3.a Fixed-Priority Scheduling

Where we look at the schedulability tests for FPS, their strength and weaknesses, we accommodate aperiodic tasks, and we review the priority assignment algorithms.

The simplest workload model
- The application consists of \( n \) tasks, for constant \( n \)
- All tasks are periodic with known periods
  - Whence the name “periodic workload model”
- All tasks are assumed independent
  - No sharing of logical resources; no precedence constraints
- All tasks have implicit deadline \( (D = T) \)
  - Each job must complete before the release of its successor
- All tasks have a single, fixed WCET
  - Which can be trusted as a safe and tight upper-bound
- All runtime overheads are collated in the tasks’ WCET
  - Context-switch times, handing of clock interrupts, etc.

Notation in this section
- \( B \): Worst-case blocking time for the task (if applicable)
- \( C \): Worst-case computation time (WCET) of the task (\( = c \))
- \( D \): Relative 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 (\( = p \))
- \( U \): The utilization of each task (\( = \frac{c}{T} \))
- \( a-Z \): The name of a task

Fixed-priority scheduling (FPS)
- Still the most widely used approach in industry
- Each task has a fixed (static) priority determined off-line
- The “priority” of a real-time task is solely derived from its temporal requirements
  - The task’s relative importance (aka criticality) to correct system operation or system integrity does not influence its scheduling priority
  - Later in this course we shall discuss mixed-criticality systems, which employ scheduling solutions that also contemplate criticality attributes
- The ready jobs are dispatched to execution in the order determined by the static priority of their corresponding task
  - FPS at run time if fully determined by the priority assignment algorithm!
Preemption and non-preemption /1

- With priority-based scheduling, a high-priority task may released a job during the execution of a lower-priority one
  - The HP job will be placed at the top of the (notional) ready queue
- In a preemptive scheme, that event will cause an immediate switch of execution to the HP job
- With non-preemption, the LP job will be allowed to complete before the job at the top of the ready queue may execute
- Preemptive schemes (such as FPS and EDF) enable higher-priority tasks to be more reactive, hence they are preferred
  - Non-preemptive scheme protect “delicate” fractions of execution

Preemption and non-preemption /2

- Alternative strategies allow a LP job to continue executing for a bounded time before being preempted
  - Earlier than its completion
- Such schemes use either deferred preemption (“give me a little bit more”) or cooperative dispatching (“I will tell you when”)
- Value-based scheduling (VBS) is another way to control preemption
  - When the system becomes overloaded, some adaptive scheme of scheduling helps mitigate the risk or the consequences of overrun
  - A utility value is attached to each task off-line, and an on-line VBS algorithm to decide which job to run next

Rate-monotonic scheduling (RMS)

- Each task is assigned a priority based on its period
  - The shorter the period, the higher the priority
  - Such priorities have to be unique; no ties allowed
- For any two tasks \( \tau_i, \tau_j : T_i < T_j \rightarrow P_i > P_j \)
  - Rate monotonic assignment is optimal under preemptive priority-based scheduling and implicit deadlines
- Oddity of nomenclature
  - Priority 1 as numerical value is the lowest (least) priority
  - Task indices are sorted highest-priority to lowest-priority

Utilization-based test

- A simple sufficient but not necessary test exists for RMS for task sets with \( D = T \)
  - It upper-bounds the schedulable utilization of FPS
    \[
    U(n) = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \left( \frac{1}{2^n - 1} \right)
    \]
    where \( \lim_{n \to \infty} n \left( \frac{1}{2^n - 1} \right) = \ln 2 \approx 0.69 \)
- This shows that the schedulable utilization of FPS (RMS) is less than that of EDF
- Utilization-based tests are simple to compute, but highly inaccurate: they often don’t know …
Example: task set A

- The combined utilization of this task set is $U_A = 0.82$
- Above the threshold for three tasks: $U_A > U(3) = 0.78$
- Task set A fails the utilization-based test
- Hence, we have no a-priori answer on its actual feasibility

<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>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

- Task Release Time
- Task Completion Time
- Deadline Met
- Preempted
- Executing

Example: task set B

- Its combined utilization is $U_B = 0.775 < U(3) = 0.78$
- It passes the utilization-based test
- Hence, this task set is guaranteed to 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

- Its combined utilization is $U_C = 1.0 > U(3) = 0.78$
- It fails the utilization-based test
- But, interestingly, the task periods are harmonic
- The timeline shows that the task set meets all its deadlines
- FPS (RMS) performs much better with harmonic-rate tasks

<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>
Response time analysis /1

- RTA is a feasibility test: it is 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 some tasks will miss their deadline at run time
    - Unless the WCET values turn out to be pessimistic
- FPS determines exactly which tasks will miss their deadline in that case

Response time analysis /2

- The worst-case response time $R_i$ of task $\tau_i$ is first calculated and then checked with its deadline $D_i$
  - $\tau_i$ is feasible if and only if $R_i \leq D_i$
- $R_i = C_i + I_i$, where $I_i$ denotes the interference that $\tau_i$ suffers from higher-priority tasks
- With feasibility analysis we reason about tasks, but scheduling applies to their jobs!

Calculating $R$

- Within the span of $R_i$, each HP task $\tau_j$ will execute at most $\frac{R_i}{T_j}$ times
  - The ceiling function $\lceil f \rceil$ 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 $2$, as it is of $6/3$
  - Using the ceiling signifies that a job of $\tau_i$ will be preempted for a full execution of a job of $\tau_j$ released exactly at $\tau_i$’s end
- The total interference suffered by $\tau_i$ from $\tau_j$ in $R_i$ where $P_i < P_j$, is upper-bounded by $\frac{R_i}{T_j} C_j$
Response time equation

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

- Where \( hpi(j) \) is the set of tasks with priority higher than \( \tau_j \)'s
- Solved by forming a recurrence relation

\[ w_i^{n+1} = C_i + \sum_{j \in hpi(i)} \left( \frac{w_j^n}{T_j} \right) C_j \]

- The set of values \( w_i^0, w_i^1, w_i^2, \ldots, w_i^n \) is monotonically non-decreasing
- \( w_i^0 \) must not be greater than \( C_i \) besides being non-negative
- When \( w_i^n = w_i^{n+1} \), the solution to the equation has been found

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>

\( \begin{align*}
R_a &= 3 \\
\left\{ \begin{array}{c} 
  w_b^0 &= 3 \\
  w_b^1 &= 3 + \frac{3}{7} \cdot 3 = 6 \\
  w_b^2 &= 3 + \frac{6}{7} \cdot 3 = 6 \\
  R_b &= 6 
\end{array} \right.
\end{align*} \)

Example (cont’d)

\( \begin{align*}
  w_a^8 &= 5 \\
  w_a^9 &= 5 + \left[ \frac{5}{7} \right] 3 + \left[ \frac{5}{12} \right] 3 = 11 \\
  w_a^{10} &= 5 + \left[ \frac{5}{7} \right] 3 + \left[ \frac{11}{12} \right] 3 = 14 \\
  w_a^{11} &= 5 + \left[ \frac{17}{7} \right] 3 + \left[ \frac{17}{12} \right] 3 = 17 \\
  w_a^{12} &= 5 + \left[ \frac{20}{7} \right] 3 + \left[ \frac{20}{12} \right] 3 = 20 \\
  R_a &= 20 
\end{align*} \)
Revisiting task set C

- Its combined utilization is $U_c = 1.0 > U(3) = 0.78$
- The utilization-based test fails, but RTA shows that 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>Response Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>80</td>
<td>40</td>
<td>1 (low)</td>
<td>80</td>
</tr>
<tr>
<td>a</td>
<td>40</td>
<td>10</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>b</td>
<td>20</td>
<td>5</td>
<td>3 (high)</td>
<td>5</td>
</tr>
</tbody>
</table>

Sporadic tasks and other extensions

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

Coexistence of hard and soft tasks /1

- In many situations, the WCET given for sporadic tasks are considerably higher than the average case
  - The WCET values are far off the center of the Gaussian
- In exceptional circumstances, interrupts may arrive in bursts, and abnormal sensor readings may require significant extra computation to restore a baseline truth
- Analyzing feasibility with WCET may lead to very low processor utilization at run time, subtracted to soft tasks
  - Hence to undesirable waste of precious (and scarce) resource and a reduction of functional throughout
- We need some common-sense rules to contain such pessimism

Coexistence of hard and soft tasks /2

- **Rule 1**: All tasks (hard and soft; periodic and sporadic) should be schedulable using average execution times and average (sporadic) arrival rates
  - Hence, there may be situations where it may not be possible to meet all deadlines
  - This condition is known as a transient overload
    - It is transient so long as not all tasks transition forever to worst-case behavior
- **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 causes unacceptably low utilization for soft tasks then WCET values or arrival rates should be “massaged”
Handing aperiodic tasks /1

- They do not have minimum inter-arrival times, and consequently cannot claim deadlines
  - We may be interested in the system being responsive to them
  - In cyclic scheduling we would use slack stealing for those tasks …
- We might run aperiodic tasks at a priority below the priorities assigned to hard tasks
  - That way, under preemption, aperiodic tasks won’t be able to steal resources from hard tasks
- But this solution would penalize soft tasks, which might miss their deadlines too often
- We need another kind of solution …

Handing aperiodic tasks /2

- … A solution that, besides preserving hard tasks and giving fair opportunities to soft tasks, should minimize
  - The response time of the jobs at the head of the aperiodic queue
  - Or the average response time of as many aperiodic jobs as possible for a given queuing discipline

Possible choices
- Execute the aperiodic jobs in the background
- Execute the aperiodic jobs by interrupting the periodic jobs
- Use slack stealing
- Use dedicated servers

Handing aperiodic tasks /3

- Slack stealing
  - Difficult to implement for preemptive systems
    - The slack $\sigma(t)$ is a not a constant for them
    - It is a function of the time $t$ at which it is computed
  - The slack stealer is ready when the aperiodic queue is not empty; it is suspended otherwise
  - When ready and $\sigma(t) > 0$, the slack stealer is assigned the highest priority; the lowest when $\sigma(t) = 0$
  - Static computation of $\sigma(t)$ for some $t$ is useful but only when the release jitter in the system is very low
  - Under EDF, $\sigma(t = 0) = \min_j [\sigma_j(0)]$ where $\sigma_j(0) = D_j - \sum_{k=1}^{j-1} d_k$ for all jobs released in the hyperperiod starting at $t = 0$

Computing the slack under EDF

- $T_1 = (4, 2)$, $T_2 = (6, 2.75)$ - EDF scheduling: $(b, p, c, R)$

- $\sigma_{11}(0) = D_1 - C_1 = 4 - 2 = 2$
- $\sigma_{12}(0) = D_2 - C_1 - C_2 = 6 - 2 - 2.75 = 1.25$
- $\sigma_{13}(0) = D_3 - 2 \times C_1 - C_2 = 8 - 2 \times 2 - 2.75 = 1.25$
- $\sigma_{21}(0) = D_{12} - 2 \times C_1 - 2 \times C_2 = 12 - 2 \times 2 - 2 \times 2.75 = 2.5$
- $\sigma_{23}(0) = D_{13} - 3 \times C_1 - 2 \times C_2 = 12 - 3 \times 2 - 2 \times 2.75 = 0.5$
Computing the slack under FPS /1

- The amount of slack that an FPS system has in a time interval may depend on when the slack is used.
- To minimize the response time of an aperiodic job $J_a$, the decision of when to schedule it, must consider the execution time of $J_a$.
- No slack stealing algorithm under FPS can minimize the response time of every aperiodic job, even with prior knowledge of their arrival and execution times.
- Better not be greedy in using the available slack.

Computing the slack under FPS /2

- The slack of periodic jobs of $\tau_i$ should be computed based on their effective deadline $D_i^e$.
  - For a job of $\tau_i$, it should be computed at the beginning of the level-$i$ busy period that precedes $D_i$ so that $D_i^e \leq D_i$.
- The initial slack $\sigma_{i,j}(0)$ of every periodic job $f_{ij}$ (the $j$th job of task $f_i$) in $H$ is determined as
  $$\max \left( 0, D_{ij}^e - \sum_{k=1}^{i} \left\lfloor \frac{D_{ij}^e}{T_k} \right\rfloor C_k \right)$$

Slack stealing defeats optimality

- **Greed is no good** for aperiodic tasks.
  - To minimize the response time of an aperiodic job, it may be necessary to schedule it later, even if slack is currently available.
  - For any periodic task set, under FPS, and any aperiodic queuing policy, no valid algorithm exists that minimizes the response time of all aperiodic jobs.
  - Similarly, no valid algorithm exists that minimizes the average response time of the aperiodic jobs.

Handing aperiodic tasks /4

- **Periodic server (PS), a general model**
  - The PS is a notional $(T_{ps}, C_{ps})$ periodic task scheduled at the highest priority solely to execute aperiodic jobs.
  - The PS has a budget $C_{ps}$ time units and a replenishment period of length $T_{ps}$.
  - When the PS is scheduled and executes aperiodic jobs, it consumes its budget at the rate of 1 unit per unit of time.
  - Budget is exhausted when $C_{ps} = 0$ and replenished periodically.
  - The PS is backlogged when the aperiodic job queue is nonempty and idle otherwise.
  - Eligible for execution only when ready, backlogged and $C_{ps} > 0$. 

---

Handing aperiodic tasks /5

- **Polling server**, a simple (naïve) kind of PS
  - 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 server is scheduled while idle
  - It is not *bandwidth preserving*, hence it is inefficient
    - An aperiodic job that arrives just after the server has been scheduled while idle, must wait until the next replenishment time
  - Bandwidth-preserving servers need additional rules for consumption and replenishment of their budget

Handing aperiodic tasks /6

- **Deferrable Server** (DS), a *bandwidth-preserving* PS
  - DS retains its budget if no aperiodic tasks require execution
    - If an aperiodic job requires execution during the DS period, it can be served immediately: when idle, the DS stays ready
  - The budget is replenished at the start of the new period
  - If an aperiodic job arrives \( \epsilon \) time units before the end of \( T_{ds} \), the request begins to be served and blocks periodic tasks
  - When the budget is replenished, new aperiodic jobs may then be served for the full budget
  - If that happens, in \( w(T) \), DS contributes a solid interference of \( C_{ds} + \left| \frac{C_{ds}}{T_{ds}} \right| C_{dx} \), longer than \( 1 \times C_{ds} \) per busy period

Handing aperiodic tasks /7

- **Sporadic Server** (SS), fixes the bug in DS
  - The budget is replenished only when exhausted and at a minimum guaranteed distance from its earlier execution
    - Hence no longer at a fixed rate
  - This places a tighter bound on its interference and makes schedulability analysis simpler and less pessimistic
  - This is the default server policy in POSIX

SS rules under FPS

- **Consumption rules**
  - At time \( t > t_f \) (the latest replenishment time), a backlogged SS consumes budget only if executing, hence when no higher-priority task is ready
  - The replenishment is limited to the quantity of actual consumption
- **Replenishment rules**
  - \( t_e \) records the time that SS’ budget was last replenished
  - \( t_e > t_f \) is the latest time at which a lower-priority task than SS executes
  - The next replenishment time is set to \( t_e + T_{sd} \)
- **Exception**
  - If only higher-priority tasks had been busy since \( t_f \), then \( t_e + T_{sd} > t_f + T_{sd} \) and SS is late: hence, budget fully replenished as soon as exhausted
SS rules unveiled

- Let $t_a$ be the time at which SS has full budget and becomes backlogged, and $t_f \geq t_a$ the time at which SS becomes idle.
- In the $[t_a, t_f]$ interval, when SS is continuously active, three cases are possible:
  1. SS has consumed no capacity: $t_{next} = t_f + T_{SS} \Rightarrow$ no replenishment, and no interference in that interval
  2. SS has consumed all capacity: $t_{next} = t_a + T_{SS} \Rightarrow$ full replenishment, and bounded interference in that interval
  3. SS has consumed fractional capacity: $t_{next} = t_f + T_{SS} \Rightarrow$ fractional replenishment, and interference lower than allowed in that interval

Handing aperiodic tasks /8

- SS is more complex than PS or DS:
  - Its rules require keeping tab of lots of data
  - Several cases to consider when making scheduling decisions
  - This complexity is acceptable because the schedulability of a SS is easy to demonstrate
    - Under FPS, SS equates to a periodic task $\tau_a$ with $(p_a, e_a)$
- EDF and LLF use a dynamic variant of SS as well as other bandwidth-preserving server algorithms known as
  - Constant utilization server
  - Total bandwidth server
  - Weighted fair queuing server

Task sets with $D < T$

- We know that, for $D = T$, Rate Monotonic priority assignment (aka RMS) is optimal.
- For $D < T$, **Deadline Monotonic** priority ordering (DMPO), where $D_1 < D_2 \Rightarrow P_i > P_j$, 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 preserves schedulability.

DMPO is optimal /1

- Let $\tau_i, \tau_j$ be two tasks with adjacent priorities in $Q$ such that under $W$ we have $P_i > P_j \land D_i > D_j$.
- 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 $P_k > P_i$ will be unaffected.
  - All tasks $\{\tau_k\}$ with priority $P_k < P_j$ will be unaffected as they will experience the same interference from $\tau_i$ and $\tau_j$.
  - 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 /2

- All that is left to show is that task $\tau_2$, which has had its priority lowered, is still schedulable.
- Under $W$ we have $R_2 \leq D_2, D_2 < D_1$ and $R_1 \leq T_1$.
- Task $\tau_1$ only interferes once during the execution of task $\tau_2$, hence $R_1' = R_1 < D_1 < D_2$.
- Under $W'$ task $\tau_1$ completes at the time task $\tau_2$ did under $W$.
- Hence task $\tau_1$ 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.

Generalized priority assignment (aka simulated annealing)

**Theorem:** If task $p$ is assigned the lowest priority and it is feasible, then, if a feasible priority ordering exists for the complete task set, one such ordering exists where task $p$ is assigned the lowest priority.

```plaintext
procedure Assign_Pri (Set : in out Task_Set;
N   : Natural; -- number of tasks
OK  : out Boolean) is
begin
for K in 1..N loop
for Next in K..N loop
Swap(Set, K, Next);
Process_Test(Set, K, OK); -- is task K feasible now?
exit when OK;
end loop;
exit when not OK; -- failed to find a schedulable task
end loop;
end Assign_Pri;
```

Summary

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

Selected readings

  DOI: 10.1007/BF01094342
- D. Faggioli, M. Bertogna, F. Checconi (2010) *Sporadic Server revisited*  
  DOI: 10.1145/1774088.1774160