Real Time Schedulers

• Real-time schedulers must support regular, periodic execution of tasks (e.g., continuous media).
  – CPU Reservations
    • “I need to execute for X out of every Y units.”
    • Scheduler exercises admission control at reservation time: application must handle failure of a reservation request.
  – Proportional Share
    • “I need $1/n$ of resources”
  – Time Constraints
    • “Run this before my deadline at time $T$.”

Assumptions

• Tasks are periodic with constant interval between requests, $T_i$ (request rate $1/T_i$)
• Each task must be completed before the next request for it occurs
• Tasks are independent
• Run-time for each task is constant (max), $C_i$
• Any non-periodic tasks are special
Task Model

\[ C_1 = 1 \]
\[ C_2 = 1 \]

Definitions

- **Deadline** is time of next request
- **Overflow** at time \( t \) if \( t \) is deadline of unfulfilled request
- **Feasible** schedule - for a given set of tasks, a scheduling algorithm produces a schedule so no overflow ever occurs.
- **Critical instant** for a task - time at which a request will have largest response time.
  - Occurs when task is requested simultaneously with all tasks of higher priority
Rate Monotonic

- Assign priorities to tasks according to their request rates, independent of run times
- Optimal in the sense that no other fixed priority assignment rule can schedule a task set which cannot be scheduled by rate monotonic.
- If feasible (fixed) priority assignment exists for some task set, rate monotonic is feasible for that task set.

Earliest Deadline First

- Dynamic algorithm
- Priorities are assigned to tasks according to the deadlines of their current request
- With EDF there is no idle time prior to an overflow
- For a given set of \( m \) tasks, EDF is feasible iff
  \[
  \frac{C_1}{T_1} + \frac{C_2}{T_2} + \ldots + \frac{C_m}{T_m} \leq 1
  \]
- If a set of tasks can be scheduled by any algorithm, it can be scheduled by EDF
Proportional Share

- Goals: to integrate real-time and non-real-time tasks, to police ill-behaved tasks, to give every process a well-defined share of the processor.
- Each client, $i$, gets a weight $w_i$
- Instantaneous share $f_i(t) = w_i/\sum_{j \in \Lambda(t)} w_j$

- Service time ($f_i$ constant in interval)
  $S_i(t_0, t_1) = f_i(t) \Delta t$
- Set of active clients varies $\Rightarrow f_i$ varies over time
  $S_i(t_0, t_1) = \int_{t_0}^{t_1} f_i(\tau) \, d\tau$

Common Proportional Share Competitors

- Weighted Round Robin – RR with quantum times equal to share
  RR:  
  WRR:  

- Fair Share – adjustments to priorities to reflect share allocation (compatible with multilevel feedback algorithms)

Linux
Common Proportional Share Competitors

- Weighted Round Robin – RR with quantum times equal to share
  RR:
  WRR:

- Fair Share – adjustments to priorities to reflect share allocation (compatible with multilevel feedback algorithms)

Linux
**Common Proportional Share Competitors**

- **Fair Queuing**
  - Weighted Fair Queuing
  - Stride scheduling
  - VT – Virtual Time advances at a rate proportional to share
    \[ VTA(t) = W_A(t) / S_A \]
  - VFT – Virtual Finishing Time: VT a client would have after executing its next time quantum
  - WFQ schedules by smallest VFT
    - \( E_A \) never below -1

<table>
<thead>
<tr>
<th>( VFT )</th>
<th>( 2/3 )</th>
<th>( 2/2 )</th>
<th>( 1/1 )</th>
</tr>
</thead>
</table>
| \( \text{Lottery Scheduling} \) | [Waldspurger96] | Elegant approach to periodic execution, priority, and proportional resource allocation.
  - Give \( W_p \) “lottery tickets” to each process \( p \).
  - \( \text{GetNextToRun} \) selects “winning ticket” randomly.
    - If \( SW_p = N \), then each process gets CPU share \( W_p/N \) ...
      ...probabilistically, and over a sufficiently long time interval.
  - **Flexible**: tickets are transferable to allow application-level adjustment of CPU shares.
  - Simple, clean, fast.
    - Random choices are often a simple and efficient way to produce the desired overall behavior (probabilistically).
**Basic Idea**

- Resource rights are represented by **lottery tickets**
  - Give $W_p$ “lottery tickets” to each process $p$.
  - abstract, relative (vary dynamically wrt contention), uniform (handle heterogeneity)
  - responsiveness: adjusting relative # tickets gets immediately reflected in next lottery
- At allocation time: hold a **lottery**; Resource goes to the computation holding the winning ticket.
  - $GetNextToRun$ selects “winning ticket” randomly..

**Fairness**

- Expected allocation is proportional to # tickets held - actual allocation becomes closer over time.
- Number of lotteries won by client
  \[ E[w] = n p \text{ where } p = t/T \]
- Response time (# lotteries to wait for first win)
  \[ E[n] = 1/p \]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w$</td>
<td># wins</td>
</tr>
<tr>
<td>$t$</td>
<td># tickets</td>
</tr>
<tr>
<td>$T$</td>
<td>total # tickets</td>
</tr>
<tr>
<td>$n$</td>
<td># lotteries</td>
</tr>
</tbody>
</table>
Example List-based Lottery

$T = 20$

\[
\begin{array}{cccc}
10 & 2 & 5 & 1 & 2 \\
\end{array}
\]

Summing:

\[
\begin{array}{ccc}
10 & 12 & 17 \\
\end{array}
\]

Random(0, 19) = 15

---

Bells and Whistles

- **Ticket transfers** - objects that can be explicitly passed in messages
  - Can be used to solve priority inversions
- **Ticket inflation**
  - Create more - used among mutually trusting clients to dynamically adjust ticket allocations
- **Currencies** - “local” control, exchange rates
- **Compensation tickets** - to maintain share
  - use only $f$ of quantum, ticket inflated by $1/f$ in next