CMPSCI 754 Multimedia Systems	**Spring 2001**

Lecture 16: 05 April '01

Lecturer: Prashant Shenoy — *Scribe: Saif Raza Hasan*

In this lecture we shall look at the issues involved in processor scheduling for multimedia systems and applications.

Application Requirements and Classes

On the basis of system requirements, applications are typically classified in the following manner:

- **Hard Real-Time Applications**: These refer to mission critical applications which demand deterministic guarantees of service from the system and failure to give them the required Quality of Service can result in catastrophic failure. Examples include: control applications for aircraft, nuclear reactors, space missions etc.
- **Soft Real-Time Applications**: These applications too require guarantees of service for satisfactory performance, but can tolerate occasional failure of service without critical consequences. Therefore, statistical guarantees are sufficient for applications in this class. Examples include: VBR Audio, Video, interactive / immersive games, etc.
- **Best-effort Applications**: These applications do not need absolute performance guarantees but perform better when the average response times are lower. Examples include: FTP, telnet, non-time-critical compute intensive applications.

On most systems it is common to find a mix of these kinds of applications. However, because of their nature, Hard Real-Time Applications usually run on dedicated systems, designed specially to to meet their requirements.

From the Processor Scheduling point of view, different classes of applications require different kinds of scheduling algorithms.

For instance, a Unix Time Sharing scheduling may work well for best-effort applications, however an Earliest Deadline First Scheduling would be more suitable for Soft Real-Time applications.

Hence we need to design an OS framework that :

- Facilitates co-existence of various application classes
- Provides protection between the various classes. (e.g. over-booking of CPU for Soft Real-Time applications should not affect Hard Real-Time Applications.).

One solution to this problem is a hierarchical partitioning of CPU bandwidth amongst various applications and application classes.

Hierarchical CPU Scheduling

In this approach towards scheduling, the applications running on a system and competing for CPU usage, are structured as a tree of arbitrary height.

- The leaf nodes correspond to an application and represent an aggregation of threads which are scheduled by an application-specific scheduler.
- The intermediate nodes in the tree represent an aggregation of application classes and are scheduled by an algorithm that achieves hierarchical partitioning.

An effective Hierarchical CPU Scheduler should fulfill the following requirements:

- It should achieve proportionate allocation of CPU bandwidth allocated among its sub-classes even when the bandwidth available to a class fluctuates over time.
- A realistic scheduling algorithm should not require to know the computational requirements of computational task beforehand eg. amount of I/O, interrupts etc.
- Provide statistical or deterministic guarantees on throughput and delays.
- The computational overhead of the scheduling algorithm itself should not be prohibitively high.

Proportionate Allocation

In the proportionate allocation scheme, tasks receive CPU bandwidth in proportion to the weights attached to them. This is formalized as follows:

Consider two tasks with weights w_1 and w_2 respectively. Let S_1 (t_1, t_2) and S_2 $(t_1, t_2$) denote the service received by task 1 and 2 respectively in the time interval $t_1 - t_2$.

An allocation is proportionate if: $\frac{S_1(t_1,t_2)}{S_2(t_1,t_2)} = \frac{w_1}{w_2}$

for any finite time range $t_1 - t_2$ during which both tasks are runnable.

However, this measure is ideal because most real schedulers use finite sized time quanta for scheduling and the above definition would not be valid if the time interval $t_1 - t_2$ is smaller than the quantum. Therefore the following expression is used to estimate the fairness of quantum based scheduling:

$$\left| \frac{S_f(t_1, t_2)}{w_f} - \frac{S_m(t_1, t_2)}{w_m} \right| \leq H(f, m)$$

where f and m are the thread identifiers.

H (f, m) is the fairness measure and the objective is to achieve a fairness measure as small as possible.

Implementing Proportionate Allocation

A large body of literature in networks and OS exists detailing various techniques for implementing Proportionate Allocation. One of the simple approaches is: **Weighted Round Robin Scheduling**

In this implementation, the various threads are scheduled in round-robin order for a quantum with a frequency such that eventually each thread gets a number of quanta in proportion to its weight. e.g. Consider 3 threads A, B, C with weights: $w_A = 1, w_B = 2, w_C = 3$ These will be scheduled for a quantum each as below:

A, B, C, B, C, C

Thus ensuring that each thread is getting CPU time in proportion to its weight. However, this technique achieves proportionate allocation only at large time scales and does not support variable length quanta.

Another approach is **Weighted Fair Queuing (WFQ)** which tries to emulate infinitely small quantum size by keeping track of how much bandwidth a processor would have gotten if quantum size had been infinitely small and using that information to schedule the threads.

However WFQ can be unfair when processor bandwidth fluctuates over time. It is also computationally extensive and requires the length of the quantum to be known a-priori.

Start Time Fair Queuing (SFQ)

In SFQ, apart from a weight, each task 'f' is given a Start Tag (S_f and a Finish Tag (F_f).

S_f represents the amount of service task 'f' has received at the beginning of this time quantum.

F_f represents the amount of service task 'f' will have received at the end of this quantum if 'f' is scheduled during this quantum. The start and finish tag are calculated as follows:

$$S_f = \max\{v(A(q_f^j)), F_f\}$$

$$F_f = S_f + \frac{l_f^j}{r_f}$$

where:

q_f^j : j^{th} quantum of thread f
l_f^j : length of q_f^j
$A(q_f^j))$: time at which the j^{th} quantum is requested r_f: weight of thread f

v(t) represents the virtual time of the scheduler and is defined as the start tag of the thread in service at time t.

Once the Start Tags have been calculated, the threads are serviced in the increasing order of the start tags.

The figure on slide 181 illustrates the behavior of two threads which get blocked from time to time. Since the start time is defined as the $\max\{v(A(q_f^j)), F_f\}$, this ensures that a blocked does not gain credit for CPU time while it is blocked. Therefore, if a thread becomes runnable after being blocked for some time, it's Start Tag is immediately increased to $v(A(q_f^j))$ so that it cannot hog the processor claiming that it hasn't used the CPU for a long time.

A value of $S_f = 1$ indicates fair sharing.

The properties of SFQ are :

- SFQ manages to achieve fair CPU allocation even if the available processing bandwidth is varying.

$$\left| \frac{S_f(t_1, t_2)}{w_f} - \frac{S_m(t_1, t_2)}{w_m} \right| \leq \frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m}$$

- SFQ does not require the length of the quantum to be known a priori.
- SFQ provides bounds on maximum delay incurred and minimum throughput achieved by threads in realistic environments.
- SFQ is computationally efficient.

Implementation - HSFQ

A Hierarchical Start Time Fair Queuing Scheduler has been implemented in the Solaris 2.4 kernel.

The system calls available to create scheduling structures are:

int hsfq_mknod (char *name, int weight, int flag, scheduler_id id) Create a node in the scheduling tree with the specified name and weight and use the specified scheduler with it.

int hsfq_parse (char *name, int hint) parse a path name describing a node in the tree to return it's node id.

int hsfq_rmnod (int id, int mode) remove a node from the tree

hsfq_move (int from, int to, ...) move a node from one point in the tree to another. All newly created nodes are attached to a default parent. This system call is used to attach a thread's node to the appropriate application class node.

hsfq_admin (int node, int cmd, void *args) perform various admin operations on the tree structure.