4. Fixed-Priority Scheduling

Credits to A. Burns and A. Wellings

Simple workload model

- The application is assumed to consist of a fixed set of tasks
- All tasks are periodic with known periods
  - This defines the periodic workload model
- The tasks are completely independent of each other
- All system overheads (context-switch times, interrupt handling and so on) are ignored
  - Assumed to have zero cost or otherwise negligible
- All tasks have a deadline equal to their period \( D = T \)
  - Each task must complete before it is next released
- All tasks have a fixed WCET (a safe and tight upper-bound)
  - Operation modes are not considered

Standard notation

- \( B \): Worst-case blocking time for the task (if applicable)
- \( C \): Worst-case computation time (WCET) of the task
- \( D \): Deadline of the task
- \( I \): The interference time of the task
- \( J \): Release jitter of the task
- \( N \): Number of tasks in the system
- \( P \): Priority assigned to the task (if applicable)
- \( R \): Worst-case response time of the task
- \( T \): Minimum time between task releases (or task period)
- \( U \): The utilization of each task (equal to \( C/T \))
- a-Z: The name of a task

Fixed-priority scheduling (FPS)

- At present this is the most widely used approach
  - And it is the distinct focus of this segment
- Each task has a fixed (static) priority computed off-line
- The ready tasks are dispatched to execution in the order determined by their priority
- In real-time systems the “priority” of a task is derived from its temporal requirements, not its importance to the correct functioning of the system or its integrity
Preemption and non-preemption – 1

- With priority-based scheduling, a high-priority task may be released during the execution of a lower priority one
- In a preemptive scheme, there will be an immediate switch to the higher-priority task
- With non-preemption, the lower-priority task will be allowed to complete before the other may execute
- Preemptive schemes enable higher-priority tasks to be more reactive, hence they are preferred

Preemption and non-preemption – 2

- Alternative strategies allow a lower priority task to continue to execute for a bounded time
- These schemes are known as deferred preemption or cooperative dispatching
- Schemes such as EDF can also take on a preemptive or non-preemptive form
- Value-based scheduling (VBS) can too
  - VBS is useful when the system becomes overloaded and some adaptive scheme of scheduling is needed
  - VBS consists in assigning a value to each task and then employing an on-line value-based scheduling algorithm to decide which task to run next

Rate-monotonic priority assignment

- Each task is assigned a (unique) priority based on its period
  - The shorter the period, the higher the priority
  - Tasks are assigned distinct priorities (!)
- For any two tasks $\tau_i, \tau_j$ we have $T_i < T_j \Rightarrow P_i > P_j$
  - Rate monotonic assignment is optimal under preemptive priority-based scheduling
- Nomenclature
  - Priority 1 as numerical value is the lowest (least) priority but the indices are still sorted highest to lowest (!)

Utilization-based analysis

- A simple schedulability test (thus sufficient but not necessary) exists for rate monotonic scheduling
  - But only for task sets with $D = T$
  
  \[
  U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N \left( 2^{1/N} - 1 \right)
  \]
  
  $U \leq 0.69$ as $N \to \infty$
Example: task set A

- The combined utilization is 0.82 (or 82%)
- This is above the threshold for three tasks (0.78), hence this task set fails the utilization test
- Then we have no a-priori answer

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Computation Time</th>
<th>Priority</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>50</td>
<td>12</td>
<td>1 (low)</td>
<td>0.24</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
<td>10</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>c</td>
<td>30</td>
<td>10</td>
<td>3 (high)</td>
<td>0.33</td>
</tr>
</tbody>
</table>

Timeline for task set A

Example: task set B

- The combined utilization is 0.775
- This is below the threshold for three tasks (0.78), hence this task set will meet all its deadlines

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Computation Time</th>
<th>Priority</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>80</td>
<td>32</td>
<td>1 (low)</td>
<td>0.40</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
<td>5</td>
<td>2</td>
<td>0.125</td>
</tr>
<tr>
<td>c</td>
<td>16</td>
<td>4</td>
<td>3 (high)</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Example: task set C

- The combined utilization is 1.0
- This is above the threshold for three tasks (0.78) but the task set will meet all its deadlines (!)

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Computation Time</th>
<th>Priority</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>80</td>
<td>40</td>
<td>1 (low)</td>
<td>0.50</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
<td>10</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>c</td>
<td>20</td>
<td>5</td>
<td>3 (high)</td>
<td>0.25</td>
</tr>
</tbody>
</table>
Critique of utilization-based tests

- They are not exact
- They are not general
- But they are \( \Omega(N) \)
  - Which makes them interesting for a large class of users
- The test is said to be sufficient but not necessary and as such falls in the class of schedulability tests

Response time analysis – 1

- The worst-case response time \( R_i \) of task \( \tau_i \) is first calculated and then checked (trivially) with its deadline
  \[ R_i \leq D_i \]
  \[ R_i = C_i + I_i \]
  Where \( I \) is the interference from higher priority tasks

Calculating \( R \)

- Within \( R_i \), each higher priority task \( \tau_j \) will execute a \( \left\lfloor \frac{R_i}{T_j} \right\rfloor \) times
  - The ceiling function \( \left\lceil f \right\rceil \) gives the smallest integer greater than the fractional number \( f \) on which it acts
    - E.g., the ceiling of 1/3 is 1, of 6/5 is 2, and of 6/3 is 2
- The total interference suffered by \( \tau_i \) from \( \tau_j \) in \( R_i \) is given by \( \left\lfloor \frac{R_i}{T_j} \right\rfloor C_j \)
Response time equation

\[ R_i = C_i + \sum_{j \in hp(i)} \left[ \frac{R_j}{T_j} \right] C_j \]

- Where \( hp(i) \) is the set of tasks with priority higher than task \( T_i \)
- Solved by forming a recurrence relationship
  \[ w^{n+1}_i = C_i + \sum_{j \in hp(i)} \left[ \frac{w^n_j}{T_j} \right] C_j \]

- The set of values \( w^n_1, w^n_2, w^n_3, ... \) is monotonically non-decreasing
- When \( w^n_i = w^{n+1}_i \) the solution to the equation has been found
- \( w_i^0 \) must not be greater than \( C_i \) (e.g. 0 or \( C_i \))

Response time algorithm

\[
\text{for } i \text{ in } 1..N \text{ loop -- for each task in turn} \\
n := 0 \\
\text{loop} \\
calculate new \( w^{n+1}_i \) \\
\text{if } w^{n+1}_i = w^n_i \text{ then}
\quad R_i = w^n_i \\
\quad \text{exit value found} \\
\text{end if} \\
\text{if } w^{n+1}_i > T_i \text{ then}
\quad \text{exit value not found} \\
\text{end if} \\
n := n + 1 \\
\text{end loop} \\
\text{end loop}
\]

Example: task set D

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Computation Time</th>
<th>Priority</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>7</td>
<td>3</td>
<td>3 (high)</td>
<td>0.4285...</td>
</tr>
<tr>
<td>b</td>
<td>12</td>
<td>3</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>c</td>
<td>20</td>
<td>5</td>
<td>1 (low)</td>
<td>0.25</td>
</tr>
</tbody>
</table>

\[ R_a = 3 \]

\[
\begin{align*}
R_a &= 3 \\
R_b &= 3 + \left[ \frac{3}{7} \right] 3 = 6 \\
R_c &= 3 + \left[ \frac{6}{7} \right] 3 = 6 \\
\end{align*}
\]

Example (cont’d)

\[
\begin{align*}
w^0_b &= 3 \\
w^1_b &= 3 + \left[ \frac{3}{7} \right] 3 = 6 \\
w^2_b &= 3 + \left[ \frac{6}{7} \right] 3 = 6 \\
R_b &= 6 \\
\end{align*}
\]

\[
\begin{align*}
w^0_c &= 3 \\
w^1_c &= 3 + \left[ \frac{5}{12} \right] 3 = 11 \\
w^2_c &= 3 + \left[ \frac{11}{7} \right] 3 = 14 \\
w^3_c &= 3 + \left[ \frac{14}{12} \right] 3 = 17 \\
w^4_c &= 3 + \left[ \frac{17}{12} \right] 3 = 20 \\
R_c &= 20 \\
\end{align*}
\]
Revisiting task set C

The combined utilization is 1.0
This is above the utilization threshold for three tasks (0.78)

- Hence the utilization-based schedulability test failed.
- But RTA shows that the task set will meet all its deadlines (cf. page 166).

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Computation Time</th>
<th>Priority</th>
<th>Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>80</td>
<td>40</td>
<td>1 (low)</td>
<td>80</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
<td>10</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>c</td>
<td>20</td>
<td>5</td>
<td>3 (high)</td>
<td>5</td>
</tr>
</tbody>
</table>

- Response time analysis – 2

- RTA is a feasibility test
  - Thus exact, hence necessary and sufficient
  - If the task set passes the test then all its tasks will meet all their deadlines
  - If it fails the test then, at run time, some tasks will miss their deadline and FPS tells us exactly which
    - Unless the computation time estimations (the WCET) themselves turn out to be pessimistic

Sporadic tasks

- Sporadic tasks have a minimum inter-arrival time
  - Which should be preserved at run time if schedulability is to be ensured, but how can it?
  - They also require D ≤ T
  - The RTA for FPS works perfectly for D<T as long as the stopping criterion becomes
    \[ W_i^{n+1} > D_i \]
  - Interestingly this also works perfectly well with any priority ordering

Hard and soft tasks

- In many situations the WCET given for sporadic tasks are considerably higher than the average case
  - Interrupts often arrive in bursts and an abnormal sensor reading may lead to significant additional computation
  - Measuring schedulability with WCET may lead to very low processor utilizations being observed in the actual running system
  - We need some common sense to contain pessimism
General common-sense guidelines

- **Rule 1**: All tasks should be schedulable using average execution times and average arrival rates for both periodic and sporadic tasks
  - There may therefore be situations in which it is not possible to meet all current deadlines
  - This condition is known as a *transient overload*
- **Rule 2**: All hard real-time tasks should be schedulable using WCET and worst-case arrival rates of all tasks (including soft)
  - No hard real-time task will therefore miss its deadline
  - If Rule 2 incurs unacceptably low utilizations for non-worst-case jobs then WCET values or arrival rates must be reduced

Handing aperiodic tasks – 1

- These do not have minimum inter-arrival times
  - But also no deadline
  - However we may be interested in the system being responsive to them
- We can run aperiodic tasks at a priority below the priorities assigned to hard tasks
  - In a preemptive system they therefore cannot steal resources from the hard tasks
- This does not provide adequate support to soft tasks which will often miss their deadlines
- To improve the situation for soft tasks, a server can be employed
- Servers protect the processing resources needed by hard tasks but otherwise allow soft tasks to run as soon as possible

Handing aperiodic tasks – 2

- Besides preserving hard tasks and giving fair opportunities to soft tasks we still would like to schedule aperiodic jobs in a manner that minimizes
  - The response time of the job at the head of the aperiodic job queue
  - Or else the average response time of all aperiodic jobs for a given queuing discipline
- **Possible solutions**
  - Execute the aperiodic jobs in the background
  - Execute the aperiodic jobs by interrupting the periodic jobs
  - Slack stealing
  - Use dedicated servers

Handing aperiodic tasks – 3

- **Slack stealing**
  - Difficult for preemptive systems because the slack \( \sigma(t) \) is a function of the time \( t \) at which it is computed
  - The slack stealer is ready when the aperiodic queue is not empty and it is suspended otherwise
  - When ready and \( \sigma(t) > 0 \) the slack stealer is assigned the highest priority and when \( \sigma(t) = 0 \) the lowest
  - Static computation of \( \sigma(t) \) for some \( t \) is useful but only when the release jitter in the system is very low
  - Under EDF \( \sigma(t = 0) = \min_1 \{ \sigma_i(0) \} \) where \( \sigma_i(0) = D_i - \sum_{k=1...i} e_k \) for all the jobs released in the hyperperiod starting at \( t = 0 \)
Handing aperiodic tasks – 4

- **Periodic server (TPS)** – general model
  - A task that behaves much like a periodic task and it is scheduled as such, but it only executes aperiodic jobs
    - It can execute up to $\varepsilon_{ps}$ time units in any time interval of length $T_{ps}$
    - The parameter $\varepsilon_{ps}$ is called the budget of the periodic server
    - When a server is scheduled and executes aperiodic jobs, it consumes its budget at the rate of 1 unit per unit of time
    - The budget is exhausted when it reaches 0
    - The budget is replenished at some given replenishment time
  - The server is backlogged when the aperiodic job queue is nonempty
  - It is idle if the aperiodic queue is empty
  - The server is eligible for execution only when scheduled and when it is backlogged and it has non-zero budget

Handing aperiodic tasks – 5

- **Polling server (PS)**
  - A simple kind of TPS
  - It is given a fixed budget that it uses to serve aperiodic task requests that is replenished at every period
  - The budget is immediately consumed if the PS is scheduled while idle
    - The unused quantum is given over to execute periodic tasks
  - It is not bandwidth preserving
    - An aperiodic job that arrives just after the PS has been scheduled while idle will have to wait until the next replenishment time
  - Bandwidth-preserving servers are PS with additional rules for consumption and replenishment of their budget

Handing aperiodic tasks – 6

- **Deferrable Server (DS)**
  - A high-priority periodic server handles aperiodic requests
    - Similar in principle to PS but bandwidth preserving
  - If no aperiodic tasks require execution, the server retains its budget
    - Hence, if an aperiodic task requires execution during the server period, it can be served immediately
      - When idle the server does not sleep but just waits for incoming ones
    - The budget is replenished at the start of the new period
      - If an aperiodic request arrives just $\varepsilon$ time units before the end of server period the request begins to be served and blocks the periodic task; then the server budget is replenished and a new request may be served for the full budget
    - Its contribution to $\omega(t)$ is $\varepsilon_{ds} + \frac{\varepsilon_{ps}}{T_{ps}}$, which delays periodic tasks longer than one server budget per period
Handing aperiodic tasks – 7

**Priority Exchange (PE)**
- A high-priority PS serves aperiodic tasks, if any
  - Similar in principle to DS
- If no aperiodic tasks require execution
  - PE exchanges its own priority with that of the pending (soft) periodic task with priority lower than that of the server and highest amongst all other pending periodic tasks
  - Hence the selected periodic task inherits a priority higher than its own

**Sporadic Server (SS)**
- A high-priority server with a sporadic nature is enabled at a controlled rate to serve aperiodic requests
  - SS ≠ DS
  - The budget is replenished only when exhausted and at a minimum guaranteed distance from its earlier execution
  - Not periodically!
    - This places a tolerable bound on the overhead caused by the server
    - And makes schedulability analysis simpler and less pessimistic
  - This is the default server policy in POSIX

Handing aperiodic tasks – 8

**Sporadic server rules under FPS**

- **Consumption rules**
  - At time t after the latest replenishment time $t_r$, a backlogged SS consumes non-exhausted budget if and only if it is executing or no higher-priority task is ready

- **Replenishment rules**
  - The latest replenishment time $t_r$ is recorded when SS' budget is set to $e_s$
    - $t_r = 0$ when the system begins execution
  - The effective replenishment time $t_e$ is determined at time $t_f$ when SS first begins to execute since $t_e = t_r$ is set to the latest time instant at which a lower-priority task executes in $[t_r, t_f)$ or to $t_r$ if higher-priority tasks had been busy in that interval
  - $t_e$ is the time at which SS should become running
  - The next replenishment time is set at $t_e + p_{ss}$

- **Exception**
  - If $t_e + p_{ss} < t_f$ budget is replenished as soon as exhausted because SS is late

Handing aperiodic tasks – 9

- **SS is more complex than PS or DS**
  - Its rules require keeping tab of a lot of data, several cases to consider when making scheduling decisions
  - This complexity is acceptable because the schedulability of a SS is easy to demonstrate
    - SS under FPS can be seen just like a periodic task $t_p$ with $(p_{ss}, e_s)$
  - Under EDF or LLF scheduling we can use a dynamic variant of SS as well as other bandwidth-preserving server algorithms
    - **Constant utilization server**
    - **Total bandwidth server**
    - **Weighted fair queuing server**
**Task sets with D < T**

- For $D = T$, Rate Monotonic priority assignment (a.k.a. ordering) is optimal.
- For $D < T$, **Deadline Monotonic** priority ordering is optimal.
  
  
  \[ D_i < D_j \Rightarrow P_i > P_j \]

---

**DMPO is optimal – 1**

- Deadline monotonic priority ordering (DMPO) is optimal any task set $Q$ that is schedulable by priority-driven scheme $W$ it is also schedulable by DMPO.
- The proof of optimality of DMPO involves transforming the priorities of $Q$ as assigned by $W$ until the ordering becomes as assigned by DMPO.
- Each step of the transformation will preserve schedulability.

---

**DMPO is optimal – 2**

- Let $\tau_i, \tau_j$ be two tasks with adjacent priorities in $Q$ such that under $W$ we have $P_i > P_j$ and $D_i > D_j$.
- Define scheme $W'$ to be identical to $W$ except that tasks $\tau_i, \tau_j$ are swapped.
- Now consider the schedulability of $Q$ under $W'$.
- All tasks $\{\tau_k\}$ with priority $P_k > P_j$ will be unaffected.
- All tasks $\{\tau_k\}$ with priority $P_k < P_i$ will be unaffected as they will experience the same interference from $\tau_j$ and $\tau_i$.
- Task $\tau_j$ which was schedulable under $W$, now has a higher priority, suffers less interference, and hence must be schedulable under $W'$.

---

**DMPO is optimal – 3**

- All that is left to show is that task $\tau_i$, which has had its priority lowered, is still schedulable.
- Under $W$ we have $R_j \leq D_j, D_j < D_i$ and $R_i \leq T_i$.
- Task $\tau_j$ only interferes once during the execution of task $\tau_i$ hence $R'_i = R_i \leq D_j < D_i$.
  - Under $W'$ task $\tau_j$ completes at the time task $\tau_i$ did under $W$.
  - Hence task $\tau_j$ is still schedulable after the switch.
- Priority scheme $W'$ can now be transformed to $W''$ by choosing two more tasks that are in the wrong order for DMPO and switching them.
Summary

- A simple (periodic) workload model
- Delving into fixed-priority scheduling
- A (rapid) survey of schedulability tests
- Some extensions to the workload model
- Priority assignment techniques