

4.a Fixed-Priority Scheduling

Credits to A. Burns and A. Wellings



Standard notation

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

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

Fixed-priority scheduling (FPS)

- At present the most widely used approach
 - 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

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 assumed to receive distinct priorities (!)
- 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
- **Nomenclature**
 - Priority 1 as numerical value is the lowest (least) priority but the indices are still sorted highest to lowest (!)

Preemption and non-preemption /2

- Alternative strategies allow a lower priority task to continue executing for a bounded time
 - Schemes known as *deferred preemption* or *cooperative dispatching*
 - Also EDF can assume a preemptive or non-preemptive form
- **Value-based scheduling** (VBS) can too
 - Useful when the system becomes overloaded and some *adaptive* scheme of scheduling is needed to mitigate the risk
 - Assigns a value to each task and then employs an *on-line value-based scheduling algorithm* to decide which task to run next

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

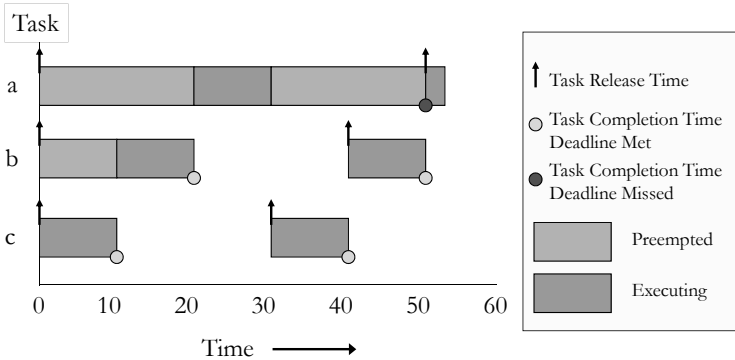
- The combined utilization is 0.82 (or 82%)
- This is above the threshold for three tasks (0.78), hence this task set fails the utilization test
- Then we have no a-priori answer

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

Timeline for task set A

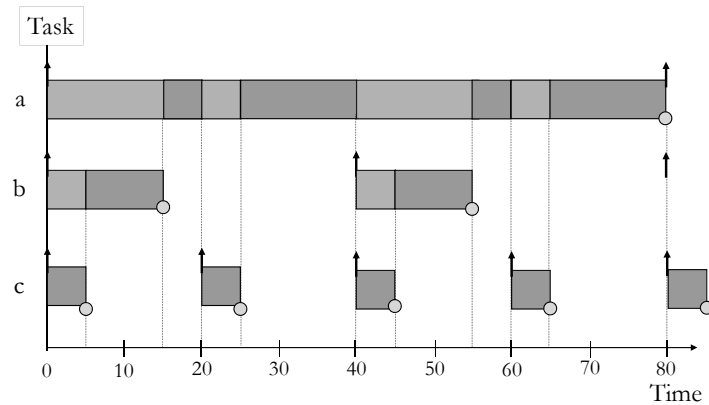


Example: task set C

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



Response time analysis /1

- The worst-case response time R_i of task τ_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

Critique of utilization-based tests

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

Calculating R

- Within R_i , each higher priority task τ_j will execute a $\left\lceil \frac{R_i}{T_j} \right\rceil$ 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, and of $6/3$ is 2
- The total interference suffered by τ_i from τ_j in R_i is given by $\left\lceil \frac{R_i}{T_j} \right\rceil C_j$

Response time equation

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

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

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

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

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

$R_a = 3$

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

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
```

If the recurrence does not converge before T_i we can still set a termination condition that attempts to determine how long past T_i job i completes

Example (cont'd)

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

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, 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. the impasse we had at pages 173-174

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 $D \leq T$
- 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 as long as the indices reflect it

Response time analysis /2

- RTA is a *feasibility test*
 - 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

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 (hard *and* soft) 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

Handling 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



Handling aperiodic tasks /1

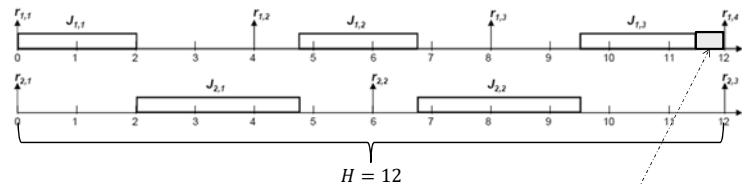
- They do not have minimum inter-arrival times
 - And consequently no deadline
 - We may be interested in the system being responsive to them
- We can run aperiodic tasks at a priority below the priorities assigned to hard tasks
 - In a preemptive system, they won't steal resources from hard tasks
- But 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

Handling aperiodic tasks /3

- **Slack stealing**
 - Difficult for preemptive systems: for them, 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, 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 $\sigma(t = 0) = \min_i \{\sigma_i(0)\}$ where $\sigma_i(0) = D_i - \sum_{k=1, \dots, i} e_k$ 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:



$$\begin{aligned}\sigma_{1,1}(0) &= 4 - 2 = 2 \\ \sigma_{2,1}(0) &= 6 - 2 - 2.75 = 1.25 \\ \sigma_{1,2}(0) &= 8 - 2 - 2.75 - 2 = 1.25 \\ \sigma_{2,2}(0) &= 12 - 2 - 2.75 - 2 - 2.75 = 2.5 \\ \sigma_{1,3}(0) &= 12 - 2 - 2.75 - 2 - 2.75 - 2 = \mathbf{0.5}\end{aligned}$$

Computing the slack under FPS /2

- The slack of periodic jobs of τ_i should be computed based on their *effective deadline* D_i^e
 - For a job of τ_i it should be computed at the beginning of the level- $i - 1$ busy period that precedes D_i so that $D_i^e \leq D_i$
- Hence the initial slack $\sigma_{i,j}(0)$ of every periodic job $J_{i,j}$ in the hyperperiod is determined as

$$\max\left(0, D_{i,j}^e - \sum_{k=1}^i \left\lceil \frac{D_{i,j}^e}{T_k} \right\rceil C_k\right)$$

Computing the slack under FPS /1

- The amount of slack an FPS system has in a time interval may depend on *when* the slack is used
- To minimise the response time of an aperiodic job J_a the decision on when to schedule it must obviously consider the execution time of J_a
 - No slack stealing algorithm under FPS can minimise 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

Handling aperiodic tasks /4

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

Handling 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
 - Ready periodic tasks – if any – execute instead
 - It is not **bandwidth preserving**
 - An aperiodic job that arrives just after the PS 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

Handling aperiodic tasks /7

- **Priority Exchange** (PE), similar in principle to DS
 - If PE server is idle when scheduled, it exchanges its own priority with that of the pending periodic task with priority lower than itself and highest amongst all other pending periodic tasks
 - The selected periodic task inherits PE's higher priority until an aperiodic task arrives or PE's ready period ends

Handling aperiodic tasks /6

- **Deferrable Server** (DS), a bandwidth-preserving TPS
 - DS retains its budget if no aperiodic tasks require execution
 - If an aperiodic task requires execution during the server period, it can be served immediately: the DS does not sleep when idle but stays ready to serve
 - The budget is replenished at the start of the new period (!)
 - If an aperiodic request arrives ε 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 requests may be served for the full budget
 - In the worst case, DS contributes to $\omega(t)$ an interference of $C_{ds} + \left\lceil \frac{t - C_{ds}}{T_{ds}} \right\rceil C_{ds}$, longer than $1 \times C_{ds}$ per period

Handling aperiodic tasks /8

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

Handling aperiodic tasks /9

- 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 τ_s with (p_s, 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*

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

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

2016/17 UniPD / T. Vardanega

Real-Time Systems

202 of 446

DMPO is optimal /3

- All that is left to show is that task τ_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 τ_j only interferes once during the execution of task τ_i hence $R_i' = R_j \leq D_j < D_i$
 - Under W' task τ_i completes at the time task τ_j did under W
 - Hence task τ_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 DMPO and switching them

2016/17 UniPD / T. Vardanega

Real-Time Systems

204 of 446

DMPO is optimal /2

- Let τ_i, τ_j be two tasks with adjacent priorities in Q such that under W we have $P_i > P_j \wedge D_i > D_j$
- Define scheme W' to be identical to W except that tasks τ_i, τ_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 $\{\tau_s\}$ with priority $P_s < P_i$ will be unaffected as they will experience the same interference from τ_j and τ_i
- Task τ_j which was schedulable under W , now has a higher priority, suffers less interference, and hence must be schedulable under W'

2016/17 UniPD / T. Vardanega

Real-Time Systems

203 of 446

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

2016/17 UniPD / T. Vardanega

Real-Time Systems

205 of 446

Selected readings

- N.C. Audsley, A. Burns, R.I. Davis, K.W. Tindell, A.J. Wellings (**1995**)
Fixed priority pre-emptive scheduling: an historical perspective
DOI: 10.1007/BF01094342
- D. Faggioli, M. Bertogna, F. Checconi (**2010**)
Sporadic Server revisited
DOI: 10.1145/1774088.1774160