2. Scheduling basics

Common approaches /1

- ***Clock-driven (time-driven) scheduling***
  - Scheduling decisions are made beforehand (at system design) and actuated at fixed time instants of execution
    - The time instants occur at intervals signaled by clock via interrupts
    - The *scheduler* first dispatches to execution the job due in the current time period and then suspends itself until then next schedule time
    - The scheduled job is supposed to complete before the next schedule time → this scheme requires no preemption
  - All scheduling parameters must be known in advance
  - The schedule, computed offline, is fixed forever
  - The scheduling overhead incurred at run time is very small

Common approaches /2

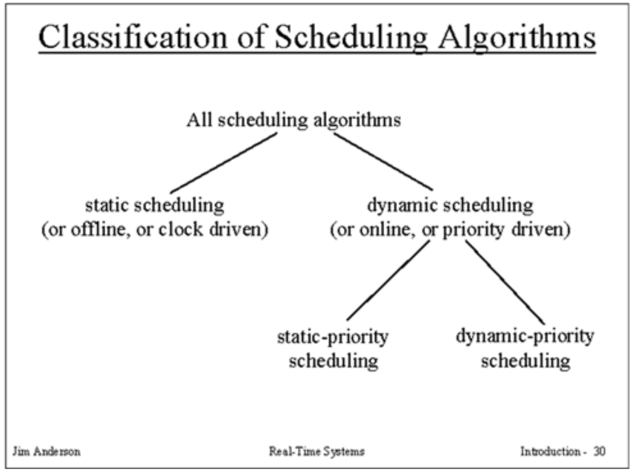
- ***Weighted round-robin scheduling***
  - With basic round-robin (which requires preemption)
    - All ready jobs are placed in a FIFO queue
    - CPU time is quantized, i.e., allotted in *time slices*
    - The job at head of queue is allowed to execute for one quantum
      - If not complete by end of quantum, it goes to the tail of the queue
      - Hence all jobs in the queue are given one quantum per round
      - Not good for jobs with precedence relations
      - Fine for producer-consumer pipelines that proceed in continual increments
  - With weighted correction to it (as for scheduling network traffic)
    - Jobs are assigned CPU time according a 'weight' (fractionary) attribute
    - Job J_i gets $\boldsymbol{\omega_i}$ time slices per round (full traversal of the queue)
      - One full round is $\sum_i \boldsymbol{\omega_i}$ of ready jobs

Common approaches /3

- ***Priority-driven (event-driven) scheduling***
  - This class of algorithms is *greedy*
    - Never leave available processing resources unutilized
    - An available resource may stay unused only if there is no job ready to use it
      - A *clairvoyant* alternative might instead defer access to the CPU to incur less contention and thus reduce job response time
    - Anomalies may occur when job parameters change dynamically
  - Scheduling decisions are made at run time when changes occur to the ready queue, hence based on present local knowledge
    - The event causing a scheduling decision is called "***dispatching point***"
  - It includes algorithms also used in non real-time systems
    - FIFO, LIFO, SETF (shortest e.t. first), LETF (longest e.t. first)

Predictability of execution

- Initial intuition
  - The execution of job set S under a given scheduling algorithm is predictable if the actual start time and the actual response time of every job in S vary within the bounds of the *maximal schedule* and *minimal schedule*
    - *Maximal schedule*: the schedule created by the scheduling algorithm under worst-case (contention) conditions
    - *Minimal schedule*: analogously for the best case
- **Theorem**: the the execution of *independent* jobs with given release times under preemptive priority-driven scheduling on a single processor is predictable
  - This notion of predictability also holds for static scheduling

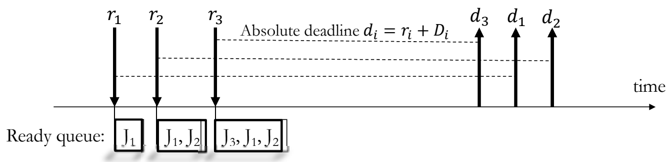


Preemption vs. non preemption

- Can we compare preemptive scheduling with non-preemptive scheduling for performance?
  - There is no single response that is valid in general
  - When all jobs have same release time, and preemption overhead is negligible (**!?**), then preemptive scheduling is provably better
- Does the improvement in the last finishing time (*minimum makespan*) under preemptive scheduling pay off the time overhead of preemption?
  - We do not know in general
  - For 2 CPUs, the minimum makespan for non-preemptive scheduling is never worse than $^4/_3$ of that for preemptive

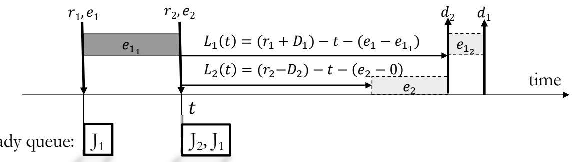
Optimality /1

- Priorities assigned *dynamically* after *absolute* deadlines
  - Ready queue reordering on job release and job completion
- ***Earliest Deadline First*** (EDF) scheduling is *optimal* for single CPU systems with independent jobs and preemption
  - For any job set, EDF produces a feasible schedule if one exists
  - The optimality of EDF breaks under other hypotheses (e.g., no preemption, multicore processing)



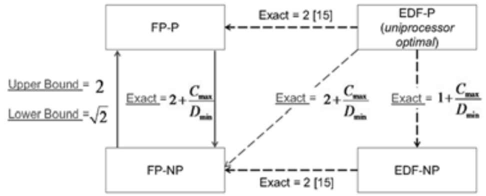
Optimality /2

- Priorities assigned dynamically after *laxity* $L(t)$
  - $L_i(t) = (r_i + D_i) - t - R_i(t)$, where $R_i(t)$ is the residual execution time needed for τ_i at time t
  - Scheduling occurs on job release and job completion
  - Jobs' priority, $L(t)$, varies with t: more dynamic than EDF and more costly to implement
- ***Least Laxity First*** (LLF) scheduling is optimal under the same hypotheses as for EDF optimality



Optimality and sub-optimality

- The *processor speed-up factor* determines the increase in processor speed that a scheduling algorithm would require to equalize an *optimal* algorithm of the same class for any task set



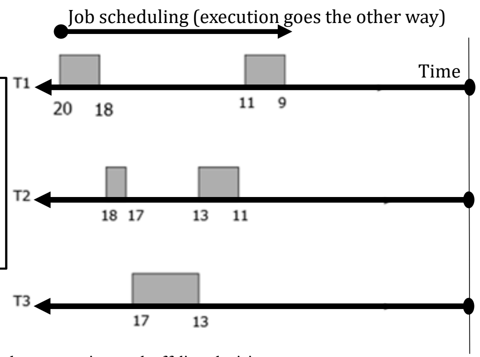
Davis et al., "Quantifying the Exact Sub-Optimality of Non-Preemptive Scheduling", RTSS 2015

Optimality /3

- If the goal were solely that jobs meet their deadlines, there would be little point in having jobs complete any earlier
  - The ***Latest Release Time*** (LRT) algorithm, the converse of EDF, follows this logic to its core, and schedules jobs backward from the latest deadline
    - LRT operates backwards treating deadlines as release times and release times as deadlines
    - LRT is *not* greedy: it may leave the CPU unused with ready tasks
- Greedy scheduling algorithms may cause jobs to suffer larger interference

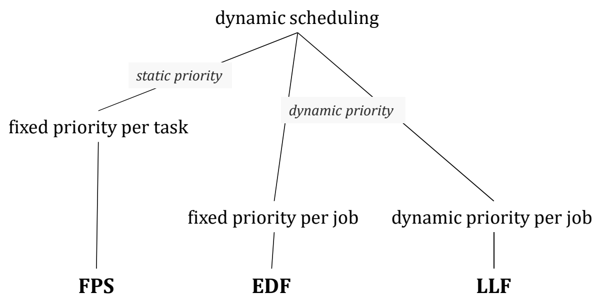
Latest Release Time scheduling

	T_1	T_2	T_3
φ_i	0	11	12
C_i	4	3	4
d_i (absolute)	20	18	17



LRT needs preemption and off line decisions

Ramifications for dynamic scheduling



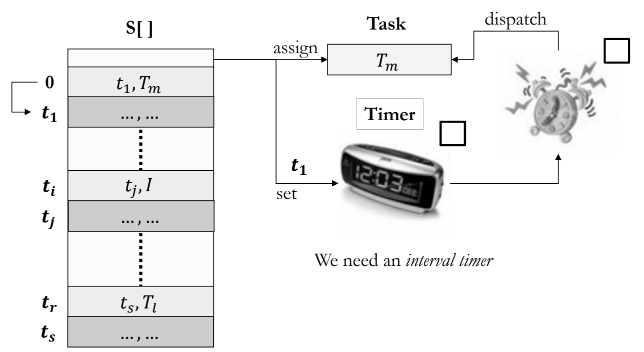
Clock-driven scheduling /1

- ***Workload model***
  - N periodic tasks, for N constant and statically defined
    - In Jim Anderson's definition of periodic (not Jane Liu's)
  - The $(\varphi_i, p_i, e_i, D_i)$ parameters of every task τ_i are constant and statically known
- The schedule is static and committed at design to a table **S** of decision times t_k where
  - $S[t_k] = \tau_i$ if a job of task τ_i must be dispatched at time t_k
  - $S[t_k] = I$ (*idle*) if no job is due at time t_k
  - Schedule computation can be as sophisticated as we like since we pay for it only at design time
  - Jobs *cannot overrun* otherwise the system is in error

Clock-driven scheduling /2

```
Input: stored schedule S[t_k], k = {0,..,N − 1}; H (hyperperiod)
SCHEDULER ::
  i := 0;
  k := 0;
  set timer to expire at t_k ;
  do forever :
    sleep until timer interrupt;
    if an aperiodic job is executing then preempt; end if;
    current task T := S[t_k] ;
    i := i + 1;
    k := i mod N;
    set timer to expire at t_k + ⌊i/N⌋ × H;
    if current task T = I
      then execute job at head of aperiodic queue;
      else execute job of task T;
    end if;
  end do;
end SCHEDULER
```


Clock-driven scheduling /3



Where the t_j values need *not* be equally spaced

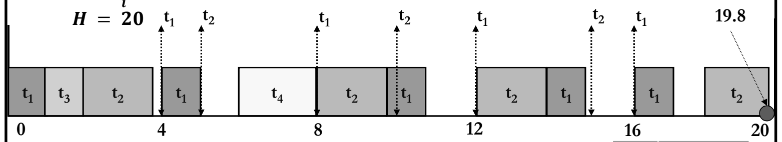
Example

$(\varphi_i, p_i, e_i, D_i)$

$\boldsymbol{J} = \{\boldsymbol{t_1} = (\mathbf{0}, \mathbf{4}, \mathbf{1}, \mathbf{4}), \boldsymbol{t_2} = (\mathbf{0}, \mathbf{5}, \mathbf{1.8}, \mathbf{5}), \boldsymbol{t_3} = (\mathbf{0}, \mathbf{20}, \mathbf{1}, \mathbf{20}), \boldsymbol{t_4} = (\mathbf{0}, \mathbf{20}, \mathbf{2}, \mathbf{20})\}$

$\boldsymbol{U} = \sum_i \frac{\boldsymbol{e_i}}{\boldsymbol{p_i}} = \mathbf{0.76}$

$\boldsymbol{H} = \mathbf{20}$



Time	Schedule
0	t_1
1	t_3
2	t_2
3.8	I
4	t_1
…	…
19.8	I
20	Goto $t\ mod(H)$

- The schedule table S for J would need 17 entries
  - That's too many and the schedule too fragmented!
- **Why 17?**

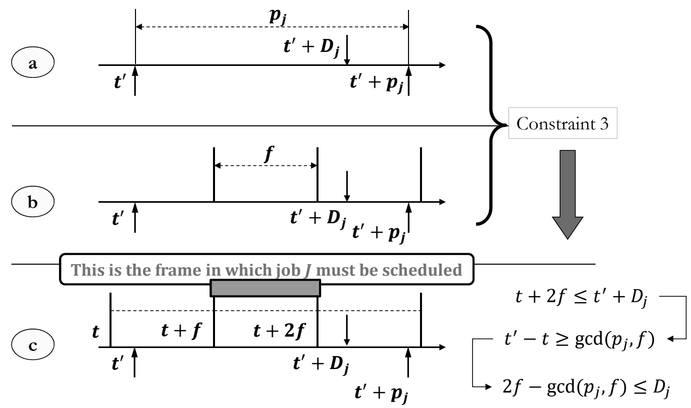
Clock-driven scheduling /4

- Reasons of complexity control suggest minimizing the size of the cyclic schedule (table S)
  - The scheduling point t_k should occur at regular intervals
    - Each such interval is termed ***minor cycle*** (*frame*) and has duration f
    - We need a (cheaper, more standard) *periodic timer* instead of a (more costly) interval timer
    - Within minor cycles there is no preemption, but a single frame may allow the execution of multiple (run-to-completion) jobs
  - For every task τ_i, φ_i must be a non-negative integer multiple of f
    - Forcedly, the first job of every task has its release time set at the start edge of a minor cycle
- To build such a schedule, we must enforce some constraints

Clock-driven scheduling /5

- **Constraint 1**: Every job J must complete within f
  - $\boldsymbol{f \geq max_{i=\{1,..n\}}(e_i)}$ so that *overruns* can be detected
- **Constraint 2**: f must be an integer divisor of the hyperperiod
  - $\boldsymbol{H : H = Nf}$ where $\boldsymbol{N} \boldsymbol{\epsilon} \mathbb{N}$
  - It suffices that f be an integer divisor of at least one task period p_i
  - The hyperperiod beginning at minor cycle kf for $k = 0, N - 1, 2N - 1$ is termed ***major cycle***
- **Constraint 3**: There must be one *full* frame f between J's release time t' and its deadline: $t' + D_j \geq t + 2f$
  - So that J can be set to be scheduled in that frame
  - This can be expressed as: $\mathbf{2}\boldsymbol{f} - \mathbf{gcd}(\boldsymbol{p_i}, \boldsymbol{f}) \leq \boldsymbol{D_i}$ for every task τ_i

Understanding constraint 3



Example

- T = {(0, 4, 1, 4), (0, 5, 2, 5), (0, 20, 2, 20)}
- H = 20
- [**c1**] : $f \geq \max(e_i)$: f ≥ **2**
- [**c2**] : $\lfloor p_i/f \rfloor - p_i/f = 0$: f = {**2**, 4, 5, 10, 20}
- [**c3**] : $2f - \gcd(p_i, f) \leq D_i$: f ≤ **2**

$f = 2 : 4 - \gcd(4,2) \leq 4$ **OK**
$4 - \gcd(5,2) \leq 5$ **OK**
$4 - \gcd(20,2) \leq 20$ **OK**
$f = 4 : 8 - \gcd(4,4) \leq 4$ **OK**
$8 - \gcd(5,4) \leq 5$ **KO**
$f = 5 : 10 - \gcd(4,2) \leq 4$ **KO**
$f = 10 : 20 - \gcd(4,2) \leq 4$ **KO**
$f = 20 : 40 - \gcd(4,2) \leq 4$ **KO**

Clock-driven scheduling /5

- It is very likely that the original parameters of some task set T may prove unable to satisfy all three constraints for any given *f* simultaneously
- In that case we must decompose task τ_i's jobs by ***slicing*** their (WCET) e_i^w into fragments small enough to artificially yield a "good" *f*

Clock-driven scheduling /6

- To construct a cyclic schedule we must make three design decisions
  - Fix an *f*
  - Slice (the large) jobs
  - Assign (jobs and) slices to minor cycles
- Sadly, these decisions are very tightly coupled
  - This defect makes cyclic scheduling *very* fragile to any change in system parameters

Clock-driven scheduling /7

```
Input: stored schedule S[k], k in 0..F − 1
CYCLIC_EXECUTIVE ::
  t := 0; k := 0;
  do forever
    sleep until clock interrupt at time t × f;
    currentBlock := S[k];
    t := t + 1; k := t mod F;
    if last job not completed then take action;
    end if;
    execute all slices in currentBlock;
    while aperiodic job queue not empty do
      execute aperiodic job at top of queue;
    end do;
  end do;
end SCHEDULER
```

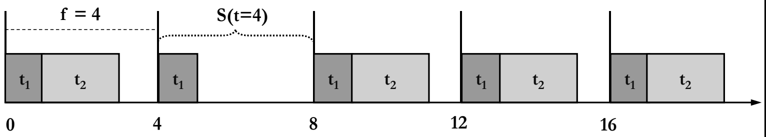
Example (slicing) – 1/2

$(\varphi_i, p_i, e_i, D_i)$

$\boldsymbol{J = \{\tau_1 = (0, 4, 1, 4), \tau_2 = (0, 5, 2, 7), \tau_3 = (0, 20, 5, 20)\}, H = 20}$

$\boldsymbol{\tau_3}$ causes disruption since we need $\boldsymbol{e_3 \le f \le 4}$ to satisfy c3

We must therefore slice $\boldsymbol{e_3}$: how many slices do we need?

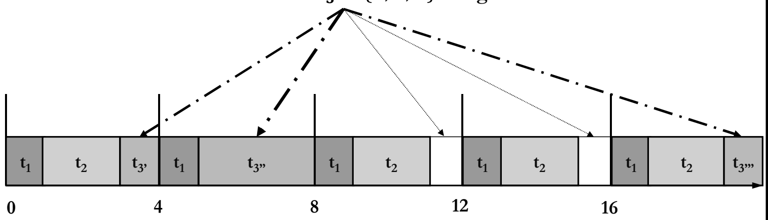


We first look at the schedule with $\boldsymbol{f = 4}$ and $\boldsymbol{F = \left(\frac{H}{f}\right) = 5}$

without $\boldsymbol{\tau_3}$, to see what least-disruptive opportunities we have …

Example (slicing) – 2/2

… then we observe that $\boldsymbol{e_3 = \{1, 3, 1\}}$ is a good choice



$\boldsymbol{\tau_3 = \{\tau_3' = (0, 20, 1, x), \tau_3'' = (0, 20, 3, y), \tau_3''' = (0, 20, 1, 20)\}}$

where $\boldsymbol{x < y \le 20}$ represent the precedence constraints that must hold between the slices (could have used phases instead)

Design issues /1

- Completing a job much ahead of its deadline is of no use
- If we have spare time we might give aperiodic jobs more opportunity to execute hence make the system more responsive
- The principle of ***slack stealing*** allows aperiodic jobs to execute in preference to periodic jobs when possible
  - Every minor cycle include some amount of slack time not used for scheduling periodic jobs
    - The slack is a *static* attribute of each minor cycle
- A scheduler does slack stealing if it assigns the available slack time at the beginning of every minor cycle (instead of at the end)
  - However, this value-added provision requires a fine-grained interval timer (again!) to signal the end of the slack time for each minor cycle

Design issues /2

- What can we do to handle ***overruns*** ?
  - Halt the job found running at the start of the new minor cycle
    - But that job may not be the one that overrun!
    - Even if it was, stopping it would only serve a useful purpose if producing a late result had no residual *utility*
  - Defer halting until the job has completed all its "critical actions"
    - To avoid the risk that a premature halt may leave the system in an inconsistent state
  - Allow the job some extra time by delaying the start of the next minor cycle
    - Plausible if producing a late result still had *utility*

Design issues /3

- What can we do to handle ***mode changes***?
  - A mode change is when the system incurs some reconfiguration of its function and workload parameters
- Two main axes of design decisions
  - With or without deadline during the transition
  - With or without overlap between outgoing and incoming operation modes

Overall evaluation

- **Pro**
  - Comparatively simple design
  - Simple and robust implementation
  - Complete and cost-effective verification
- **Con**
  - Very fragile design
    - Construction of the schedule table is a NP-hard problem
    - High extent of undesirable architectural coupling
  - All parameters must be fixed a priori at the start of design
    - Choices may be made arbitrarily to satisfy the constraints on f
    - Totally inapt for sporadic jobs

Priority-driven scheduling

- Base principle
  - Every job is assigned a priority
  - The job with the highest priority is selected for execution
- ***Dynamic-priority scheduling***
  - Distinct jobs of the same task may have *distinct* priorities
    - For EDF, the job priority is *fixed* at release but changes across releases
    - For LLF, the job priority may change at every dispatching point
- ***Static-priority scheduling***
  - All jobs of the same task have one *and the same* priority

Dynamic-priority scheduling

- **Theorem** [Liu, Layland: 1973] EDF is optimal for independent jobs with preemption
  - Also true for task sets that include sporadic jobs
  - The allowable relative deadline for this theorem to hold is implicit or constrained
- Result trivially applicable to LLF

- EDF is *not* optimal for jobs that do *not* allow preemption
  - Preemption is an aid to optimality

Static (fixed)-priority scheduling (FPS)

- Two main variants with respect to the strategy for priority assignment
  - ***Rate monotonic***
    - A task with lower period (faster rate) gets higher priority
  - ***Deadline monotonic***
    - A task with higher urgency (shorter deadline) gets higher priority
- Before looking at those strategies in more detail we need to fix some basic notions

Dynamic scheduling: comparison criteria /1

- Priority-driven scheduling algorithms that disregard job urgency (deadline) perform poorly
  - The WCET is not a factor of interest for priority assignment
  - Weighed round-robin is "*utilization*-monotonic", but is of scarce practical use for real-time
- ***Schedulable utilization*** helps compare the performance of scheduling algorithms
  - A scheduling algorithm S can produce a feasible schedule for a task set T on a single processor if and only if $U(T)$ does not exceed the schedulable utilization of S

Dynamic scheduling: comparison criteria /2

- **Theorem** [Liu, Layland: 1973] for single processors and implicit or constrained deadlines, *the schedulable utilization of EDF is 1*
- Checking for $\Delta = \sum_{i=1}^{n} \frac{e_i}{\min(d_i, p_i)} \leq 1$, known as ***density***, is a *sufficient* schedulability test for EDF
- For constrained deadlines, we may have $\Delta \geq 1 \geq U$

Dynamic scheduling: comparison criteria /3

- The schedulable utilization criterion alone is not sufficient: we must also consider predictability
  - Recall its intuition at page 59
- On transient overload, the behavior of static-priority scheduling can be determined a-priori and is reasonable
  - The overrun of any job of a given task τ does not harm the tasks with higher priority than τ
- Under transient overload, EDF becomes instable
  - A job that missed its deadline is *more urgent* than a job with a deadline in the future: one lateness may cause many more!

Dynamic scheduling: comparison criteria /4

- Other figures of merit for comparison exist
  - ***Normalized Mean Response Time*** (NMRT)
    - Ratio between the job response time and the CPU time actually consumed with the job being ready
    - The larger the NMRT value, the larger the task idle time
  - ***Guaranteed Ratio*** (GR)
    - Number of tasks whose jobs can be guaranteed versus the total number of tasks with jobs that request execution
  - ***Bounded Tardiness*** (BT)
    - Number of tasks whose job tardiness can be guaranteed to stay within given bounds
    - With some BT, soft real-time systems can have some utility

Example (EDF) /1

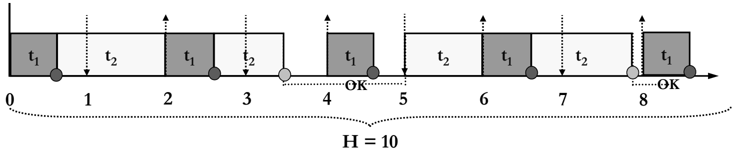
$(\varphi_i, p_i, e_i, D_i)$

$\boldsymbol{T = \{\tau_1 = (0, 2, 0.6, 1), \tau_2 = (0, 5, 2.3, 5)\}}$

Density $\boldsymbol{\Delta(T) = \frac{e_1}{D_1} + \frac{e_2}{D_2} = 1.06 > 1}$

Utilization $\boldsymbol{U(T) = \frac{e_1}{p_1} + \frac{e_2}{p_2} = 0.76 < 1}$

What happens to $\boldsymbol{T}$ under EDF?

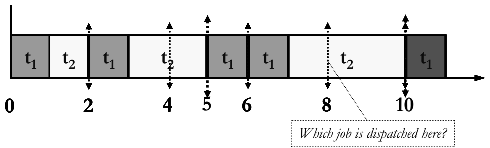


Example (EDF) /2

$(\varphi_i, p_i, e_i, D_i)$

T = {t_1= (0, 2, 1, 2), t_2= (0, 5, 3, 5)} ⇨ $\boldsymbol{U(t) = \frac{e_1}{p_1} + \frac{e_2}{p_2} = 1.1}$

T has *no* feasible schedule: what job suffers most under EDF?



T = {t_1= (0, 2, 0.8, 2), t_2= (0, 5, 3.5, 5)} ⇨ $\boldsymbol{U(t) = \frac{e_1}{p_1} + \frac{e_2}{p_2} = 1.1}$

T has *no* feasible schedule: what job suffers most under EDF?

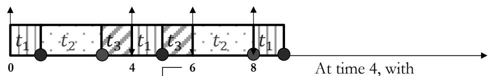
What about

T = {t1 = (0, 2, 0.8, 2), t2 = (0, 5, 4, 5)} with $\boldsymbol{U(t) = \frac{e_1}{p_1} + \frac{e_2}{p_2} = 1.2}$?

Example (EDF vs FPS) /3

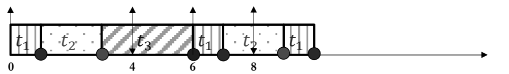
$\boldsymbol{T = \{t_1 = (0, 4, 1, 4), t_2 = (0, 6, 2, 6), t_3 = (0, 8, 3, 8)\}, U = \frac{23}{24}, \mathrm{H} = 24}$

With fixed-priority scheduling (FPS), rate-monotonic priority assignment



At time 4, with
t_3's (absolute) deadline = 8, fixed priority = low
t_1's (absolute) deadline = 8, fixed priority = high
FPS has t_1 preempt t_3
At time 6, with
t_2's (absolute) deadline = 12, fixed priority = medium
FPS has t_2 preempt t_3, <u>which misses its deadline</u>

With earliest-deadline first (EDF)



EDF may incur less preemptions and schedule more task sets than FPS

Critical instant /1

- Feasibility and schedulability tests must consider the ***worst case*** for all tasks
  - The worst case for task τ_i occurs when the worst possible relation holds between its release time and that of all higher-priority tasks
  - The actual case may differ depending on the admissible relation between D_i and p_i
- The notion of ***critical instant*** – if one exists – captures the worst case
  - The response time R_i for a job of task τ_i with release time on the critical instant is the longest possible value for τ_i

Critical instant /2

- **Theorem**: under FPS with $D_i \leq p_i \ \forall i$, the critical instant for task τ_i occurs when the release time of *any* of its jobs is in phase with a job of every higher-priority task in the set
- We seek $\max(\omega_{i,j})$ for all jobs $\{j\}$ of task τ_i for

$$\omega_{i,j} = e_i + \sum_{(k=1,..,i-1)} \left\lceil \frac{(\omega_{i,j} + \varphi_i - \varphi_k)}{p_k} \right\rceil e_k - \varphi_i$$

For task indices assigned in decreasing order of priority

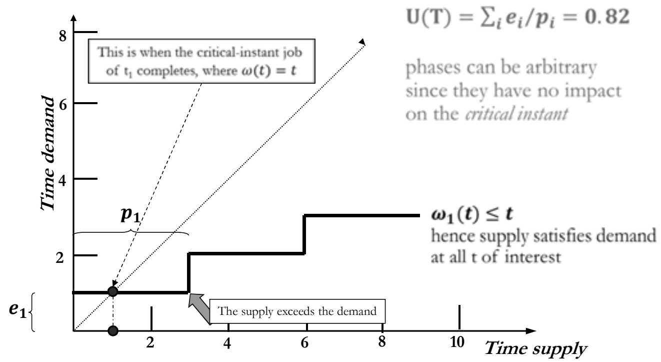
- The $\sum$ component captures the ***interference*** that any job j of task τ_i incurs from jobs of higher-priority tasks $\{\tau_k\}$ between the release time of the first job of task τ_k (with phase φ_k) to the response time of job j, which occurs at $\varphi_i + \omega_{i,j}$

Time-demand analysis /1

- When φ is 0 for all jobs considered, this equation captures the *absolute worst case* for task τ_i
- This equation stands at the basis of ***Time Demand Analysis***, which investigates how ω varies as a function of time
  - As long as $\omega(t) \leq t$ *for some (important)* t for the job of interest, the supply satisfies the demand, hence the job can complete in time
- **Theorem** [Lehoczky, Sha, Ding: 1989] condition $\omega(t) \leq t$ is an *exact feasibility test* (necessary and sufficient)
  - The obvious question is for which 't' to check
  - The method proposes to check at *all periods of all higher-priority tasks* until the deadline of the task under study

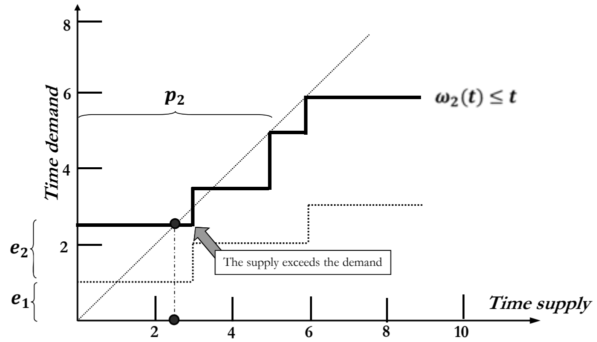
Time demand analysis /2

T = {t_1= (-, 3, 1, 3), t_2=(-, 5, 1.5, 5), t_3= (-, 7, 1.25, 7)} $(\varphi_i, p_i, e_i, D_i)$



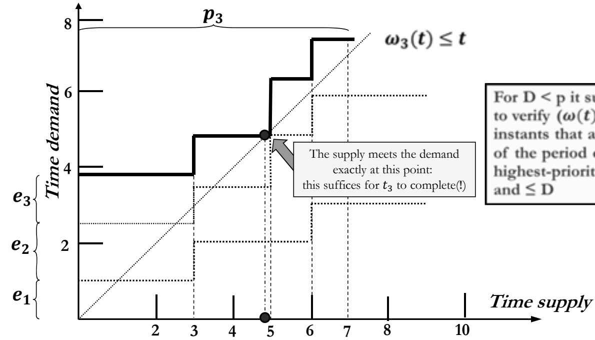
Time demand analysis /3

T = {t_1= (-, 3, 1, 3), t_2=(-, 5, 1.5, 5), t_3= (-, 7, 1.25, 7)}



Time demand analysis /4

T = {t_1= (-, 3, 1, 3), t_2=(-, 5, 1.5, 5), t_3= (-, 7, 1.25, 7)}



For D < p it suffices to verify ($\omega(t) \leq t$) at time instants that are multiple of the period of the highest-priority tasks and ≤ D

Time demand analysis /5

- It is straightforward to extend TDA to determine the *response time* of tasks

The smallest value t that satisfies

$$t = e_i + \sum_{(k=1,..i-1)} \left\lceil \frac{t}{p_k} \right\rceil e_k$$

is the *worst-case response time* of task τ_i

- Solutions methods to calculate this value were independently proposed by
  - [Joseph, Pandia: 1986]
  - [Audsley, Burns, Richardson, Tindell, Wellings: 1993]

Time demand analysis /6

- What changes in the definition of critical instant when $D>p$?
- **Theorem** [Lehoczky, Sha, Strosnider, Tokuda: 1991] The first job of task τ_i may *not* be the one that incurs the worst-case response time
- Hence we must consider *all* jobs of task τ_i within the so-called ***level-i busy period***
  - The (t_0, t) time interval within which the processor is busy executing jobs with priority $\geq i$, release time in (t_0, t), response time falling within t
  - The release time in (t_0, t) captures the full backlog of interfering jobs
  - The response time of all those jobs falling within t ensures that the busy period includes their completion

Example

T_1 = {-, 70, 26, 70}, T_2 = {-, 100, 62, 120} $(\varphi_i, p_i, e_i, D_i)$

Let's look at the level-2 busy period

Ready queue: $J_{1,1}$, $J_{2,1}$

Time window 1 [0,70)
Time left for $J_{2,1}$: 70-26 = 44
Still to execute: 62-44 = 18

Ready queue: $J_{1,2}$, $J_{2,1}$

Time window 2 [70,100)
Time left for $J_{2,1}$: 30-26 = 4
Still to execute: 18-4 = 14
Release time of job $J_{2,2}$

Ready queue: $J_{2,1}$, $J_{2,2}$

Time window 3 [100,140)
Time left for $J_{2,1}$ = 40
$J_{2,1}$ completes at: 114 (R = 114)
Time available for $J_{2,2}$: 40-14 = 26
Still to execute: 62-26 = 36

Ready queue: $J_{1,3}$, $J_{2,2}$

Time window 4 [140,200)
Time available for $J_{2,2}$: 60-26 = 34
Still to execute: 36-34 = 2

Ready queue: $J_{2,2}$, $J_{2,3}$

Time window 5 [200,210)
Release time of job $J_{2,3}$
$J_{2,2}$ completes at: 202 (R = 102)
Time available for $J_{2,3}$: 10-2 = 8
Still to execute: 62-8 = 54

Ready queue: $J_{1,4}$, $J_{2,3}$

Time window 6 [210,280)
Time available for $J_{2,3}$: 70-26 = 44
Still to execute: 54-44 = 10

Ready queue: $J_{1,4}$, $J_{2,3}$

Time window 7 [280,300)
Time available for $J_{2,3}$: 20-20 = 0
Release time of job $J_{2,4}$

Ready queue: $J_{1,5}$, $J_{2,3}$, $J_{2,4}$

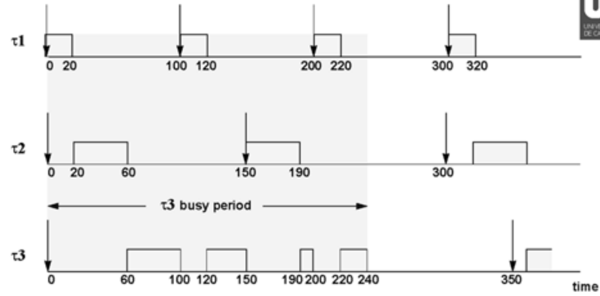
Time window 8 [300,350)
Time available for $J_{2,3}$: 50-6 = 44
$J_{2,3}$ completes at: 300+6+10 = 316 (R = 116)

$J_{2,1}$'s response time is **not** worst-case!

Still in ready queue: $J_{2,4}$
The T_2 busy period extends beyond this point (!)

Level-i busy period

T_1 = {-, 100, 20, 100}, T_2 = {-, 150, 40, 150}, T_3 = {-, 350, 100, 350} ⇒ U = 0.75
The same definition of level-i busy period holds also for D ≤ p but its width is obviously shorter!



Demand bound analysis (EDF)

- When $\boldsymbol{df}$ is the *demand function* (as in time demand analysis) and t_i is time, an *exact* test for a task set T to be schedulable by EDF is
  $$\forall t_1, t_2 : t_2 > t_i, \boldsymbol{df}(t_1, t_2) \le t_2 - t_1$$
- For periodic tasks with no offsets and $U \le 1$, the following holds:
  $$\boldsymbol{df}(t_1, t_2) \le \boldsymbol{df}(0, t_2 - t_1)$$
- The ***demand bound function*** helps generalize the test
  $$\boldsymbol{dbf}(L) = \max_t\big(df(t, t + L)\big) = df(0, L), L > 0$$
- <u>**Theorem**</u> [Baruah, Howell, Rosier: 1990] Exact test for EDF:
  $$\boxed{\forall L \in D(T), \boldsymbol{dbf}(L) \le L, U < 1}$$
  - Where $D(T)$ is the set of deadlines for T in $[0, L_m]$, $L_m = min(L_a, L_b)$, $L_a = max\left\{D_1, \dots, D_n, \frac{\sum_{i=1}^{n}(T_i - D_i)U_i}{1-U}\right\}$, $L_b =$ the first idle time in the busy period of the task set

Summary

- Initial survey of scheduling approaches
- Important definitions and criteria
- Detail discussion and evaluation of main scheduling algorithms
- Initial considerations on feasibility analysis techniques

Selected readings

- T. Baker, A. Shaw
  The cyclic executive model and Ada
  DOI: 10.1109/REAL.**1988**.51108
- C.L. Liu, J.W. Layland
  Scheduling algorithms for multiprogramming in a hard-real-time environment
  DOI: 10.1145/321738.321743 (**1973**)