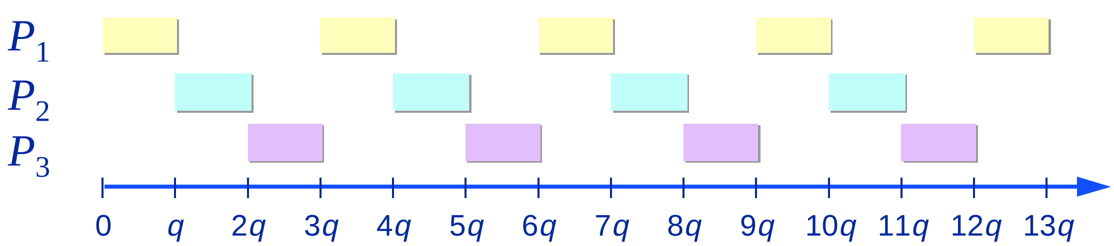
Proportional Share Resource Allocation
Outline

- Fluid-flow resource allocation models
  - » Packet scheduling in a network
- Proportional share resource allocation models
  - » CPU scheduling in an operating system
- On the duality of proportional share and traditional real-time resource allocation models
  - » How to make a provably real-time general purpose operating system

Proportional Share Resource Allocation
Concept

- Processes are allocated a *share* of a shared resource
  - » a *relative percentage* of the resource’s total capacity
- Processes make progress at a uniform rate according to their share
- OS Example — time sharing tasks allocated an equal share ($1/n^{th}$) of the processor’s capacity
  - » round robin scheduling, fixed size quantum



Proportional Share Resource Allocation
Formal model

- Processes are allocated a *share* of the processor's capacity
  - » Process i is assigned a ***weight*** w_i
  - » Process i's *share* of the CPU at time t is

$$f_i(t) = \frac{w_i}{\sum_j w_j}$$

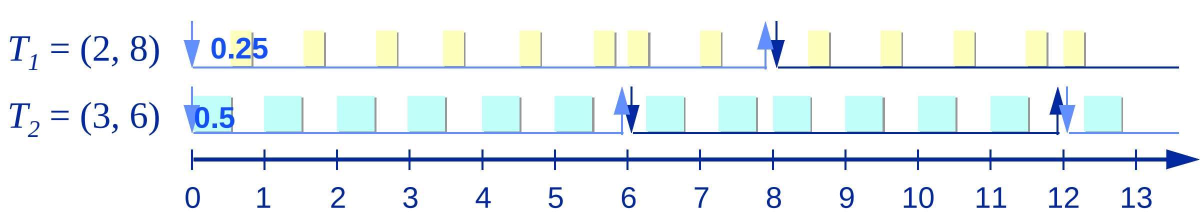
- If processes' weights remain constant in $[t_1, t_2]$ then process i receives

$$S_i(t_1,t_2) = \int_{t_1}^{t_2} f_i(\tau)\, dt = \frac{w_i}{\sum_j w_j}(t_2 - t_1)$$

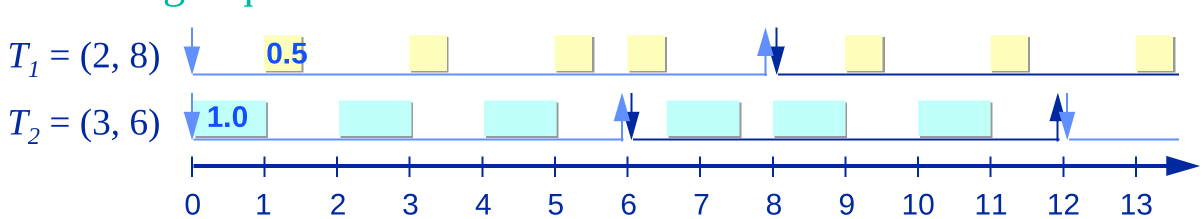
units of execution time in $[t_1, t_2]$

Proportional Share Resource Allocation
Real-time scheduling example

- Periodic tasks allocated a share equal to their processor utilization c/p
  - » round robin scheduling with infinitesimally small quantum

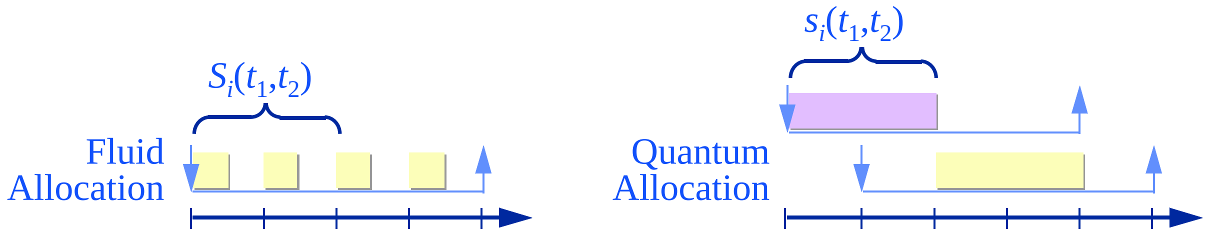


  - » with integer quantum = 2 time units



Proportional Share Resource Allocation

Task scheduling metrics & goals



- Schedule tasks such that their performance is as close as possible to their performance in the *fluid* system

$$S_i(t_1,t_2) = \int_{t_1}^{t_2} \frac{w_i}{\sum_j w_j}\, d\tau$$

- Define the allocation error for task *i* at time *t* as

$$lag_i(t) = S_i(t_0,t) - s_i(t_0,t)$$

Proportional Share Resource Allocation

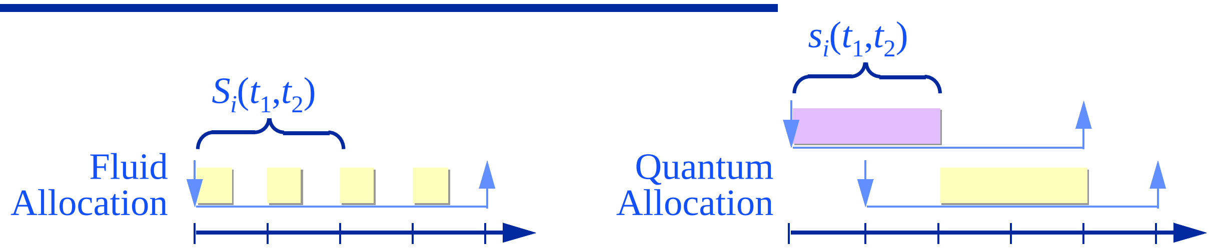
Task scheduling metrics & goals

$$lag_i(t) = S_i(t_0,t) - s_i(t_0,t)$$

- Because allocation is quantum-based, tasks can be either *behind* or *ahead of* the fluid schedule
  - » if lag is *negative* then a task has received *more* service time than it would have received in the fluid system
  - » if lag is *positive* then a task has received *less* service time than it would have received in the fluid system

Proportional Share Resource Allocation

Task scheduling metrics & goals



$$lag_i(t) = S_i(t_0,t) - s_i(t_0,t)$$

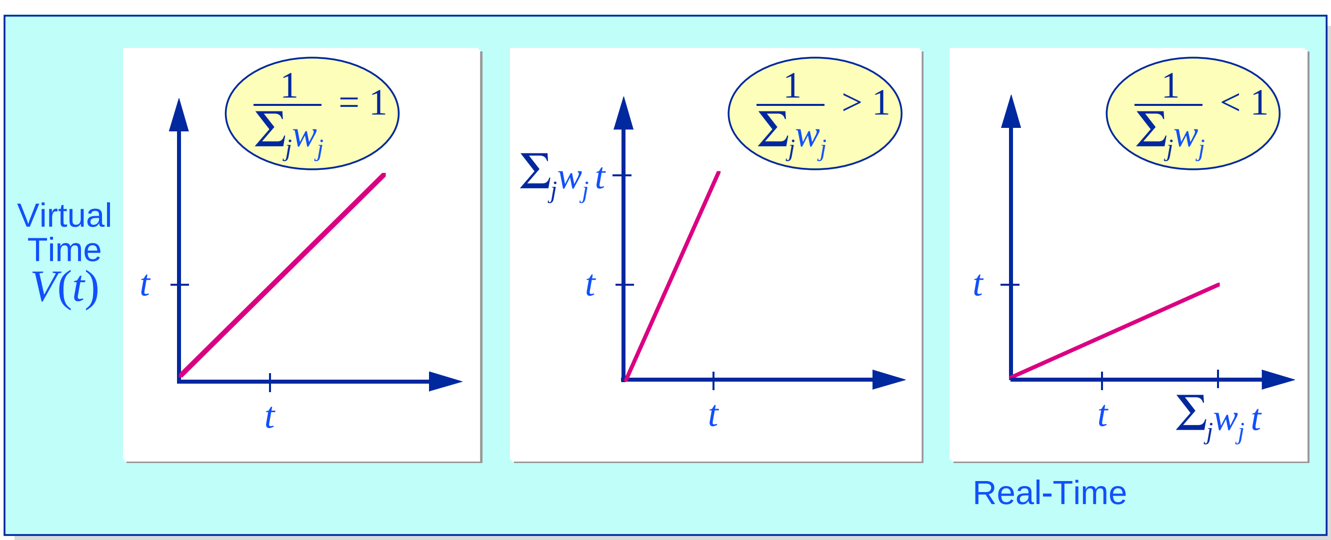
- Goal: Schedule tasks such that their performance is as close as possible to that in the *fluid* system
- Schedule tasks such that the lag is:
  - » bounded, and
  - » minimized over all tasks and time intervals

Scheduling to Bound Lag

The virtual time domain

- Tasks are scheduled in a *virtual time* domain

$$V(t) = \int_0^t \frac{1}{\Sigma_j w_j} \, d\tau$$

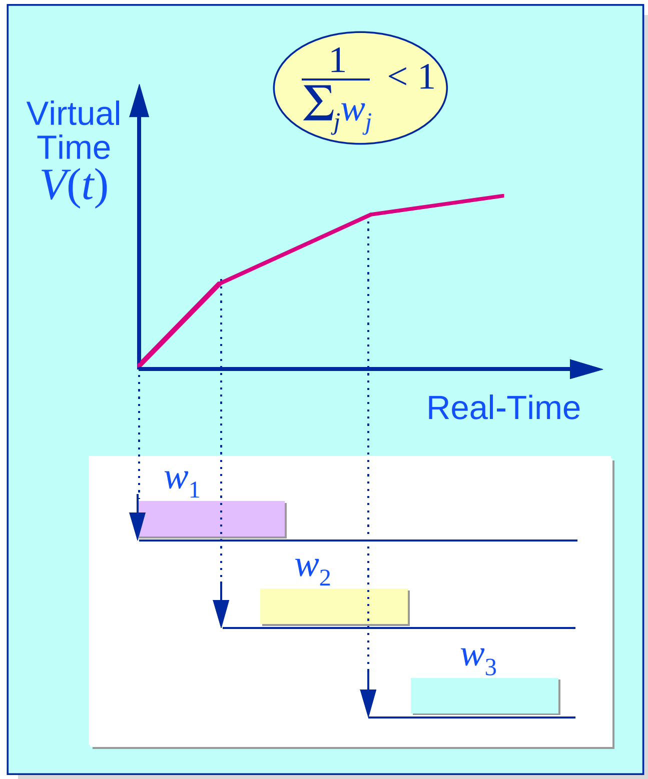


Scheduling to Bound Lag

The virtual time domain

- Slope of virtual time changes are tasks enter and leave the system

$$V(t) = \int_0^t \frac{1}{\Sigma_j w_j} \, d\tau$$



Scheduling to Bound Lag

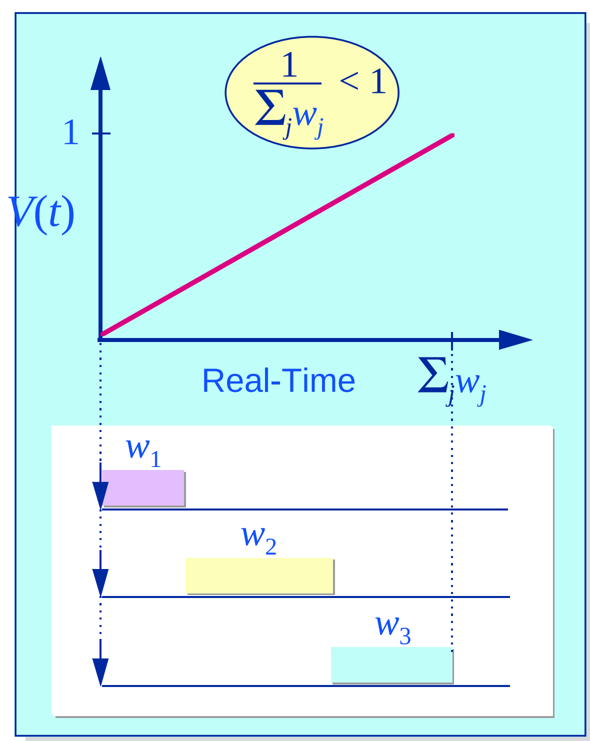
The virtual time domain

- Task's execute for w_i real-time time units in each virtual-time time unit
  - » Thus ideally, task *i* executes for

$$S_i(t_1,t_2) = w_i \int_{t_1}^{t_2} \frac{1}{\Sigma_j w_j} \, d\tau$$

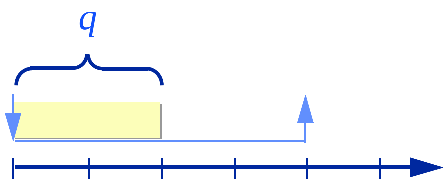
$$= (V(t_2) - V(t_1))w_i$$

time units in any real-time interval



Scheduling to Bound Lag

Virtual time scheduling principles



$$lag_i = S_i(t_1,t_2) - s_i(t_1,t_2)$$

$$S_i(t_1,t_2) = \int_{t_1}^{t_2} \frac{w_i}{\sum_j w_j} \, d\tau$$

- Schedule tasks only when their lag is non-negative
  - » If a task with negative lag makes a request for execution at time t, it is not considered until a future time t' when $lag(t') \geq 0$
  - » Let $e > t$ be the earliest time a task can be scheduled
    - ❖ the time at which

$$S(t_i, e) = s(t_i, t)$$

  - » This time occurs in the virtual time domain at time

$$S(t_i, e) = s(t_i, t)$$
$$(V(e) - V(t_i))w_i = s(t_i, t)$$
$$V(e) = V(t_i) + s(t_i,t)/w_i$$

Scheduling to Bound Lag

Virtual time scheduling principles

- Task requests should not be considered before their "eligible" time e
- Requests should be completed by *virtual* time

$$V(d) = V(e) + c_i/w_i$$

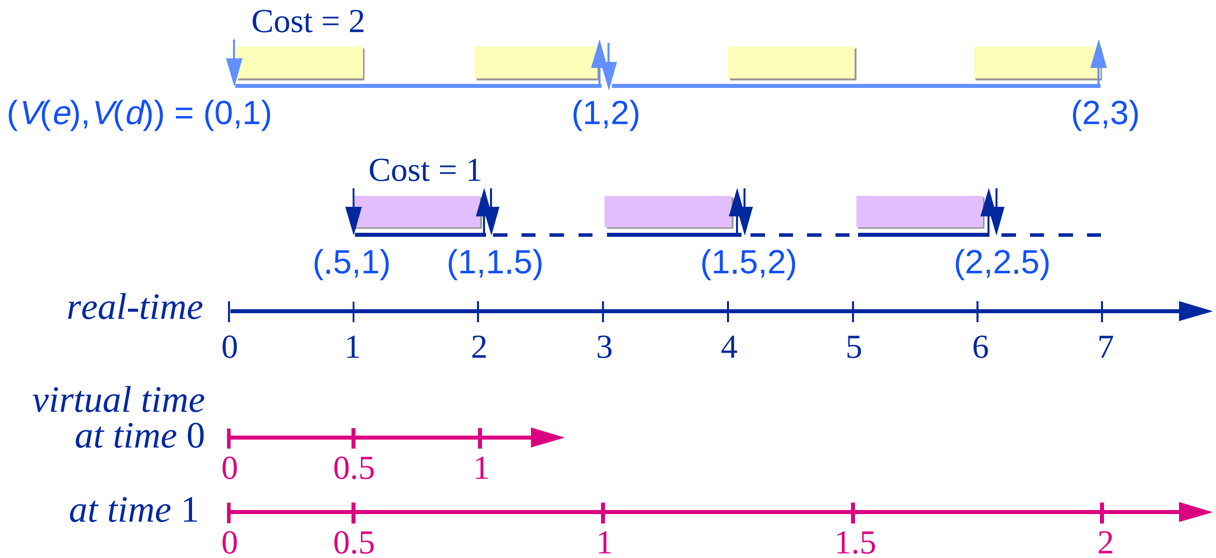
  - » where c_i is the cost of executing the request

- Our candidate scheduling algorithm: *Earliest Eligible Virtual Deadline First* (*EEVDF*)

> At *each scheduling point, a new quantum is allocated to the eligible process with the nearest earliest virtual deadline*

Earliest Eligible Virtual Deadline First Scheduling

Example: Two tasks with equal weight = 2 (q = 1)



$$V(t) = \int_0^t \frac{1}{\sum_j w_j} \, d\tau \qquad V(e) = V(t_0) + s(t_0, t)/w$$

$$V(d) = V(e) + c/w$$

Proportional Share Resource Allocation

Issues

- How to use proportional share scheduling for real-time computing
  - » How to ensure deadlines are respected in the real-time domain
  - » Bounding the allocation error
- Practical considerations — Maintaining virtual time
  - » Policing errant tasks
  - » Dealing with tasks that complete “early”

Using Proportional Share Allocation For Real-Time Computing

- Deadlines in a proportional share system ensure *uniformity of allocation*, not *timeliness*

- Weights are used to allocate a *relative* fraction of the CPU's capacity to a task

$$f_i(t) = \frac{w_i}{\Sigma_j w_j}$$

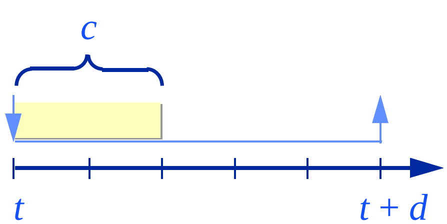
- Real-time tasks require a *constant* fraction of a resource's capacity

$$f_i(t) = \frac{c_i}{p_i}$$

- Thus real-time performance can be achieved by adjusting weights dynamically so that the share remains constant

Supporting Real-Time Computing

Dynamically adjusting weights



$$V(e) = V(t) + s(t, t)/w_i = V(t)$$
$$V(d) = V(e) + c/w_i$$

- Consider task i that arrives at time t with a deadline at time $t + d$
  - » In the interval $[t, t+d]$ the task requires a share of the processor equal to c/d

$$\frac{w_i}{\Sigma_j w_j} = \frac{c}{d}$$

$$w_i = \frac{c}{d}\left(\Sigma_{j \neq i} w_j + w_i\right)$$

$$w_i = \frac{\frac{c}{d}\ \Sigma_{j \neq i} w_j}{1 - \frac{c}{d}}$$

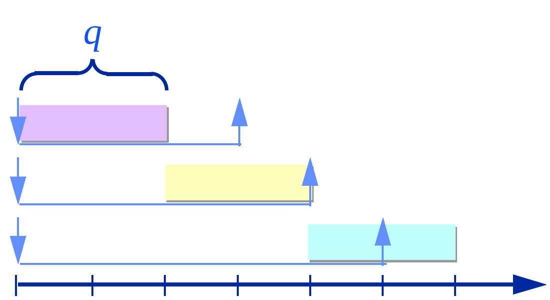
Supporting Real-Time Computing
Admission control

- If real-time tasks require a fixed share of the CPU's capacity, only a finite number of tasks may be guaranteed to execute real-time

- Admission criterion:
  - » a simple sufficient condition — $\sum_i \frac{c_i}{d_i} \leq 1$
  - » a necessary condition??
    - ❖ it depends...

Supporting Real-Time Computing
Bounding the allocation error

- Is a task guaranteed to complete before its deadline?

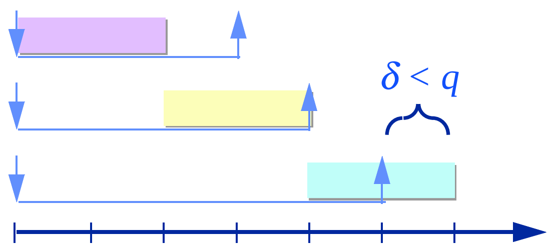


- How late can a task be?
  - » Theorem: By at most q time units

Supporting Real-Time Computing

Bounding the allocation error

- Consider a task system wherein tasks always terminate with zero lag
- Theorem: *Let d be the current deadline of a request made by task k. Let f be the actual time the request is fulfilled.*
  - » $f < d + q$ (the request is fulfilled no later than time $d + q$)
  - » if $f > d$ then for all t, $d \leq t < f$, $lag_k(t) < q$



Bounding the Allocation Error

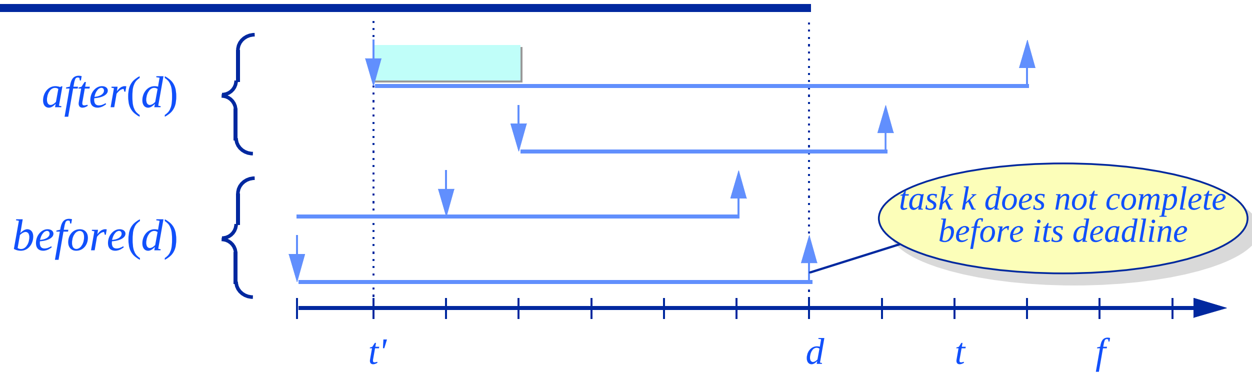
Some properties of *lag*(*t*)

$$lag_i(t) = S_i(t_0,t) - s_i(t_0,t) \qquad V(e) = V(t_0) + s(t_0, t)/w \qquad V(d) = V(e) + c/w$$

- $lag_i(t) < 0$ implies that task i has received “too much” service, $lag_i(t) > 0$ implies that it has received “too little”
- Eligibility law
  - » If a task has non-negative lag then it is eligible
- Lag conservation law
  - » For all t, $\sum_i lag_i(t) = 0$
- Missed deadline law
  - » If a task misses its deadline d then $lag_i(t)$ = *remaining required service time*
- Preserved lateness law
  - » If a task that misses a deadline at d completes execution at T, then
    - ❖ for all t, $T \geq t > d$, $lag_i(t) > 0$
    - ❖ $lag_i(t)$ > *remaining service time*

Bounding the Allocation Error

Proof sketch of Theorem

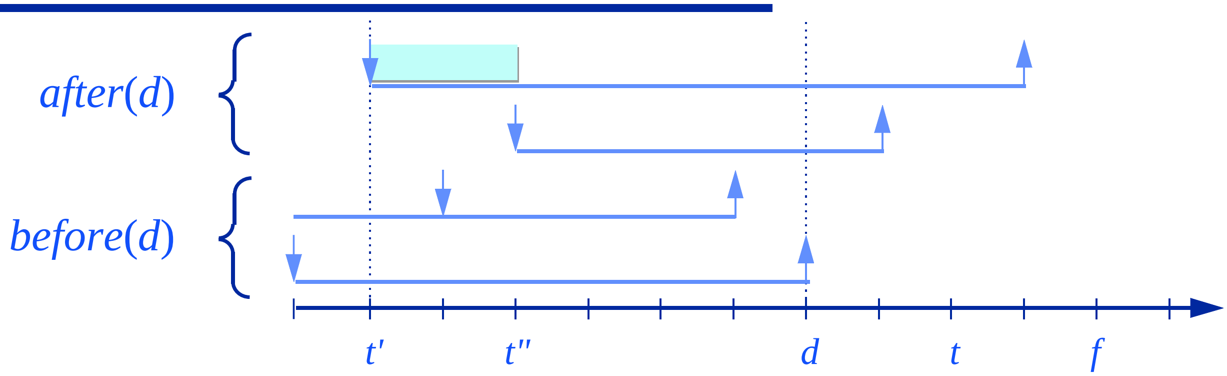


- Let $t' < f$ be the latest time a task with deadline after d receives a quantum
- At any time t partition tasks into those with requests with deadlines before d and those with deadlines after d

$$\Sigma_{i \text{ in } before(t')} lag_i(t') < 0 \qquad \Sigma_{i \text{ in } after(t')} lag_i(t') > 0$$

Bounding the Allocation Error

Proof sketch of Theorem



- $\Sigma_{i \text{ in } before(t')} lag_i(t') < 0, \quad \Sigma_{i \text{ in } after(t')} lag_i(t') > 0$
- $\Sigma_{i \text{ in } after(t)} lag_i(t) > -q, \quad \Sigma_{i \text{ in } before(t)} lag_i(t) < q, \quad lag_k(t) < q$
- $\Sigma_{i \text{ in } before(d)} lag_i(d) < q,$

  This implies that all requests in *before*(d) must be completed by time $d + q$

Supporting Real-Time Computing
Bounding the allocation error

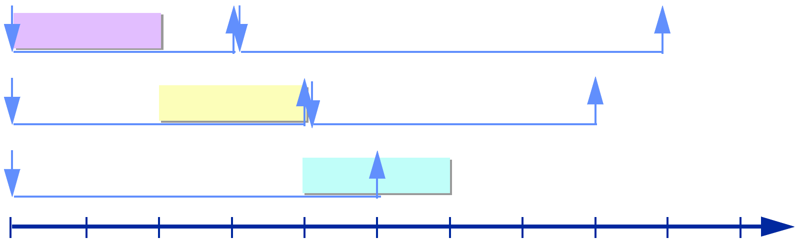
- Theorem: *Let c be the size of the current request of task k. Task k's lag is bounded by*

$$-c < lag_k(t) < max(c, q)$$

Proportional Share Resource Allocation
Issues

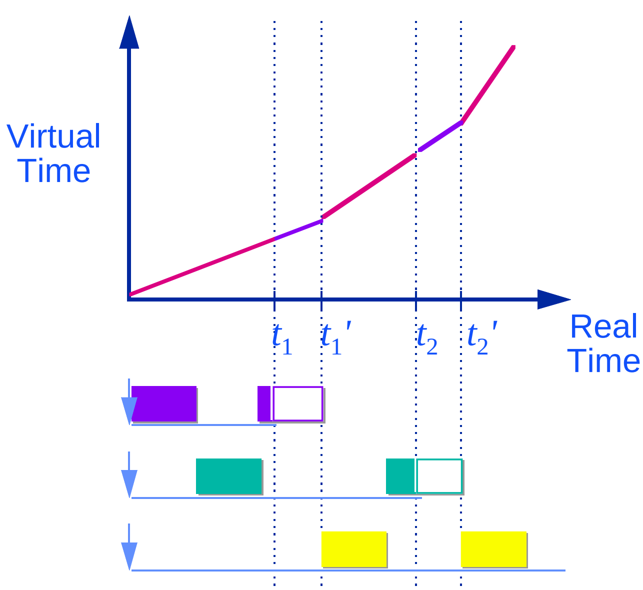
- How to use proportional share scheduling for real-time computing
  - » How to ensure deadlines are respected in the real-time domain
  - » Bounding the allocation error
- Practical considerations — Maintaining virtual time
  - » Policing errant tasks
  - » Dealing with tasks that complete "early"

Practical Considerations
Maintaining virtual time

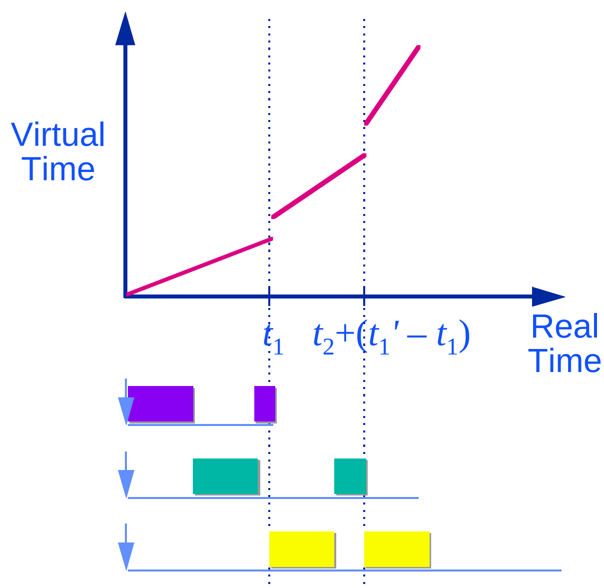


- What happens when a task completes after its deadline?
  - » Preserved lateness law: $\forall t$, $T \geq t > d$, $lag_i(t) > 0$
- If the task makes another request immediately, the request is eligible
- If the task terminates the total lag in the system is negative
  - » Lag conservation law requires that $\forall t$, $\Sigma_i lag_i(t) = 0$

Maintaining Virtual Time
A task terminates with positive lag



Ideal system



Actual system

Maintaining Virtual Time

A task terminates with positive lag

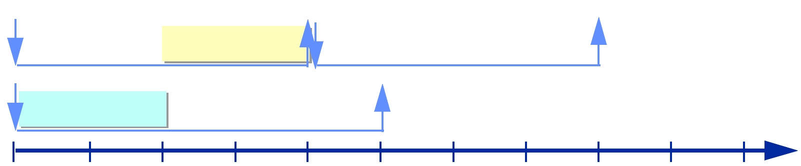
- When task k terminates with positive lag we must:
  - » update virtual time to the next point in time $V(t)$ at which $lag_k(t) = 0$
  - » update each task's lag to reflect the discontinuities in virtual time
- If t_k is the time a task with positive lag terminates, then

$$V(t_k) = V(t_k') + \frac{lag_k(t_k)}{\sum_{j \neq k} w_j}$$

$$lag_i(t_k) = lag_i(t_k') + w_i \frac{lag_k(t_k)}{\sum_{j \neq k} w_j}$$

Practical Considerations

Maintaining virtual time



- What happens when a task completes before its deadline?
  - » Task's lag will be negative
- If the task makes another request immediately, the request is ineligible
- If the task terminates, the termination can be delayed until the task's lag is 0
  - » If the task correctly estimated its execution time this will occur at the task's deadline
  - » Otherwise, this time may be either before or after its deadline

Practical Considerations
Policing tasks

- What happens when a task is not complete by its deadline but its lag is negative?
  - » The task under estimated its execution time
- Several alternatives:
  - » Have the operating system issue a new request on behalf of the task
  - » Issue a new request for the task but penalize it by reducing its weight
- In all cases, the "errant" task has *no* effect on the performance of other tasks!

Practical Considerations
Bounding the allocation error in practice

- Theorem: *Let c be the size of the current request of task k. Task k's lag is bounded by*

$$-c < lag_k(t) < max(c, q)$$

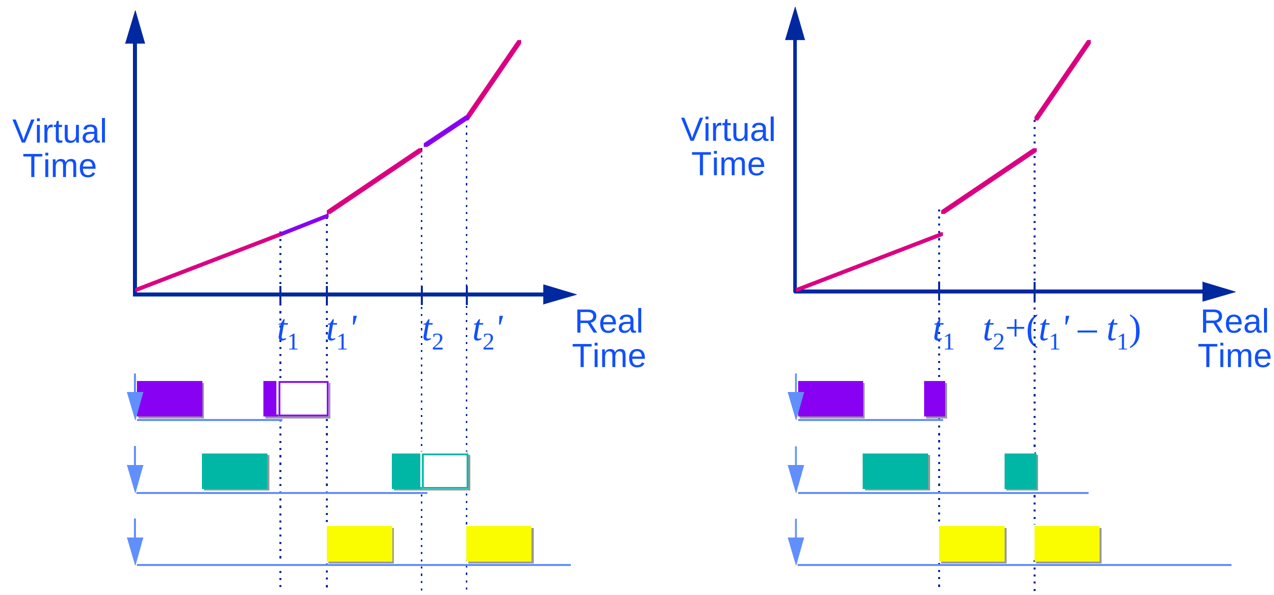
- Theorem: *If tasks terminate with positive lag then a task k's lag is bounded by*

$$-c < lag_k(t) < max(c_{max}, q)$$

where c_{max} is the largest request made by any task in the system

Practical Considerations
Bounding the allocation error in practice

- Theorem: *If tasks terminate with positive lag then a task k's lag is bounded by* $-c < lag_k(t) < max(c_{max}, q)$



Practical Considerations
Bounding the allocation error in practice

- Theorem: *If tasks terminate with positive lag then a task k's lag is bounded by* $-c < lag_k(t) < max(c_{max}, q)$
  - » Thus a trade-off exists between the size of a task's request (*i.e.*, scheduling overhead) and the accuracy of allocation

- Corollary: *If tasks requests are always less than a quantum then for all tasks k, then* $-q < lag_k(t) < q$

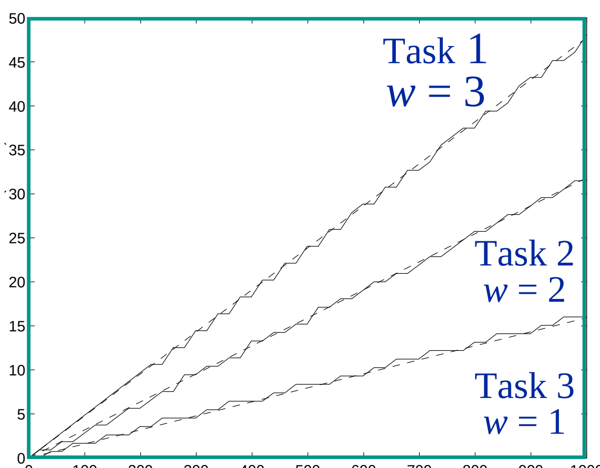
Experimental Evaluation

EEVDF Implementation in FreeBSD

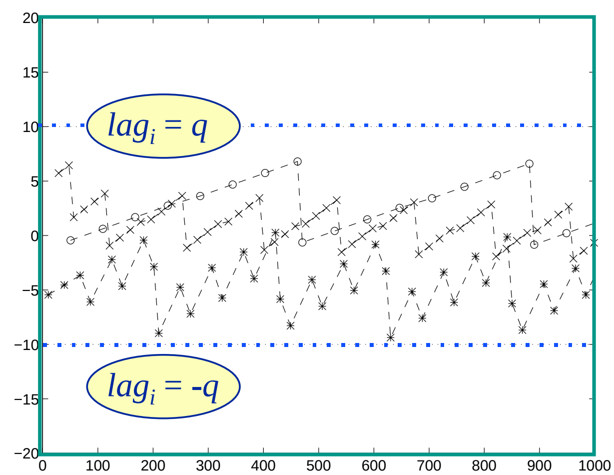
- Platform
  - » PC compatible, 75 Mhz Pentium processor, 16 MB RAM
- Implementation
  - » Replaced FreeBSD CPU scheduler
  - » Time quantum = 10 ms
- Experiments
  - » Non-real-time tasks making uniform progress
  - » Speeding up and slowing down task progress by manipulating weights
  - » Real-time execution (of non-real-time programs!)

Proportional Share Scheduling Example

Uniform allocation to non-real-time processes



Number of Dhrystone Iterations (x 1,000) *v.* Elapsed Time (secs)



Measured Lag *v.* Elapsed Time (secs)

Proportional Share Resource Allocation

Summary

- A “real-time” neutral model
  - » Supports both real-time and non-real-time
- EEVDF scheduling provides optimal lag bounds ($\pm q$)
  - » Allocation error & hence timeliness guarantees are as good as possible
- But maintaining virtual time is non-trivial
  - » Better than sorting but $O(n)$ in the worst case
- Unclear how to solve the integrated systems problem