182.086 Real-Time Scheduling

SS 2015

Prof. Ulrich Schmid

http://ti.tuwien.ac.at/ecs/teaching/courses/rt_sched

Technische Universität Wien
Institut für Technische Informatik
Embedded Computing Systems Group (E182/2)
Course Overview
Paradigm

Use an advanced scheduling algorithm, namely, Earliest Deadline First (EDF), for an in-depth treatment of:

- Modeling and Analysis
- Algorithms
- Schedulability results

Prerequisites:

- Basic mathematics and complexity theory
- Basic real-time scheduling knowledge
Course Content

What you will hear about:

- Uniprocessor systems under EDF
- Feasibility and response time analysis
- Complexity analysis
- Competitive analysis
Course Content

What you will hear about:
- Uniprocessor systems under EDF
- Feasibility and response time analysis
- Complexity analysis
- Competitive analysis

What you will NOT hear about:
- Different scheduling algorithms
- Real-time programming
- Worst-case execution time analysis
- Scheduling in multiprocessor and distributed systems
Some Background Info


- Schedule of lectures available on course homepage [http://ti.tuwien.ac.at/ecs/teaching/courses/rt_sched](http://ti.tuwien.ac.at/ecs/teaching/courses/rt_sched)
What to Do?

- **2 Quizzes** (45%), 15–20 min. each, covering the material taught since the predecessor quiz (+ very basics)
- **Final exam [optional]** (10%), either oral or 30-40 min. quiz covering the whole content of the course. Used as a make-up quiz in case you miss one of the former two quizzes.
- **2 Homework assignments** (45%), to be carefully, rigorously and completely worked out by yourself
  - \LaTeX\ paper version
  - presentation on blackboard in class
- **Participation in discussions in class** (± 10%)
General Rules

- Passing requires $\geq 60\%$ of the achievable maximum
- Presence in class is expected!
- Advance reading of textbook is recommended.
- All work must be done on your own and written up in your own words; all sources of information must be properly referenced (except textbook and slides)
- Graduate courses like this one adhere to “pull-based” learning — it is up to you to make the best use of this course!
Expected Achievements

Having passed this course, you should

- have got a basis for own work in this area
- have a somewhat improved formal/mathematical understanding (major rationale of most graduate basic courses in Mag-TI)
- have seen another example of “computer science ⊃ programming”

http://ti.tuwien.ac.at/ecs/teaching/courses/rt_sched
Follow-up Courses

Problems in Distributed Computing 182.703
- Overview lectures of advanced topics in distributed algorithms
- Common reading of papers and book chapters
- Student’s lectures

Embedded Systems Engineering Scientific Project 182.706
- First steps in own scientific work in a (self-)assigned distributed algorithms project
- Guided writing of a small scientific paper + presentation
- Learning about the scientific community in the field

⇒ Master thesis, Dissertation
Questions ?
Introduction to Real-Time Scheduling (Ch. 2)
Tasks (I)

A task is an executable entity of work, characterized by

- task execution properties
- task release constraints
- task completion constraints
- task dependencies

Tasks are executed in a system made up of

- one processor (CPU)
- other resources (communication links, shared data etc.)
Tasks (II)

Task execution properties:

- **Worst-case execution time**
- **Importance value**
- **Preemptive / non-preemptive** execution
- Task can / cannot be suspended during execution
Tasks (II)

Task execution properties:
- Worst-case execution time
- Importance value
- Preemptive / non-preemptive execution
- Task can / cannot be suspended during execution

Task release constraints:
- Periodic tasks: Activated periodically
- Sporadic tasks: Minimum inter-activation time
- Aperiodic tasks: No constraints
Tasks (III)

Task completion constraints:

- **Deadlines:**
  - Hard: Completion by the deadline mandatory
  - Firm: Completion by the deadline or no execution
  - Soft: Best-effort completion

- **[Jitter: Completion within some time interval]**
Tasks (III)

Task completion constraints:

- **Deadlines:**
  - Hard: Completion by the deadline mandatory
  - Firm: Completion by the deadline or no execution
  - Soft: Best-effort completion

- `[Jitter]: Completion within some time interval`

Task dependencies:

- **Precedence** relations among tasks
- **Resource sharing** during task execution (shared data, communication links, ...)

182.086 Real-Time Scheduling (http://ti.tuwien.ac.at/ecs/teaching/courses/rt_sched) – p. 14/208
Real-Time Scheduling

Schedule execution of a set of tasks
- on the processor and resources available in the system
- such that all timing constraints are met
⇒ Need a feasible schedule
Real-Time Scheduling

Schedule execution of a set of tasks
- on the processor and resources available in the system
- such that all timing constraints are met
⇒ Need a feasible schedule

2 dimensions of classification for real-time scheduling approaches:
- Static / dynamic
- Off-line / on-line
Static vs. Dynamic

Static scheduling assumes a priori knowledge of
- characteristic parameters of all tasks
- all future release times (e.g. periodic tasks)
⇒ Typically allows guarantees and optimal algorithms

Dynamic scheduling
- knows currently active task set
- does not know future arrivals
⇒ Guarantees and optimality more difficult to establish [if at all existent]
On-line vs. Off-line

**On-line** scheduling:
- All scheduling decisions are taken at run-time
- On-line algorithm could be
  - **Planning-based**: Admission control allows guarantees
  - **Best-effort**: Deadline misses possible

**Off-line** scheduling:
- Given a priori knowledge of tasks and future arrivals
- Compute off-line some information that simplifies on-line scheduling, like
  - dispatch tables for table-driven cyclic scheduler
  - priority mapping for rate-monotonic scheduling
Basic Notation

Task and Jobs:

- System of \( n \) tasks \( \tau_i \), \( 1 \leq i \leq n \), with WCET \( C_i \)
- Task execution consists of a sequence of jobs [= task instances]: \( J_{i,j} \) denotes \( j \)-th job of task \( \tau_i \)

Characteristics of job \( J_{i,j} \):

- **Release time** \( r_{i,j} \) when the job is ready for execution
- **Deadline** \( d_{i,j} \) when the job must have completed
- Sometimes: Relative deadline \( D_{i,j} = d_{i,j} - r_{i,j} \)

Sometimes no need to distinguish different tasks: \( J_j \) denotes \( j \)-th job, according to some unique enumeration
Task Release Characteristics

Distinguish 3 types of tasks:

- **Periodic tasks**: \( r_{i,j} = s_i + (j - 1)T_i \), with
  - period \( T_i \)
  - offset \( s_i \) (synchronous tasks: \( \forall i : s_i = 0 \))
  - Typical: Deadline = next release time \( d_{i,j} = r_{i,j+1} \)

- **Sporadic tasks**: \( r_{i,j+1} \geq r_{i,j} + T_i \), with
  - sporadicity interval \( T_i \)
  - Typical: Deadline = earliest next release time \( d_{i,j} = r_{i,j} + T_i \)

- **Aperiodic tasks**: No release restrictions, typically relative deadlines
Fundamentals of EDF Scheduling (Ch. 3)
Underlying Assumptions

General constraints in this part:

- Independent tasks on single processor

Varying constraints:

- Preemptive/non-preemptive
- Idling/non-idling
- Periodic/sporadic task sets
- Synchronous/asynchronous task sets
- Arbitrary/restricted deadlines
- With/without release jitter, scheduling overhead, . . .
Basic Terms

A schedule is
- **feasible** if all deadlines are met
- **non-idling** if the processor is never idle when there are jobs ready for execution

A real-time scheduling algorithm \( \mathcal{A} \) is **optimal** if
- whenever a certain task set \( \mathcal{T} \) can be feasibly scheduled by some scheduling algorithm \( \mathcal{B} \),
- then \( \mathcal{T} \) can also be feasibly scheduled by \( \mathcal{A} \)

Note that optimality results are typically restricted:
- Certain classes of algorithms (e.g. preemptive ones)
- Certain task sets (e.g. periodic ones)
Flavor of Upcoming Results

Optimality & Complexities:

- EDF optimality for single processor scheduling [no overloads]
- NP-hardness of feasibility checking for asynchronous periodic task sets with arbitrary deadlines

Feasibility analysis: EDF guarantees feasible schedule if

- maximum loading factor [processor demand/time] \( u \leq 1 \)
- processor utilization \( U \leq 1 \)
- maximum loading factor during busy period \( L = W(L) \)
  - \( u \leq 1 \) (preemptive EDF)
  - \( u < 1 \) (non-preemptive EDF)
Optimality and Complexity
Optimality of Jackson’s Rule (I)

Consider the following scheduling problem:

- $n$ independent jobs $J_i$ with execution time $C_i$ and deadline $d_i$, $1 \leq i \leq n$
- all jobs simultaneously released at time 0
- non-preemptive scheduling
- minimize maximum lateness $\ell = \max_{1 \leq i \leq n} \ell_i$ with $\ell_i = f_i - d_i$, where $f_i$ is the completion time of job $J_i$
Consider the following scheduling problem:

- $n$ independent jobs $J_i$ with execution time $C_i$ and deadline $d_i$, $1 \leq i \leq n$
- all jobs simultaneously released at time 0
- non-preemptive scheduling
- minimize maximum lateness $\ell = \max_{1 \leq i \leq n} \ell_i$ with $\ell_i = f_i - d_i$, where $f_i$ is the completion time of job $J_i$

**Theorem 25** (Jackson’s rule (= EDF)). *Any schedule that puts the jobs in order of non-decreasing deadlines minimizes the maximum lateness $\ell$.*

Note: EDF schedule not unique if $d_i = d_j$ for some $i \neq j$. 
Optimality of Jackson’s Rule (II)

Proof. (By contradiction.)

Assume maximum lateness is minimized by schedule \( S = J_1 J_2 \cdots J_n \) different from any EDF schedule.

We inductively construct schedules \( S_k, S_0 = S \), with non-increasing maximum lateness, which coincide with EDF schedule \( \mathcal{E} = J_{\mathcal{E}_1} J_{\mathcal{E}_2} \cdots J_{\mathcal{E}_n} \) in the first \( k \) jobs.

Given \( S_k \), let \( i = \max_{\mathcal{E}} \min_{i} \{ i | \forall \ell < i : J_\ell = J_{\mathcal{E}_\ell} \wedge J_i \neq J_{\mathcal{E}_i} \} \) be first index with \( J_i \neq J_{\mathcal{E}_i} \) in maximal matching EDF execution \( \mathcal{E} \). Then, \( i \geq k + 1, S_y = S_k \) for \( k + 1 \leq y < i \), and

- both \( J_i \) and \( J_{\mathcal{E}_i} \) are started at the same time \( t \) in \( S_k \) and \( \mathcal{E} \)
- \( J_{\mathcal{E}_i} = J_j \) with \( j > i \) since \( i \) first violation of EDF
- \( d_i > d_j \), since \( d_i = d_j \) would allow choosing another EDF schedule where \( J_{\mathcal{E}_i} = J_i \), violating the definition of \( i \)
- \( d_{i+1} \geq d_j, \ldots d_{j-1} \geq d_j \), since \( d_j \) nearest deadline in \( S_k \) after \( J_i \)
Optimality of Jackson’s Rule (III)

Proof. (cont.) Hence, in \( S_k \),

- \( J_j \) is the task with maximum lateness \( \ell_j \) among \( J_i, \ldots, J_j \)
- lateness of job \( J_m \), for \( i \leq m \leq j - 1 \), satisfies \( \ell_m \leq \ell_j - C_j \)

Exchanging \( J_i \) and \( J_j \) thus yields schedule \( S_i \) where

- \( J_j \) starts and completes much earlier, i.e., \( \ell'_j < \ell_j \)
- \( J_x, i + 1 \leq x \leq j - 1 \), start and complete at most \( C_j - C_i < C_j \) later, i.e., \( \ell'_x < \ell_j \)
- \( J_i \) completes at same time as \( J_j \) but has later deadline, i.e., \( \ell'_i < \ell_j \)

Thus, \( S_i \) coincides with \( E \) in the first \( i \) jobs and

- decreases the maximum lateness \( \ell = \max_{i \leq k \leq j} \ell_k \) of the jobs \( k \in [i, j] \) in \( S \), and preserves \( \ell'_x = \ell_x \) for \( x \not\in [i, j] \)

\( \square \)
Optimality of EDF

Is EDF still optimal if we enrich the model?

Adding non-zero release times:

- Non-preemptive scheduling becomes NP-hard

⇒ Non-preemptive EDF cannot be optimal since it is not clairvoyant (cannot predict future arrivals)
Optimality of EDF

Is EDF still optimal if we enrich the model?

Adding non-zero release times:
- Non-preemptive scheduling becomes NP-hard
- Non-preemptive EDF cannot be optimal since it is not clairvoyant (cannot predict future arrivals)

How to re-establish optimality:
- Restriction to non-idling policies: Non-idling non-preemptive EDF is optimal
- Allowing preemption \(\Rightarrow\) Preemptive EDF is optimal
- [EDF optimal also under various stochastic models.]
Non-idling Scheduling (I)

Useful lemma for every non-idling policy:

**Lemma 29.** Consider two non-idling schedules $S$ and $S'$ of the same task set $\mathcal{T}$. Then, the processor is idle at time $t$ in $S$ iff it is idle in $S'$. 
Non-idling Scheduling (I)

Useful lemma for every non-idling policy:

**Lemma 29.** Consider two non-idling schedules $S$ and $S'$ of the same task set $T$. Then, the processor is idle at time $t$ in $S$ iff it is idle in $S'$.

_Proof._ By contradiction (embedded in induction on number of idle/busy transitions):

Let

- $t$ be the first time when, w.l.o.g., the processor is idle in $S$ but busy in $S'$,
- $t_0 < t$ be the time of the processor's last transition from busy to idle before $t$ (with $t_0 = 0$ if none), both in $S$ and in $S'$.
- Clearly, the processor is idle in $[t_0, t)$ both in $S$ and in $S'$. 
Non-idling Scheduling (II)

Proof. (cont.)

Since the set $J$ of job releases in $[0, t)$ is the same in $S$ and in $S'$,

- the cumulative workload $C = \sum_{j \in J} C_j$ is the same in $S$ and $S'$,
- every job $\in J$ must have been processed by $t_0 < t$, i.e., $C \leq t_0 < t$. [Note: $C_j$ is independent of when a task is scheduled!]
- If the processor is indeed busy in $S'$ at time $t$, there must be a job release at $t$.
- This job should be assigned to the processor also in $S$ at time $t$, since the policy is non-idling.

Hence, the processor cannot be idle at $t$ in $S$. \hfill \square
Optimality non-idling non-pre EDF (I)

**Theorem 31** (George et.al). *Non-preemptive EDF is an optimal non-idling scheduling strategy that minimizes maximum lateness.*
Optimality non-idling non-pre EDF (I)

**Theorem 31** (George et.al). *Non-preemptive EDF is an optimal non-idling scheduling strategy that minimizes maximum lateness.*

*Proof.* Let any feasible schedule $S = J_1 J_2 \cdots J_n$ be given. We inductively construct schedules $S_k$, $S_0 = S$, with non-increasing maximum lateness, which coincide with EDF schedule $E = J_{E_1} J_{E_2} \cdots J_{E_n}$ in the first $k$ jobs.

According to the “non-idling” lemma, we can restrict our attention to the case where the processor is busy in any $S_k$ iff it is busy in $E$.

Given $S_k$, let again $i = \max_E \min_i \{ i | \forall \ell < i : J_\ell = J_{E_\ell} \land J_i \neq J_{E_i} \}$. Then,

- $J_{E_i} = J_j$ with $j > i$ since $i$ first violation of EDF
- $d_i > d_j$, since $d_i = d_j$ would allow choosing another EDF schedule where $J_{E_i} = J_i$, violating the definition of $i$
- $d_{i+1} \geq d_j, \ldots d_{j-1} \geq d_j$, since $d_j$ nearest deadline in $S_k$ after $J_i$
Optimality non-idling non-pre EDF (II)

Proof. (cont.)

Hence,

- in $S_k$, every job $J_m$ for $i \leq m \leq j - 1$ must actually complete by time $d_j - C_j \leq d_m - C_j$ at latest

- $J_j$ is already started in $E$ at the time $t$ when $J_i$ is started in $S_k$.

We can hence again swap $J_j$ and $J_i$ in $S_k$ to derive schedule $S_i$, without violating any deadline:

- $J_j$ starts and hence completes much earlier

- $J_i, \ldots, J_{j-1}$ start and complete at most $C_j - C_i < C_j$ later

- $J_i$ completes at same time as $J_j$ but has later deadline

Clearly, $S_i$ now coincides with $E$ in the first $i$ jobs and does not increase the maximum lateness.  \[ \square \]
Optimality of Preemptive EDF (I)

Preemptive scheduling is

- based upon discrete time model
- unit time "1"
- unit time = granularity of preemption
- usually non-idling (but of course not necessarily so)
Preemptive scheduling is

- based upon discrete time model
  - unit time "1"
  - unit time = granularity of preemption
- usually non-idling (but of course not necessarily so)

**Theorem 33** (Dertouzos). *Preemptive EDF is an optimal preemptive scheduling policy that minimizes maximum lateness.*
Optimality of Preemptive EDF (I)

Preemptive scheduling is

- based upon discrete time model
- unit time "1"
- unit time = granularity of preemption
- usually non-idling (but of course not necessarily so)

**Theorem 33** (Dertouzos). *Preemptive EDF is an optimal preemptive scheduling policy that minimizes maximum lateness.*

*Proof.* Let any feasible schedule $S$ be given. We inductively construct (by swapping time-slices) schedules $S_k$, $S_0 = S$, with non-increasing maximum lateness, which coincide with a preemptive EDF schedule $E$ in the first $k$ slots.

Induction basis: For $t = 0$, $S$ and $E$ are of course the same. 

□
Proof. (cont.)
Induction step: Suppose we have \( S^t \) that coincides with \( E \) in \([0, t)\).

Consider the time slice \([t, t + 1)\):

- If no job is executed in \( S^t \), schedule the first unassigned slot of the nearest-deadline ready job (or an idle slot if none) in \( E \).

- If \( J_i \) with deadline \( d_i \) is executed, let

  - \( J_j \) be the job with the smallest deadline \( d_j < d_i \) pending for execution at \( t \) [must exist, otherwise we have already EDF]
  - \( t_j > t \) be the first time when \( J_j \) is executed (in slot \([t_j, t_j + 1]\)) after \( t \)

- Since \( t_j < d_j < d_i \), we may swap \( J_i \) and \( J_j \) in \([t, t + 1)\) without violating feasibility.

\( \square \)
Optimality of Preemptive EDF (III)

Proof. (cont.)
Why is the maximum lateness not increased?

All latencies \( \ell'_m = \ell_m \) unchanged, except possibly for \( m = i, j \):

- If \([t, t + 1)\) last slot of \( J_i \) in \( S \): \( \ell_i < \ell_j \leq \ell_{\text{max}} \)
  - \( \ell'_j \leq \ell_j \) and
  - \( \ell'_i > \ell_i \), but still: \( \ell'_i < \ell_j \leq \ell_{\text{max}} \) since \( d_j < d_i \)

- If \([t, t + 1)\) not last slot of \( J_i \):
  - \( \ell'_j \leq \ell_j \) and
  - \( \ell'_i = \ell_i \)

Hence, the maximum lateness is never increased.

The induction step is completed, since we have transformed \( S^t \) into \( S^{t+1} \) that coincides with \( E \) in \([0, t + 1)\). \( \square \)
Feasibility Analysis

The results established for EDF so far reveal that no algorithm can do better in many settings.

- Just always using EDF should solve the real-time scheduling issue?
- What else can we do?
The results established for EDF so far reveal that no algorithm can do better in many settings.

- Just always using EDF should solve the real-time scheduling issue?
- What else can we do?

Unfortunately:

- Optimality results deal with feasible schedules only
- How can we know that some task set can be feasibly scheduled at all?
Feasibility Analysis

The results established for EDF so far reveal that no algorithm can do better in many settings.

- Just always using EDF should solve the real-time scheduling issue?
- What else can we do?

Unfortunately:

- Optimality results deal with feasible schedules only
- How can we know that some task set can be feasibly scheduled at all?

One needs techniques for feasibility analysis.
Feasibility Analysis

The purpose of feasibility tests is to decide whether some task set can be feasibly scheduled:

- Off-line tests: Allow to determine feasible settings of parameters like task periods
- On-line tests: Allows acceptance tests for guaranteed real-time properties [firm deadlines]
Feasibility Analysis

The purpose of feasibility tests is to decide whether some task set can be feasibly scheduled:

- Off-line tests: Allow to determine feasible settings of parameters like task periods
- On-line tests: Allows acceptance tests for guaranteed real-time properties [firm deadlines]

Discouraging result:

- Deciding whether a general asynchronous periodic task can be feasibly scheduled is NP-hard in the strong sense
- We hence need to restrict our task sets to get computationally tractable feasibility analysis techniques
Complexity Theory Refresher
Complexity Theory Refresher (I)

Recommended literature:

Complexity Theory Refresher (I)

Recommended literature:

In real-time scheduling, we are interested in 2 types of problems:
- **Optimization problems**: Given problem instance $I$, compute minimum or maximum solution $S(I)$
- **Decision problems**: Given problem instance $I$, compute decision $D(I) \in \{\text{yes, no}\}$
Optimization example: Travelling Salesman Problem (TSP)

- Problem instance $I$: A fully connected weighted graph with $n$ nodes
- Solution $S(I)$: A tour of the $n$ nodes with minimum weight sum

Decision example: Travelling Salesman Decision (TSD)

- Problem instance $I$: A fully connected weighted graph with $n$ nodes and a bound $B$
- Decision $D(I)$: Is there a tour of the $n$ nodes with weight sum $\leq B$?
- Complement problem coTSD: Is there no tour of the $n$ nodes with weight sum $\leq B$?
Problem complexity depends upon its size.

Input parameters in problem instances:
- Problem size (e.g. number of nodes \( n \) in TSP)
- Data size (e.g. size of variables holding the weights)

Problem complexity severely depends upon encoding of data:
- Binary encoding: \( s \) symbols (0 or 1) allow to encode \( V(s) = 2^s \) values \( \Rightarrow \) exponential complexity w.r.t. \( s \)
- Unary encoding: \( s \) symbols (1 only) allow to encode \( V(s) = s \) values \( \Rightarrow \) linear complexity w.r.t. \( s \)
Complexity Theory Refresher (IV)

Theory of NP-completeness is restricted to (language) decision problems

- **P**: Problems solvable by a deterministic Turing machine in polynomial time

- **NP**: Problems solvable by a non-deterministic Turing machine in polynomial time

We distinguish

- **Polynomial** complexity: Polynomial time w.r.t. code length $s$

- **Pseudo-polynomial** complexity: Polynomial time w.r.t. numerical value $V(s)$ [= polynomial when using only unary encoding].
Example: Travelling Salesman Decision TSD ∈ NP

- Non-deterministically guessing the [right] out of the $n!$ permutations of nodes in $I$
- Summing all weights along the resulting cycle and comparing whether it has weight sum at most $B$
- All these operations can be done in polynomial time $\Rightarrow$ TSD ∈ NP
Complexity Theory Refresher (VI)

Example: Complement Travelling Salesman Decision
\[ \text{coTSD} \in \text{coNP} \]

- Cannot solve coTSD via solution algorithm for TSD:
  - Checking Yes-instance of coTSD requires checking No-instance of TSD
  - TSD solution algorithm need not be polynomial (need not even halt) for No-instances!
- Since coNP \( \neq \) NP (if P \( \neq \) NP), it follows that coTSD \( \not\in \) NP
Alternative characterization of NP: Problems where

- yes-instances provide a certificate of polynomial size that can be verified in polynomial time
- no-instances need not provide such a certificate at all
Alternative characterization of NP: Problems where
- yes-instances provide a certificate of polynomial size
  that can be verified in polynomial time
- no-instances need not provide such a certificate at all

Example: Travelling Salesman Decision TSD $\in$ NP
- Take the tour $C(I)$ that is proposed as certificate
- $C(I)$ has polynomial size and can polynomially be verified to have weight $\leq B$
- Recall: No certificate needs to be provided for No-instances $\Rightarrow$ cannot derive a certificate for coTSD from TSD certificate!
Reduction of decision problems:

Consider two decision problems $P$ and $Q$

$P \leq Q$: $P$ can be (pseudo-)polynomially reduced to $Q$ if there is a (pseudo-)polynomially computable function $f$ that maps every $I_P$ to some $I_Q = f(I_P)$

$D(I_P) = \text{yes} \iff D(I_Q) = \text{yes}$ for every $I_Q = f(I_P)$
Complexity Theory Refresher (VIII)

Reduction of decision problems:
- Consider two decision problems $P$ and $Q$
- $P \leq Q$: $P$ can be (pseudo-)polynomially reduced to $Q$ if
  - there is a (pseudo-)polynomially computable function $f$ that maps every $I_P$ to some $I_Q = f(I_P)$
  - $D(I_P) = \text{yes} \iff D(I_Q) = \text{yes}$ for every $I_Q = f(I_P)$

Properties: If $P \leq Q$ and $Q \leq R$, then
- $P \leq R$ (transitivity) [but not $Q \leq P$ (symmetry)]
- if $Q$ is polynomially solveable, then $P$ is also polynomially solveable
- if $P$ cannot be solved polynomially, then $Q$ cannot be solved polynomially
Complexity Theory Refresher (IX)

A decision problem $P$ is
- **NP-hard** if all problems $Q \in \text{NP}$ satisfy $Q \leq P$
- **coNP-hard** if all problems $Q \in \text{NP}$ satisfy $Q \leq coP$
- **NP-complete** if $P$ is NP-hard and $P \in \text{NP}$
- **coNP-complete** if $P$ is coNP-hard and $coP \in \text{NP}$

A decision problem $P$ is
- (co)NP-hard/complete in the **strong** sense if it is w.r.t. polynomial complexity [unary data encoding]
- (co)NP-hard/complete in the **ordinary** sense if it is w.r.t. pseudo-polynomial time [binary data encoding]
How to do a NP-hardness/completeness proof?

- Need to show that all problems \( Q \in \text{NP} \) satisfy \( Q \leq P \)
- Since we do not know all \( Q \), this is impossible in a direct way
How to do a NP-hardness/completeness proof?

- Need to show that all problems $Q \in \text{NP}$ satisfy $Q \leq P$
- Since we do not know all $Q$, this is impossible in a direct way

But:

- There are problems like SATISFYABILITY $\in \text{NP}$, which have been shown (directly, via Turing machines) to satisfy $P \leq \text{SATISFYABILITY}$ for all $P \in \text{NP}$
- For a new problem $Q$, it hence suffices to show that some problem $P$ known to be NP-hard can be reduced to $Q$, i.e., satisfies $P \leq Q$
Complexity Theory Refresher (XI)

Scheduling often deals with optimization problems, so theory of NP-completeness not directly applicable. Remedy:

- Typically, it is possible to compute some lower bound $LB$ and upper bound $UB$ for [discrete value] optimal solution $S(I)$
  - in polynomial time
  - with $UB - LB$ being polynomial
- Use a (binary) search $\in [LB, UB]$ for determining bound $B$ and run an instance of the corresponding decision problem with bound $B$

$\Rightarrow$ Solution to optimization problem, with complexity mainly determined by the decision problem
NP-Hardness of Feasibility Analysis
A well-known decision problem:

**Definition 51** (Simultaneous Congruences Problem (SCP)). Given a set $A = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of ordered pairs of positive integers, and an integer $k$, $2 \leq k \leq n$. Determine whether there is a subset $A' \subseteq A$ of $k$ ordered pairs and a positive integer $z$ such that for all $(x_i, y_i) \in A'$ it holds $z \equiv x_i \mod y_i$.

A well-known result:
A well-known decision problem:

**Definition 51** (Simultaneous Congruences Problem (SCP)). Given a set $A = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of ordered pairs of positive integers, and an integer $k$, $2 \leq k \leq n$. Determine whether there is a subset $A' \subseteq A$ of $k$ ordered pairs and a positive integer $z$ such that for all $(x_i, y_i) \in A'$ it holds $z \equiv x_i \mod y_i$.

A well-known result:

**Theorem 51** (Leung and Witehead; Baruah et.al.). The SCP is NP-hard in the strong sense.
We will use polynomial reduction to show:

**Theorem 52** (Leung and Merrill; Baruah et.al.). *The feasibility analysis problem for asynchronous periodic task systems is coNP-hard in the strong sense.*

Note that this can be extended to arbitrarily small utilizations!
We will use polynomial reduction to show:

**Theorem 52** (Leung and Merrill; Baruah et.al.). *The feasibility analysis problem for asynchronous periodic task systems is coNP-hard in the strong sense.*

Note that this can be extended to arbitrarily small utilizations!

*Proof.* We have to show that every problem instance $I_{SCP}$ can be mapped to a problem instance $I_{coFA}$, where coFA means that some task set cannot be feasibly scheduled.

Note that we can select specific problem instances in coFA that suit our needs [with $D_i = 1, C_i = 1/(k - 1)$]; we do not need to cover the whole set coFA!
NP-Hard Feasibility Analysis (III)

Proof. (cont.)

Proof intuition:

- \( z \equiv x_i \mod y_i \) means that there is some \( m_i \) ensuring \( x_i + m_i \cdot y_i = z \)

- if this is simultaneously true for \( k \) indices \( i \), this can be mapped to \( k \) jobs that are simultaneously released at time \( z \)

- if we make their cumulative execution time larger than their deadlines, the task set cannot be feasibly scheduled

Given \( I_{SCP} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \) and some \( k \geq 2 \), we choose the following periodic task set:

- Offset \( s_i = x_i \), period \( T_i = y_i \)

- Relative deadline \( D_i = 1 \), WCET \( C_i = \frac{1}{k-1} \)
NP-Hard Feasibility Analysis (IV)

Proof. Given a yes-instance of $I_{SCP}$, this implies that

- $k$ jobs collide at time $z$
- their cumulative execution time is $k/(k - 1) > 1$, so some job misses its deadline

$\Rightarrow$ The task set cannot be feasibly scheduled

Given an infeasible task set (not schedulable by EDF)

- let $t_0$ be the first time of a deadline violation
- all task parameters (except WCET) are integers $\Rightarrow t_0$ is an integer
- for a deadline miss, $\geq k$ jobs must have arrived at time $z = t_0 - 1$
- all $k$ colliding tasks $\tau_i$ must satisfy $s_i + m_i \cdot T_i = z$ for some $m_i$

$\Rightarrow$ the corresponding $I_{SCP}$ is a yes-instance.

The mapping function can of course be computed in polynomial time. $\Box$
EDF Optimality Revisited (I)

We established earlier that preemptive EDF is optimal and minimizes maximum lateness:

- For every \textbf{preemptive} schedule \( S \), there is some as least as good preemptive EDF schedule \( E \).

- We proved even more: For every schedule \( S \),
  - preemptive or not
  - idling or not
there is some as least as good preemptive EDF schedule \( E \).
EDF Optimality Revisited (I)

We established earlier that preemptive EDF is optimal and minimizes maximum lateness:

- For every preemptive schedule \( S \), there is some as least as good preemptive EDF schedule \( E \).
- We proved even more: For every schedule \( S \), preemptive or not, idling or not, there is some as least as good preemptive EDF schedule \( E \).

Why not using EDF as a coFA test: If preemptive EDF cannot schedule some task set, return ‘´No”
- can schedule some task set, return “Yes”
EDF Optimality Revisited (II)

Does this contradict coNP hardness of feasibility analysis for asynchronous task sets with arbitrary deadlines?
EDF Optimality Revisited (II)

Does this contradict coNP hardness of feasibility analysis for asynchronous task sets with arbitrary deadlines?

There is no contradiction here:

- How long do we need to run EDF until we find some deadline violation, if any?
- May have arbitrary running time for periodic task sets with arbitrary deadlines $D_i \leq T_i$
Does this contradict coNP hardness of feasibility analysis for asynchronous task sets with arbitrary deadlines?

There is no contradiction here:

- How long do we need to run EDF until we find some deadline violation, if any?
- May have arbitrary running time for periodic task sets with arbitrary deadlines $D_i \leq T_i$

But:

- Will establish running time bounds later on, which are polynomial in case of $D_i = T_i$ and $D_i \geq T_i$, for example.
- There are indeed efficient preemptive EDF-based feasibility tests for restricted cases.
Feasibility Analysis Techniques
Informal definitions:

- **Processor demand** is the computation time requested by timely tasks in a given interval of time.
- **Loading factor** is the proportion of processor demand over the time interval.

Intuitively,

- no scheduler can handle a task set where the processor demand in some interval exceeds this interval.
- the best one can hope is an algorithm that schedules task sets with loading factor 1.

We show that preemptive EDF accomplishes this ⇒ optimal.
**Definitions**

**Definition 59.** Given a set of real-time jobs and a release time interval $[t_1, t_2)$, which is the interval $[t_1, t_2]$ with task releases excluded at $t_2$, the processor demand $h_{[t_1,t_2)}$ of the job set in the interval $[t_1, t_2)$ is

$$h_{[t_1,t_2)} = \sum_{t_1 \leq r_k, d_k \leq t_2} C_k.$$ 

**Definition 59.** Given a set of real-time jobs, its loading factor $u_{[t_1,t_2)}$ in the interval $[t_1, t_2)$ is

$$u_{[t_1,t_2)} = \frac{h_{[t_1,t_2)}}{t_2 - t_1}.$$ 

The maximum loading factor $u$, simply called loading factor, is

$$u = \sup_{0 \leq t_1 < t_2 < \infty} u_{[t_1,t_2)}.$$
Example

Consider 3 jobs:

- $J_1$ with WCET $C_1 = 4$, $r_1 = 3$ and $d_1 = 18$
- $J_2$ with WCET $C_2 = 4$, $r_2 = 5$ and $d_2 = 12$
- $J_3$ with WCET $C_3 = 6$, $r_3 = 6$ and $d_3 = 14$

| $u[3,18]$ | $\frac{4+4+6}{15} = \frac{14}{15}$ |
| $u[5,14]$ | $\frac{4+6}{9} = \frac{10}{9}$ |
| $u[5,12]$ | $\frac{4}{7}$ |
| $u[6,14]$ | $\frac{6}{8}$ |

Hence, $u = \frac{10}{9} > 1$ in this example.
Busy Periods (I)

If the processor is busy at time $t$, we say that

- $t$ is properly within a busy period, iff some job released before $t$ is executed after $t$

- $t$ delimits a busy period, iff no job released before $t$ is executed after $t$

If the processor is busy at $t$, then the unique busy period $BP(t) = [t_1, t_2)$ containing $t$ is defined by its

- start delim. time
  \[ t_1 = \inf_{t' \leq t} \{ t' \mid t' \text{ is properly within } BP(t) \} \]

- end delim. time
  \[ t_2 = \sup_{t'' \geq t} \{ t'' \mid t'' \text{ is properly within } BP(t) \} \]
Busy Periods (II)

Consequently,

- a busy period is a “maximal” period of continuous processor activity
- every busy period is started by a task release and ends upon a task completion
- two busy periods \([t_1, t_2]\) and \([t'_1, t'_2]\) may occur back-to-back, i.e., \(t'_1 = t_2\)

Any schedule consists of alternating busy and idle periods,

- independently of the scheduling policy (for non-idling policies)
- possibly involving idle periods of duration 0
Deadline-\( t \) Busy Periods (I)

We say that some time \( t_0 \) where the processor is busy with a job with deadline \( \leq t \) is

- **properly within a deadline-\( t \) busy period**, iff some job with deadline \( \leq t \) released before \( t_0 \) is executed after \( t_0 \)
- **delimits a deadline-\( t \) busy period**, iff no job with deadline \( \leq t \) [but possibly a job with deadline \( > t \)] released before \( t_0 \) is executed after \( t_0 \)

If the processor is busy with a job with deadline \( \leq t \) at \( t_0 \), then the unique **deadline-\( t \) busy period** \( BP_t(t_0) = [t_1, t_2] \) containing \( t_0 \) is defined by its

- start del. time \( t_1 = \inf_{t' \leq t_0} t' \) is properly within \( BP_t(t_0) \)
- end delim. time \( t_2 = \sup_{t'' \geq t} t'' \) is properly within \( BP_t(t_0) \)
Deadline-\(t\) Busy Periods (II)

In case of preemptive EDF scheduling, when there are pending jobs with deadline \(\leq t\), then
- the processor cannot be idle
- no job with deadline \(> t\) is executed

Consequently, in case of preemptive EDF scheduling, a deadline-\(t\) busy period
- is a busy period where only jobs with deadlines \(\leq t\) are executed
- is started by a deadline \(\leq t\) task release and ends upon a deadline \(\leq t\) task completion
- must end before \(t\) if there is no deadline miss
- may immediately be followed by another deadline-\(t\) b.p.
EDF Loading Factor Condition (I)

**Theorem 65** (Spuri). A set of real-time jobs is feasibly scheduled by preemptive EDF if and only if \( u \leq 1 \).
EDF Loading Factor Condition (I)

**Theorem 65** (Spuri). A set of real-time jobs is feasibly scheduled by preemptive EDF if and only if \( u \leq 1 \).

**Proof.** Direction \( \Leftarrow \): We show that \( u \leq 1 \Rightarrow \) feasible schedule by showing that non-feasible schedule \( \Rightarrow u > 1 \).

Assuming that the first deadline miss occurs at time \( t \), this must happen in a deadline-\( t \) busy period \([t_1, t_2)\) with \( t < t_2 \). Hence,

- the processor demand \( h_{[t_1, t)} \) must be excessive, i.e.,

\[
h_{[t_1, t)} = \sum_{t_1 \leq r_k, d_k \leq t} C_k > t - t_1
\]

\( \Rightarrow u_{[t_1, t)} > 1 \) and hence \( u > 1 \).

This contradicts \( u \leq 1 \) and thus completes the proof for direction \( \Leftarrow \). \( \Box \)
Proof. (cont.)

**Direction \(\Rightarrow\):** Since the schedule is feasible,

- the processor demand in any interval \([t_1, t_2)\) cannot exceed the interval’s length, i.e.,

\[
\forall [t_1, t_2) : h_{[t_1, t_2)} = \sum_{t_1 \leq r_k, d_k \leq t_2} C_k \leq t_2 - t_1
\]

\[\Rightarrow\] \(\forall [t_1, t_2) : u_{[t_1, t_2)} \leq 1\) and hence \(u \leq 1\),

This completes the proof for direction \(\Rightarrow\). \(\square\)
EDF Loading Factor Condition (III)

Since preemptive EDF is optimal and the loading factor test works for all task sets

- there is a loading-factor-based feasibility test for arbitrary task sets

- contradicts coNP hardness of feasibility analysis for asynchronous task sets with arbitrary deadlines?
EDF Loading Factor Condition (III)

Since preemptive EDF is optimal and the loading factor test works for all task sets

- there is a loading-factor-based feasibility test for arbitrary task sets
- contradicts coNP hardness of feasibility analysis for asynchronous task sets with arbitrary deadlines?

However, there is no contradiction here:

- **All** \([t_1, t_2)\) must be examined to determine \(u\)
- Leads to exponential complexity for periodic task sets with arbitrary deadlines \(D_i \leq T_i\)
EDF Loading Factor Condition (III)

Since preemptive EDF is optimal and the loading factor test works for all task sets

- there is a loading-factor-based feasibility test for arbitrary task sets
- contradicts coNP hardness of feasibility analysis for asynchronous task sets with arbitrary deadlines?

However, there is no contradiction here:

- All \( [t_1, t_2) \) must be examined to determine \( u \)
- Leads to exponential complexity for periodic task sets with arbitrary deadlines \( D_i \leq T_i \)

Again: One can derive efficient preemptive EDF feasibility tests for restricted cases [in particular, \( D_i = T_i \) and \( D_i \geq T_i \)].
The following lemma will prove useful on several occasions:

**Lemma 68.** Consider a periodic task with period $T_i$ and relative deadline $D_i = T_i$, and two arbitrary points in time $t_1, t_2$. There are

(a) at most $\left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor$ jobs from $\tau_i$ completely within $[t_1, t_2)$, as well as within $[t_1, t_2]$

(b) at most $\left\lceil \frac{t_2 - t_1}{T_i} \right\rceil$ job releases from $\tau_i$ within $[t_1, t_2)$

(c) at most $1 + \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor$ job releases from $\tau_i$ within $[t_1, t_2]$
Invocations Lemma (II)

Proof. The number of job invocations within $[t_1, t_2]$ is obviously maximized when there is a job release at time $t_1$.

Assume a job release at $t_1$ and distinguish 2 cases:

Case $T_i|(t_2 - t_1)$:

(a) exactly $\frac{t_2-t_1}{T_i} = \left\lfloor \frac{t_2-t_1}{T_i} \right\rfloor$ complete jobs fit into $[t_1, t_2)$ (as well as into $[t_1, t_2]$)

(b) exactly $\frac{t_2-t_1}{T_i} = \left\lceil \frac{t_2-t_1}{T_i} \right\rceil$ job releases occur within $[t_1, t_2)$ (the release at $t_2$ is not counted here)

(c) exactly $1 + \frac{t_2-t_1}{T_i} = 1 + \left\lfloor \frac{t_2-t_1}{T_i} \right\rfloor$ job releases occur within $[t_1, t_2]$ (the release at $t_2$ is counted here)
Invocations Lemma (III)

Proof. (cont.)

Case $T_i \not| (t_2 - t_1)$:

Here, $[t_1, t_2)$ and $[t_1, t_2]$ are the same since no release can take place at $t_2$ [remember that we assumed a release at $t_1$]

(a) exactly $\left\lfloor \frac{t_2-t_1}{T_i} \right\rfloor$ complete jobs fit completely into $[t_1, t_2)$

(b)+(c) exactly $1 + \left\lfloor \frac{t_2-t_1}{T_i} \right\rfloor = \left\lceil \frac{t_2-t_1}{T_i} \right\rceil$ job releases occur within $[t_1, t_2)$.

\qed
Invocations Lemma (III)

Proof. (cont.)

Case \( T_i \nmid (t_2 - t_1) \):

Here, \([t_1, t_2]\) and \([t_1, t_2]\) are the same since no release can take place at \( t_2 \) [remember that we assumed a release at \( t_1 \)]

(a) exactly \( \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor \) complete jobs fit completely into \([t_1, t_2]\)

(b)+(c) exactly \( 1 + \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor = \left\lceil \frac{t_2 - t_1}{T_i} \right\rceil \) job releases occur within \([t_1, t_2]\).

Note that, for any \( x \), it holds that

\[
\left\lfloor x \right\rfloor \leq x \leq \left\lceil x \right\rceil \leq \left\lceil x \right\rceil + 1
\]
Invocations Lemma with Deadlines

**Lemma 71.** Consider a task $\tau_i$ with period $T_i$ and arbitrary relative deadline $D_i$, and an interval $[t_1, t_2]$ with $t_2 - t_1 - D_i \geq 0$. The number of jobs with release times within $[t_1, t_2)$ and deadlines within $[t_1, t_2]$ is bounded by

$$1 + \left\lfloor \frac{t_2 - t_1 - D_i}{T_i} \right\rfloor.$$
**Invocations Lemma with Deadlines**

**Lemma 71.** Consider a task $\tau_i$ with period $T_i$ and arbitrary relative deadline $D_i$, and an interval $[t_1, t_2]$ with $t_2 - t_1 - D_i \geq 0$. The number of jobs with release times within $[t_1, t_2)$ and deadlines within $[t_1, t_2]$ is bounded by

$$1 + \left\lfloor \frac{t_2 - t_1 - D_i}{T_i} \right\rfloor.$$ 

**Proof.** According to the Invocations Lemma (c),

1. at most $1 + \left\lfloor \frac{t_2 - t_1 - D_i}{T_i} \right\rfloor$ task releases of $\tau_i$ may take place within $[t_1, t_2 - D_i]$

$\Rightarrow$ this is also the maximum number of complete jobs within $[t_1, t_2)$ in the presence of an arbitrary deadline $D_i \leq t_2 - t_1$. 

\[\square\]
Theorem 72. A set of \( n \) synchronous periodic tasks with \( \forall i : D_i = T_i \) can be feasibly scheduled by preemptive EDF if and only if

\[
U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.
\]
Feasibility Test by Liu & Layland (I)

**Theorem 72.** A set of $n$ synchronous periodic tasks with $\forall i : D_i = T_i$ can be feasibly scheduled by preemptive EDF if and only if

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.$$ 

**Proof.** Recalling the Invocation Lemma (a), for any $[t_1, t_2)$ it follows that

$$h_{[t_1, t_2)} = \sum_{t_1 \leq r_k, d_k \leq t_2} C_k \leq \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i \leq \sum_{i=1}^{n} \frac{t_2 - t_1}{T_i} C_i$$

such that $h_{[t_1, t_2)} \leq (t_2 - t_1)U$ and thus $u_{[t_1, t_2)} \leq U.$ \qed
Feasibility Test by Liu & Layland (II)

Proof. (cont.)

Choosing \( t_1 = 0 \) and \( t_2 = H \) with \( H = \text{lcm}(T_1, \ldots, T_n) \) denoting the length of the hyper-period, we get

\[
h_{[0,H)} = \sum_{0 \leq r_k, d_k \leq H} C_k = \sum_{i=1}^{n} \frac{H}{T_i} C_i
\]

such that \( h_{[0,H)} = H \cdot U \) and hence \( u_{[0,H)} = U \).

Recalling \( \forall (t_1, t_2) : u_{[t_1,t_2)} \leq U \), this implies \( u = U \).

The condition \( U \leq 1 \) hence follows immediately from the loading factor condition \( u \leq 1 \).

\( \square \)
Theorem 74. Any set of \( n \) asynchronous periodic tasks with \( \forall i : D_i = T_i \) can be feasibly scheduled by preemptive EDF if and only if

\[
U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.
\]
Theorem 74. Any set of \( n \) asynchronous periodic tasks with
\( \forall i : D_i = T_i \) can be feasibly scheduled by preemptive EDF if and only if

\[
U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1.
\]

Proof. Recalling the Invocation Lemma (a), for any \([t_1, t_2)\) it follows again that

\[
h_{[t_1,t_2)} = \sum_{t_1 \leq r_k, d_k \leq t_2} C_k \leq \sum_{i=1}^{n} \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i \leq \sum_{i=1}^{n} \frac{t_2 - t_1}{T_i} C_i
\]

such that \( h_{[t_1,t_2)} \leq (t_2 - t_1)U \) and thus \( u_{[t_1,t_2)} \leq U. \)
Feasibility Test by Coffman (II)

Proof. (cont.)

Choosing $t_1 = 0$ and $t_2 = s + m \cdot H$, where

- $s = \max\{s_1, \ldots, s_n\}$ is the maximum of all offsets
- $H = \text{lcm}(T_1, \ldots, T_n)$ is the length of the hyper-period
- $m \geq 1$ is an arbitrary integer

$$h_{[0,s+mH]} \geq \sum_{i=1}^{n} h_{[s_i,s_i+mH]}^i = \sum_{i=1}^{n} \frac{mH}{T_i}C_i = mHU$$

such that $u_{[0,s+mH]} \geq \frac{mH}{s+mH}U$ for any $m$.

Recalling $\forall [t_1, t_2) : u_{[t_1, t_2)} \leq U$, taking the limit $m \to \infty$ reveals $u = U$. The condition $U \leq 1$ hence follows again from the loading factor test condition $u \leq 1$. \qed

182.086 Real-Time Scheduling (http://ti.tuwien.ac.at/ecs/teaching/courses/rt_sched) – p. 75/208
Feasibility of Sporadic/Hybrid Tasks
Definition 77. A sporadic task set is feasible if, for any choice of release times compatible with the specified sporadicity intervals, the resulting job set is feasible.
Definition 77. A sporadic task set is feasible if, for any choice of release times compatible with the specified sporadicity intervals, the resulting job set is feasible.

Note that

- Dertouzos’ theorem revealed that preemptive EDF is also optimal for sporadic tasks
- Intuition tells that the worst case for sporadic task sets is the corresponding synchronous periodic one
Feasibility of Sporadic Tasks (I)

Definition 77. A sporadic task set is feasible if, for any choice of release times compatible with the specified sporadicity intervals, the resulting job set is feasible.

Note that

- Dertouzos’ theorem revealed that preemptive EDF is also optimal for sporadic tasks
- Intuition tells that the worst case for sporadic task sets is the corresponding synchronous periodic one

Theorem 77 (Sporadic demand). Given a set of \( n \) sporadic tasks \( \tau_i \) with sporadicity interval \( T_i \), and the corresponding synchronous periodic task set \( \tau'_i \) with period \( T_i \) and maximum loading factor \( u' \), any sequence of jobs \( J \) of the sporadic task set has a loading factor \( u \leq u' \).
Feasibility of Sporadic Tasks (II)

Proof. Let \( h_{[t_1,t_2]} \) and \( h'_{[t_1,t_2]} \) be the processor demand of the sporadic job set \( \mathcal{J} \) and the periodic job set \( \mathcal{J}' \), respectively.

We will show that, for all \([t_1, t_2)\), there is some \([t'_1, t'_2)\) with \( t_2 - t_1 = t'_2 - t'_1 \) such that \( h_{[t_1,t_2]} \leq h'_{[t'_1,t'_2]} \), which obviously implies \( u \leq u' \).

Consider the interval \([t_1, t_2)\). From the Invocations Lemma with Deadlines [71], we immediately obtain

\[
h_{[t_1,t_2]} \leq \sum_{i:D_i \leq t_2-t_1} \left( 1 + \left\lfloor \frac{t_2 - t_1 - D_i}{T_i} \right\rfloor \right) C_i
\]

\( \Box \)
Feasibility of Sporadic Tasks (III)

Proof. (cont.)

Choosing $t'_1 = 0$ and $t'_2 = t_2 - t_1$, the synchronous set $\mathcal{J}'$ obviously satisfies

$$h'_{[t'_1, t'_2)} = \sum_{i: D_i \leq t'_2 - t'_1} \left( 1 + \left\lfloor \frac{t'_2 - t'_1 - D_i}{T_i} \right\rfloor \right) C_i$$

Putting things together, $h_{[t_1, t_2)} \leq h'_{[t'_1, t'_2)}$ and hence $u \leq u'$ follows. □
Feasibility of Hybrid Task Sets

A hybrid task set consists of synchronous periodic and/or sporadic tasks with arbitrary deadlines.

- Due to the previous theorem, all results for synchronous periodic task sets generalize to hybrid task sets.

⇒ It makes sense to further study feasibility analysis of synchronous periodic task sets.

What is known for synchronous periodic/hybrid task sets with arbitrary deadlines?

- The general asynchronous problem is coNP-hard.
- The general synchronous problem is (weakly) coNP-hard.
- Necessary / sufficient conditions for EDF schedulability do exist.
Theorem 81 (Baruah et.al.). If a given hybrid task set is feasible under preemptive EDF scheduling, then $U \leq 1$. 
Necessary Hybrid Feasibility Condition

**Theorem 81** (Baruah et.al.). *If a given hybrid task set is feasible under preemptive EDF scheduling, then $U \leq 1$.*

**Proof.** Consider the interval $[0, mH + D_{\text{max}})$, where

- $D_{\text{max}} = \max\{D_1, \ldots, D_n\}$ is the maximum relative deadline
- $H = \text{lcm}(T_1, \ldots, T_n)$ is the length of the hyper-period
- $m \geq 1$ is an arbitrary integer

For the worst case of periodic jobs, it follows that

$$h_{[0,mH+D_{\text{max}})} \geq \sum_{i=1}^{n} \frac{mH}{T_i} C_i = mHU$$

such that $u_{[0,mH+D_{\text{max}})} \geq \frac{mH}{mH+D_{\text{max}}} U$ for any $m$. Taking $m \to \infty$ reveals $u \geq U$, so $U \leq 1$ by the EDF loading factor condition.  \qed
Hybrid Feasibility Condition for $D_i \geq T_i$

**Theorem 82** (Baruah et.al.). *Any hybrid task set with $\forall i : D_i \geq T_i$ is feasible under preemptive EDF scheduling if and only if $U \leq 1$.***
**Theorem 82** (Baruah et.al.). *Any hybrid task set with $\forall i : D_i \geq T_i$ is feasible under preemptive EDF scheduling if and only if $U \leq 1$.**

**Proof.** The previous theorem established already that feasible EDF schedule $\Rightarrow U \leq 1$. Hence, we only need to show the other direction $U \leq 1 \Rightarrow$ feasible schedule.

Consider the corresponding synchronous task set and any interval $[t_1, t_2)$. The processor demand is

$$h_{[t_1,t_2)} \leq \sum_{i:D_i \leq t_2-t_1} \left( 1 + \left\lfloor \frac{t_2 - t_1 - D_i}{T_i} \right\rfloor \right) C_i$$

$$\leq \sum_{i:D_i \leq t_2-t_1} \left( \frac{t_2 - t_1}{T_i} - \frac{D_i - T_i}{T_i} \right) C_i$$
Proof. (cont.)

\[ h_{[t_1, t_2]} \leq (t_2 - t_1) \sum_{i: D_i \leq t_2 - t_1} \frac{C_i}{T_i} \]

\[ \leq (t_2 - t_1) U \leq t_2 - t_1, \]

from where \( u \leq 1 \) follows. Applying the EDF loading factor condition completes the proof of our theorem. \( \square \)
Sufficient Hybrid Feasibility Condition (I)

**Theorem 84.** A hybrid task set of \( n \) tasks is feasible under preemptive EDF scheduling if

\[
\sum_{i=1}^{n} \frac{C_i}{\min\{D_i, T_i\}} \leq 1.
\]
Theorem 84. A hybrid task set of \( n \) tasks is feasible under preemptive EDF scheduling if

\[
\sum_{i=1}^{n} \frac{C_i}{\min\{D_i, T_i\}} \leq 1.
\]

Proof. Consider the corresponding synchronous task set and any interval \([t_1, t_2)\). The processor demand is

\[
h_{[t_1,t_2)} \leq \sum_{i: D_i \leq t_2-t_1} \left( 1 + \left[ \frac{t_2 - t_1 - D_i}{T_i} \right] \right) C_i
\]

\[
\leq \sum_{i: D_i \leq t_2-t_1} \left( 1 + \frac{t_2 - t_1 - \min\{D_i, T_i\}}{\min\{D_i, T_i\}} \right) C_i
\]
Sufficient Hybrid Feasibility Condition (II)

Proof. (cont.)

\[ h_{[t_1,t_2]} \leq \sum_{i: D_i \leq t_2 - t_1} \frac{t_2 - t_1}{\min\{D_i, T_i\}} C_i \]

\[ \leq (t_2 - t_1) \sum_{i=1}^{n} \frac{C_i}{\min\{D_i, T_i\}} \leq t_2 - t_1 \]

from where \( u \leq 1 \) follows. Applying the EDF loading factor condition completes the proof of our theorem.

\( \square \)
Sufficient Hybrid Feasibility Condition (II)

Proof. (cont.)

\[
\begin{align*}
  h_{[t_1,t_2]} & \leq \sum_{i: D_i \leq t_2 - t_1} \frac{t_2 - t_1}{\min\{D_i, T_i\}} C_i \\
  & \leq (t_2 - t_1) \sum_{i=1}^{n} \frac{C_i}{\min\{D_i, T_i\}} \leq t_2 - t_1
\end{align*}
\]

from where \( u \leq 1 \) follows. Applying the EDF loading factor condition completes the proof of our theorem.

Unfortunately, this condition is sufficient but not necessary (example Fig. 3.5).
The general EDF loading factor condition can be simplified for hybrid task sets:

- For any interval \([t_1, t_2)\), it holds that \(h_{[t_1, t_2]} \leq h'_{[0, t_2 - t_1]}\),
  where \(h'_{[0, t_2 - t_1]}\) is the processor demand of the corresponding synchronous task set [see the proof of the sporadic demand theorem]

- Consider processor demand \(h(t) = h'_{[0, t)}\) only
Hybrid Loading Factor Condition

The general EDF loading factor condition can be simplified for hybrid task sets:

- For any interval \([t_1, t_2)\), it holds that \(h_{[t_1,t_2)} \leq h'_{[0,t_2-t_1)}\), where \(h'_{[0,t_2-t_1)}\) is the processor demand of the corresponding synchronous task set [see the proof of the sporadic demand theorem]

- Consider processor demand \(h(t) = h'_{[0,t)}\) only

Theorem 86. A hybrid task set is feasible under preemptive EDF scheduling if and only if \(\forall t : h(t) \leq t\).
Hybrid Loading Factor Condition

The general EDF loading factor condition can be simplified for hybrid task sets:

- For any interval \([t_1, t_2)\), it holds that \(h_{[t_1, t_2)} \leq h'_{[0, t_2 - t_1)}\), where \(h'_{[0, t_2 - t_1)}\) is the processor demand of the corresponding synchronous task set [see the proof of the sporadic demand theorem]

- Consider processor demand \(h(t) = h'_{[0, t)}\) only

Theorem 86. A hybrid task set is feasible under preemptive EDF scheduling if and only if \(\forall t : h(t) \leq t\).

Proof. Follows immediately from \(h_{[t_1, t_2)} \leq h'_{[0, t_2 - t_1)}\) and the general loading factor condition.
Hybrid Loading Factor Test (I)

Feasibility test based upon hybrid loading factor condition
\[ \forall t : h(t) \leq t? \]

- Testing all intervals \([0, t)\) is not practical, since even incremental computation needs \(O(n)\) time per step

- Note that looking at intervals \([0, t]\) with \(t = d_{i,j}\) is sufficient here, since \(h(t)\) increases only at deadlines

- Are there conditions under which just all \(t \leq t_{\text{max}}\) need to be tested?
Hybrid Loading Factor Test (I)

Feasibility test based upon hybrid loading factor condition
\[ \forall t : h(t) \leq t? \]

- Testing all intervals \([0, t]\) is not practical, since even incremental computation needs \(O(n)\) time per step
- Note that looking at intervals \([0, t]\) with \(t = d_{i,j}\) is sufficient here, since \(h(t)\) increases only at deadlines
- Are there conditions under which just all \(t \leq t_{\text{max}}\) need to be tested? Yes.
Hybrid Loading Factor Test (I)

Feasibility test based upon hybrid loading factor condition
\[ \forall t : h(t) \leq t? \]

- Testing all intervals \([0, t]\) is not practical, since even incremental computation needs \(O(n)\) time per step
- Note that looking at intervals \([0, t]\) with \(t = d_{i,j}\) is sufficient here, since \(h(t)\) increases only at deadlines
- Are there conditions under which just all \(t \leq t_{\text{max}}\) need to be tested? Yes.

**Theorem 87** (Baruah et.al.). *If a hybrid task set of \(n\) tasks is not feasible under preemptive EDF scheduling and \(U < 1\), then \(h(t) > t\) implies*

- \(t < D_{\text{max}}\), or \(t < \frac{U}{1-U} \max_{1 \leq i \leq n} \{T_i - D_i\}\)
**Proof.** Assume $h(t) > t$. If $t < D_{\text{max}}$, we are done. Hence, assume that also $t \geq D_{\text{max}}$.

\[
h(t) \leq \sum_{i:D_i \leq t} \left( 1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor \right) C_i
\]

\[
\leq \sum_{i=1}^{n} \left( \frac{t + T_i - D_i}{T_i} \right) C_i
\]

\[
\leq t \sum_{i=1}^{n} \frac{C_i}{T_i} + \sum_{i=1}^{n} \frac{C_i}{T_i} (T_i - D_i)
\]

\[
\leq tU + \max_{1 \leq i \leq n} \{T_i - D_i\} U.
\]

\[\square\]
Proof. (cont.)

Recalling \( t < h(t) \), it follows that

\[
t(1 - U) < U \max_{1 \leq i \leq n} \{ T_i - D_i \}
\]

from where the statement in our theorem follows. \( \square \)
From this theorem, we conclude the following:

- Testing all intervals \([0, t]\) for
  \[t \leq t_{\text{max}} = \max \left\{ D_{\text{max}}, \frac{U}{1-U} \max_{1 \leq i \leq n} \{T_i - D_i\} \right\}\]
  suffices

- If \(U \leq c < 1\), the complexity of feasibility testing is pseudo-polynomial, since
  \[\frac{U}{1-U} \max_{1 \leq i \leq n} \{T_i - D_i\} \leq \frac{c}{1-c} \max_{1 \leq i \leq n} \{T_i - D_i\}\]
  checking \(h(t) \leq t\) can be done incrementally in \(O(n)\) time per step (= per job deadline)
  \[\Rightarrow\] test needs only \(O \left( n \max_{1 \leq i \leq n} \{D_{\text{max}}, T_i - D_i\} \right)\) time
Hybrid Loading Factor Test (IV)

From this theorem, we conclude the following:

- Testing all intervals $[0, t]$ for
  
  $t \leq t_{\text{max}} = \max \{ D_{\text{max}}, \frac{U}{1-U} \max_{1 \leq i \leq n} \{ T_i - D_i \} \}$ suffices

- If $U \leq c < 1$, the complexity of feasibility testing is pseudo-polynomial, since

  $\frac{U}{1-U} \max_{1 \leq i \leq n} \{ T_i - D_i \} \leq \frac{c}{1-c} \max_{1 \leq i \leq n} \{ T_i - D_i \}$

  checking $h(t) \leq t$ can be done incrementally in $O(n)$ time per step (= per job deadline)

  $\Rightarrow$ test needs only $O(\max_{1 \leq i \leq n} \{ D_{\text{max}}, T_i - D_i \})$ time

Are there further bounds on $t_{\text{max}}$?
Some additional results regarding the maximum time $t_{\text{max}}$ where $\forall t \leq t_{\text{max}} : h(t) \leq t$ must be checked:

- $t_{\text{max}} \leq H + D_{\text{max}}$, where $H = \text{lcm}(T_1, \ldots, T_n)$ for $U = 1$ [which implies exponential complexity]
- Zheng and Shin’s bound:

$$t_{\text{max}} = \max\left\{D_{\text{max}}, \sum_{i=1}^{n} \frac{(1 - D_i/T_i)C_i}{1 - U}\right\}$$

- George et.al.’s bound:

$$t_{\text{max}} = \sum_{D_i \leq T_i} \frac{(1 - D_i/T_i)C_i}{1 - U}$$
Busy Period Analysis
Recall Busy Periods

Given some time $t$ where the processor is busy, the unique busy period $[t_1, t_2)$ with $t_1 \leq t \leq t_2$ is a period of continuous processor activity, such that

- $t_1$ is the last instant where no job that was released before $t_1$ is executed after $t_1$
- $t_2$ is the first instant where no job that was released before $t_2$ is executed after $t_2$.

Any schedule consists of alternating busy and idle periods, although idle periods may have duration 0.

The synchronous busy period is the first busy period in the schedule of a synchronous task set.

The cumulative workload within some interval $[t_1, t_2)$ is the sum of the WCET of all jobs released within $[t_1, t_2)$. 
Recall Deadline-$t$ Busy Periods

A deadline-$t$ busy period $[t_1, t_2)$ is a busy period where only jobs with deadlines $\leq t$ are executed:

- $t_1 < t$ is the last instant where no jobs with deadline $\leq t$ that were released before $t_1$ are executed after $t_1$
- $t_2$ is the first instant where no jobs with deadline $\leq t$ [but possibly jobs with deadline $> t$] that were released before $t_2$ are executed after $t_2$.

**Corollary 94.** In case of preemptive EDF scheduling, a deadline-$t$ busy period $[t_1, t_2)$

- is initiated by the arrival of a task with deadline $\leq t$
- is non-idle throughout $[t_1, t_2]$
- guarantees that if $t_2 > t$, then the task with deadline $t$ cannot make it.
Synchronous Busy Period Feasibility (I)

The following generic theorem allows to further restrict the interval $[0, t_{\text{max}})$ where $h(t) \leq t$ must be checked:

**Theorem 95** (Liu and Layland and others). *If a synchronous periodic task set is not feasible under preemptive EDF, then there is a deadline miss in the synchronous busy period.*
Synchronous Busy Period Feasibility (I)

The following generic theorem allows to further restrict the interval $[0, t_{\text{max}})$ where $h(t) \leq t$ must be checked:

**Theorem 95** (Liu and Layland and others). *If a synchronous periodic task set is not feasible under preemptive EDF, then there is a deadline miss in the synchronous busy period.*

**Proof.** Suppose there is a deadline miss of a job of $\tau_i$ at time $t$ somewhere in the schedule.

- Let $[t', t'']$ with $t' < t < t''$ be the deadline-$t$ busy period containing the deadline miss.
- Note that $t$ must be the deadline of a job of $\tau_i$.
- $\Rightarrow$ The processor demand in $[t', t)$ must satisfy $h_{[t', t)} > t - t'$.
Synchronous Busy Period Feasibility (II)

Proof. (cont.)

Consider a schedule derived from the original one by:

- Dropping all task releases before $t'$. This cannot not change $h_{[t',t)}$ since a task released before $t'$ either
  - has a deadline $> t$ and is hence not included in $h_{[t',t)},$ or
  - has deadline $d \leq t$ but belongs to some preceding deadline-$d$ busy period, which must have been completed before $t'$
- Shifting all jobs from tasks $\tau_j \neq \tau_i$ that are within the deadline-$t$ busy period left, so that they are all synchronously released at $t'$
  - This can at most increase $h_{[t',t)},$ so $\tau_i$ will still miss its deadline at $t$
Synchronous Busy Period Feasibility (IV)

Proof. (cont.)

Now let also the first job of $\tau_i$ be released synchronously at $t'$:

- The processor demand $h[t',t]$ in $[t',t)$ can only increase
- Let $\hat{t}$ be the largest deadline before or at $t$ in the resulting schedule, with $\tau_\ell$ being the according task
- $\tau_\ell$ experiences a deadline miss at $\hat{t}$:
  - The processor demand $h[t',\hat{t}) = h[t',t)$, since $\hat{t}$ was last deadline included
  - Length of deadline-$\hat{t}$ busy period is larger than $t - t' \geq \hat{t} - t'$
- The resulting schedule is equivalent to the synchronous busy period (except that it starts at $t'$) and contains a deadline miss.

□
**Theorem 98.** Independently of the [non-idling] scheduling policy employed, the length $L$ of the synchronous busy period of a synchronous task set of $n$ tasks is the [unique] limiting point of

$$L^{(m+1)} = W \left( L^{(m)} \right)$$

$$L^{(0)} = \sum_{i=1}^{n} C_i,$$

where

$$W(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil C_i.$$
Busy Period Length (II)

Proof. For a given interval $[0, t)$ the (proof of the) Invocation Lemma (b) reveals that $W(t)$ is the cumulative workload within $[0, t)$. Clearly,

- $W(t)$ is piecewise constant
- $W(\varepsilon) = \sum_{i=1}^{n} C_i > 0$, for any arbitrarily small $\varepsilon > 0$
- $W(t + \Delta) \geq W(t)$ for all $t, \Delta \geq 0$ (weak monononicity)
- if $W(t + \Delta) \neq W(t)$ then $W(t + \Delta) \geq W(t) + C_{\text{min}}$
- $\lim_{t\to\infty} W(t) = \infty$

Define $L^{(-1)} = \varepsilon$. Using induction on $m \geq 0$, we show that

(1) $L^{(m)} \geq L^{(m-1)}$

(2) the length of the synchronous busy period $L$ is at least $L^{(m)}$
Busy Period Length (III)

*Proof. (cont.)*

**Basis m = 0:** Dealing with a synchronous periodic task set, all tasks are released at time 0, hence:

- The resulting busy period(s) must have length at least
  \[ L^{(0)} = \sum_{i=1}^{n} C_i > \varepsilon = L^{(-1)}, \]
  since all tasks released at 0 must be executed to completion

- There is no idle time within \([0, L^{(0)}]\) \(\Rightarrow\) contained in the synchronous busy period. Hence, \(L \geq L^{(0)}\). 

\(\square\)
Busy Period Length (IV)

Proof. (cont.)

Induction step $m \rightarrow m + 1$: Assuming $L^{(m)} \geq L^{(m-1)}$ and $L \geq L^{(m)}$,

- the cumulative workload within the interval $[0, L^{(m)}]$ is
  
  \[ W(L^{(m)}) = L^{(m+1)} \geq L^{(m)} = W(L^{(m-1)}) \], since otherwise
  
  \[ L^{(m)} < L^{(m-1)} \] by weak monotonicity of $W(.)$

- There is no idle time within $[0, L^{(m+1)}]$, since
  
  - there is no idle time within $[0, L^{(m)}]$, and
  
  - any additional workload [if any] is caused by task releases within
    $[0, L^{(m)})$

  \[ \Rightarrow \] the processor must be busy within $[L^{(m)}, L^{(m+1)}]$ as well.

  \[ \Rightarrow L \geq L^{(m+1)}. \]

\[ \square \]
Busy Period Length (V)

Proof. (cont.)

Since $L^{(m)}$ is monotonically increasing, it has a limit point:

- $L = \lim_{m \to \infty} L^{(m)} = \infty$: Non-terminating synchronous busy period.

- $L = \lim_{m \to \infty} L^{(m)} < \infty$: Since $W(\cdot)$ and hence $L^{(\cdot)}$ increases at least by $C_{\min}$ if it is increased at all, there must be a finite $m \geq 0$ where $L^{(m)} = L^{(m+1)}$.

- There are no additional releases in $[0, L^{(m)}]$ as compared to the releases in $[0, L^{(m-1)}]$.

- There must be an idle period following $t = L^{(m)}$.

$\Rightarrow$ The length of the synchronous busy period must be exactly $L = L^{(m)} = L^{(m+1)} = \ldots$
Convergence Condition (I)

**Theorem 103** (Spuri). If $U \leq 1$, then the synchronous busy period length $L < \infty$ is computed in a finite number of iterations.
Theorem 103 (Spuri). If $U \leq 1$, then the synchronous busy period length $L < \infty$ is computed in a finite number of iterations.

Proof. Let $H = \text{lcm}(T_1, \ldots, T_n)$ be the length of the hyper-period. Then, $U \leq 1$ implies

$$W(H) = \sum_{i=1}^{n} \left\lceil \frac{H}{T_i} \right\rceil C_i = H \cdot U \leq H$$

and, due to weak monotonicity, $W(t) \leq H$ for all $0 \leq t \leq H$. 

□
Convergence Condition (II)

Proof. (cont.)

Hence,

- $\forall m : L^{(m)} \leq H$ since the iteration cannot exceed the value $H$
- Since $W(.)$ and hence $L^{(m)}$ increases by at least $C_{\text{min}}$ in every step where it increases at all, there is a finite $m$ such that $L = L^{(m)} = L^{(m+1)} = \ldots$

Consequently, the iteration for computing $L^{(m)}$ terminates after finitely many steps. \qed
Example

Consider task set of previous example:

\[
\begin{align*}
L^{(0)} &= W(0) = 3 + 2 + 1 = 6 \\
L^{(1)} &= W(6) = 2 \cdot 3 + 1 \cdot 2 + 1 \cdot 1 = 9 \\
L^{(2)} &= W(9) = 3 \cdot 3 + 1 \cdot 2 + 1 \cdot 1 = 12 \\
L^{(3)} &= W(12) = 3 \cdot 3 + 1 \cdot 2 + 2 \cdot 1 = 13 \\
L^{(4)} &= W(13) = 4 \cdot 3 + 1 \cdot 2 + 2 \cdot 1 = 16 \\
L^{(5)} &= W(16) = 4 \cdot 3 + 1 \cdot 2 + 2 \cdot 1 = 16 \\
L &= 16
\end{align*}
\]
Recast Busy Period Length Theorem

Theorem 106. Independently of the [non-idling] scheduling policy employed, the length $L$ of the synchronous busy period of a synchronous task set of $n$ tasks is the smallest positive solution of

$$ t = W(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil C_i. $$
Recast Busy Period Length Theorem

**Theorem 106.** Independently of the [non-idling] scheduling policy employed, the length $L$ of the synchronous busy period of a synchronous task set of $n$ tasks is the smallest positive solution of

$$t = W(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil C_i.$$

**Proof.** We only need to show that there cannot be a smaller solution $L' > 0$ satisfying $L' < L = L^{(m)} = L^{(m+1)}$.

- Clearly, $L' > \varepsilon = L^{(-1)}$ since $W(\varepsilon) = \sum_{i=1}^{n} C_i > \varepsilon$

- Since $W(t) \leq L'$ for all $t \leq L'$ due to weak monotonicity, the iteration cannot exceed $L'$

$$\Rightarrow \forall m': L^{(m')} \leq L' < L,$$ which contradicts $L^{(m)} = L$. 

182.086 Real-Time Scheduling (http://ti.tuwien.ac.at/ecs/teaching/courses/rt_sched) – p. 106/208
Being able to restrict feasibility analysis of a task set to $t \leq t_{\text{max}} = L$

can speedup feasibility analysis, although worst case complexity is not improved:

In particular, for $U \leq c < 1$,

\[
L = \sum_{i=1}^{n} \left\lceil \frac{L}{T_i} \right\rceil C_i \leq \sum_{i=1}^{n} \left( 1 + \frac{L}{T_i} \right) C_i \\
\leq \sum_{i=1}^{n} C_i + L \sum_{i=1}^{n} \frac{C_i}{T_i} \leq \sum_{i=1}^{n} C_i + Lc,
\]
Complexity of Busy Period Analysis (II)

Since this leads to $L \leq \frac{1}{1-c} \sum_{i=1}^{n} C_i$, we obtain:

- Pseudo-polynomial time complexity $O\left(n \sum_{i=1}^{n} C_i\right)$ for computing $L$ since every iteration takes $O(n)$

- Pseudo-polynomial time complexity $O\left(n \sum_{i=1}^{n} C_i\right)$ for feasibility analysis since incrementally computing $h(t + \Delta t) = h(t) + \sum_{i \in I_{\Delta t}} C_i$ takes $O(n)$ per deadline

Final loading factor feasibility test for synchronous/hybrid task sets with arbitrary deadlines and $U \leq 1$:

- Check $h(t) \leq t$ for $t = d_k \in [0, t_{\text{max}}]$, where $t_{\text{max}} = \min\{L, t_{\text{max}}^{(ZS)}, t_{\text{max}}^{(George)}, \ldots\}$

- At most exponential complexity; pseudo-polynomial complexity in many cases ($U \leq c < 1$)
Response Time Analysis (Ch. 4)
Alternative to Feasibility Analysis

Consider any job $J_{i,j}$ of task $\tau_i$:

- Release time $r_{i,j}$, relative deadline $D_i$
- Compute worst case response time $rt_{i,j}$ for $J_{i,j}$
- If $rt_i = \max_j rt_{i,j} \leq D_i$ for all $i, j$, the task set is feasible

Response time analysis

- is more difficult than feasibility analysis (= simple check $h(t) \leq t$) since response times must be computed recursively
- is more general and allows even end-to-end response time analysis in distributed systems (holistic schedulability analysis)
Worst Case Response Time (I)

Lemma 111. The worst case response time of job $J_{i,j}$ with deadline $d = d_{i,j}$ occurs in a $\tau_i$-peer-synchronous (PS) deadline-$d$ busy period $[t_1, t_2)$ containing $J_{i,j}$, in which all tasks except $\tau_i$ are released simultaneously at $t_1$ and with maximum rate.
Worst Case Response Time (I)

Lemma 111. The worst case response time of job $J_{i,j}$ with deadline $d = d_{i,j}$ occurs in a $\tau_i$-peer-synchronous (PS) deadline-$d$ busy period $[t_1, t_2)$ containing $J_{i,j}$, in which all tasks except $\tau_i$ are released simultaneously at $t_1$ and with maximum rate.

Proof. Consider a specific job $J_{i,j}$ of $\tau_i$, and let

- $a$ be its release time and $d = a + D_i$ its absolute deadline
- $f \geq a + C_i$ be its finishing time
- $[t_1, t_2)$ be the deadline-$d$ busy period containing $f$ (actually, containing the whole interval $[a, f]$), with
  - $t_1$ denoting the last time before $f$ without a job with deadline $\leq d$ released before and executed after $t_1$
  - $t_2$ denoting the first time at or after $f$ without a job with deadline $\leq d$ released before and executed after $t_2$
Worst Case Response Time (II)

Proof. (cont.) Since \([t_1, t_2]\) is a deadline-\(d\) busy period,

- \(t_1\) must be the release time of a task with deadline \(\leq d\)
- only jobs with deadline \(\leq d\) are executed during \([t_1, t_2]\), without idle time in between
- \(J_{i,j}\) need not be the last in deadline-\(d\) BP if there are jobs \(J_{k,l}\) with same deadline \(d\)

Consider the schedule where all tasks except \(\tau_i\) are shifted left such that they are released simultaneously and at their maximum rate from \(t_1\) on.

- Jobs released before \(t_1\) did not contribute to \([t_1, t_2]\) ⇒ do not contribute after the shift
- The proc. demand \(h_{[t_1,t]}\) for any \(t \in [t_1, t_2]\) cannot decrease
- Neither \(J_{i,j}\)’s completion time \((t = f)\) nor the length of the deadline-\(d\) busy period \((t = t_2)\) can decrease.
Worst Case Response Time (III)

Consider initial $\tau_i$-peer-synchronous (IPS) deadline-$d$ busy periods $[0, t')$, where

- $\tau_i$ has job release times $r_{i,j} = s_i + (j - 1)T_i$ with $s_i \geq 0$
- all tasks $\tau_k \neq \tau_i$ are released synchronously at time 0

The worst case response time for task $\tau_i$

- need not occur in an IPS deadline-$d$ busy period (i.e., possibly $t_1 \neq 0$ in the PS deadline-$d$ busy period $[t_1, t_2)$)
- BUT: We can nevertheless restrict our attention to IPS deadline busy periods ...
Worst Case Response Time (IV)

Given some arbitrary job $J_{i,j}$ released at $a \geq 0$, contained in the PS deadline-$d$ busy period $[t_1, t_2)$,

- we can choose $t_1$ as our new time origin 0
- $J_{i,j}$ is released at time $a' = a - t_1$, and has deadline $d' = d - t_1$
- the resulting IPS deadline-$d'$ busy period $[0, t_2 - t_1)$ ends at time $L_i(a') = t_2 - t_1$
- $a' + C_i \leq L_i(a') \leq L$, since
  - $a' + C_i$ is trivial lower bound for $L_i(a')$
  - the synchronous busy period must include all possible IPS deadline-$d$ busy periods
Worst Case Response Time (V)

Obviously,

- $L_i(a')$ is also the length of the original PS deadline-$d$ busy period $[t_1, t_2)$

- The worst case response time of $J_{i,j}$ released at time $a$ satisfies $rt_i(a) \leq t_2 - a = L_i(a') - a'$

- Hence, $rt_i(a)$ is the same as $rt_i(a')$ of a job $J_{i,j}'$ released at time $a'$

It hence suffices to study all IPS deadline-$d$ busy periods, where a specific job $J_{i,j}$ of $\tau_i$

- arrives at some arbitrary $0 \leq a \leq L_i(a) - C_i$

- has deadline $d = a + D_i$

- Note that the first job of $\tau_i$ has offset $s_i(a) = a - \lfloor \frac{a}{T_i} \rfloor T_i$
Lemma 116. Given a schedule where some job of $\tau_i$ is released at time $a$ with deadline $d = a + D_i$, and all other tasks are released at 0. The worst case deadline-$d$ relevant cumulated workload $W_i(a, t)$ arriving in $[0, t)$ is

$$W_i(a, t) = \sum_{k \neq i, D_k \leq a + D_i} \min \left\{ \left\lceil \frac{t}{T_k} \right\rceil, 1 + \left\lfloor \frac{a + D_i - D_k}{T_k} \right\rfloor \right\} C_k + \Delta_i(a, t)C_i$$

where

$$\Delta_i(a, t) = \begin{cases} \min \left\{ \left\lceil \frac{t - s_i(a)}{T_i} \right\rceil, 1 + \left\lfloor \frac{a - s_i(a)}{T_i} \right\rfloor \right\} & \text{if } t \geq s_i(a), a \geq s_i(a), \\ 0 & \text{otherwise.} \end{cases}$$
\(L_i(a)\) Computation (II)

**Proof.** For any \(\tau_k \neq \tau_i\), the deadline-\(d\)-relevant workload created by \(\tau_k\) within \([0, t]\) is the minimum of

- the max. number \(\left\lceil \frac{t}{T_k} \right\rceil\) of \(\tau_k\) job releases in \([0, t]\)
- the max. number \(1 + \left\lfloor \frac{d-D_k}{T_k} \right\rfloor\) of \(\tau_k\) jobs with deadline \(\leq d\) in \([0, d]\)

The deadline-\(d\)-relevant workload created by \(\tau_i\) within \([0, t]\) is 0 if \(t\) is before or at the first job release, or else determined by the minimum of

- the max. number \(\left\lceil \frac{t-s_i(a)}{T_i} \right\rceil\) of job releases in \([s_i(a), t]\)
- the max. number \(1 + \left\lfloor \frac{a-s_i(a)+D_i-D}{T_i} \right\rfloor = 1 + \left\lfloor \frac{a-s_i(a)}{T_i} \right\rfloor\) of jobs with deadline \(\leq d\) in \([s_i(a), d]\) (and hence in \([0, d]\])
- Note that only jobs released before and at time \(a\) are relevant for \(L_i(a)\), since later jobs of \(\tau_i\) have a deadline past \(d = a + D_i\)
Theorem 118. Given the schedule where some job of $\tau_i$ is released at time $a$ with deadline $d = a + D_i$, the worst case length of the IPS deadline-$d$ busy period $L_i(a)$ can be computed iteratively as

$$L_i^{(0)}(a) = \sum_{j \neq i, D_j \leq a + D_i} C_j + \delta_{s_i(a)=0} C_i$$

$$L_i^{(m+1)}(a) = W_i(a, L_i^{(m)}(a))$$

where

$$\delta_{s_i(a)=0} = \begin{cases} 1 & \text{if } s_i(a) = 0, \\ 0 & \text{otherwise}. \end{cases}$$
Proof. The construction is exactly the same as for ordinary busy periods, except that only deadline-\(d\) relevant cumulative workload must be considered.

This is ensured by the minimum functions in \(W_i(a, t)\), which guarantee that

- only the workload that arrived before \(t\) is considered
- only the workload that is relevant for a deadline-\(d\) busy period is considered

Hence, the iterative construction indeed builds up the IPS deadline-\(d\) busy period.
Worst Case Response Time Analysis (I)

\( W_i(a, t) \) has the same properties as the cumulative workload \( W(t) \) for ordinary busy periods. Hence,

- the fixed point iteration used for determining the length \( L \) of the synchronous busy period can be used

\( L_i(a) \) is the unique limit \( \lim_{m \to \infty} L_i^{(m)}(a) \)

\( L_i(a) \) is also meaningful for irrelevant \( a \) (where \( a + C_i > L_i(a) \)), in which case \( L_i(a) \) is the length of the whole IPS busy period

If \( U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \), it again follows

- \( L_i(a) = L_i^{(m)}(a) = L_i^{(m+1)}(a) \) for some finite \( m \)

- \( L_i(a) \) is the smallest positive solution of \( t = W_i(a, t) \)
Worst Case Response Time Analysis (II)

To find out whether a task set is feasible, determine:

- Worst case response time of \( J_{i,j} \) released at time \( a \), which is \( rt_i(a) = \max\{C_i, L_i(a) - a\} \) [\( C_i \) is obvious lower bound]

- Note that this definitions also covers irrelevant \( a \) (where \( L_i(a) < a + C_i \))

- The worst case response time of \( \tau_i \) is \( rt_i = \max_{a \geq 0} rt_i(a) \)

- Checking all \( a \geq 0 \) not practical, but
  - \( L_i(a) \leq L \) for all \( \tau_i \)
  - Checking \( 0 \leq a \leq L - C_i \) is hence sufficient

If \( \forall i : rt_i \leq D_i \), the task set is feasible.
Worst Case Response Time Analysis (III)

The range for $a$ where $rt_i(a) \leq D_i$ must be checked for can be further reduced by exploiting some additional facts:

- $W_i(a, t)$ and hence $L_i(a)$ has a local maximum if $a$ is such that $a + D_i - D_k = mT_k$ for some $m$
- $D_i \leq D_j \Rightarrow \max_a L_i(a) \leq \max_a L_j(a)$
- $L_i(a)$ is non-decreasing in $a$ for all $\tau_i$

The detailed algorithm can be found in the textbook.
Variants of Hybrid Models
Non-Idling Non-preemptive Hybrid EDF (I)

Feasibility analysis for non-idling non-preemptive EDF for hybrid task sets is slightly more complicated:

- Urgent tasks released during execution of less urgent one is delayed (priority inversion)
- May even lead to infeasible schedule, as already shown by earlier example
- Fortunately, the maximum delay of any urgent job can be bounded
Non-Idling Non-preemptive Hybrid EDF (I)

Feasibility analysis for non-idling non-preemptive EDF for hybrid task sets is slightly more complicated:

- Urgent tasks released during execution of less urgent one is delayed (priority inversion)
- May even lead to infeasible schedule, as already shown by earlier example
- Fortunately, the maximum delay of any urgent job can be bounded

How can this be done?
Non-Idling Non-preemptive Hybrid EDF (II)

For some $t$, let $t' < t$ be the release time of the job that initiates the last deadline-$t$ busy period $[t_1, t_2)$:

- $t'$ is the release time of a job with deadline $\leq t$
- A job $J_{low}$ with deadline $> t$ could be executing at $t'$ that cannot be preempted $\Rightarrow$ priority inversion during $[t', t_1]$. There is no idle time during $[t', t_2]$.

After $J_{low}$ has completed, only jobs $J_{high}$ with deadline $\leq t$ released at or after $t'$ [if any] are executed

$\Rightarrow$ Any $J_{high}$ can experience at most one priority inversion

$\Rightarrow$ The maximum penalty of $J_{high}$ due to non-preemption is $\max_{t'+D_j > t} \{C_j - 1\}$, with penalty 0 if no such $D_j$ exists.
Non-Idling Non-preemptive Hybrid EDF (III)

**Theorem 126** (George, Rivierre and Spuri 1996). Any hybrid task set with processor utilization $U \leq 1$ is feasible under non-preemptive EDF if and only if

$$\forall t \in S : h(t) + \max_{D_j > t} \{C_j - 1\} \leq t,$$

where $t_{\text{max}} = \max\{L, t_1, t_2\}$ with

$$S = \bigcup_{i=1}^{n} \left\{ kT_i + D_i, k = 0, \ldots, \left[ \frac{t_{\text{max}} - D_i}{T_i} \right] \right\}$$

$$t_1 = \max \left\{ D_{\text{max}}, \frac{\sum_{i=1}^{n} (1 - D_i/T_i)C_i}{1 - U} \right\}$$

$$t_2 = \frac{\sum_{D_i \leq T_i} (1 - D_i/T_i)C_i + \max_{i=1,\ldots,n} \{C_i - 1\}}{1 - U}$$
tasks with release jitter (I)

Up to now,
- jobs are ready for execution from their release time on
- relative deadlines are measured from release times on

In reality, however, it makes sense to distinguish two instants for the $j$-th job $J_{i,j}$ of task $\tau_i$:
- **Arrival time** $a_{i,j}$ [starting the relative deadline]
- release time $a_{i,j} \leq r_{i,j} \leq a_{i,j} + J_i$, where $J_i \geq 0$ is the release jitter of task $\tau_i$

A non-zero release jitter
- models scheduling/dispatching overhead
- needs to be accounted for in feasibility analysis
Tasks with Release Jitter (II)

If $J_i > 0$, then two task instances of $\tau_i$ may occur closer to each other than $T_i$, namely, $T_i - J_i$. The worst case arrival pattern of a hybrid task set is:

- All jobs experience their shortest interarrival time at the (synchronous) beginning of the schedule
- First job released at $t = 0$, consecutive ones at $\max\{kT_i - J_i, 0\}$ for $k > 0$
Tasks with Release Jitter (II)

If \( J_i > 0 \), then two task instances of \( \tau_i \) may occur closer to each other than \( T_i \), namely, \( T_i - J_i \). The worst case arrival pattern of a hybrid task set is:

- All jobs experience their shortest interarrival time at the (synchronous) beginning of the schedule
- First job released at \( t = 0 \), consecutive ones at \( \max\{kT_i - J_i, 0\} \) for \( k > 0 \)

This is equivalent to the following arrival pattern:

- The first job \( J_{i,1} \) of \( \tau_i \) arrives at \( t = -J_i \), all subsequent jobs \( J_{i,k}, k \geq 2 \), arrive equally spaced by \( T_i \)
- All jobs arriving at \( t < 0 \) are simultaneously released at time 0, all others are released when they arrive
Tasks with Release Jitter (III)

The processor demand $h(t)$ hence becomes

$$h(t) = \sum_{i:D_i \leq t + J_i} \left( 1 + \left\lfloor \frac{t + J_i - D_i}{T_i} \right\rfloor \right) C_i$$

The maximum time $t_{\text{max}}$ where the demand must be checked is

$$t_{\text{max}} = \max \left\{ \max_{i=1,\ldots,n} \{ D_i - J_i \}, \frac{U}{1 - U} \max_{1 \leq i \leq n} \{ T_i + J_i - D_i \} \right\}$$

 Appropriately modified variants of the Zheng & Shin and George et.al. upper bounds also exist.
Tasks with Release Jitter (IV)

By the same token, it is possible to adapt the busy period analysis to non-zero release jitter: The cumulative workload reads

\[ W(t) = \sum_{i=1}^{n} \left\lfloor \frac{t + J_i}{T_i} \right\rfloor C_i \]

everything else remains unchanged.

Consequently,

- most results for \( \forall i : J_i = 0 \) can be adapted to non-zero release jitter
- handling this important model extension is not too difficult.
Sporadically Periodic Tasks (I)

In real systems, processing requirements for a task $\tau_i$ are often bursty:

- Started by some initial job, arriving sporadically with sporadicity interval $T_i$ (outer period)
- Followed by at most $m_i - 1 \geq 0$ additional jobs each, arriving sporadically with sporadicity interval $t_i$ (inner period)
- Individual deadline for every job, but mandatory requirement: $m_i t_i \leq T_i$

Examples:
- Data reception via serial line
- Sensor polling triggered by some event, e.g., an alarm
Sporadically Periodic Tasks (II)

Worst case arrival pattern for loading factor analysis:
- Packing the maximum number of job releases at the beginning of the schedule
- Incorporating release jitter as before, with same jitter bound $J_i$ applying to every (outer, inner) job of $\tau_i$

Adaption of busy period and processor demand analysis:
Need to consider
- number $k_i$ of complete outer periods of $\tau_i$ fitting into the time interval $[0, t)$ in question
- number of completely fitting inner periods from the last (incompletely fitted) outer period
Sporadically Periodic Tasks (III)

The cumulative workload then becomes

\[ W(t) = \sum_{i=1}^{n} I_i(t)C_i \] with

\[ I_i(t) = \left\lceil \frac{t + J_i}{T_i} \right\rceil m_i + \min \left\{ m_i, \left\lceil \frac{t + J_i - \left\lfloor \frac{t+J_i}{T_i} \right\rfloor T_i}{t_i} \right\rceil \right\} \]

The processor demand becomes

\[ h(t) = \sum_{D_i \leq t+J_i} H_i(t)C_i \] with

\[ H_i(t) = \left\lceil \frac{t + J_i - D_i}{T_i} \right\rceil m_i + \min \left\{ m_i, 1 + \left\lceil \frac{t + J_i - D_i - \left\lfloor \frac{t+J_i-D_i}{T_i} \right\rfloor T_i}{t_i} \right\rceil \right\} \]

The feasibility test conditions etc. remain unchanged.
Scheduling Overhead (I)

In real systems,

- the processor must also execute scheduler and dispatcher \( \Rightarrow \) is not 100% available for task execution

- Example: Periodic tick interrupt activating the scheduler, which moves tasks from an arrival queue to the (dynamic) priority-ordered processing queue and dispatches into the first ranked one.

Consider processors with availability functions satisfying
\[ a(t) + a(T) \leq a(t + T) \] for all \( T, t \). Then, it can be shown that

- the Liu & Layland Synchronous Busy Period Theorem is still valid

- incorporating scheduling overhead in feasibility analysis is possible
Scheduling Overhead (II)

Scheduling overhead model by Tindell et.al.:
- $C_{\text{tick}}$ is the WCET of the tick interrupt service routine
- $C_{QL}$ is the WCET to move the first job
- $C_{QS}$ is the WCET to move subsequent jobs

The scheduling overhead over $[0, w]$ is then $OV(w) =$

$T(w)C_{\text{tick}} + \min\{T(w), K(w)\}C_{QL} + \max\{K(w) - T(w), 0\}C_{QS}$

with
- $T(w) = \left\lceil \frac{w}{T_{\text{tick}}} \right\rceil$ bounds the number of tick interrupts in $[0, w)$
- $K(w) = \sum_{i=1}^{n} \left\lfloor \frac{w+J_i}{T_i} \right\rfloor$ is the number of task moves
The cumulative workload then becomes

\[ W(t) = OV(t) + \sum_{i=1}^{n} \left\lfloor \frac{t + J_i}{T_i} \right\rfloor C_i, \]

and if the availability function \( a(t) \) is such that

\[ a(t) \geq \max\{t - OV(t), 0\}, \]

then a sufficient feasibility test condition is

\[ t - OV(t) \geq \sum_{i: D_i \leq t + J_i} \left( 1 + \left\lfloor \frac{t + J_i - D_i}{T_i} \right\rfloor \right) C_i \]

Further generalizations by Jeffay & Stone.
EDF under Overloads (Ch. 5.1)
Overloads (I)

All previous “positive” results regarding EDF somehow assumed that

- a feasible schedule of the task set in question exists
- although maybe only God knows this schedule

Examples: EDF is guaranteed to find a feasible schedule if

- Maximum loading factor $u \leq 1$
- Processor utilization $U \leq 1 (D_i \geq T_i)$
Overloads (I)

All previous “positive” results regarding EDF somehow assumed that

- a feasible schedule of the task set in question exists
- although maybe only God knows this schedule

Examples: EDF is guaranteed to find a feasible schedule if

- Maximum loading factor \( u \leq 1 \)
- Processor utilization \( U \leq 1 \) \( (D_i \geq T_i) \)

In reality, however, overloads and hence infeasible task sets can and do occur.
Overloads (II)

Overloads are impossible to circumvent because of limited control over the environment, leading to:

- unanticipated changes of the arrival load over time
- excessive job execution times
- failures
Overloads (II)

Overloads are impossible to circumvent because of limited control over the environment, leading to

- unanticipated changes of the arrival load over time
- excessive job execution times
- failures

When modeling and analyzing fault-tolerant distributed real-time systems in such advanced settings,

- the environment is considered as the computer system’s adversary
- the computer system plays against the adversary
- a correct system design implements a winning strategy
How does EDF Perform under Overloads?

In the presence of overload,
- it is impossible for any scheduling algorithm to meet all deadlines
- some [less important] jobs must be aborted

EDF is optimal in the absence of overloads. However, under overloads,
- EDF aborts jobs in an “uncontrolled” way
- arrival of a new job may cause all preempted ones to miss their deadlines (domino effect)

EDF performs poorly under overload!
Importance vs. Urgency

Apparently, one has to distinguish two properties of a task:
- Urgency
- Importance

In pure EDF [as well as in rate/deadline monotonic priority assignments],
- urgency and importance are reflected by a single (dynamic) priority value
  \[ \Rightarrow \text{sub-optimal behavior under overloads.} \]

Let’s assign each task \( T_i \) an additional importance value \( V_i \).
Utility Functions (I)

However, for the successful operation of a system,

- the importance of task $\tau_i$ may depend upon the relative finishing times $F_{i,j} = f_{i,j} - r_{i,j}$ of its jobs
- a static importance value is often not sufficient

Generalize importance value $V_i$ by utility function $V_i(F_{i,j})$:

- hard
- firm
- soft
- non-real-time
Utility Functions (II)

Scheduling with utility functions:

- Scheduling algorithm $\mathcal{A}$ selects a schedule $S$, denoted $S = \mathcal{A} \vdash J_{i,j}$

- Scheduling goal: Maximize cumulative utility
  \[ \Gamma = \sum_{\mathcal{A} \vdash J_{i,j}} V_i(F_{i,j}). \]
Utility Functions (II)

Scheduling with utility functions:

- Scheduling algorithm $A$ selects a schedule $S$, denoted $S = A \vdash J_{i,j}$
- Scheduling goal: Maximize cumulative utility $\Gamma = \sum_{A \vdash J_{i,j}} V_i(F_{i,j})$.

What can be achieved here?
On-line vs. Clairvoyant Scheduling

An on-line scheduling algorithm $\mathcal{A}$

- does not have any information (execution time, deadline etc.) about a job before it is released
- is easy to implement
On-line vs. Clairvoyant Scheduling

An on-line scheduling algorithm $\mathcal{A}$
- does not have any information (execution time, deadline etc.) about a job before it is released
- is easy to implement

A clairvoyant scheduler $\mathcal{C}$
- has all information about all future job releases a priori
- is a theoretical algorithm that cannot be implemented
On-line vs. Clairvoyant Scheduling

An on-line scheduling algorithm $\mathcal{A}$

- does not have any information (execution time, deadline etc.) about a job before it is released
- is easy to implement

A clairvoyant scheduler $\mathcal{C}$

- has all information about all future job releases a priori
- is a theoretical algorithm that cannot be implemented

Interesting question:

- How does cumulative utility $\Gamma_A$ relate to $\Gamma_C$?

$\Rightarrow$ Competitive analysis
Competitive Analysis Basics (I)

Given a set of real-time tasks $\mathcal{T}$ with firm deadlines, let
- $v_i$ be the importance value density of a task/job $\tau_i$
  - uniform value density: $\forall i : v_i = 1$
  - non-uniform value density: $\mathcal{T}$’s importance ratio
    \[ k = \frac{\max_{\tau_i \in \mathcal{T}} \{v_i\}}{\min_{\tau_i \in \mathcal{T}} \{v_i\}} > 1 \]
- $V_i = V_i(F_{i,j}) = v_iC_i$ be the importance value of $\tau_i$

Given any sequence $\mathcal{J} = \{J_{i,j}\}$ of job releases of $\mathcal{T}$, let
- $\Gamma^*(\mathcal{J}) = \sum_{\mathcal{C}^* \vdash J_{i,j}} V_i$ be the maximum cumulative value obtained by the optimal clairvoyant algorithm $\mathcal{C}^*$
- $\Gamma_A(\mathcal{J}) = \sum_{\mathcal{A} \vdash J_{i,j}} V_i$ be the maximum cumulative value obtained by on-line algorithm $\mathcal{A}$
Question 1: Does an optimal on-line algorithm exist?

- Algorithm $A^*$ that maximizes $\Gamma_{A^*}(J)$ for every sequence $J$ of job releases of $T$

- Simple example reveals that this requires knowledge of future arrivals

$\Rightarrow$ No optimal on-line algorithm exists
Competitive Analysis Basics (II)

Question 1: Does an optimal on-line algorithm exist?

- Algorithm $A^*$ that maximizes $\Gamma_{A^*}(J)$ for every sequence $J$ of job releases of $T$
- Simple example reveals that this requires knowledge of future arrivals

⇒ No optimal on-line algorithm exists

Question 2: How well performs some given algorithm $A$?

- Determine competitive factor $\varphi_A$:

$$\varphi_A = \inf_J \frac{\Gamma_A(J)}{\Gamma^*(J)}$$
Competitive Analysis Basics (III)

Properties $\varphi_A$:

For every sequence $\mathcal{J}$ of job releases, it must hold that

$$\Gamma_A(\mathcal{J}) \geq \varphi_A \Gamma^*(\mathcal{J})$$
Properties $\varphi_A$:

- For every sequence $\mathcal{J}$ of job releases, it must hold that $\Gamma_A(\mathcal{J}) \geq \varphi_A \Gamma^*(\mathcal{J})$

- Sometimes: Restrict sequence of job releases for computing $\varphi_A$, to guarantee things like:
  - sporadicity of task releases
  - limited maximum or average load
Properties $\varphi_A$:

- For every sequence $J$ of job releases, it must hold that $\Gamma_A(J) \geq \varphi_A \Gamma^*(J)$

- Sometimes: Restrict sequence of job releases for computing $\varphi_A$, to guarantee things like:
  - sporadicity of task releases
  - limited maximum or average load

Further Questions:

- What is the best achievable $\varphi_A$?
- What is the algorithm $A$ achieving it?
Competitive Analysis Basics (IV)

How to determine $\varphi_A$?

- Two-player game between
  - on-line algorithm $A$
  - adversary (constructing $J$ [+ running clairvoyant algorithm $C^*$])

- Adversary tries to make ratio between $A$’s and $C^*$’s gain as small as possible.
Competitive Analysis Basics (IV)

How to determine $\varphi_\mathcal{A}$?

- Two-player game between
  - on-line algorithm $\mathcal{A}$
  - adversary (constructing $\mathcal{J}$ [+ running clairvoyant algorithm $\mathcal{C}^*$])

- Adversary tries to make ratio between $\mathcal{A}$’s and $\mathcal{C}^*$’s gain as small as possible.

To show that $\varphi_\mathcal{A} \leq r$, a counter-example suffices:

- Construct some sequence $\mathcal{J}$ of job releases of $\mathcal{T}$
- Determine $\Gamma^*(\mathcal{J})$ provided by $\mathcal{C}^*$
- Show that $\Gamma_\mathcal{A}(\mathcal{J}) \leq r \Gamma^*(\mathcal{J})$
Simple Example: Competitive Factor EDF

Consider $\mathcal{J}$ consisting of two zero-slack time jobs $J_1$ and $J_2$:

- $V_1 = C_1 = K$ and $V_2 = C_2 = \varepsilon K$
- $J_1$, $J_2$ simultaneously released but $d_1 > d_2$
- Both preemptive and non-preemptive EDF provide cumulative value $\Gamma_{EDF}(\mathcal{J}) = \varepsilon K$
- Clairvoyant scheduler $C^*$ provides $\Gamma_{C^*}(\mathcal{J}) = K$

$\Rightarrow \varphi_{EDF} \leq \varepsilon$
Simple Example: Competitive Factor EDF

Consider $\mathcal{J}$ consisting of two zero-slack time jobs $J_1$ and $J_2$:

- $V_1 = C_1 = K$ and $V_2 = C_2 = \varepsilon K$
- $J_1$, $J_2$ simultaneously released but $d_1 > d_2$
- Both preemptive and non-preemptive EDF provide cumulative value $\Gamma_{EDF}(\mathcal{J}) = \varepsilon K$
- Clairvoyant scheduler $C^*$ provides $\Gamma_{C^*}(\mathcal{J}) = K$

$\Rightarrow \varphi_{EDF} \leq \varepsilon$

Hence, EDF has in fact zero competitive factor!
Simple Example: Competitive Factor EDF

Consider $\mathcal{J}$ consisting of two zero-slack time jobs $J_1$ and $J_2$:

- $V_1 = C_1 = K$ and $V_2 = C_2 = \varepsilon K$
- $J_1$, $J_2$ simultaneously released but $d_1 > d_2$
- Both preemptive and non-preemptive EDF provide cumulative value $\Gamma_{EDF}(\mathcal{J}) = \varepsilon K$
- Clairvoyant scheduler $C^*$ provides $\Gamma_{C^*}(\mathcal{J}) = K$

$\Rightarrow \quad \varphi_{EDF} \leq \varepsilon$

Hence, EDF has in fact zero competitive factor!

What is the competitive factor of other algorithms?
Lower-Bounds on Competitive Factors (I)

Arbitrary loading factor (in particular, $u > 2$):

- Uniform value density: No on-line algorithm achieves $\varphi_A > \frac{1}{4}$
- Value density ratio $k > 1$: No on-line algorithm achieves $\varphi_A > \frac{1}{(1+\sqrt{k})^2}$

We will now exercise the detailed proof of the first result, based upon the following assumptions:

- Infinitely fine-grained preemptability
- Finite durations of periods with overload
- Adversary and player execute at same speed
Theorem 151 (Baruah et. al.). *For uniprocessor systems under finite durations of overload, no online algorithm achieves a competitive factor \( \varphi > 1/4 \).*
Theorem 151 (Baruah et. al.). *For uniprocessor systems under finite durations of overload, no online algorithm achieves a competitive factor $\varphi > 1/4$.*

*Proof.* Quite complex — done on blackboard . . .

[You may want to refresh your basic knowledge of difference equations, generating functions, partial fraction expansions and complex numbers.]

□
An Algorithm with $\varphi_A = \frac{1}{4}$?

The competitive factor lower-bounds can be shown to be tight, by giving an online algorithm:

- TD1, which has $\varphi_A = \frac{1}{4}$ (uniform value density)
- $D^{over}$, which has $\varphi_A = \frac{1}{(1 + \sqrt{k})^2}$ (value density ratio $k > 1$)

We will study a simple version of algorithm TD1, which

- works only for task sets with zero laxity
- [but can be extended to work also with task sets with non-zero laxity]
- exploits the adversary argument used in the lower bound proof
Algorithm TD1 (I)

Intuition behind algorithm TD1:

- We consider zero-laxity task sets only, hence
  - if, in a sequence $J = J_1, J_2, \ldots, J_m$ of jobs, any successive pair of jobs overlaps (i.e. $r_j \leq r_{j+1} < d_j$), at most one job $J_i \in J$ can be feasibly scheduled
  - choose a task $J_i$ that has sufficiently high value w.r.t. the accumulated total value of $J$
- More specifically, inspired by the lower-bound proof,
  - abort the current job in favor of a newly arrived one if its value is $< 1/4$ of the duration $\Delta$ of the currently “expected” busy period
  - the clairvoyant scheduler can accumulate a value of at most $\Delta$
Algorithm TD1 (II)

Hence, given a sequence of overlapping jobs $J_1, J_2 \ldots$, starting a busy period:

- TD1 eventually chooses a **single** job $J_i$ that completes $J_1, \ldots, J_{i-1}$ are discarded right away or aborted before completion
- $J_{i+1}, \ldots, J_m$, with $J_m$ being the last job released before $J_i$’s termination, are discarded

For this purpose, TD1 maintains two intervals, both starting at the beginning of a busy period:

- $\Delta'$ ends at deadline of $J_{run}$, i.e., is busy period that would be generated when $J_{run}$ was not aborted later.
- $\Delta$ extends $\Delta'$ by maximum deadline of tasks discarded during $J_{run}$; $\Delta = \Delta'$ after an abort.
Algorithm TD1 (III)

Pseudo-Code Algorithm TD1 (Simple Version)

1. \( \Delta' = 0; \Delta = 0; v_{\text{run}} = 0 \)
2. \( J_{\text{run}} := \emptyset \) /* currently no scheduled job */

When \( J_{\text{next}} \) is released, where \( k \geq 0 \) is remaining processing time of \( J_{\text{run}} \):

3. \( \Delta := \max\{\Delta, \Delta' - k + C_{\text{next}}\} \)
4. if \( v_{\text{run}} < \Delta/4 \) then
5. abort \( J_{\text{run}} \)
6. set \( \Delta' := \Delta \) /* Note: \( \Delta' - k + C_{\text{next}} > \Delta \) here */
7. schedule \( J_{\text{run}} := J_{\text{next}} \) and set \( v_{\text{run}} := v_{\text{next}} = C_{\text{next}} \)

Whenever \( J_{\text{run}} \) completes:

8. \( \Delta' = 0; \Delta = 0; v_{\text{run}} = 0 \)
9. \( J_{\text{run}} := \emptyset \) /* currently no scheduled job */
Lemma 156. Consider the sequence $\Delta_k = \Delta'_k$ and $v_k$ maintained by TD1 in $\Delta$ and $v_{run}$ at line 7, in a sequence of aborted overlapping job arrivals $J_1, \ldots, J_i$. For any $1 \leq k \leq i$, it holds that $v_k > \Delta_k / 2$. 
Algorithm TD1 (IV)

**Lemma 156.** Consider the sequence $\Delta_k = \Delta'_k$ and $v_k$ maintained by TD1 in $\Delta$ and $v_{run}$ at line 7, in a sequence of aborted overlapping job arrivals $J_1, \ldots, J_i$. For any $1 \leq k \leq i$, it holds that $v_k > \Delta_k / 2$.

**Proof.** By induction. For $k = 1$, obviously $v_1 = \Delta_1 > \Delta_1 / 2$.

Assuming that $v_k > \Delta_k / 2$, consider arrival of $J_{k+1}$:

- $J_k$ is aborted, hence $v_k < \Delta_{k+1} / 4$ by line 4.
- Must have $\Delta' - k + C_{next} > \Delta$ in line 3, otherwise preceding (discarded) job would already cause abort of $J_{run}$. Hence, $\Delta_{k+1} = \Delta'_{k+1} < \Delta'_k + v_{k+1} = \Delta_k + v_{k+1}$.

Combining this leads to

$$v_{k+1} > \Delta_{k+1} - \Delta_k > \Delta_{k+1} - 2v_k > \Delta_{k+1} - \Delta_{k+1} / 2$$

and hence $v_{k+1} > \Delta_{k+1} / 2$ as claimed.  $$\square$$
Algorithm TD1 (VI)

**Theorem 157** (Baruah et. al.). *For task sets with zero laxity and finite durations of overload, the simple algorithm TD1 achieves a competitive factor $\varphi_{TD1} = \frac{1}{4}$.***
Algorithm TD1 (VI)

Theorem 157 (Baruah et. al.). For task sets with zero laxity and finite durations of overload, the simple algorithm TD1 achieves a competitive factor $\varphi_{TD1} = 1/4$.

Proof. Consider a sequence $J_1, J_2 \ldots, J_{m'}$ (finite, because the period of overload must be finite) of overlapping job arrivals. It suffices to distinguish two cases:

- $i = m'$, so no overlapping job is released after the chosen $J_i = J_m$: By our lemma, $v_{m'} > \Delta_{m'}/2 > \Delta_{m'}/4$.

- $i < m'$, so jobs $J_{i+1}, \ldots, J_m$ are released and discarded after $J_i$ is released [but before $J_i$ terminates, hence $m \leq m'$]: By the threshold rule (line 4) of TD1, the running task's $v_{\text{run}} = v_i$ must satisfy $v_i \geq \Delta_{\ell}/4$ for any $i + 1 \leq \ell \leq m$, hence $v_i \geq \Delta_m/4$.

Since a clearvoyant scheduler can obtain a cumulative value of at most $\Delta_m$ in $J_1, \ldots, J_m$, TD1 indeed provides $\varphi_{TD1} = 1/4$ as asserted. $\square$
Infinite Duration of Overloads?

The competitive factor lower-bounds hold only for finite durations of overload.

Is this restriction necessary?
Infinite Duration of Overloads?

The competitive factor lower-bounds hold only for finite durations of overload

- Is this restriction necessary?
- YES (at least in the uniprocessor case)
Infinite Duration of Overloads?

The competitive factor lower-bounds hold only for finite durations of overload

Is this restriction necessary?

YES (at least in the uniprocessor case)

For infinite durations of overload, no on-line uniprocessor algorithm has non-zero competitive factor:

Adversary can generate a sequence of overlapping jobs of ever increasing values

Player either abandons current task (= no gain) or schedules it (= loses gain)

This goes on forever \(\Rightarrow\) adversary infinitely often “ahead” of player
Extended Competitive Lower-Bounds

**Restricted loading factor** $1 < u \leq 2$:

- **Uniform value density**: No on-line algorithm achieves
  \[ \varphi_A \geq p, \text{ where } p \text{ satisfies } 4 \left(1 - (u - 1)p\right)^3 = 27p^2 \]

- **Value density ratio** $k > 1$:
  - If $q = k(u - 1) \geq 1$, no on-line algorithm achieves
    \[ \varphi_A \geq \frac{1}{(1 + \sqrt{q})^2} \]
  - If $q = k(u - 1) < 1$, no on-line algorithm achieves
    \[ \varphi_A \geq p, \text{ where } p \text{ satisfies } 4 \left(1 - qp\right)^3 = 27p^2 \]

- Can also be “simulated” by allowing the player to execute faster than the adversary \( \Rightarrow \) (much) better lower bounds [Baruah et.al.]
Summarizing Observations

The considered lower-bounds

- can be extended to multiprocessor scheduling
- are quite conservative (infinite preemptability etc.)

⇒ real algorithms can be expected to work better in most cases

Interesting practical algorithms that can cope with finite duration overloads:

- $D_{over}$ by Koren and Shasha
- RED by Buttazzo and Stankovic
EDF Scheduling with Shared Resources (Ch. 6)
Where do we Stand?

We considered uniprocessor CPU scheduling using EDF for independent (simple) tasks

- No shared resources
- No precedence constraints
Where do we Stand?

We considered uniprocessor CPU scheduling using EDF for independent (simple) tasks
  - No shared resources
  - No precedence constraints

In reality, tasks are not independent:
  - shared data
  - shared communication channels
  - precedence constraints
Shared Resources—An Overview (I)

Accessing shared resources:
- Semaphores resp. Critical sections
- [Transactions]
Shared Resources—An Overview (I)

Accessing shared resources:

- Semaphores resp. **Critical sections**
- [Transactions]

Consider joint problem:

- Scheduling periodic tasks with deadlines
- Tasks access shared resources modeled as critical sections

Goals:

- Meeting all deadlines
- Avoid unnecessary blocking
Shared Resources—An Overview (II)

Requires resource access protocol:

- If job $J$ wants to enter CS and no job is currently in CS, it enters and holds CS while in
- If job $J$ wants to enter CS that is held by some job, it has to wait until it is granted access to CS
- If the job currently holding CS leaves, the CS becomes free and some job waiting for CS is granted access
Shared Resources—An Overview (II)

Requires resource access protocol:

- If job $J$ wants to enter CS and no job is currently in CS, it enters and holds CS while in
- If job $J$ wants to enter CS that is held by some job, it has to wait until it is granted access to CS
- If the job currently holding CS leaves, the CS becomes free and some job waiting for CS is granted access

Interferes with real-time scheduling . . .
Shared Resources—An Overview (III)

Complications:

- Tasks may suffer from blocking:
  - Low priority job $J_L$ executed at time $t$, currently in CS $z$
  - High priority job $J_H$ released at $t$, also wants to enter CS $z$

  $\Rightarrow J_H$ is blocked until $J_L$ leaves $z$

- Preemption within CS not always allowed:
  - CS representing access to communication channel
  - CS representing access to I/O-device

- Feasibility checking for scheduling with shared resources is NP-hard
Priority Inversion

Preemptive priority driven scheduling principle:

- Low priority job $J_L$ preempted upon release of high priority job $J_H$
- Principle “controllably” (while in CS only) violated during blocking
- Principle uncontrollably violated during priority inversion:
  - Low-priority $J_L$ blocks high-priority $J_H$ via some CS
  - Medium-priority $J_M$ preempts $J_L$ (for its entire execution time $C_M$!)
  ⇒ medium-priority $J_M$ indirectly blocks high priority $J_H$
- Easily causes deadline violations
Some days into the mission: Total SW resets

- Press statements:
  - “software glitches”
  - “the computer was trying to do too many things at once”

- Wind River reported (VxWorks was the OS):
  - Preemptive priority scheduling
  - “Information bus”: shared memory to pass information
  - Bus access controlled via a mutex
Part of the on-board task structure:

- **High priority** bus management task \( H \) (access to bus)
- **Low priority** meteo data gather task \( L \) (access to bus)
- **Medium priority** communication task \( M \)
- Watchdog resets system if bus mgmt. task takes too long
Part of the on-board task structure:

- **High priority** bus management task $H$ (access to bus)
- **Low priority** meteo data gather task $L$ (access to bus)
- **Medium priority** communication task $M$

Watchdog resets system if bus mgmt. task takes too long

Pathfinder problem caused by classic **priority inversion**:

- $H$ was released while $L$ held the bus
- $H$ waited for $L$ to terminate
- $L$ was preempted by long running $M$
- Watchdog timed out $\Rightarrow$ reset
Assumptions and Notations (I)

Tasks:

- A set of \( n \geq 2 \) preemptible periodic (and/or sporadic) tasks \( \tau_1, \ldots, \tau_n \), with \( D_i \leq T_i \)

- The sequence of jobs \( J_{i,1}, J_{i,2}, \ldots \) of \( \tau_i \) sequentially enters critical sections (CS) \( z_{i,1}, z_{i,2}, \ldots \)

Constraints on critical sections:

- Every CS is entirely in some job \( z_{i,k} \subseteq J_{i,\ell} \)

- For every \( \tau_i \), CSs \( z_{i,1}, z_{i,2}, \ldots \) may be different, but must be
  - disjoint \( (z_{i,k} \cap z_{i,\ell} = \emptyset) \)
  - properly nested \( (z_{i,k} \subset z_{i,\ell}) \)

- The WCET for CS \( z_{i,k} \) is \( c_{i,k} \)
Assumptions and Notations (II)

CS and resources:

- Every CS $z_{i,k}$ is associated with a single shared resource $R_{i,k}$
- For every resource $R$, let $Z_R = \{z_{i,k} | R_{i,k} = R\}$ denote the set of CS associated with $R$

In case of nested CS $z_{i,k_1} \subset z_{i,k_2} \subset \cdots \subset z_{i,k_m}$,

- (a job in) $z_{i,k_\ell}$ holds all resources $R_{i,k_\ell}, \ldots, R_{i,k_m}$
- resources are
  - acquired in the order $R_{i,k_m}, \ldots, R_{i,k_1}$
  - freed in the order $R_{i,k_1}, \ldots, R_{i,k_m}$
The Classic Priority Inheritance Protocol

Avoid priority inversion as follows:

- Schedule jobs according to static priority scheduling
- If a job $J_L$ blocks a higher-priority job $J_H$ via some shared resource (= both in CS),
  - $J_L$ temporarily inherits the priority of $J_H$
  - $J_L$ reverts to original priority when it exits CS
- Can also be adapted to nested CS
The Classic Priority Inheritance Protocol

Avoid priority inversion as follows:

- Schedule jobs according to static priority scheduling
- If a job $J_L$ blocks a higher-priority job $J_H$ via some shared resource (= both in CS),
  - $J_L$ temporarily inherits the priority of $J_H$
  - $J_L$ reverts to original priority when it exits CS
- Can also be adapted to nested CS

Will look into it in detail ...
Blocking with Static Priority Scheduling

With static priority scheduling, capturing blocking is easy:

- A job $J_{i,k}$ is **blocked** by a job $J_{j,\ell}$ in CS $z_{j,m}$ if
  - $J_{i,k}$ cannot resume execution before $J_{j,\ell}$ has completed $z_{j,m}$, and
  - $J_{j,\ell}$ has **lower priority** than $J_{i,k}$

- We can also say that task $\tau_j$ blocks task $\tau_i$ since all jobs of a task have same priority

- If $R$ is a resource held by $J_{j,\ell}$ in CS $z_{j,m}$, we also say that $J_{i,k}$ (resp. task $\tau_i$) is blocked by the resource $R$
Blocking with Static Priority Scheduling

With static priority scheduling, capturing blocking is easy:

- A job $J_{i,k}$ is blocked by a job $J_{j,\ell}$ in CS $z_{j,m}$ if
  - $J_{i,k}$ cannot resume execution before $J_{j,\ell}$ has completed $z_{j,m}$, and
  - $J_{j,\ell}$ has lower priority than $J_{i,k}$
- We can also say that task $\tau_j$ blocks task $\tau_i$ since all jobs of a task have same priority

- If $R$ is a resource held by $J_{j,\ell}$ in CS $z_{j,m}$, we also say that $J_{i,k}$ (resp. task $\tau_i$) is blocked by the resource $R$

For dynamic priority scheduling, capturing blocking is more difficult . . .
Blocking with EDF

Adapt blocking definition from static priority scheduling:

- A job $J_{i,k}$ is blocked by a job $J_{j,\ell}$ in CS $z_{j,m}$ if
  - $J_{i,k}$ cannot resume execution before $J_{j,\ell}$ has completed $z_{j,m}$, and
  - $J_{j,\ell}$ has later deadline than $J_{i,k}$

Unfortunately, this property is not static (i.e., need not hold for all jobs $J_{j,\ell}$ of task $\tau_j$)
Blocking with EDF

Adapt blocking definition from static priority scheduling:

- A job $J_{i,k}$ is **blocked** by a job $J_{j,\ell}$ in CS $z_{j,m}$ if
  - $J_{i,k}$ cannot resume execution before $J_{j,\ell}$ has completed $z_{j,m}$, and
  - $J_{j,\ell}$ has later deadline than $J_{i,k}$

- Unfortunately, this property is not static (i.e., need not hold for all jobs $J_{j,\ell}$ of task $\tau_j$)

We need a **static** measure $\pi_i$ associated with task $\tau_i$ to express worst case blocking times in terms of $c_{i,m}$ only . . .
Preemption Levels (I)

With EDF, $J_{j,\ell}$ could block $J_{i,k}$ on a shared resource only iff

- $J_{j,\ell}$ has later deadline: $d_{j,\ell} > d_{i,k}$
- otherwise, $J_{i,k}$ would just be regularly preempted by $J_{j,\ell}$
- $J_{j,\ell}$ is released earlier: $r_{j,\ell} < r_{i,k}$
- otherwise, $J_{i,k}$ would never give $J_{j,\ell}$ a chance to execute
Preemption Levels (II)

To capture blocking in EDF as in static priority scheduling, assign preemption level $\pi_i$ to task $\tau_i$:

- Job $J_{i,k}$ of $\tau_i$ could only be blocked by $J_{j,\ell}$ of $\tau_j$ on a shared resource iff $\pi_j < \pi_i$
- The choice $\pi_i > \pi_j$ iff $D_i < D_j$ does the job

Preemption levels will allow us

- to analyze potential blocking, in a not overly conservative way
- express worst case blocking times in terms of $c_{i,m}$ only
The Priority Inheritance Protocol (I)

Deadline (= dynamic priority) of a job $J_{i,j}$ of task $\tau_i$:

- **nominal** deadline $d_{i,j}$
- **active** deadline $d_i(t)$
  - initialized to $d_{i,j}$ upon job release
  - possibly modified when entering/leaving a CS according to rules below

Jobs are scheduled EDF, based on their active deadlines.
The Priority Inheritance Protocol (II)

PIP update rules (I):

- When \( J_{i,k} \) tries to enter \( z_{i,m} \) but \( R_{i,m} \) is already held by a higher-deadline job \( J_{j,\ell} \), then \( J_{i,k} \) is blocked by \( J_{j,\ell} \) on \( R_{i,m} \) (as well as by all other jobs also blocked by \( J_{j,\ell} \) on \( R_{i,m} \)).

- Otherwise, \( J_{i,k} \) enters \( z_{i,m} \) and thus holds \( R_{i,m} \).

- If \( J_{i,k} \) is blocked by some \( J_{j,\ell} \), then \( J_{j,\ell} \) inherits \( J_{i,k} \)’s deadline:
  - \( J_{j,\ell} \)’s active deadline \( d_j(t) \) is set to \( d_i(t) \)
  - whenever \( d_i(t) \) changes (by blocking some even smaller-deadline job), \( d_j(t) := \min\{d_j(t), d_i(t)\} \)
The Priority Inheritance Protocol (III)

PIP update rules (II):

- If $J_{j,\ell}$ exits a CS, it resumes
  - the active deadline it had when entering the CS (for outermost CS)
  - the active deadline of the lowest active-deadline job queued for enclosing CS (for nested CS)

- The jobs blocked by a resource $R$ are queued according to increasing active deadlines

- When freed, resource $R$ is acquired by the smallest-deadline job queued for $R$
Observations Priority Inheritance Protocol

From the PIP rules, we immediately observe:

- Priority inheritance is transitive: If $J_L$ blocks $J_M$, and $J_M$ blocks $J_H$, then $J_L$ inherits the active deadline of $J_H$ via $J_M$.

- The job holding resource $R$ always executes with an active deadline equal to the smallest of all jobs queued for $R$ (or any of its enclosing $R'$).
From the PIP rules, we immediately observe:

- Priority inheritance is transitive: If $J_L$ blocks $J_M$, and $J_M$ blocks $J_H$, then $J_L$ inherits the active deadline of $J_H$ via $J_M$.

- The job holding resource $R$ always executes with an active deadline equal to the smallest of all jobs queued for $R$ (or any of its enclosing $R'$)

But there are subtle details . . .
Priority Inheritance Blocking (I)

PIP leads to different types of blocking:

- **Direct blocking** occurs when a job tries to acquire a resource already held by some other job.
- **Transitive blocking** occurs in case of nested CS:
  - large-deadline job $J_L$ acquires $R_1$ in CS
  - medium-deadline job $J_M$ has nested CS, acquiring $R_2$, $R_1$
  - small-deadline job $J_H$ acquires $R_2$ in CS
  - $\Rightarrow J_H$ is transitively blocked by $J_L$
Priority Inheritance Blocking (I)

PIP leads to different types of blocking:

- **Direct blocking** occurs when a job tries to acquire a resource already held by some other job.

- **Transitive blocking** occurs in case of nested CS:
  - large-deadline job \( J_L \) acquires \( R_1 \) in CS
  - medium-deadline job \( J_M \) has nested CS, acquiring \( R_2, R_1 \)
  - small-deadline job \( J_H \) acquires \( R_2 \) in CS
  - \( \Rightarrow J_H \) is transitively blocked by \( J_L \)

There is even a worse form of blocking . . .
Priority Inheritance Blocking (II)

Push-through blocking occurs when

- large-deadline $J_L$ blocks small-deadline $J_H$ (directly or transitively) via some resources
- $J_L$ also blocks every medium-deadline $J_M$ via its inherited active deadline from $J_H$

Note:
- $J_M$ need not access any shared resource for being blocked!
- Preemption levels are not meaningful here: $J_M$ can have $\pi_M > \pi_H$ ("anomalous" p.t.b.), yet cannot preempt $J_H$ since it has later deadline!
  \[ \Rightarrow \] Push-through blocking can hit any $\tau_M$ with $\pi_M > \pi_L$
Properties of PIP (I)

We start our analysis with some technical lemmas . . .

Lemma 182. After a job $J_{i,k}$ has been released, only jobs with active deadline less than or equal to $d_{i,k}$ are executed until $J_{i,k}$ completes.

Proof. The lemma follows directly from the PIP rules:

- If the job $J_{i,k}$ is never blocked, the lemma is an obvious consequence of EDF scheduling.
- Otherwise, any task that blocks $J_{i,k}$ inherits its deadline or an even smaller one.
Properties of PIP (II)

Lemma 183. A job $J_H$ can be blocked by a larger-deadline job $J_L$ only if, at the time $J_H$ is released, $J_L$ is already within a CS that can (later) block $J_H$.

Proof. Suppose $J_L$ is not within a CS that can either (a) directly block $J_H$ or (b) can lead to an active deadline of $J_L$ that is less than or equal to the one of $J_H$ (transitive or push-through blocking).

- By Lemma 182, only tasks with active deadline shorter than $J_H$ will be executed after its release.
- Since $J_L$’s active deadline when $J_H$ is released is larger than $J_H$, the lemma follows.
Properties of PIP (III)

Lemma 184. A job $J_{i,k}$ of the task $\tau_i$ can only be blocked by a higher-deadline job $J_{j,\ell}$ of task $\tau_j$ if $\tau_i$ has a greater preemption level than $\tau_j$, i.e., $\pi_i > \pi_j$.

Proof. By Lemma 183, $J_{j,\ell}$ must already be in a potentially blocking CS when $J_{i,k}$ is released. Hence,

- $J_{j,\ell}$ is released before $J_{i,k}$
- $J_{j,\ell}$ has a larger deadline than $J_{i,k}$

$\Rightarrow \tau_j$ has a greater relative deadline than $\tau_i$, hence $\pi_i > \pi_j$. □
Blocking Critical Sections

Definition 185 (a). Let $\beta_{i,j}$ denote the set of all CS of jobs of $\tau_j$ that can block jobs of the task $\tau_i$ (via any kind of blocking):

$$\beta_{i,j} = \{ z_{j,m} : \pi_j < \pi_i \text{ and } z_{j,m} \text{ can block } J_{i,k} \}$$

Since we have properly nested CS, we are primarily interested in maximal elements of $\beta_{i,j}$:

Definition 185 (b). Maximal elements in $\beta_{i,j}$:

$$\beta_{i,j}^* = \{ z_{j,m} : (z_{j,m} \in \beta_{i,j}) \text{ and } (\not\exists z_{j,m'} \in \beta_{i,j} \text{ such that } z_{j,m} \subset z_{j,m'}) \}$$

Note carefully that a job $J_{j,\ell}$ exiting $z_{j,m} \in \beta_{i,j}^*$ always assumes its nominal deadline in PIP afterwards.
Lemma 186. For any $j \neq i$, a job $J_{i,k}$ can be blocked by larger-deadline jobs $J_{j,\ell}$ of task $\tau_j$ for at most the duration of one critical section of $\beta_{i,j}^*$.

Proof. By Lemma 183 and 184, $J_{j,\ell}$ must be in some potentially blocking CS $z_{j,m}$ when $J_{i,k}$ is released:

1. $z_{j,m}$ must be in $\beta_{i,j}^*$
2. When $J_{j,\ell}$ exits $z_{j,m}$, it returns to its nominal deadline, which is larger than the deadline of $J_{i,k}$
3. By Lemma 182, $J_{i,k}$ cannot be blocked by $J_{j,\ell}$ (or any later job of $\tau_j$) again.

□
Theorem 187.  Under EDF scheduling with PIP, each job of a task $\tau_i$ can be blocked by at most the duration of one critical section in each of $\beta^*_i,j$, $1 \leq j \leq n$ and $\pi_i > \pi_j$.

Proof.  The theorem follows from:

- Lemma 186 can be applied for all tasks $\tau_j$ that can block $\tau_i$
- Lemma 184 implies that these are exactly the tasks $\tau_j$ with $\pi_j < \pi_i$.

This theorem only considered blocking by other tasks.  Still open:

- Consider blocking via specific resources
- Computing the worst case blocking times
Definition 188 (a). Let \( \zeta_{i,j,k}^* \) denote the set of all longest critical sections of \( \tau_j \) associated with resource \( R_k \), which can block \( \tau_i \)'s jobs (in any way).

\[
\zeta_{i,j,k}^* = \{ z_{j,h} : z_{j,h} \in \beta_{i,j}^* \text{ and } R_{j,h} = R_k \}
\]

Definition 188 (b). Let \( \zeta_{i,,k}^* \) denote the set of all longest critical sections associated with resource \( R_k \) that can block jobs of \( \tau_i \).

\[
\zeta_{i,,k}^* = \bigcup_{\pi_j < \pi_i} \zeta_{i,j,k}^*
\]
Longest Critical Sections (II)

**Theorem 189.** For each resource $R_k$, a job of $\tau_i$ can be blocked by at most one critical section in $\zeta_{i^*}^{i^*},k$.

**Proof.** By Lemma 183 and Lemma 184, a lower priority job $J_{j,\ell}$ must be within some CS $z_{j,m}$ associated with $R_k$ when $J_{i,k}$ is released, and $\pi_j < \pi_i$.

- When $J_{j,\ell}$ frees $R_k$, either
  - $R_k$ is acquired by $J_{i,k}$
  - $R_k$ is acquired by a smaller or equal active-deadline job
- Anyway, subsequently, $J_{i,k}$ cannot be blocked by higher-deadline jobs still queued for $R_k$ anymore.

\[\square\]
Corollary 190. Under EDF scheduling with PIP, if there are $m$ resources than can block the jobs of $\tau_i$, each of these jobs can experience a maximum blocking time $B_i =$

$$
\min \left( \sum_{\pi_j < \pi_i} \max \{ c_{j,\ell} : z_{j,\ell} \in \beta_{i,j}^* \} , \sum_{k=1}^{m} \max \{ c_{j,\ell} : z_{j,\ell} \in \zeta_{i,k}^* \} \right)
$$

Meaning:
- Left part: tasks
- Right part: resources
Maximum Blocking Times

**Corollary 190.** Under EDF scheduling with PIP, if there are $m$ resources than can block the jobs of $\tau_i$, each of these jobs can experience a maximum blocking time $B_i =$

$$
\min \left( \sum_{\pi_j < \pi_i} \max \left\{ c_{j,\ell} : z_{j,\ell} \in \beta_{i,j}^* \right\}, \sum_{k=1}^{m} \max \left\{ c_{j,\ell} : z_{j,\ell} \in \zeta_{i,..,k}^* \right\} \right)
$$

Meaning:
- Left part: tasks
- Right part: resources

Computation of this expression is somewhat tricky . . .
Computation of Blocking Times

Computing $\beta_{i,j} = \beta_{i,j}^D \cup \beta_{i,j}^T \cup \beta_{i,j}^P$ (required for $\beta_{i,j}^*$) actually involves all CS $z_{j,\ell}$ of $\tau_j$ that can block a job of $\tau_i$ via

- direct blocking ($\beta_{i,j}^D$) - simple
- transitive blocking ($\beta_{i,j}^T$) - complex
- push-through blocking ($\beta_{i,j}^P$) - even more complex

Let $\beta_{i,j}^{DT} = \beta_{i,j}^D \cup \beta_{i,j}^T$ and consider the latter two . . .
Transitive Blocking

Recall example: 3 tasks $\tau_j, \tau_h, \tau_i$ with $\pi_j < \pi_h < \pi_i$:

- Large-deadline task $\tau_j$ acquires $R_1$ in CS
- Medium-deadline task $\tau_h$ has nested CS, acq. $R_2$, $R_1$
- Small-deadline task $\tau_i$ acquires $R_2$ in CS

$\Rightarrow$ Job $J_i$ can be transitively blocked by job $J_j$

Incorporate this in $\beta^T_{i,j}$, for all $\tau_h$ with $\pi_j < \pi_h < \pi_i$, via

$$\theta_{i,h,j} = \{z_{j,\ell} \in \beta^D_{h,j} : (R_{j,\ell} = R_{h,p}) \land (z_{h,p} \subset z_{h,q}) \land (z_{h,q} \in \beta^{DT}_{i,h})\}$$

Note that any CS in $\theta_{i,h,j} \subseteq \beta^T_{i,j}$ will also cause push-through blocking in $\beta^P_{h,j}$ (next slide).
Push-Through Blocking Times

Recall example: 3 tasks $\tau_j, \tau_h, \tau_i$ with $\pi_j < \pi_h, \pi_i$:

- Large-deadline task $\tau_j$ blocks small-deadline $\tau_i$ (directly or transitively) via some resources
- $\tau_j$ then also blocks every medium-deadline $\tau_h$ via its inherited active deadline from $J_i$
- $\tau_h$ need not share any resource with $\tau_j$ or $\tau_i$!

Incorporate this in $\beta^P_{h,j}$, for every $\pi_h > \pi_j$, by including all CS $\in \beta^{DT}_{i,j}$, for all $\tau_i$ with $\pi_i > \pi_j$ via

$$\beta^P_{h,j} = \bigcup_{i: \pi_i > \pi_j} \beta^{DT}_{i,j} =: \psi_j$$

As obviously $\forall h : \theta_{i,h,j} \subset \psi_j$, we actually have $\beta^T_{i,j} \subset \beta^P_{i,j}$.
In order to finally compute $\beta_{i,j}$, we introduce:

**Definition 194.** Let $\sigma_i$ be the set of resources accessed by jobs of $\tau_i$.

For each $j$ such that $\pi_j < \pi_i$, we get

$$\beta_{i,j} = \{ z_{j,k} : R_{j,k} \in \sigma_i \} \cup \psi_j$$

where

- $\beta_{i,j}^D = \{ z_{j,k} : R_{j,k} \in \sigma_i \}$ covers direct blocking
- $\psi_j$ covers both push-through and transient blocking

Textbook: Algorithm for computing $B_i$ via $\beta_{i,j}$ in $O(cn^3)$ time, with $c$ being the max. number of CS per task.
Feasibility Check (I)

Assume now that

- $B_i$ is known for each $\tau_i$
- Tasks are ordered by decreasing preemption level (increasing relative deadlines), i.e., $\pi_i \geq \pi_j$ for $i \leq j$.

**Theorem 195.** Given a set $\mathcal{T}$ of $n$ periodic/sporadic tasks, any job set generated by $\mathcal{T}$ is feasibly scheduled with EDF scheduling and PIP, if

$$
\sum_{j=1}^{i-1} \frac{C_j}{D_j} + \frac{C_i + B_i}{D_i} \leq 1 \quad \text{for } 1 \leq i \leq n.
$$
Feasibility Check (II)

Proof. By induction on $i$.

Basis $i = 1$: Since

$$\frac{C_1 + B_1}{D_1} \leq 1$$

and every job of $\tau_1$ can be blocked at most by $B_1$ time, Theorem 84 for $n = 1$ [note that $\min\{T_i, D_i\} = D_i$ due to our basic assumption $D_i \leq T_i$] proves that $\tau_1$ is feasibly schedulable.

Induction step: $i - 1 \rightarrow i$ for $i - 1 \geq 1$: Assume that

- tasks $\tau_1, \ldots, \tau_{i-1}$ are feasibly schedulable
- the $i$-st condition also holds:

$$\sum_{j=1}^{i-1} \frac{C_j}{D_j} + \frac{C_i + B_i}{D_i} \leq 1$$
Feasibility Check (III)

Proof. (cont.)

We have to show that our theorem holds for the first $i$ tasks.

1. Choose any $J_{j,\ell}$ out of the first $i$ tasks (with completion time $f_{j,\ell}$).

2. Let $t \leq r_{j,\ell}$ to be the beginning of the deadline-$d_{j,\ell}$ busy period containing the release of $J_{j,\ell}$.

3. In $[t, f_{j,\ell}]$, only the following can be executed:
   - Jobs $\mathcal{J}(t, f_{j,\ell})$ of $\tau_1, \ldots, \tau_i$ released at time $\geq t$ and with deadline $\leq d_{j,\ell}$
   - CS of jobs of $\tau_h$, $h > i$, that can cause blocking of $\tau_1, \ldots, \tau_i$.

Let $B(t, f_{j,\ell})$ be cumulative blocking time.

Then,

$$f_{j,\ell} \leq t + B(t, f_{j,\ell}) + \sum_{J_{h,\ell} \in \mathcal{J}(t,f_{j,\ell})} C_h$$
**Feasibility Check (IV)**

*Proof. (cont.)*

Distinguish 2 cases:

- If no jobs of \( \tau_i \) are in \( \mathcal{J}(t, f_{j, \ell}) \), then the resulting schedule is identical to the one for \( \tau_1, \ldots, \tau_{i-1} \), which is feasible by the induction hypothesis. Hence, \( f_{j, \ell} \leq d_{j, \ell} \) as required.

- Otherwise, if at least one job of \( \tau_i \) is in \( \mathcal{J}(t, f_{j, \ell}) \):
  
  - All blocking times of jobs of \( \tau_i \) in \( \mathcal{J}(t, f_{j, \ell}) \) are accounted for in \( B_i \).
  
  - All blocking times of the first \( i - 1 \) tasks are also push-through blocking for \( \tau_i \), and thus also included in \( B_i \).

Hence, all blocking times are accounted for in \( B_i \). \qed
Feasibility Check (V)

Proof. (cont.)

Now construct modified schedule:

- Change blocking:
  - First $i - 1$ tasks scheduled without blocking
  - “Attribute” their blocking to additional blocking of some jobs of $\tau_i$

Since the $i$-th condition holds, the modified schedule is feasible by Theorem 84

- Original schedule feasible, since
  - Modification does not change $f_{j,\ell}$
  - The loading factor in $[t, f_{j,\ell}]$ did not change (i.e., remains $\leq 1$)
The Priority Ceiling Protocol for EDF

PIP’s major problem:
- High priority job may be blocked at every CS
  ⇒ Chained blocking

The Priority Ceiling Protocol (PCP) for EDF:
- ∀\( R \), let \( c(R) = c(R, t) \) be the ceiling of \( R \) at time \( t \)

\[
c(R) = \min_i \{ d_{i,k} | J_{i,k} \text{ will acquire or did not yet release } R \}
\]

- Let \( R_H \) be the resource with the smallest ceiling of all resources acquired at time \( t \) (by jobs \( \not\in \tau_i \))
- \( J_i \) can enter and hold CS only if \( d_i(t) < c(R_H) \), otherwise \( J_i \) blocks outside CS and \( J_H \) holding \( R_H \) inherits \( J_i \)’s deadline.
The Dynamic Priority Ceiling Protocol (II)

How the PCP works:
- Let $J_H$ be a small-deadline job, with deadline $d_H$
- At most one lower priority job $J_L$ may enter CS $R_k$ and block $J_H$ later on, since $c(R_H) = c(R_k) = d_H$
- $R_k$’s ceil. cannot become larger before $J_L$ leaves CS
- Subsequently, only jobs with deadline less than $d_H$ may enter any CS

Properties Priority Ceiling Protocol (PCP)
- Any job is blocked at most once, when it enters its first critical section
- No deadlock regardless of locking order
EDF Scheduling with Precedence Constraints (Ch. 7)
Precedence Constraints

Motivation:
- Results of jobs may be inputs to other jobs
- If $J_i$’s result is input to $J_j$ this induces a partial order $J_i \preceq J_j$
- $J_i$ and $J_j$ must be scheduled in this order
- We assume $J_i \preceq J_j \Rightarrow r_i \leq r_j$

Upcoming results:
- Simple techniques to handle precedence constraints
- [With additional shared resources]
EDF and Precedence Constraints

EDF can easily deal with precedence constraints

- Idea: Modify active deadlines to also reflect precendences
- Ensure that a job that causally depends on $J_i$ has a larger deadline

We already introduced the terms nominal and active deadlines:

- **nominal** deadline $d_{i,j}$ is deadline of $J_{i,j}$
- **active** deadline $d_i^*(t)$ is updated during execution
EDF* (I)

The algorithm:

- For each job $J_i$, compute active deadline $d_i^*$ as

$$d_i^* = \min \left\{ d_i, \min_h \{ d_h - C_h : J_i \prec J_h \} \right\}$$

- Schedule jobs using EDF, where ties are broken according to precedence: If $d_i^* = d_j^*$, then
  - schedule $J_i$ before $J_j$ if $J_i \prec J_j$
  - arbitrary otherwise
**EDF* (I)**

The algorithm:

- For each job $J_i$, compute active deadline $d_i^*$ as

$$d_i^* = \min \left\{ d_i, \min_h \{ d_h - C_h : J_i \prec J_h \} \right\}$$

- Schedule jobs using EDF, where ties are broken according to precedence: If $d_i^* = d_j^*$, then
  - schedule $J_i$ before $J_j$ if $J_i \prec J_j$
  - arbitrary otherwise

**Lemma 205.** It holds that $J_i \prec J_j \Rightarrow d_i^* \leq d_j^*$. 
**EDF***(II)***

*Proof.* Let \( \min_i = \min_h \{ d_h - C_h : J_i \prec J_h \} \). If \( J_i \prec J_j \), we have

\[
\begin{align*}
& \min_i \leq \min_j \\
& \min_i \leq d_j - C_j < d_j
\end{align*}
\]

Hence,

\[
\begin{align*}
& d_i^* \leq \min_i < d_j \\
& d_i^* \leq \min_i \leq \min_j \\
\Rightarrow & \quad d_i^* \leq d_j^* \text{ as asserted}
\end{align*}
\]

Note that \( d_i^* = d_j^* \) if both \( J_i \prec J_h \) and \( J_j \prec J_h \) for some low-deadline \( J_h \) (error in textbook)
EDF* (III)

**Theorem 207.** EDF* is optimal, i.e., EDF* finds a feasible schedule if there exists one.

**Proof.** Assume that there is a feasible and precedence-compliant non-EDF* schedule. We transform it into a feasible EDF* schedule:

- Assume $J_j$ is scheduled (in part) before $J_i$, yet $d_i^* < d_j^*$
- from $d_i^* < d_j^*$ it follows $J_j \not≺ J_i$
- from feasibility it follows $J_i \not≺ J_j$
- Hence: There cannot be a precedence relation between $J_i$ and $J_j$
EDF* (IV)

Proof. (cont.) No precedence relation between $J_i$ and $J_j$:

- Let $t_i$ resp. $t_j$ be the first time $J_i$ resp. $J_j$ is executed
- Swap maximum-length execution segment in $[\max\{t_j, r_i\}, f_j)$ and $[t_i, f_i)$
  - completion time of $J_i$ shortened
  - completion time of $J_j$ is $f'_j = f_i \leq d_i^* < d_j^* \leq d_j$
- Produces feasible schedule

A finite number of such inversions eventually yields EDF*-compliant schedule. □