus related.

- Confine attention to first period of each task.
- Tasks keep processor busy until end of period p_n.



$$e_k = p_{k+1} - p_k$$
$$e_n = p_n - 2\sum_{k=1}^{n-1} e_k$$

call this <u>Property A</u>

Proof Liu&Layland (cont)

- Step 3: Show that any set of D-T-S tasks that are not related according to Property A has higher utilization.
- What happens if we deviate from Property A?
- Deviate one way: Increase execution of some high-priority task by ε:

$$e'_1 = e_1 + \varepsilon = p_2 - p_1 + \varepsilon$$

Must reduce execution time of some other task:

$$e'_k = e_k - \varepsilon$$

$$U' - U = \frac{e_1'}{p_1} + \frac{e'_k}{p_k} - \frac{e_1}{p_1} - \frac{e_k}{p_k} = \underbrace{\frac{\varepsilon}{p_1} - \frac{\varepsilon}{p_k}}_{>0}$$

Proof Liu&Layland (cont)

- Deviate other way:

Reduce execution time of some high-priority tasks by ε:

$$e''_1 = e_1 - \varepsilon = p_2 - p_1 - \varepsilon$$

Must increase execution time of some lower-priority task:

$$e''_k = e_k + 2\varepsilon$$

$$U'' - U = \underbrace{\frac{2\varepsilon}{p_k} - \frac{\varepsilon}{p_1}}_{>0}$$

Proof Liu&Layland (cont)

- Step 4: Express the total utilization of the M-D-T-S task system (which has Property A).
- Define $gi := \frac{p_n - p_i}{p_i} \Rightarrow \begin{cases} e_i &= g_i p_i - g_{i+1} p_{i+1} \\ e_n &= p_n - 2 g_1 p_1 \end{cases}$

$$U = \sum_{i=1}^{n} \frac{e_i}{p_i} = \sum_{i=1}^{n-1} \left\{ g_i - g_{i+1} \frac{p_{i+1}}{p_i} \right\} + 1 - 2g_1 \frac{p_1}{p_n} = 1 + g_1 \frac{g_1 - 1}{g_1 + 1} + \sum_{i=2}^{n-1} g_i \frac{g_i - g_{i-1}}{g_i + 1}$$

- Find least upper bound on utilization: Set first derivative of U with respect to each of g_i's to zero:

$$\frac{\partial U}{\partial g_i} = \frac{(g_j^{\,2} + 2g_j - g_{j-1})}{(g_j + 1)^2} - \frac{g_{j+1}}{g_{j+1} + 1} = 0$$

for $j=1,2,3,\ldots,n-1$

$$g_j = 2^{(n-j)/n} - 1$$

$$\Rightarrow U = n(2^{1/n} - 1).$$

Q.E.D.

Period Ratios > 2

- We show:
  1. Every D-T-S task system ***T*** with period ratio > 2 can be transformed into D-T-S task system ***T'*** with period ratio <= 2.
  2. The total utilization of the task set decreases during the transformation step.
- We can therefore confine search to systems with period ratio < 2.

- 1. Transformation ***T-T'***:

```
while ∃T_k with l·p_k < p_n ≤ (l+1)p_k   (l ≥ 2)
  T_k(p_k,e_k)  →  (l·p_k, e_k)
  T_n(p_n,e_n)  →  (p_n, e_n+(l-1)e_k)
end
```

- Compare utilizations:

$$U - U' = \frac{e_k}{p_k} + \frac{e_n}{p_n} - \frac{e_k}{l \cdot p_k} - \frac{e_n + (l-1)e_k}{p_n} = \frac{e_k}{p_k} - \frac{e_k}{l \cdot p_k} - \frac{(l-1)e_k}{p_n}$$

$$= \left(\frac{1}{l \cdot p_k} - \frac{1}{p_n} \right)(l-1)e_k > 0$$

Q.E.D.

Regarding that Little Question about the Phasing...

- Definition:

  [Critical Instant]
  [Liu&Layland] If the maximum response time of all jobs in T_i is less than D_i, then the job of T_i released in the critical instant has the maximum response time.
  [Baker] If the response time of some jobs in T_i exceeds D_i, then the response time of the job release during the critical instant exceeds D_i.

- Theorem:

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

Proof (informal)

- Assume: Theorem holds for $k<i$.
- WLOG: $\forall k < i : \phi_k = 0$, and we look at $J_{i,1}$:
- Observation: The completion time of higher-priority jobs is independent of the release time of $J_{i,1}$.
- Therefore: The sooner $J_{i,1}$ is released, the longer it has to wait until it is completed.

Q.E.D.

Proof 2 (less informal)

- WLOG: $\min\{\phi_k | k = 1,\ldots,i\} = 0$
- Observation: Need only consider time processor is busy executing jobs in $T_1, T_2, \ldots, T_{i-1}$ before ϕ_i.
  If processor idle or executes lower-priority jobs, ignore that portion of schedule and redefine the ϕ_k's.
- During interval $[\phi_k, \phi_i + R_{i,1}]$ a total of $\left\lceil \frac{R_{i,1} + \phi_i - \phi_k}{p_k} \right\rceil$ jobs in T_k become ready for execution.
- so: $R_{i,1} + \phi_i = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_{i,1} + \phi_i - \phi_k}{p_k} \right\rceil e_k$

  $R_{i,1}$ is smallest solution, if such a solution exists.
- and: $R_{i,1} = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_{i,1} + \phi_i - \phi_k}{p_k} \right\rceil e_k - \phi_i$

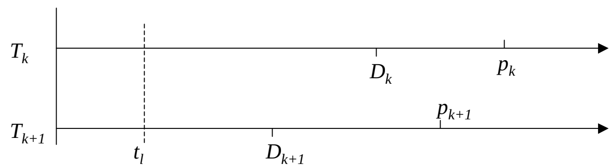
Why Utilization-Based Tests?

- If no parameter ever varies, we could use simulation.
- But:
  - Execution times may be smaller than e_i
  - Interrelease times may vary.
- Tests are still robust.
- Useful as methodology to define execution times or periods.

Optimality of Deadline-Monotonic, Rate-Monotonic

- Theorem: If a task set can be feasibly scheduled by some static-priority algorithm, it can be feasibly scheduled by DM.

- Proof:
  - Assume: A feasible schedule exists for a task set T. The priority assignment is $T_1, T_2, \ldots, T_n$.
    For some k, we have $D_k > D_{k+1}$.
  - We show that we can swap the priority of T_k and T_{k+1} and the resulting schedule remains feasible.



Time-Demand Analysis

- Compute total demand on processor time of job released at a critical instant and by higher-priority tasks as function of time from the critical instant.
- Check whether demand can be met before deadline.
- Determine whether T_i is schedulable:
  - Focus on a job in T_i, suppose release time is critical instant of T_i:
    $w_i(t)$: Processor-time demand of this job and all higher-priority jobs released in (t_0, t):

$$w_i(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k$$

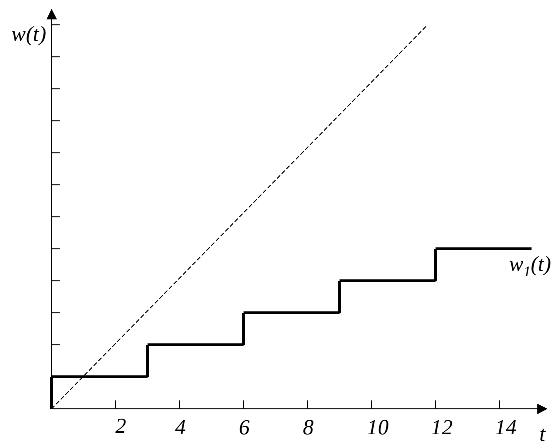
- This job in T_i meets its deadline if, for some

$$t_1 \le D_i \le p_i \quad : \quad w_i(t_1) \le t_1$$

- If this does not hold, job cannot meet its deadline, and system of tasks is not schedulabe by given static-priority algorithm.

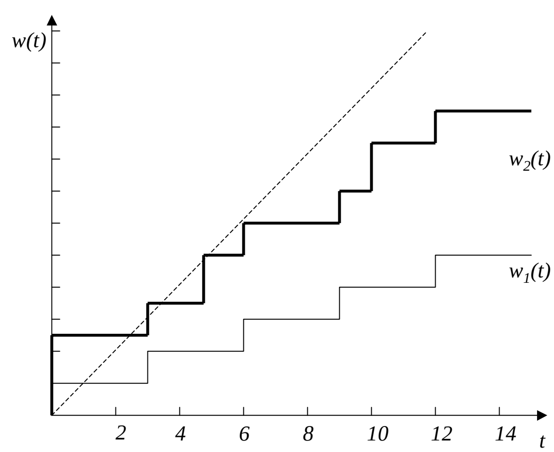
Example

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



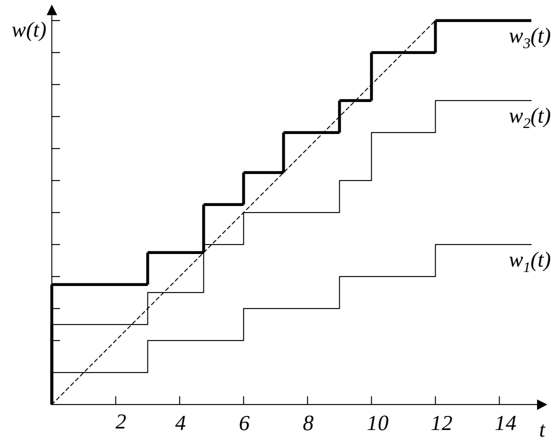
Example

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



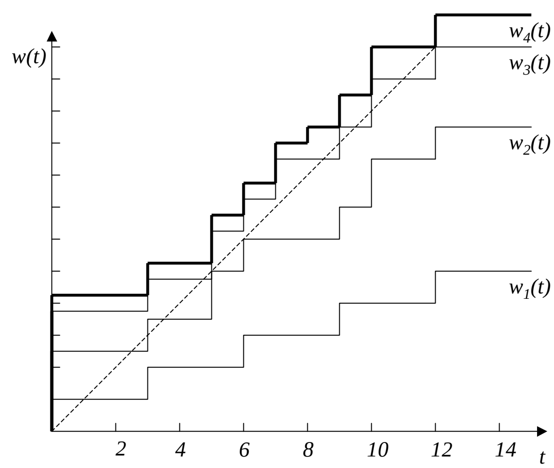
Example

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



Example

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



Practical Factors

- Non-Preemptable Portions (*)
- Self-Suspension of Jobs (*)
- Context Switches (*)
- Insufficient Priority Resolutions (Limited Number of Distinct Priorities)
- Time-Driven Implementation of Scheduler (Tick Scheduling)
- Varying Priorities in Fixed-Priority Systems

Practical Factors I: Non-Preemptability

- Jobs, or portions thereof, may be non-preemptable.
- Definition: **[non-preemptable portion]**
  ρ_i : largest non-preemptable portion of jobs in T_i.
- Definition: **[blocked job]**
  A job is said to be **blocked** if it is prevented from executing by lower-priority job. (priority-inversion)
- When testing schedulability of a task T_i, we must consider
  - higher-priority tasks

  and
  - non-preemptable portions of lower-priority tasks

Analysis with Non-Preemptable Portions

- Definition: **[blocking time]**
  The **blocking time** b_i of Task T_i is the longest time by which any job of T_i can be blocked by lower-priority jobs:

$$b_i = \max_{i+1 \le k \le n} \rho_k$$

- Time-demand function with blocking:

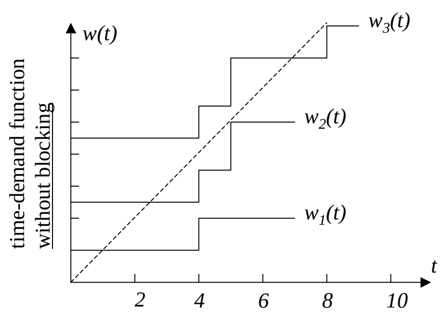
$$w_i(t) = e_i + b_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k$$

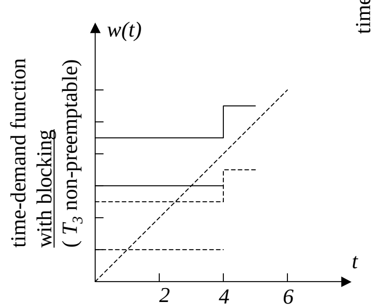
- Utilization bounds with blocking:
  test one task at a time:

$$\frac{e_1}{p_1} + \frac{e_2}{p_2} + \ldots + \frac{e_i + b_i}{p_i} = \sum_{k=1}^{i} \frac{e_k}{p_k} + \frac{b_i}{p_i} \le U_{RM}(i)$$

Non-Preemptability: Example

$T_1 = (4, 1)$
$T_2 = (5, 1.5)$
$T_3 = (9, 2)$



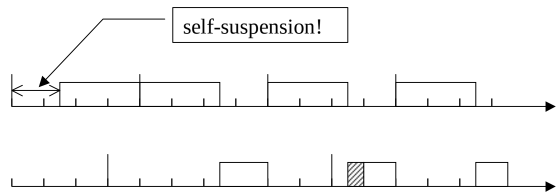


Practical Factors II: Self-Suspension

- Definition: **[Self-Suspension]** Self-suspension of a job occurs when the job waits for an external operation to complete (RPC, I/O operation).
- Assumption: We know the maximum length of external operation; i.e., the duration of self-suspension is bounded.
- Example:

$T_1 = (\phi_1=0, p_1=4, e_1=2.5)$

$T_2 = (\phi_2=3, p_2=7, e_2=2.0)$



- Analysis: b_i^{SS} : Blocking time of T_i due to self-suspension.

$$b_i^{SS} = \text{max. self-suspension time of } T_i + \sum_{k=1}^{i-1} \min(e_k, \text{max. self-suspension time of } T_k)$$

Self-Suspension with Non-Preemptable Portions

- Whenever job self-suspends, it loses the processor.
- When tries to re-acquire processor, it may be blocked by tasks in non-preemptable portions.
- Analysis:
  - b^{NP}_i: Blocking time due to non-preemptable portions
  - K_i: Max. number of self-suspensions
  - b_i: Total blocking time

$$b_i = b^{SS}_i + (K_i + 1)\, b^{NP}_i$$

Practical Factors III: Context Switches

- Definition:

  > **[Job-level fixed priority assignment]**
  > In a job-level fixed priority assigment, **e**ach job is given a fixed priority for its entire execution.

- Case I: No self-suspension
  - In a job-level fixed-priority system, each job preempts at most one other job.
  - Each job therefore causes at most two context switches
  - Therefore: Add the context switch time twice to the execution time of job: $e_i = e_i + 2\ CS$
- Case II: Self-suspensions can occur
  - Each job suffers two more context switches each time it self-suspends
  - Therefore: Add more context switch times appropriately:

    $e_i = e_i + 2\ (K_i + 1)\ CS$