Handling Overload

(G. Buttazzo, Hard Real-Time Systems, Ch. 9)

Causes for Overload

- Bad system design
  - e.g. poor estimation of worst-case execution times
- Simultaneous arrival of unexpected events
- Malfunctioning of input devices
  - “babbling idiot” problem
- Unpredicted variations of environmental conditions
- Operating system exceptions
  - Caused by anomalous combination of data
  - exceptions handlers may starve real-time workload
Definitions of “Load”

In standard queueing theory:
\[ \lambda = \text{average arrival rate} \]
\[ \mu = \text{mean service time} \]
\[ \rho = \lambda \mu = \text{average load} \]

For periodic tasks:
\[ \rho = U = \Sigma C_i / T_i \]
\[ \rho : \text{“system load”} \]
\[ U : \text{“utilization factor”} \]

For generic set of jobs:
\[ \rho(t_a, t_b) = \max_{t_1, t_2 \in [t_a, t_b]} \frac{g(t_1, t_2)}{t_2 - t_1} \]
\[ g(t_1, t_2) : \text{processor demand during interval } [t_1, t_2] \]

Instantaneous Load

Definition (Instantaneous Load):
\[ \rho_i(t) = \frac{\sum_{d_k \leq d_i} c_k(t)}{(d_i - t)} \]

\( \rho_i \) = “partial load” for job \( J_i \) at time \( t \).
\( c_k(t) \) = remaining execution time of \( J_k \) at time \( t \).

Total load at time \( t \):
\[ \rho(t) = \max_i \rho_i(t) \]
Handling Overloads

**Definition (Overload):**
A computing system experiences Overload when the computation time demanded by task set in an interval exceeds the available processing time.

**Definition (Overrun):**
A task (job) experiences Overrun when it exceeds its expected utilization. Overruns may occur because of:
- **Activation Overrun:** job is activated before expected arrival time.
- **Execution Overrun:** job computation time exceeds expected value.
Transient vs. Permanent Overload

Definition (Transient Overload):
Overload condition occurs for a limited duration in a system with average load less than schedulable utilization (*), e.g.,
\[ \rho_{\text{avg}} \leq 1 \]
\[ \rho_{\text{max}} > 1 \]

Definition (Permanent Overload):
Overload condition occurs for an unpredictable duration in a system with average load higher than schedulable utilization (*), e.g.,
\[ \rho_{\text{avg}} > 1 \]

Types of Overload Conditions

**Transient overloads due to aperiodic jobs:**
- can happen in event-triggered systems.

**Transient overloads due to task overruns:**
- tasks execute (or are activated) more than expected.
- can happen in event-triggered and time-triggered systems.

**Permanent overloads due in periodic task systems:**
- total utilization factor is larger than schedulable utilization (*).
Performance Metric

- \( V(f_i) \) = Value of task as function of finish time \( f_i \)
  - reflects importance of task

- Cumulative Value of Algorithm A:
  \[ \Gamma_A = \sum_i v(f_i) \]

- Maximum achievable cumulative value \( \Gamma^* \):
  \[ \Gamma^* := \max_A (\Gamma_A) \]

Dynamic (On-Line) vs. Clayrvoaynt Schedulers

Example:

(a) \[ \Gamma_{\text{max}} = 10 \]

(b) \[ \Gamma_{\text{max}} = 12 \]

(c) \[ \Gamma_{\text{max}} = 14 \]
Competitive Factor

Question: What is the minimum cumulative value that can be achieved by an algorithm on any task set?

Definition (Competitive Factor):
A scheduling algorithm \( A \) has a competitive factor \( \phi_A \) iff it can guarantee a cumulative value \( \Gamma_A \geq \phi_A \Gamma^* \), where \( \Gamma^* \) is the cumulative value achieved by the optimal clairvoyant scheduler.

EDF has Competitive Factor Zero (\( \phi_{EDF} = 0 \))

Figure 9.8 Situation in which EDF has an arbitrarily small competitive factor.
What is the Cost of no Clairvoyance?

Theorem (Baruah et al.):
In a system where the loading factor is greater than 2 ($\rho > 2$) and tasks' values are proportional to their computation times ($V_i = C_i$), no on-line algorithm can guarantee a competitive factor greater than 25%.

Proof by adversarial argument:
- Scheduler is player, clairvoyant scheduler is adversary
- Adversary generates sequence of tasks to minimize $\Gamma_A / \Gamma^*$
- At the end of game, compare $\Gamma_A$ and $\Gamma^*$

Adversarial Argument: Task Generation
- Major tasks, of length $C_i$, and associated tasks, of length $\varepsilon$ arbitrarily small.
- All tasks have zero laxity.
- After releasing a major task $J_i$, adversary releases next major task $J_{i+1}$ at time before the deadline of $J_i$, that is, $r_{i+1} = d_i - \varepsilon$
- For each major task $J_i$, adversary may also create a sequence of associated tasks, $[r_i, d_i]$, such that each subsequent associated task is released at the deadline of the previous one in the sequence.
  - Resulting load is $\rho = 2$.
  - Any algorithm that schedules an assoc. tasks cannot schedule $J_i$ within its deadline.
- If player schedules assoc. task, adversary stops sequence of assoc. tasks.
- If player schedules $J_i$, sequence of tasks stops with release of $J_{i+1}$
- Sequence has finite length, i.e., until $J_m$ for some value for $m$. 
Handling Overloads

### Task Generation Strategy (II)

- Player will never abandon major task for associated task. (value would be negligible)
- However, values of major tasks are chosen by adversary.
- Let \( J_0, J_1, \ldots, J_p, \ldots, J_m \) be worst-case sequence of tasks, WLOG \( C_0 = 1 \)

### Three Cases

**Case 1:** Player decides to schedule \( J_0 \):
- sequence terminates with \( J_1 \).
- cumulative value obtained by player is \( C_0 \)
- cumulative value obtained by adversary is \( C_0 + C_1 - \epsilon \)
- ratio is
  \[
  \phi_0 = \frac{C_0}{C_0 + C_1} = \frac{1}{1 + \frac{1}{k}} = \frac{1}{k}.
  \]
  (We let \( C_1 = k-1 \))

© R. Bettati

© R. Bettati
Three Cases (II)

**Case 2:** Player decides to schedule $J_i$:
- sequence terminates with $J_2$.
- cumulative value obtained by player is $C_i$.
- cumulative value obtained by adversary is $C_0 + C_i + C_2$.
- ratio is $\varphi_i = \frac{C_1}{C_0 + C_1 + C_2} = \frac{k-1}{k+C_2}$ (remember: $C_i = k-1$).
- Observation 1: $\varphi_i = \varphi_0$, otherwise adversary would have “stopped earlier”: $(k-1)/(k+C_2) <= 1/k$.
- Observation 2: $\varphi_i = \varphi_0$, otherwise player would “stuck with” $J_i$: $(k-1)/(k+C_2) >= 1/k$.
- Therefore: $\varphi_i = \varphi_0 = (k-1)/(k+C_2) = 1/k$.
- And: $C_2 = k^2 - 2k$.

Three Cases (III)

**Case 1:** Player decides to schedule $J_i$:
- sequence terminates with $J_{i+1}$.
- cumulative value obtained by player is $C_i$.
- cumulative value obtained by adversary is $C_0 + C_i + ... + C_{i+1}$.
- ratio is $\varphi_i = \frac{C_1}{\sum_{j=0}^{i+1} C_j} = \frac{1}{k+1}$.
- Observation: $\varphi_i = \varphi_{i-1} = ... = \varphi_0 = 1/k$.
- Thus, $\varphi_i = \frac{C_i}{\sum_{j=0}^{i} C_j + C_{i+1}} = \frac{1}{k}$.
- and $C_{i+1} = kC_i - \sum_{j=0}^{i} C_j$.

Worst-case computation times:

\[
\begin{align*}
C_0 & = 1 \\
C_{i+1} & = kC_i - \sum_{j=0}^{i} C_j.
\end{align*}
\]
So, what about the Bound?!

- \[ \frac{C_{m-1}}{\sum_{j=0}^{m-1} C_i + C_m} = \frac{1}{k} \]
- \[ j_i = \frac{C_i}{\sum_{j=0}^{i} C_j + C_{i+1}} = \frac{1}{k} \]
- \[ \varphi_i = \frac{C_i}{\sum_{j=0}^{i} C_j + C_{i+1}} = \frac{1}{k} \]
- \[ \varphi_m = \frac{C_m}{\sum_{j=0}^{m} C_j} \]
- \[ C_m = kC_{m-1} - \sum_{j=0}^{m-1} C_i \]
- \[ \sum_{j=0}^{i} C_j \leq \frac{1}{k} \]
- \[ \sum_{j=0}^{i} C_j \leq \frac{1}{k} \]
- \[ C_m \leq C_{m-1} \]

The Bound (II)

- Recall: \[ \frac{C_m}{\sum_{j=0}^{m} C_j} = \frac{C_{m-1}}{kC_{m-1}} \leq \frac{1}{k} \iff C_m \leq C_{m-1} \]
- We can rewrite:
  \[ C_{i+2} = kC_{i+1} - \sum_{j=0}^{i+1} C_j \]
  \[ C_{i+1} = kC_i - \sum_{j=0}^{i} C_j \]
- \[ \frac{C_{m-1}}{\sum_{j=0}^{m-1} C_i + C_m} = \frac{1}{k} \]
- \[ \sum_{j=0}^{m-1} C_i + C_m = \frac{1}{k} \]
- \[ C_{i+1} = k(C_{i+1} - C_i) - C_{i+1} \]
- \[ kC_{m-1} = \sum_{j=0}^{m} C_i + C_{m-1}k(C_{i+1} - C_i) \]
- \[ C_m = kC_{m-1} - \sum_{j=0}^{m-1} C_i = \frac{1}{k} - \frac{1}{k} \]
- \[ k(C_{i+1} - C_i) \]
The Bound (III)

- Recurrence Relation:

\[
\begin{aligned}
C_0 &= 1 \\
C_1 &= k - 1 \\
C_{i+2} &= k(C_{i+1} - C_i).
\end{aligned}
\]

The tightest bound on the competitive factor is given by the smallest ratio \(1/k\) such that recurrence relation satisfies

\[ C_m \leq C_{m-1} \]

- All we need to do is solve the recurrence relation. (Standard Discrete Math)
- We get 3 cases (\(k > 4; k = 4; k < 4\))
- Only Case \(k < 4\) has solution.