Multimedia Operating Systems

Multimedia Operating Systems

- Support multiple kinds of applications
  - Multimedia applications: Streaming audio, video, games, etc.
  - Traditional applications: Editors, compilers, web servers, etc.
- Satisfy different application characteristics and requirements
- Traditional Operating Systems:
  - Goal is to maximize system throughput and utilization
  - No differentiation between various application classes

Application Requirements

- Soft real-time applications: statistical guarantees
  - Examples: Streaming media, virtual games
- Interactive applications: no absolute performance guarantees, but low average response times
  - Examples: Editors, compilers
- Throughput-intensive Applications: no performance guarantees, but high throughput
  - Examples: http, ftp servers

OS Design Requirements

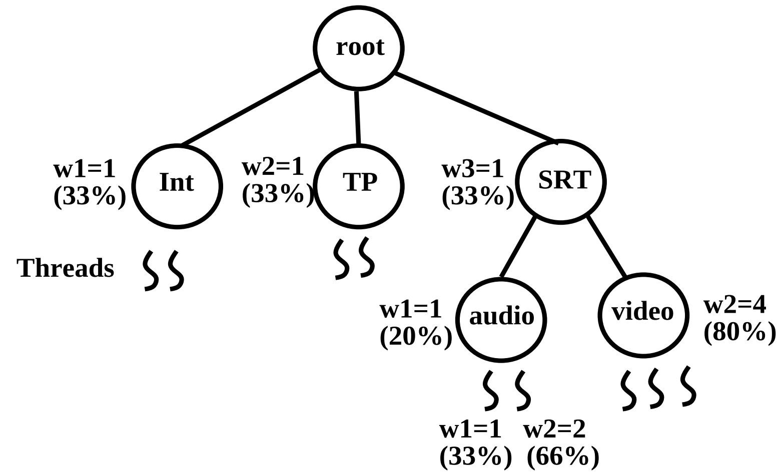
- Fair, Proportionate resource allocation:
  - Divide resources according to application requirements
  - Example: 30% of CPU to streaming, 20% to http server, etc.
- Application Isolation:
  - Prevent misbehaving or overloaded applications from affecting others
  - Example: overloaded web server should not affect streaming media server
- Service Differentiation:
  - Scheduling policy appropriate for the application class

Processor Scheduling

- Different application classes ⇒ different scheduling algorithms
  - Example: Time-sharing for best-effort applications, proportional-share for soft real-time
- Need a scheduling framework for service differentiation
- Solution: Hierarchical partitioning of CPU bandwidth

Hierarchical CPU Scheduling

- Hierarchical partitioning specified as a *tree*
- Leaf nodes:
  - Aggregation of threads
  - Scheduled by application-specific scheduler
- Intermediate nodes:
  - Aggregation of application classes
  - Scheduled by an algorithm that achieves hierarchical partitioning



Requirements of a Hierarchical CPU Scheduler

- Should achieve proportionate allocation of CPU bandwidth allocated to a class among its sub-classes, even when the bandwidth available to a class fluctuates over time
- Should not require computational requirements of tasks to be known a priori
- Should provide throughput and delay guarantees
- Should be computationally efficient

Proportionate Allocation

- Assign weights to tasks
- Tasks receive CPU bandwidth in proportion to weights
- Ideal definition: $\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m} = 0$

$W_f(t_1, t_2)$	:	aggregate work done by thread f in interval in $[t_1, t_2]$
r_f	:	weight of thread f

- Quantum-based scheduling: $\left|\frac{W_f(t_1,t_2)}{r_f} - \frac{W_m(t_1,t_2)}{r_m}\right| \leq H(f,m)$
- $H(f,m)$: fairness measure
- Objective: achieve small fairness measure

Generalized Processor Sharing (GPS)

- Idealized Algorithm:
  - Infinitesimally small quanta
  - No scheduling overhead
- Achieves perfect proportionate allocation
  - Each task m gets a virtual CPU with capacity $(\frac{r_m}{\sum_i r_i}) \cdot C$
- Lower bound on Fairness Measure of any algorithm
  - $H(f, m) = 0$

Start-Time Fair Queuing (SFQ)

- Start tag S_f and finish tag F_f:

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

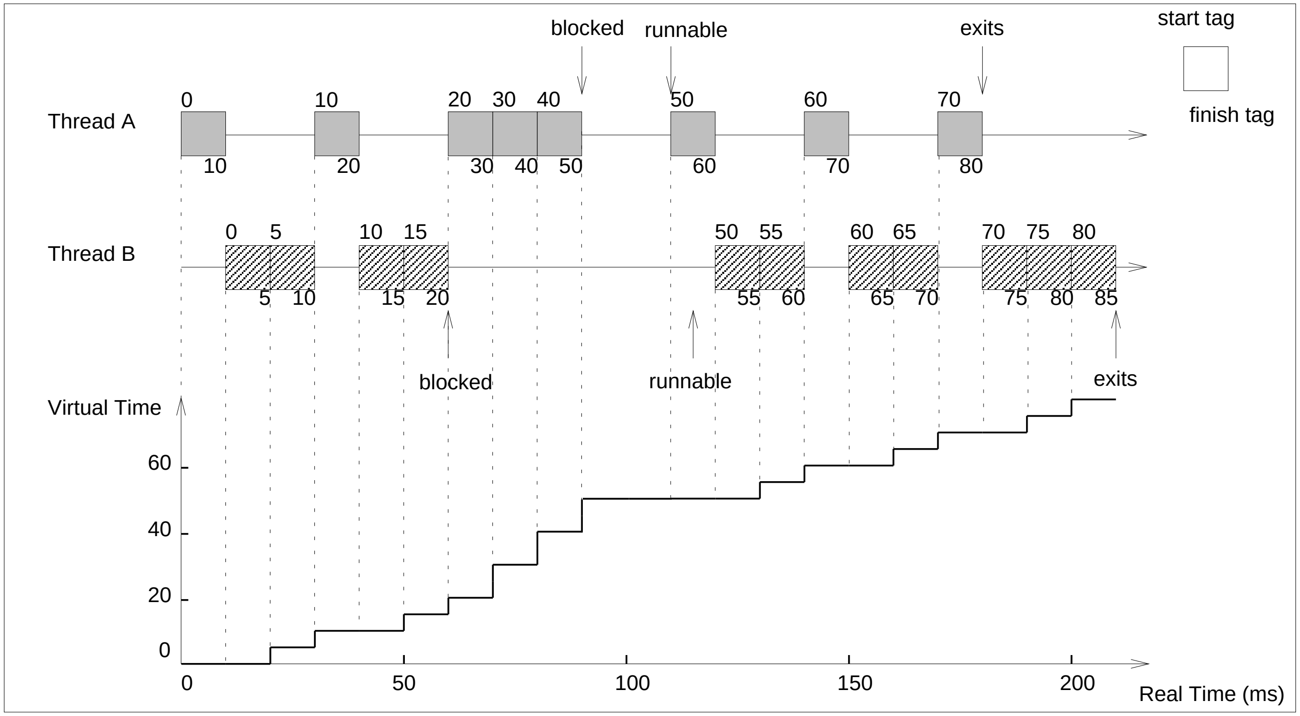
$$F_f = S_f + \frac{l_f^j}{r_f}$$

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

- Virtual time $v(t)$: start tag of the thread in service at time t
- Threads are serviced in the increasing order of start tags

SFQ: An Example

- Threads A and B s.t. $r_A : r_B = 1 : 2$



Properties of SFQ

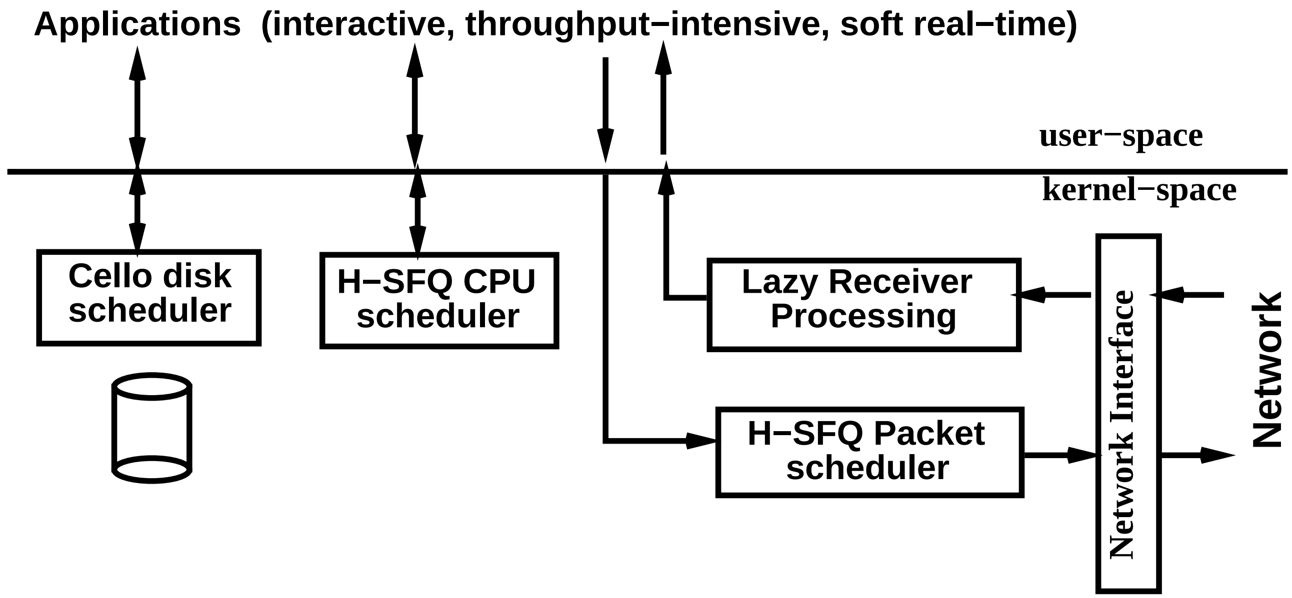
- SFQ achieves fair allocation of CPU regardless of variation in available processing bandwidth

$$\left| \frac{W_f(t_1, t_2)}{r_f} - \frac{W_m(t_1, t_2)}{r_m} \right| \leq \frac{l_f^{max}}{r_f} + \frac{l_m^{max}}{r_m}$$

- 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

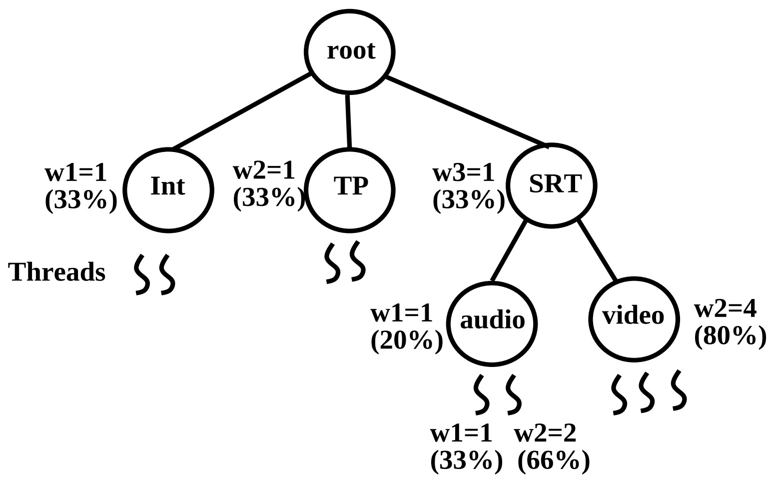
Multimedia OS Case Study: QLinux

- QoS-Enhanced version of Linux
- Replaces traditional Linux resource schedulers



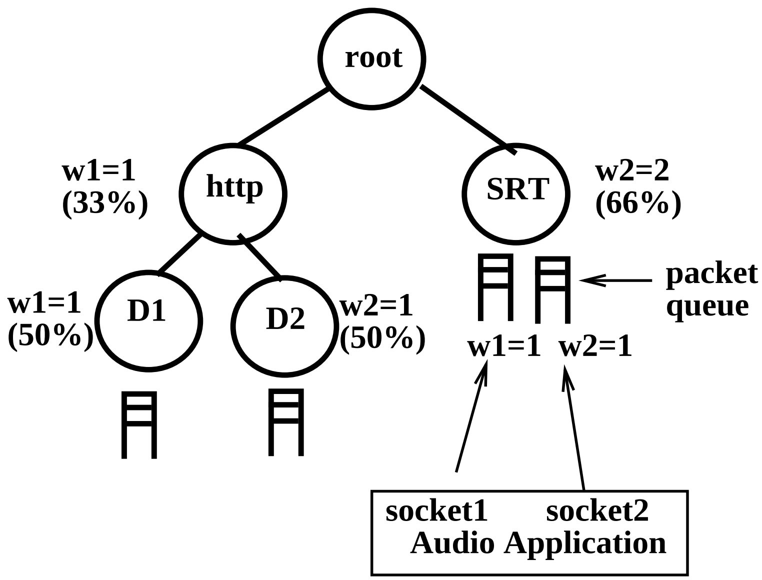
QLinux Components: CPU Scheduler

- Hierarchical SFQ (HSFQ):
  - Leaf nodes: Class-specific schedulers
  - Intermediate nodes: SFQ



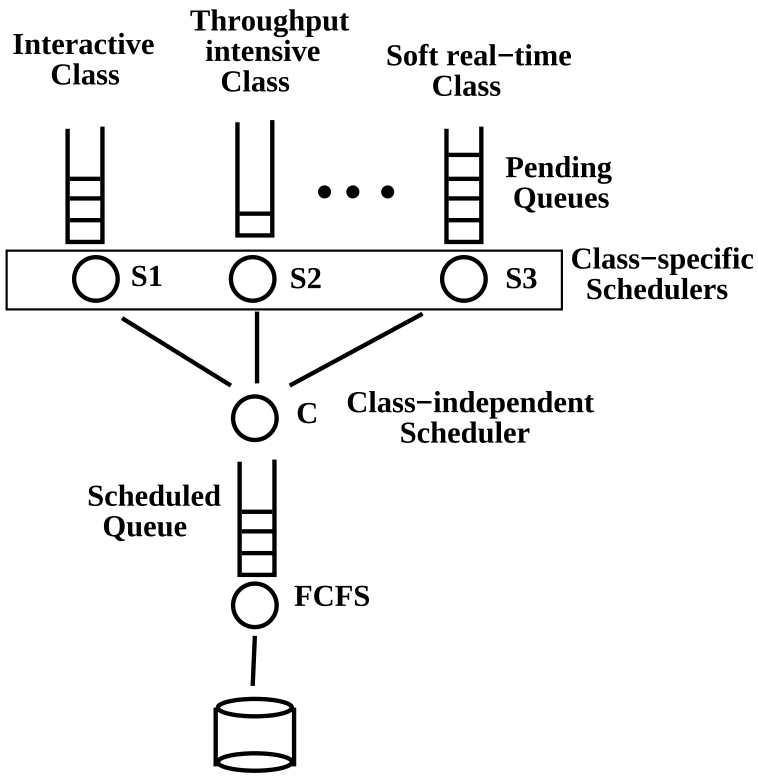
QLinux Components: Packet Scheduler

- HSFQ:
  - Sockets attached to queues
  - Queues scheduled hierarchically



QLinux Components: Disk Scheduler

- Cello:
  - Class-independent scheduler: Weighted bandwidth allocation
  - Class-specific scheduler: Service differentiation



QLinux Components: Network Subsystem

- Lazy Receiver Processing (LRP)
- Traditional OS network subsystem:
  - Interrupt driven processing of incoming packets
  - Inappropriate accounting of resource usage
- LRP:
  - Delays protocol processing: accurate resource accounting
  - Early demultiplexing: application isolation