CSCE 990: *Real-Time Systems*

Proportional Share Scheduling

Steve Goddard
goddard@cse.unl.edu

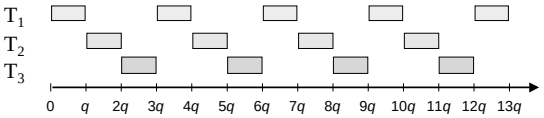
http://www.cse.unl.edu/~goddard/Courses/RealTimeSystems

Introduction

- We have looked at systems consisting exclusively of hard real-time tasks and systems consisting of real-time and non-real-time tasks.
  - » In both cases, we assumed that all jobs were scheduled according to normal real-time scheduling disciplines.
- We now look at another way of integrating real-time and non-real-time computing: Fairness
  - » Proportional Share Resource Allocation, Stoica *et. al.* (1996)
  - » P-fair Scheduling, Baruah *et. al*. (1996)
- We will show that the notion of Fairness can be used to execute jobs in real-time.

Proportional Share 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



Formal Allocation Model

- Processes are allocated a share of the processor's capacity:
  - » Process *i* is assigned a weight w_i
    - Note: w_i is a rational number. That is, it need not be an integer.
  - » Process *i*'s share of the CPU at time *t* is

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

    - Note: in the denominator of $f_i(t)$ we are summing over only the active processes in the system; all tasks in the case of a real-time system.

Formal Allocation Model

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

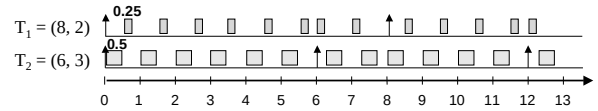
$$S_i(t_1,t_2) = \int_{t_1}^{t_2} f_i(\tau)dt = \frac{w_i}{\sum_j w_j}(t_2 - t_1)$$

  units of execution time in $[t_1, t_2]$ for a fluid-allocation system.

- Some comments on $S_i(t_1,t_2)$:
  - » The definition of the integral $S_i()$ is always correct.
  - » The assumption that weights remain constant over the interval lets us reduce the integral to a linear equation.
  - » Otherwise, we can generate a piecewise linear equation.

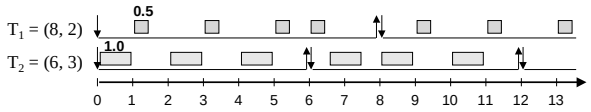
Real-Time Scheduling Example

- Ignoring weights for a moment, we note that Al Mok gave a simple proof (in his dissertation) that utilization ≤ 1 is a sufficient condition for schedulability when each task T_i is allocated a fraction of each time unit equal to its utilization e_i/p_i.
- Thus, we can schedule periodic tasks with a round robin scheduling algorithm if we assume an infinitesimally small quantum and allocate each task a share equal to their processor utilization e_i/p_i within each time unit.



Real-Time Scheduling Example

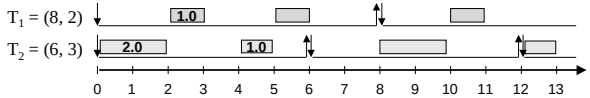
- Now consider the same task set with RR scheduling and a variable-sized quantum. Assume a round length of 2.
- Thus, the share of task T_i in each round (of length 2) is $2(e_i/p_i)$.



- Note: This slide and the rest of the presentation on Proportional Share uses a slightly different notation for the release and deadline of a task:
  - » down arrows represent releases (or a job entering the system).
  - » up arrows represent deadlines (or a job leaving the system).

Real-Time Scheduling Example

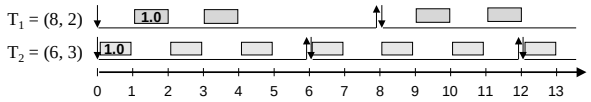
- Round robin scheduling with an integer, variable-sized quantum and a round length of 4:



- Notice that, in this and the last two examples, the processor is idle at least 25% of each round, even when there are ready tasks.
  - » In these examples, the scheduling algorithm is non-work-conserving.
- However, each process always receives its share in each round.

Real-Time Scheduling Example

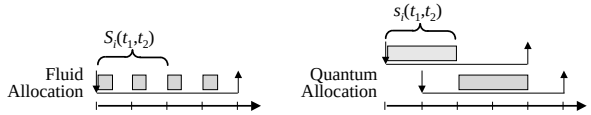
- A more realistic scheduling scenario would be round-robin scheduling with a fixed size quantum.
- Consider a fixed sized quantum of 1 and a round length of 2:



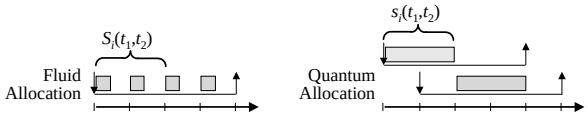
- Notice that T_1 is not scheduled in [4,8) because it has already completed execution.
  - » It received more that its share in [0,4) and less in [4,8)

Task Scheduling Goal

- Practical constraints require us to allocate the CPU with a fixed-sized quantum of q time units.
- Our goal is to approximate, as close as possible, a pure fluid-allocation system using quantum based scheduling.



Task Scheduling Metrics

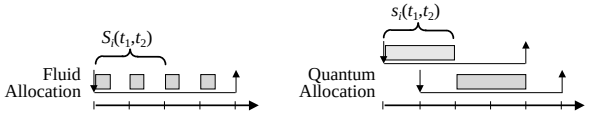


- The time allocated to task T_i in a pure fluid allocation during the interval $[t_1,t_2)$ is

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

- Let $s_i(t_1,t_2)$ be the time allocated to task T_i using a fixed quantum allocation in the interval $[t_1,t_2)$.

Quantifying the Allocation Error

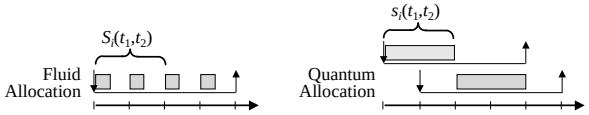


- To minimize the error incurred because we are allocating the CPU with a fixed quantum, we need to schedule tasks such that their performance is as close as possible to their performance in the *fluid* system.
- Define the allocation error for T_i at time t as

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

where t_0 is the time a task is initially released.

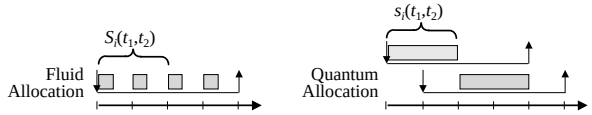
What Does $lag_i(t)$ Mean?



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

- 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.

Real-Time Computing using Proportional Share



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

- 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

Issues

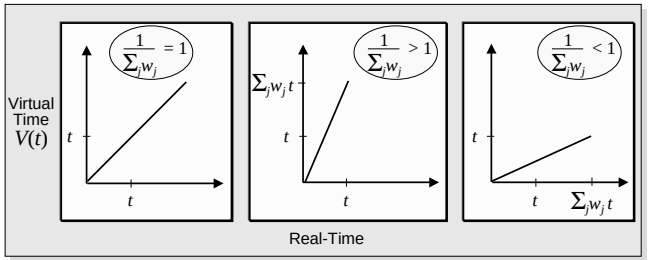
- What happens when a task's period/deadline is small relative to the size of a quantum?
- A large quantum is more efficient, but it introduces a larger allocation error.
- For example, a task's lag can be negative when
  - » The quantum is large and the task executes very soon after being released.
  - » Some other task does not use the share of the CPU that it is allocated (*i.e.*, the task finishes "early") and the task in question consumes the left-over capacity.
- Under what conditions is a task's lag positive?

Scheduling to Bound Lag

- Tasks are scheduled in a *virtual time* domain

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

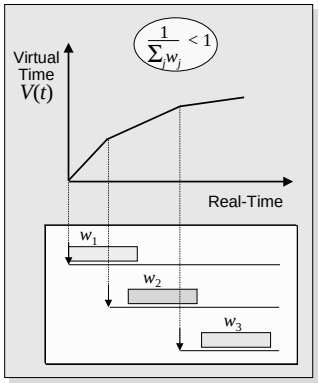
- Each task executes for w_i real-time units during each virtual time unit.



The Virtual Time Domain

- Slope of virtual time changes as (non-real-time) tasks enter and leave the system

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



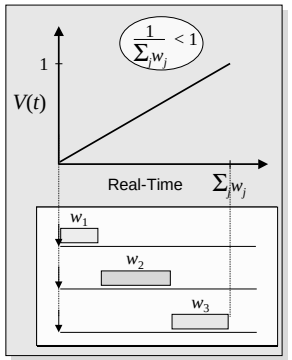
The Virtual Time Domain

- Tasks execute for w_i real-time time units in each virtual-time time unit
  - » Thus ideally, task T_i executes for

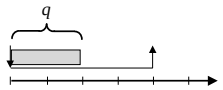
$$S_i(t_1,t_2) = w_i \int_{t_1}^{t_2} \frac{1}{\sum_j w_j} \, d\tau$$

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

  time units in any real-time interval



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

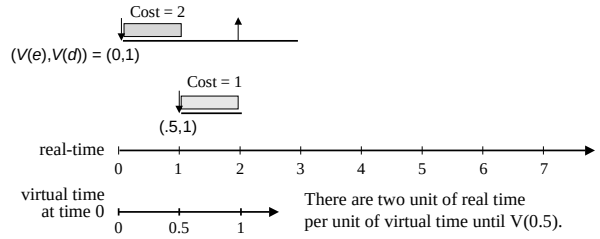
Virtual time scheduling principles

- Task requests should not be considered before their "eligible" time *e*. Why?
- 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 task with the earliest virtual deadline

EEVDF Scheduling

Example: Two tasks with equal weight = 2 (q = 1 in real-time). These tasks are not periodic, but each task makes a request immediately after it completes. The first task arrives at (real) time 0 and the second at (real) time 1.

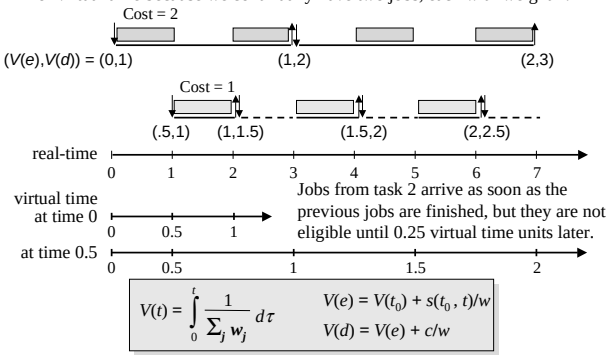


There are two unit of real time per unit of virtual time until V(0.5).

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

EEVDF Scheduling

At time V(0.5) = 1, virtual time slows down to four units of real time per unit of virtual time because we continually have two jobs, each with weight 2.



Jobs from task 2 arrive as soon as the previous jobs are finished, but they are not eligible until 0.25 virtual time units later.

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

Note: These are *not* periodic tasks! (The arrows just indicate when tasks enter and leave the system.

In particular, assume that a task makes its next request immediately after it completes.

Note that:

— task 1 always has a virtual deadline 1 time unit after its eligible time (it requires one virtual time unit to complete)

— task 2 always has a virtual deadline 0.5 time units after its eligible time

Issues

- To use proportional share scheduling for real-time computing:
  - » We need to ensure deadlines are respected in the real-time domain.
  - » We must also bound the allocation error.

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

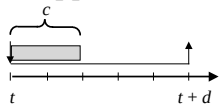
$$f_i(t) = \frac{w_i}{\Sigma_j w_j}$$

- 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

Dynamically Adjusting Weights to Support Real-Time Computing



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

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

$$\frac{w_i}{\sum_j w_j} = \frac{c_i}{d_i}$$

$$w_i = \frac{c_i}{d_i}\left(\sum_{j \neq i} w_j + w_i\right)$$

$$w_i = \frac{\frac{c_i}{d_i}\sum_{j \neq i} w_j}{1 - \frac{c_i}{d_i}}$$

Dynamically Adjusting Weights to Support Real-Time Computing

- Example: task $T_i = (t,4,1,4)$ joins a system with a total weight of 3 at time t with a deadline at time $t + 4$.
  - » In the interval $[t, t+4]$ the task requires a share of the processor equal to 1/4=0.25.
  - » Its weight should be

$$w_i = \frac{\frac{c_i}{d_i}\sum_{j \neq i} w_j}{1 - \frac{c_i}{d_i}} = \frac{\frac{1}{4} \times 3}{1 - \frac{1}{4}} = 1$$

Issues

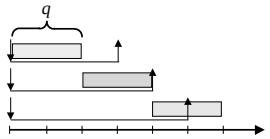
- Every time the slope of the virtual time changes, we must recompute the weight and deadline for the task.
  - » Will its lag change?
- If there are a *n* real-time tasks then we end up with *n* linear equations with *n* unknowns.
- If real-time tasks require a fixed share of the CPU's capacity, only a finite number of tasks may be guaranteed to execute in real-time.
  - » How do we limit the number of real-time tasks?

Admission control

- We need simple (efficient) on-line admission control algorithm!
- Admission criterion:
  - » a simple sufficient condition — $\Sigma_i \frac{c_i}{d_i} \leq 1$
  - » a necessary condition??
    - it depends...

Bounding the Allocation Error

- Is a task guaranteed to complete before its deadline?



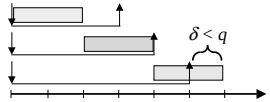
- How late can a task be?

Theorem: A task will miss its deadline by at most q time units.

Bounding the Allocation Error

- Consider a task system wherein jobs always terminate with zero lag.

Theorem: Let d be the current deadline of a request made by task T_k. Let f be the actual time the request is fulfilled:
(1) $f < d + q$ (the request is fulfilled no later than time $d + q$)
(2) if $f > d$ then for all t, $d \le t < f$, $lag_k(t) < q$



- Note: (2) says that if a task is late, it still must have been executing uniformly

Some properties of *lag*(*t*)

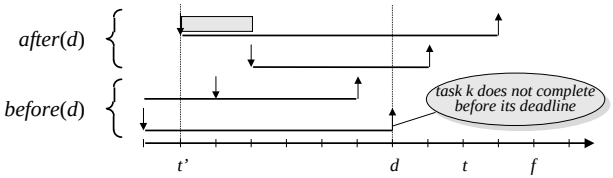
$lag_i(t) = S_i(t_0,t) - s_i(t_0,t)$

$V(e) = V(t_0) + s(t_0, t)/w$

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

- $lag_i(t) < 0$ 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*, $\Sigma_i lag_i(t) = 0$
- 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*

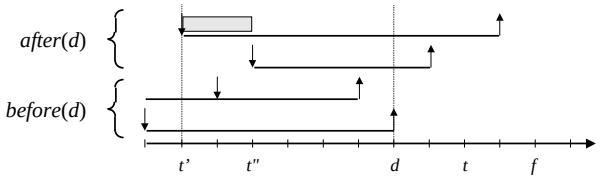
Proof Sketch of Theorem



- 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*

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

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 $d + q$

Bounding the Allocation Error

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

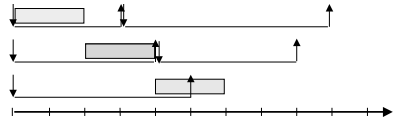
$$-c < lag_k(t) < max(c, q)$$

- The previous Theorem only dealt with lag after a deadline was missed.
- This theorem bounds the overall uniformity of resource allocation.

Remaining Issues

- Practical considerations:
  - » Maintaining virtual time
  - » Dealing with tasks that complete “early”
  - » Policing errant tasks

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 $\forall t, \Sigma_i lag_i(t) = 0$

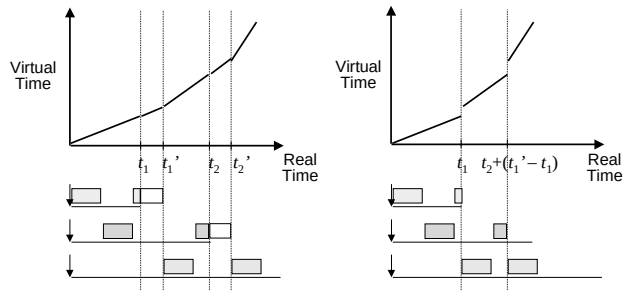
Positive Lag

- A task with positive lag is behind schedule. It needs to execute "more."
- A task can terminate with positive lag if:
  - » it did not complete by its deadline
  - » it completes before its deadline but it does not execute for as long as it expected to. (But simply not using enough execution time does not imply that lag is positive upon termination.)

Tasks that Terminate Early

- When a task terminates "early" (*i.e.*, executes for less than its wcet), it creates a discontinuity in virtual time.
- What's the effect/implication of discontinuities in virtual time?
  - » lags have to be recomputed to their values at the "primed" times in the left-hand figure on the next slide.

A Task Terminates with Positive Lag



A Task Terminates with Positive Lag

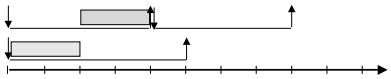
- When task T_k terminates with positive lag we must:
  - » update virtual time to the next point in time $V(t)$ at which $lag_k(t) = 0$
  - » update each task's lag to reflect the discontinuities in virtual time
- If t_k is the time a task with positive lag terminates, then

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

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

where t_k' is the virtual time at t_k before task T_k terminates.

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.

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* affect on the performance of other tasks!

Bounding the Allocation Error in Practice

- Recall our previous theorem on bounding lag.

> **Theorem:** Let c be the size of the current request of task T_k. Task T_k's lag is bounded by
>
> $$-c < lag_k(t) < max(c, q)$$

- We are now able to generalize this theorem.

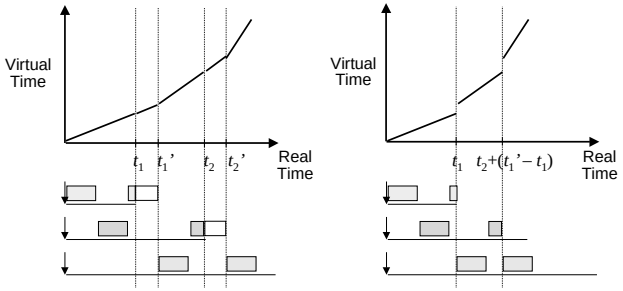
> **Theorem:** If tasks terminate with positive lag then task T_k's lag is bounded by
>
> $$-c < lag_k(t) < max(c_{max}, q)$$
>
> where c_{max} is the largest request made by any task in the system

Bounding the Allocation Error in Practice

- The previous Theorem only dealt with the case where every task terminated with zero lag.
  - » The new theorem is more general.
- But what's the impact of this theorem?
  - » Ultimately when tasks terminate with positive lag they are completing "early" in the real-time domain.
  - » Thus other tasks have received "more" time than they should have.
- While lag bounds are potentially worse, this is primarily a statement about the uniformity of allocation.
  - » Tasks are actually going to complete earlier in real-time.

Explanation of the Theorem

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



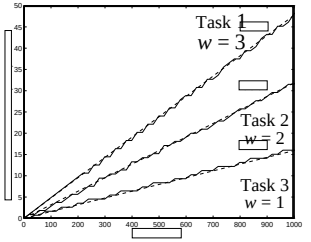
Exploring the Theorem Further

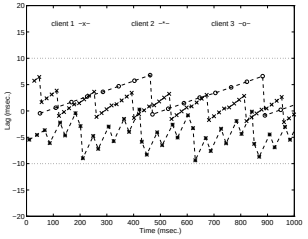
- Theorem: *If tasks terminate with positive lag then a task k's lag is bounded by* $-c < lag_k(t) < max(c_{max}, q)$
  - » Thus a trade-off exists between the size of a task's request (*i.e.*, scheduling overhead) and the accuracy of allocation

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

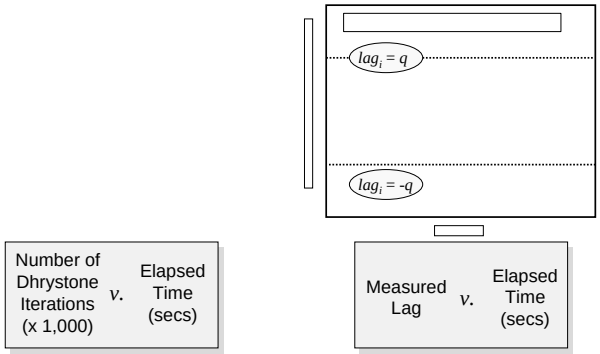
Experimental Evaluation
EEVDF Implementation in FreeBSD

- 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
    - Dhrystone benchmark tasks executing in parallel
  - » Each iteration of the benchmark requires approximately 9 *ms*
  - » 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



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
  - » How do you solve the system of linear equations efficiently?