3. Scheduling issues

Common approaches /1

- ***Clock-driven (time-driven) scheduling***
  - Scheduling decisions are made beforehand (off line) and carried out at predefined time instants
    - The time instants normally occur at regular intervals signaled by a clock interrupt
    - The scheduler first dispatches jobs to execution as due in the current time period and then suspends itself until then next schedule time
    - The scheduler uses an off-line schedule to dispatch
  - All parameters that matter must be known in advance
  - The schedule is static and cannot be changed at run time
  - The run-time overhead incurred in executing the schedule is minimal

Common approaches /2

- ***Weighted round-robin scheduling***
  - With basic round-robin
    - All ready jobs are placed in a FIFO queue
    - The job at head of queue is allowed to execute for one *time slice*
      - If not complete by end of time slice it is placed at the tail of the queue
    - All jobs in the queue are given one time slice in one round
  - Weighted correction (as applied to scheduling of network traffic)
    - Jobs are assigned differing amounts of CPU time according a given 'weight' (fractionary) attribute
    - Job J_i gets $\boldsymbol{\omega_i}$ time slices per round – one round is $\sum_i \boldsymbol{\omega_i}$ of ready jobs
    - Not good for jobs with precedence relations
      - Response time gets worse than basic RR which is already bad
    - Fit for producer-consumer jobs that operate concurrently in a pipeline

Common approaches /3

- ***Priority-driven (event-driven) scheduling***
  - This class of algorithms is *greedy*
    - They never leave available processing resources unutilized
      - Seeking local optimization
    - An available resource may stay unused iff there is no job ready to use it
    - A *clairvoyant* alternative may 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 on 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)
      - Normally applied at every round of RR scheduling

Preemption vs. non preemption

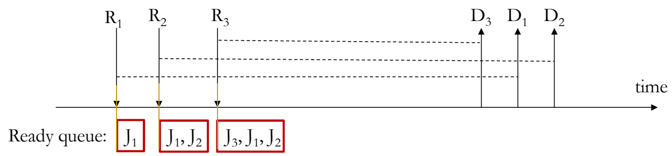
- Can we compare preemptive scheduling with non-preemptive scheduling in terms of performance?
  - There is no response that is valid in general
    - When all jobs have the same release time and the time overhead of preemption is negligible then preemptive scheduling is provably better
  - It would be interesting to know whether the improvement of the last finishing time (a.k.a. *minimum makespan*) under preemptive scheduling pays off the time overhead of preemption
- For 2 CPU we do know that the minimum makespan for non-preemptive scheduling is never worse than 4/3 of that for preemptive

Further definitions

- Precedence constraints effect release time and deadline
  - One job's release time cannot follow that of a successor job
  - One job's deadline cannot precede that of a predecessor job
- ***Effective release time***
  - For a job with predecessors this is the *latest* value between its own release time and the maximum of the effective release time of its predecessors plus the WCET of the corresponding job
- ***Effective deadline***
  - For a job with successors this is the *earliest* value between its deadline and the effective deadline of its successors less the WCET of the corresponding job
- For single processor with preemptive scheduling we may disregard precedence constraints and just consider ERT and ED

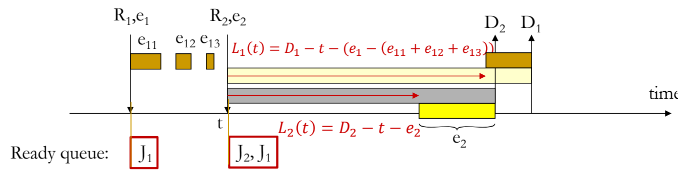
Optimality /1

- Priorities assigned in accord to (effective) deadlines
  - ***Earliest Deadline First*** scheduling is *optimal* for single processor systems with independent jobs and preemption
    - For any given job set, EDF produces a feasible schedule if one exists
    - The optimality of EDF falls short under other hypotheses (e.g., no preemption, multicore processing)



Optimality /2

- Priorities assigned in accord to *slack* (i.e., *laxity*)
  - ***Least Laxity First*** scheduling is optimal under the same hypotheses as for EDF optimality
    - LLF is far more onerous than EDF to implement as it has to keep tab of execution time!



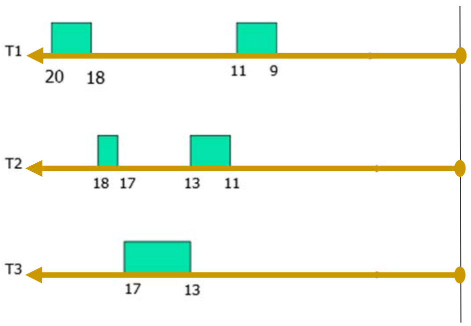
Optimality /3

- If the goal is that jobs just make their deadlines then having jobs complete any earlier has not much point
  - The ***Latest Release Time*** algorithm (converse of EDF) follows this logic and schedules jobs backwards from the latest deadline
    - LRT operates backward treating deadlines as release times and release times as deadlines
    - LRT is not greedy as it may leave the CPU unused with ready tasks
- Greedy scheduling algorithms may cause jobs to incur larger interference

Latest Release Time scheduling

	T1	T2	T3
A	0	11	12
C	4	3	4
D	20	18	17

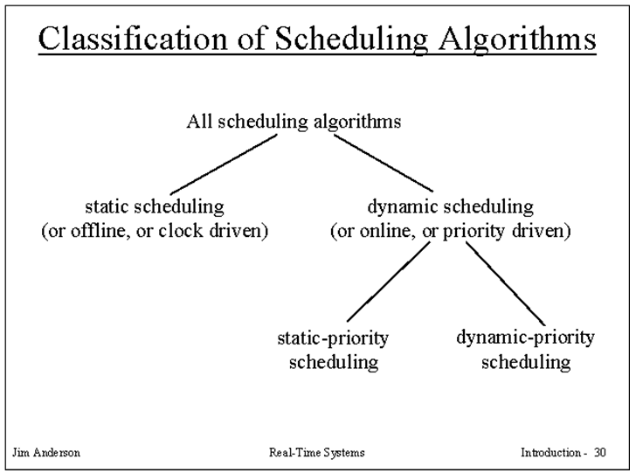
(D=absolute deadline)



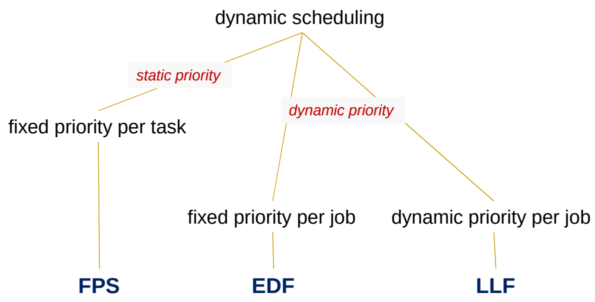
Needs preemption and off line decisions

Predictability of execution

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



Ramifications for dynamic scheduling



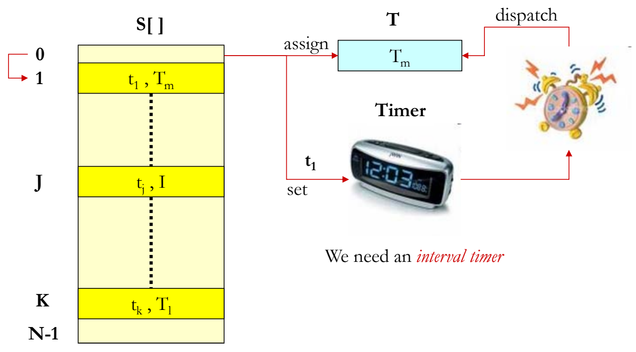
Clock-driven scheduling /1

- ***Workload model***
  - N periodic tasks with 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 off line before system start to a table **S** of decision times t_k
  - $S[t_k] = \tau_i$ if a job of task τ_i must be dispatched at time t_k
  - $S[t_k] = I$ (*idle*) otherwise
  - Schedule computation can be as sophisticated as we like since we pay for it only once and before execution
  - Jobs cannot overrun otherwise the system is in error

Clock-driven scheduling /2

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


Clock-driven scheduling /3



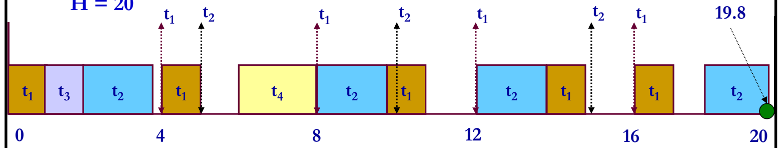
Example

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

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

U = 0.76

H = 20



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

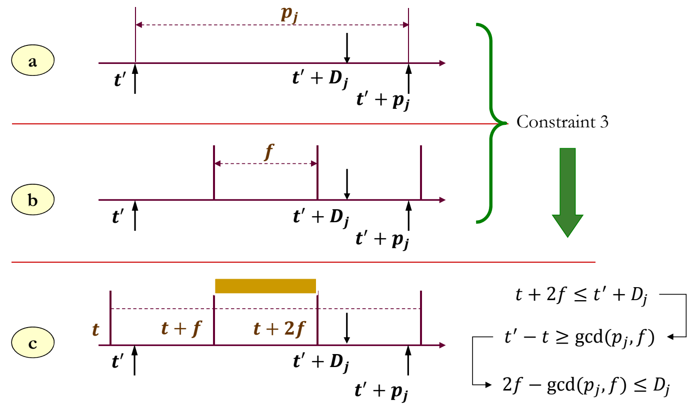
Clock-driven scheduling /4

- Obvious reasons suggest we should minimize the size and complexity 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 *periodic timer*
    - Within minor cycles there is no preemption but a single minor cycle may contain the execution of multiple (run-to-completion) jobs
  - φ_i for every task τ_i must be a non-negative integer multiple of f
    - The first job of every task has release time (forcedly) set at the beginning of a minor cycle
- We must therefore enforce some artificial constraints

Clock-driven scheduling /5

- **Constraint 1**: Every job J must complete within f
  - $f \geq max_{i=\{1,..n\}}(e_i)$ so that *overruns* can be detected
- **Constraint 2**: f must be an integer divisor of hyper-period H : $H = Nf$ where N is an integer
  - Satisfied if f is an integer divisor of at least one task period p_i
  - The hyper-period beginning at minor cycle kf for $k = 0, .. N-1$ 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 scheduled in that frame
  - This can be expressed as: $2f - \gcd(p_i, f) \leq D_i$ 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 T's jobs by ***slicing*** their larger e_{max} into fragments small enough to artificially yield a "good" *f*

Clock-driven scheduling /6

- To construct a cyclic schedule we must therefore make three design decisions
  - Fix an *f*
  - Slice (the large) jobs
  - Assign (jobs and) slices to minor cycles
- There is a very unfortunate inter-play among these decisions
  - Cyclic scheduling thus is very fragile to any change in system parameters

Clock-driven scheduling /7

```
Input: stored schedule S(k) for k = 0,..,F-1;
CYCLIC_EXECUTIVE:
  t := 0; k = 0;
  do forever:
    sleep until clock interrupt @ time t × f;
    currentBlock = S(k);
    t := t+1; k := t mod F;
    if last job not completed take action;
    end if;
    execute slices in currentBlock;
    while the aperiodic job queue is not empty do
      execute aperiodic job at top of queue;
    end do;
  end do;
end SCHEDULER
```

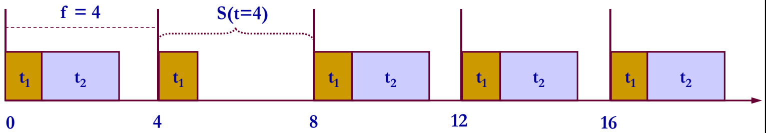
Example (slicing) – 1/2

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

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

τ_3 causes disruption since we need $e_3 \leq f \leq 4$ to satisfy c3

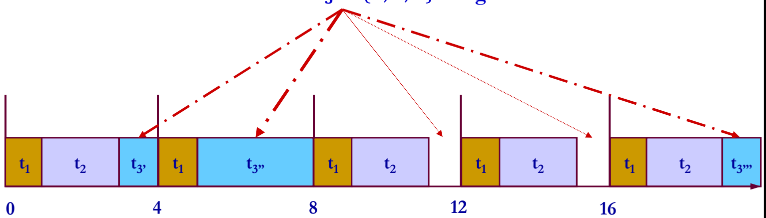
We must therefore slice e_3 : how many slices do we need?



We first look at the schedule with $f = 4$ and $F = \left(\frac{H}{f}\right) = 5$ without τ_3, to see what least-disruptive opportunities we have …

Example (slicing) – 2/2

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



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

where $x < y \leq 20$ 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)
  - This 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
- ***Static-priority scheduling***
  - All jobs of the same task have one and same priority

Dynamic-priority scheduling

- Two main algorithms
  - ***Earliest Deadline First*** (EDF)
  - ***Least Laxity First*** (LLF)
- **Theorem** [Liu, Layland: 1973] EDF is optimal for independent jobs with preemption
  - Also true with sporadic tasks
  - The relative deadline for periodic tasks may be arbitrary with the respect to period (<, =, >)
- Result trivially applicable to LLF
- EDF is not optimal for jobs that do not allow preemption

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
  - What about "*execution*-monotonic"?
- 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!
- How to compare the performance of scheduling algorithms?
- ***Schedulable utilization*** is a useful criterion
  - A scheduling algorithm can produce a feasible schedule for a task set T on a single processor if $U(T)$ does not exceed its schedulable utilization

Dynamic scheduling: comparison criteria /2

- **Theorem** [Liu, Layland: 1973] for single processors the schedulable utilization of EDF is 1
- For arbitrary deadlines, the ***density*** $\delta_k = \frac{e_k}{\min(p_k, D_k)}$ is an important feasibility factor
  - $\Delta = \sum_k \delta_k > U$ if $D_i < p_i$ for some τ_i
  - Hence $\Delta \leq 1$ is a sufficient *schedulability test* for EDF

Dynamic scheduling: comparison criteria /3

- The schedulable utilization criterion alone is not sufficient: we must also consider predictability
- 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 hinder the tasks with higher priority than τ
- Under transient overload EDF becomes instable
  - For EDF a job that missed its deadline is more urgent than a job with a deadline in the future!

Dynamic scheduling: comparison criteria /3

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

Example (EDF) /1

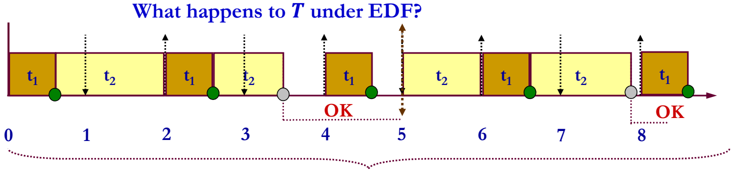
$(\varphi_i, p_i, e_i, D_i)$

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

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

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

What happens to T under EDF?

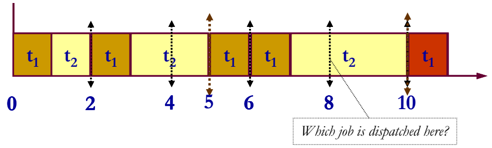


Example (EDF) /2

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

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

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)} ⇨ $U(t) = \frac{e_1}{p_1} + \frac{e_2}{p_2} = 1.1$

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 $U(t) = \frac{e_1}{p_1} + \frac{e_2}{p_2} = 1.2$?

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 $D_i \le p_i \; \forall i$, 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 task 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

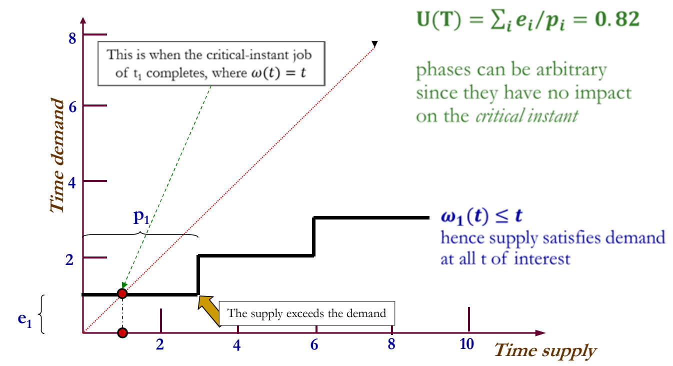
  - The summation term captures the *interference* that any job j of task τ_i incurs from jobs of all higher-priority tasks $\{\tau_k\}$ between the release time of the first job of task τ_k (with phase φ_k) to the response time of job j of task τ_i , which occurs at $\varphi_i + \omega_{i,j}$

Time-demand analysis /1

- When φ is 0 for all jobs considered then 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
  - So long as $\omega(t) \le t$ *for some t* within the time interval of interest the supply satisfies the demand, hence the job can complete in time
- **Theorem** [Lehoczky, Sha, Ding: 1989] condition $\omega(t) \le t$ 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

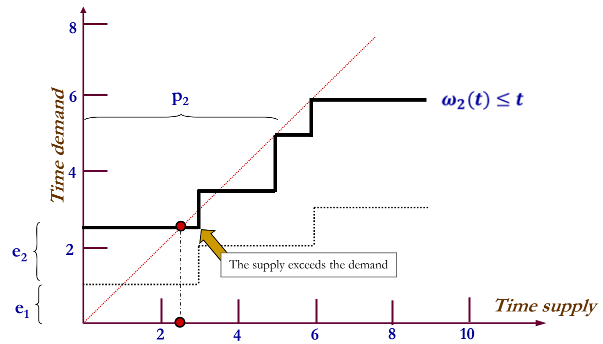
Time demand analysis /2

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



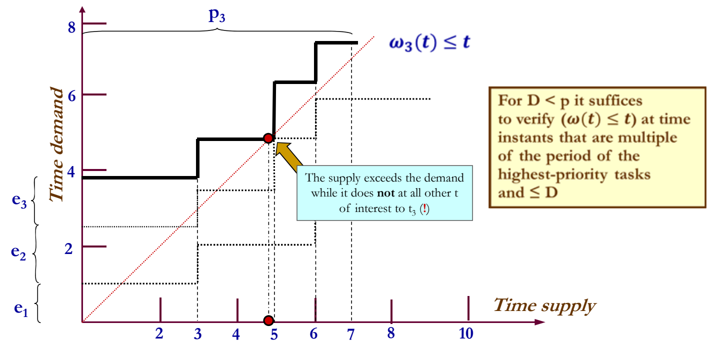
Time demand analysis /3

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)}



Time demand analysis /5

- It is straightforward to extend TDA to determine the *response time* of tasks
- The smallest value t that satisfies the fixed-point equation $t = e_i + \sum_{(k=1,..i-1)} \left\lceil \frac{t}{p_k} \right\rceil e_k$ 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]

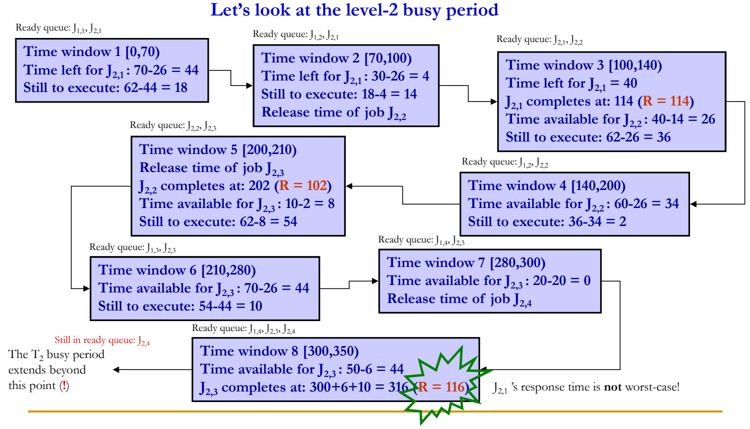
Time demand analysis /6

- 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

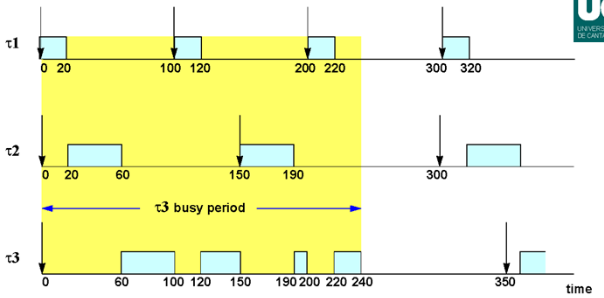


Level-i busy period

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!



Summary

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