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 (i.e., static) priority which is 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 `i` and `j`

$$T_i < T_j \Rightarrow P_i > P_j$$

- This assignment is optimal
  - If any task set can be scheduled (using preemptive priority-based scheduling) with a fixed-priority assignment scheme, then the given task set can also be scheduled with a rate monotonic assignment scheme
  - This is termed ***rate monotonic 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

$$U \equiv \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N\,(2^{1/N} - 1)$$

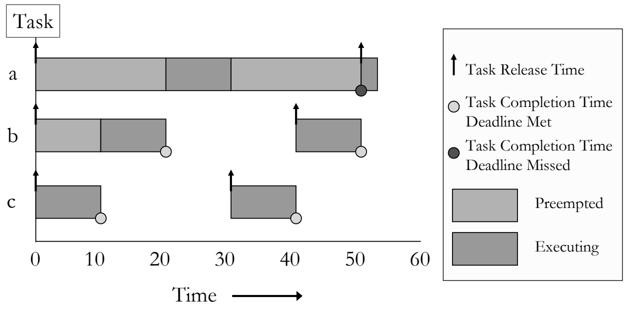
$$U \leq 0.69 \text{ as } N \rightarrow \infty$$

Example: task set A

Task	Period	Computation Time	Priority	Utilization
	T	C	P	U
a	50	12	1 (low)	0.24
b	40	10	2	0.25
c	30	10	3 (high)	0.33

- 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

Timeline for task set A



Example: task set B

Task	Period	Computation Time	Priority	Utilization
	T	C	P	U
a	80	32	1 (low)	0.40
b	40	5	2	0.125
c	16	4	3 (high)	0.25

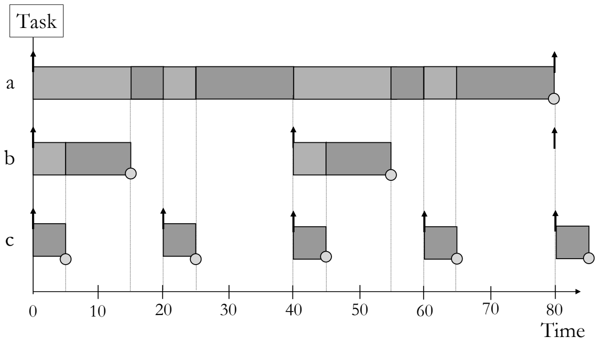
- The combined utilization is 0.775
- This is below the threshold for three tasks (0.78), hence this task set will meet all its deadlines

Example: task set C

Task	Period	Computation Time	Priority	Utilization
	T	C	P	U
a	80	40	1 (low)	0.50
b	40	10	2	0.25
c	20	5	3 (high)	0.25

- 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 (!)

Timeline for task set C



Critique of utilization-based tests

- They are not exact
- They are not general
- But they are Ω(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* of task *i* is calculated first and then checked (trivially) with its deadline

$$R_i \le D_i$$

$$R_i = C_i + I_i$$

Where *I* is the interference from higher priority tasks

Calculating R

- During *R*, each higher priority task *j* will execute a number of times

$$\text{Number of Releases} = \left\lceil \frac{R_i}{T_j} \right\rceil$$

  - The ceiling function ⌈ ⌉ gives the smallest integer greater than the fractional number 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 is given by $\left\lceil \frac{R_i}{T_j} \right\rceil C_j$

Response time equation

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

- Where *hp*(*i*) is the set of tasks with priority higher than task *i*
- Solved by forming a recurrence relationship

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

- The set of values $w_i^0, w_i^1, w_i^2, \ldots, w_i^n, \ldots$ is monotonically non-decreasing when $w_i^n = w_i^{n+1}$ the solution to the equation has been found, must not be greater than w_i^0(e.g. 0 or C_i)

Response time algorithm

```
for i in 1..N loop -- for each task in turn
  n := 0
  w_i^n := C_i
loop
    calculate new w_i^(n+1)
    if w_i^(n+1) = w_i^n then
      R_i = w_i^n
      exit value found
    end if
    if w_i^(n+1) > T_i then
      exit value not found
    end if
    n := n + 1
  end loop
end loop
```

Example: task set D

Task	Period	Computation Time	Priority	Utilization
	T	C	P	U
a	7	3	3 (high)	0.4285…
b	12	3	2	0.25
c	20	5	1 (low)	0.25

$$R_a = 3$$

$$\begin{cases} w_b^0 = 3 \\ w_b^1 = 3 + \left\lceil \frac{3}{7} \right\rceil 3 = 6 \\ w_b^2 = 3 + \left\lceil \frac{6}{7} \right\rceil 3 = 6 \\ \boxed{R_b = 6} \end{cases}$$

Example (cont'd)

$$\begin{cases} w_c^0 = 5 \\ w_c^1 = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 3 = 11 \\ w_c^2 = 5 + \left\lceil \frac{11}{7} \right\rceil 3 + \left\lceil \frac{11}{12} \right\rceil 3 = 14 \\ w_c^3 = 5 + \left\lceil \frac{14}{7} \right\rceil 3 + \left\lceil \frac{14}{12} \right\rceil 3 = 17 \\ w_c^4 = 5 + \left\lceil \frac{17}{7} \right\rceil 3 + \left\lceil \frac{17}{12} \right\rceil 3 = 20 \\ w_c^5 = 5 + \left\lceil \frac{20}{7} \right\rceil 3 + \left\lceil \frac{20}{12} \right\rceil 3 = 20 \\ \boxed{R_c = 20} \end{cases}$$

Revisiting task set C

Task	Period	Computation Time	Priority	Response Time
	T	C	P	R
a	80	40	1 (low)	80
b	40	10	2	15
c	20	5	3 (high)	5

- 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 response time analysis shows that the task set will meet all its deadlines

Response time analysis – 2

- RTA is an exact feasibility test (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, a task will miss its deadline
  - 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 response time algorithm 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

General guidelines

- **Rule 1** : All tasks should be schedulable using average execution times and average arrival rates
  - 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 "normal execution" 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

- **Periodic server** (TPS)
  - A task that behaves much like a periodic task and it is scheduled as such, but it only executes aperiodic jobs
    - It never executes for > ePS units of time in any time interval of length pPS
      - The parameter ePS 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 per unit time
      - The budget is exhausted when it reaches 0
    - The budget is replenished at a given *replenishment time*
  - The server is *backlogged* when the aperiodic job queue is nonempty
    - It is *idle* if the 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 – 4

- **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 – 5

- **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
      - In the absence of pending requests the server does not sleep but just waits for any incoming one
  - The budget is replenished at the start of the new period
      - If an aperiodic request arrives just ε time before the end of server period the request begins to be served and blocks the periodic task; the server budget is then replenished and the request is served for the full budget
      - Hence periodic tasks may be blocked **longer** than the server budget

Handing aperiodic tasks – 6

- **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

Handing aperiodic tasks – 7

- **Sporadic Server** (SS)
  - A high-priority periodic server is enabled at a sufficiently high rate to serve requests from sporadic tasks
    - SS ≠ DS
    - The budget is replenished only when exhausted, rather than at each server period
      - 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

- A SS is more complex than a PS or a 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
    - A simple SS (p_s, e_s) under FPS can be seen just like a periodic task T_i with $p_i=p_s$ and $e_i=e_s$
- Under EDF or LLF scheduling we can use SS as well as any of three 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

if 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 `i`, `j` be two tasks with adjacent priorities in `Q` such that under `W`
  $$P_i > P_j \wedge D_i > D_j$$
- Define scheme `W'` to be identical to `W` except that tasks `i` and `j` are swapped
- Now consider the schedulability of `Q` under `W'`
- All tasks with priorities greater than `j` will be unaffected by this change to lower-priority tasks
- All tasks with priorities lower than `i` will be unaffected as they will experience the same interference from `i` and `j`
- Task `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 is the need to show that task `i`, which has had its priority lowered, is still schedulable
- Under `W`

$$R_j < D_j, D_j < D_i \; and \; D_i \leq T_i$$

- Hence task `j` only interferes once during the execution of task `i`
- It follows that:

$$R'_i = R_j \leq D_j < D_i$$

- Hence task `i` 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 DMP 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