

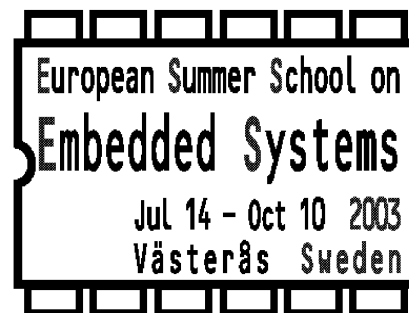
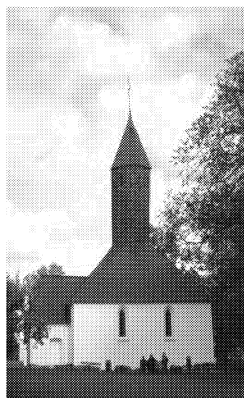
MÄLARDALENS HÖGSKOLA

ESSES 2003

European Summer School on Embedded Systems

Lecture Notes Part XX

Real-Time Systems: Real-Time Scheduling



Editors: Ylva Boivie, Hans Hansson, Jane Kim, Sang Lyul Min

Västerås, Oct 6-8, 2003

ISSN 1404-3041

ISRN MDH-MRTC-113/2003-1-SE

MRTC

MÄLARDALEN REAL-TIME
RESEARCH CENTRE

www.mrtc.mdh.se

Predictably Flexible Real-Time Systems –

*from power plants to home entertainment
applications*

Gerhard Fohler 2003

Mälardalen University, Sweden

gerhard.fohler@mdh.se

Roadmap

- event triggered vs. time triggered
implications of activation paradigms
- novel application requirements
- predictable flexibility
 - combined approach
- applications

Roadmap

- event triggered vs. time triggered
implications of activation paradigms
- novel application requirements
- predictable flexibility
 - combined approach
- applications

Activation Paradigms

- activation of activities - tasks
 - when are events recognized?
 - who initiated activities?
 - when are decisions taken?
- event triggered – ET
 - event initiates activities in system immediately
- time triggered – TT
 - activities initiated at predefined points in time

Properties – Time Triggered

- offline scheduling
- scheduling table
- slots – time triggered activation of dispatcher
- runtime dispatcher executes decision in table

predictable

- ☺ deterministic – known beforehand which activity running when
- ☺ complex demands, distributed, end-to-end, jitter, ...
- ☺ low runtime overhead - table
- ☹ inflexible – can only handle what is completely known before

deterministic

TT

2003

Properties – Event Triggered

- online scheduling, priority driven
- event activates scheduler which takes over
- rules + test
 - earliest deadline first (dynamic)
 - fixed priority

☺ flexible – not completely known

☺ widely used

☹ only simple constraints

☹ high runtime overhead for some

☹ limited predictability – keeps deadlines
when exactly

flexible

ET

Effects on Design

- activation paradigm is central design decision
- “either – or” decision
 - advantages of one method at expense of those of other
 - demands outside paradigm need to be “squeezed in”
- system wide implications
 - same properties for *all* activities
 - mostly highest level

monolithic approaches - “power plant” approaches

- single system for single application
- single paradigm for single class of demands
- high effort and cost

Roadmap

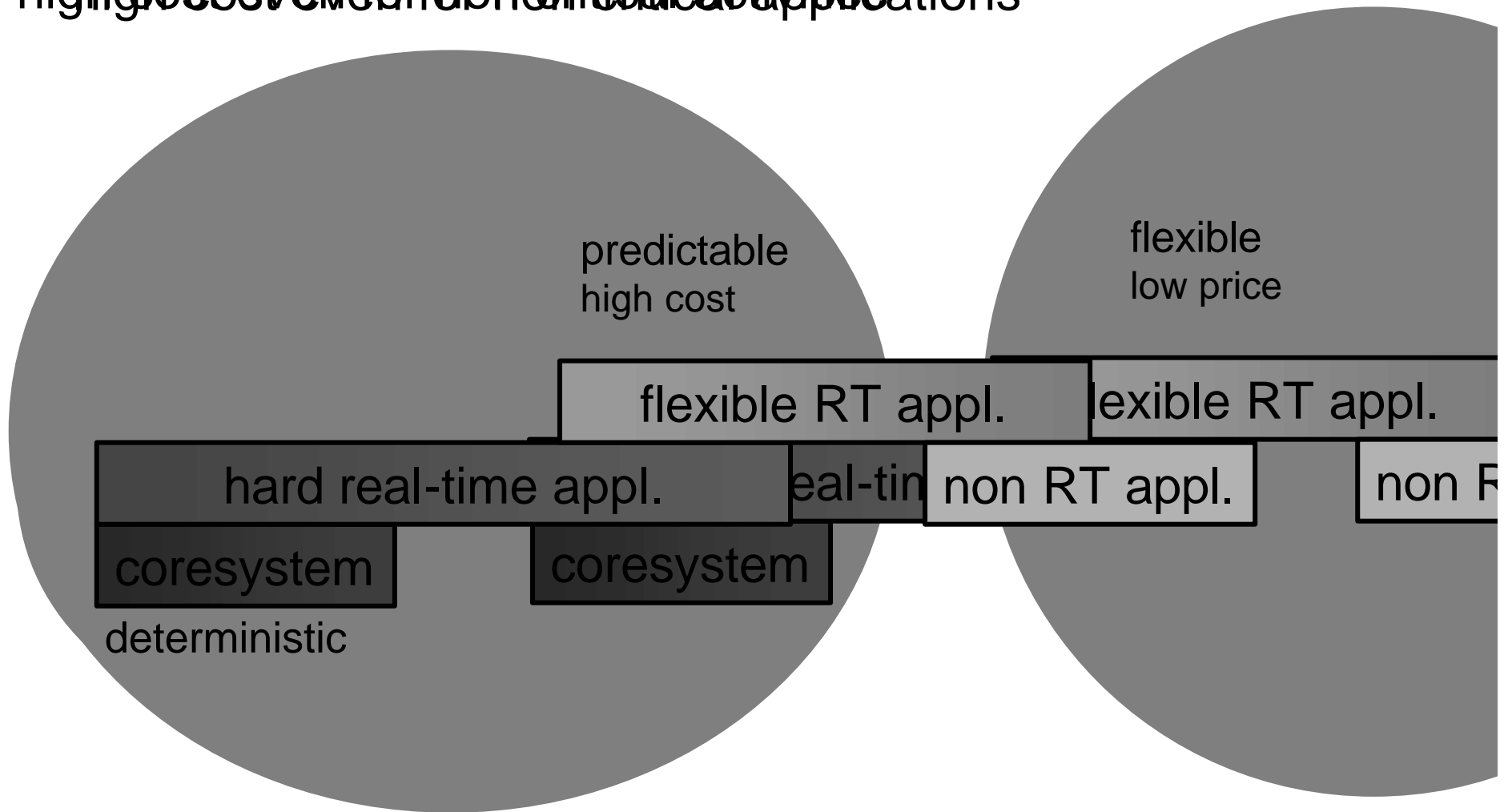
- event triggered vs. time triggered
implications of activation paradigms
- novel application requirements
- predictable flexibility
 - combined approach
- applications

Novel Applications

mix of activities and demands

- core system with high demands
 - strict timing behavior
 - safety critical, fault tolerant
 - proven and tested for worst case
- hard real-time applications
 - temporal correctness, etc.
- flexible real-time applications
 - not completely known
 - some deadlines can be missed
- non real-time activities
 - must not disturb real-time activities

high cost even for non-critical applications



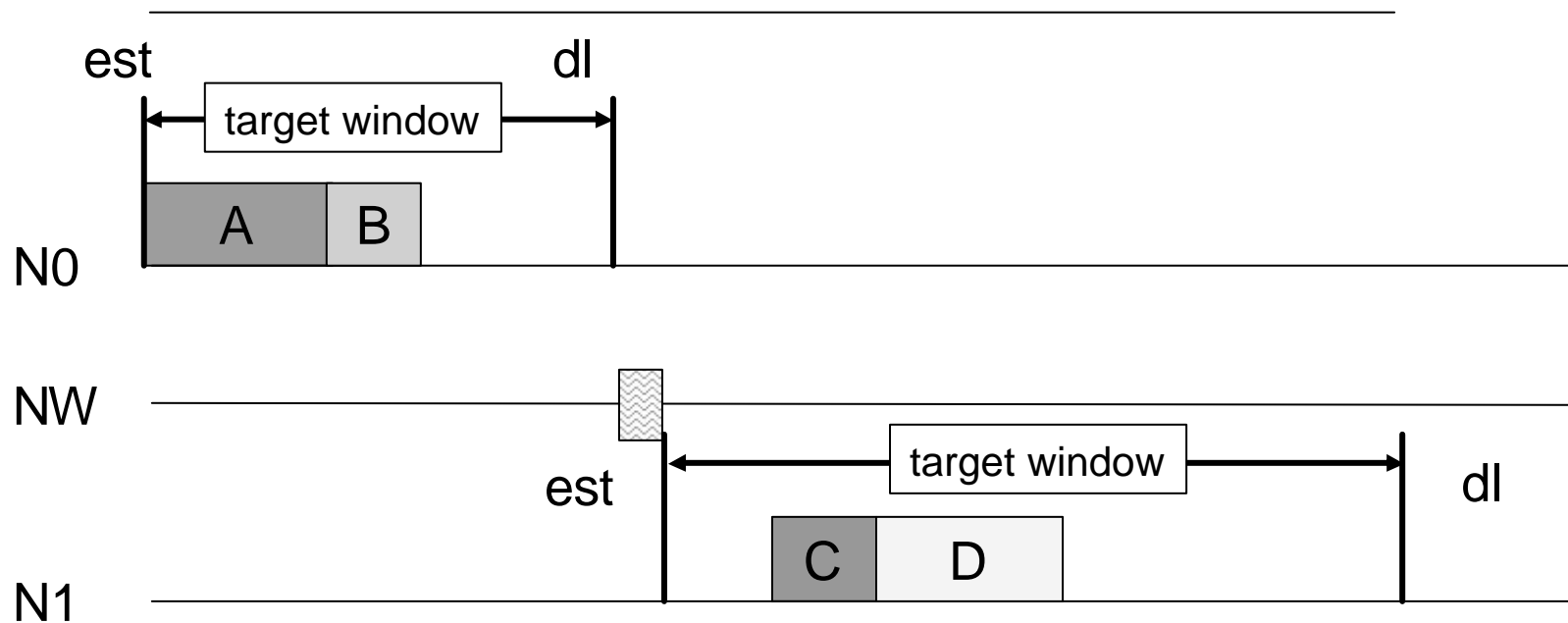
not deterministic behavior of critical activities

Offline Schedules – a Closer Look

- general, complex (temporal) constraints
- offline scheduler
 - resolves demands
 - constructs *single* solution meeting all demands
 - table for least common multiple of periods
- no flexibility

- analysis of offline schedule and demands
- limit task executions - target windows
 - demands fulfilled, if tasks execute within target windows
 - starttime, deadline pairs
- ready for dynamic, event triggered scheduling

$dl(PG)$



offline, TT

original timing constraints

flexibility - complexity

original - NP

offline scheduler

complexity reduction

scheduling table

0 - 0

flexibility analysis

target windows of tasks

ok - ok



online, ET

EDF tasks

standard EDF scheduling

ints., spare cap.

protected offline tasks

Roadmap

- event triggered vs. time triggered
implications of activation paradigms
- novel application requirements
- predictable flexibility
- combined approach
- applications

Predictable Flexibility

target windows control flexibility of task execution

- target window = original task execution
no flexibility, original schedule
- target window after flexibility analysis
flexibility of execution while meeting demands
- reduced target windows
reduced flexibility, e.g., for jitter control
- modifying target windows selects flexibility of tasks individually

Fixed Priority Scheduling

- so far time triggered to event triggered, earliest deadline first (EDF) – dynamic priorities
- how about fixed priority scheduling (FPS)?
- simple, but limited constraints
- transformation method:
 - takes offline schedule
 - determines task attributes, such that when execute with FPS at runtime: offline schedule reenacted
 - target windows, set of priority inequalities, integer linear programming
- now transformations between offline, EDF, and FPS

offline, TT

original temporal constraints

offline scheduler

scheduling table

flexibility analysis

target windows of tasks

online, ET

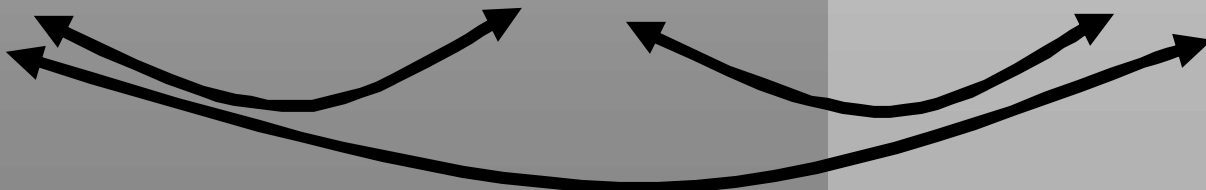
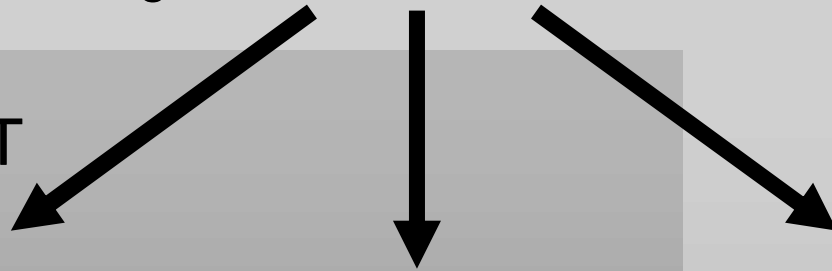
EDF tasks

FPS tasks

EDF scheduling

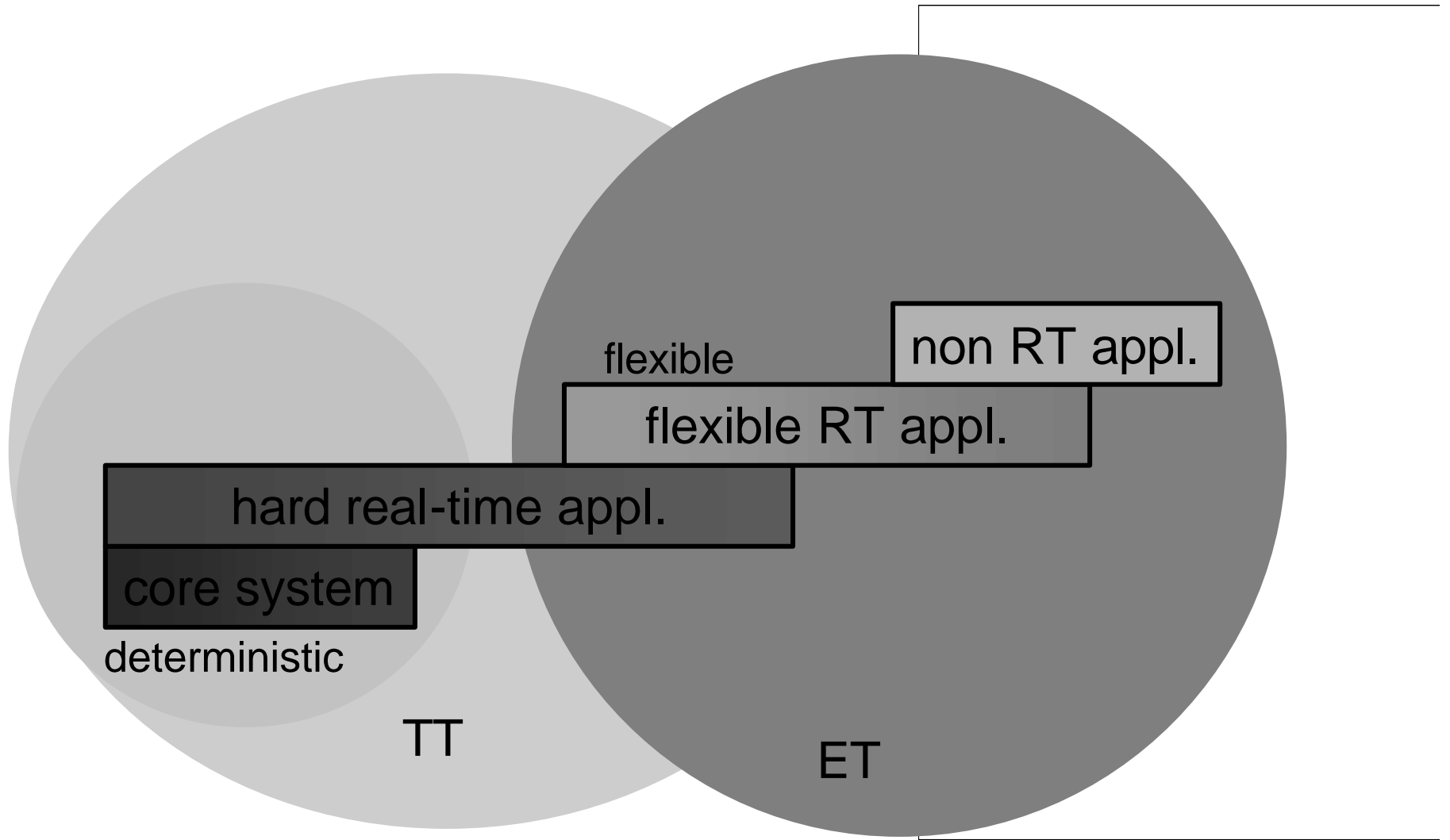
FPS scheduling

offline scheduling



-
- core system
offline scheduling
 - hard real-time applications
offline scheduling or online scheduling
 - flexible real-time applications
combined offline/online approach
 - non real-time activities
together with combined offline/online
 - flexibility individually configured

 - guaranteed tasks protected



Roadmap

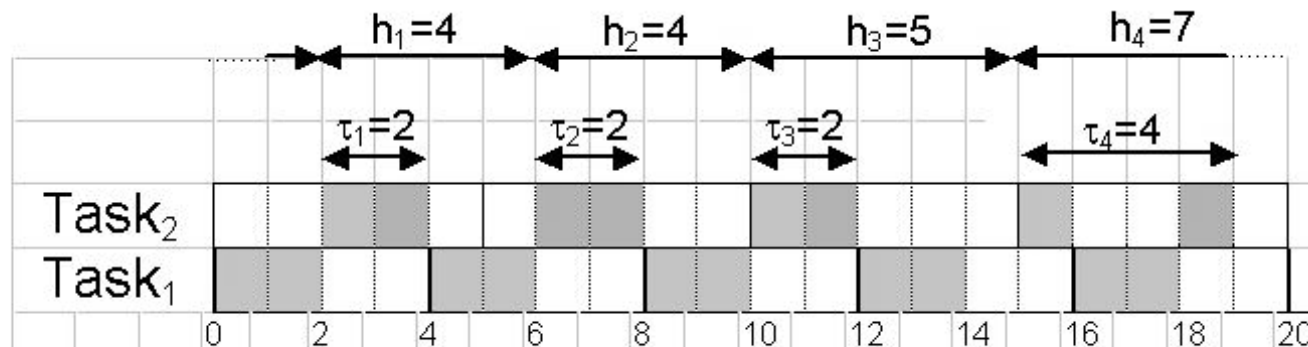
- event triggered vs. time triggered
implications of activation paradigms
- novel application requirements
- predictable flexibility
 - combined approach
- applications

Applications

- two example applications for predictable flexibility
 - real-time control systems
 - home entertainment networks – video streaming

Real-time Control Systems

- mix of task demands
- sampling – actuating tasks: very strict constraints
- deviations results in jitter, error



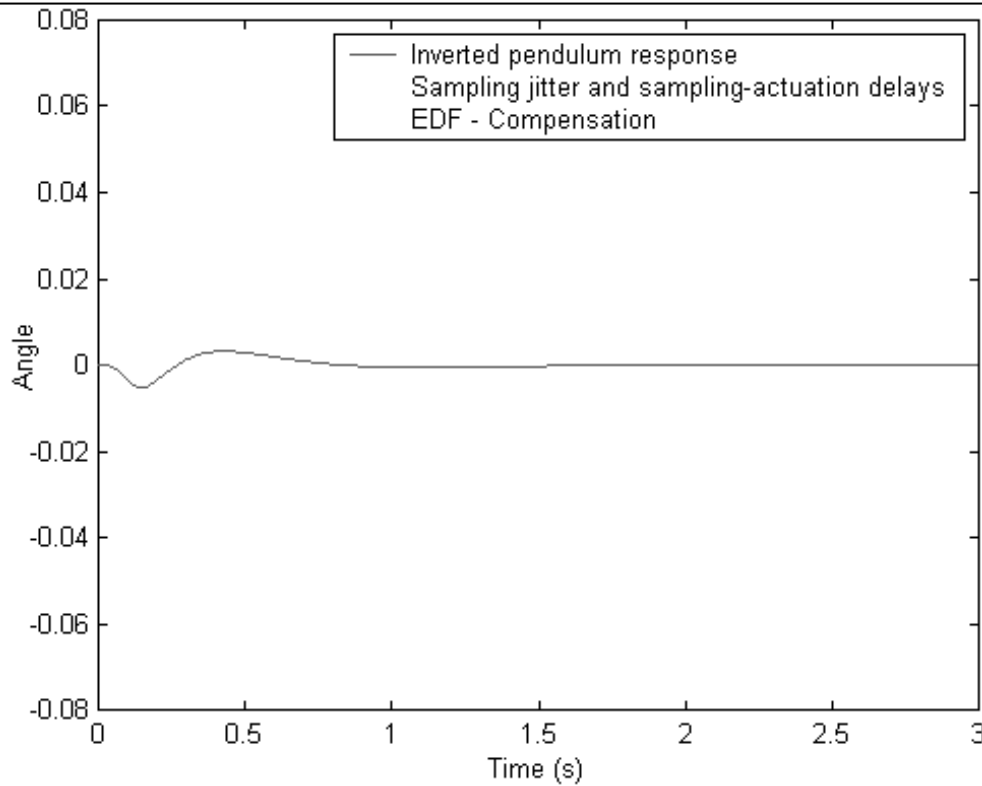
- *sampling jitter*
- *sampling-actuation (delay) jitter*
- other tasks flexible

predictable flexibility:

restrict (=zero) flexibility for sampling actuating tasks

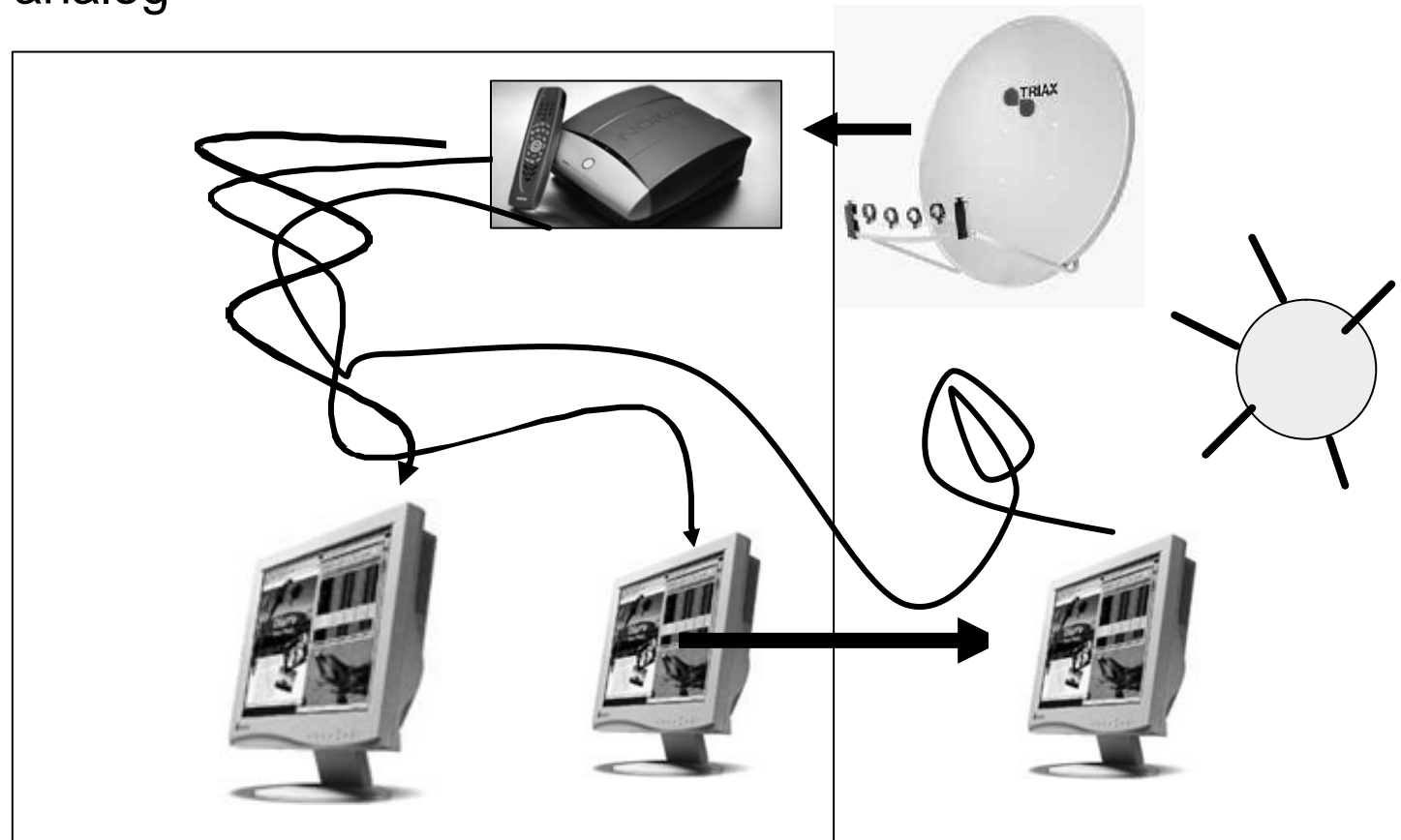
(predict exact times, even non periodic for compensation)

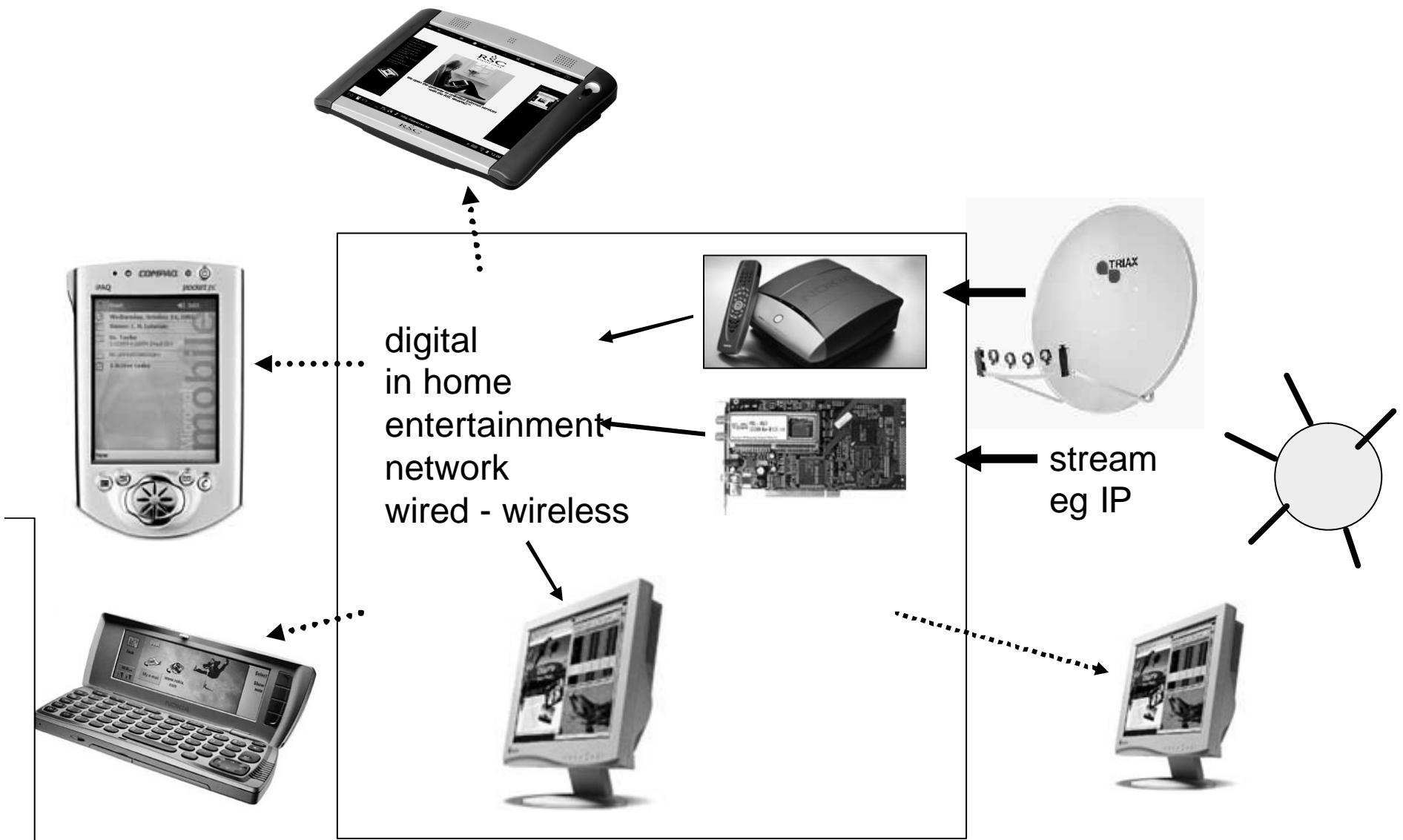
other tasks flexible



Home Entertainment Networks

- current, analog





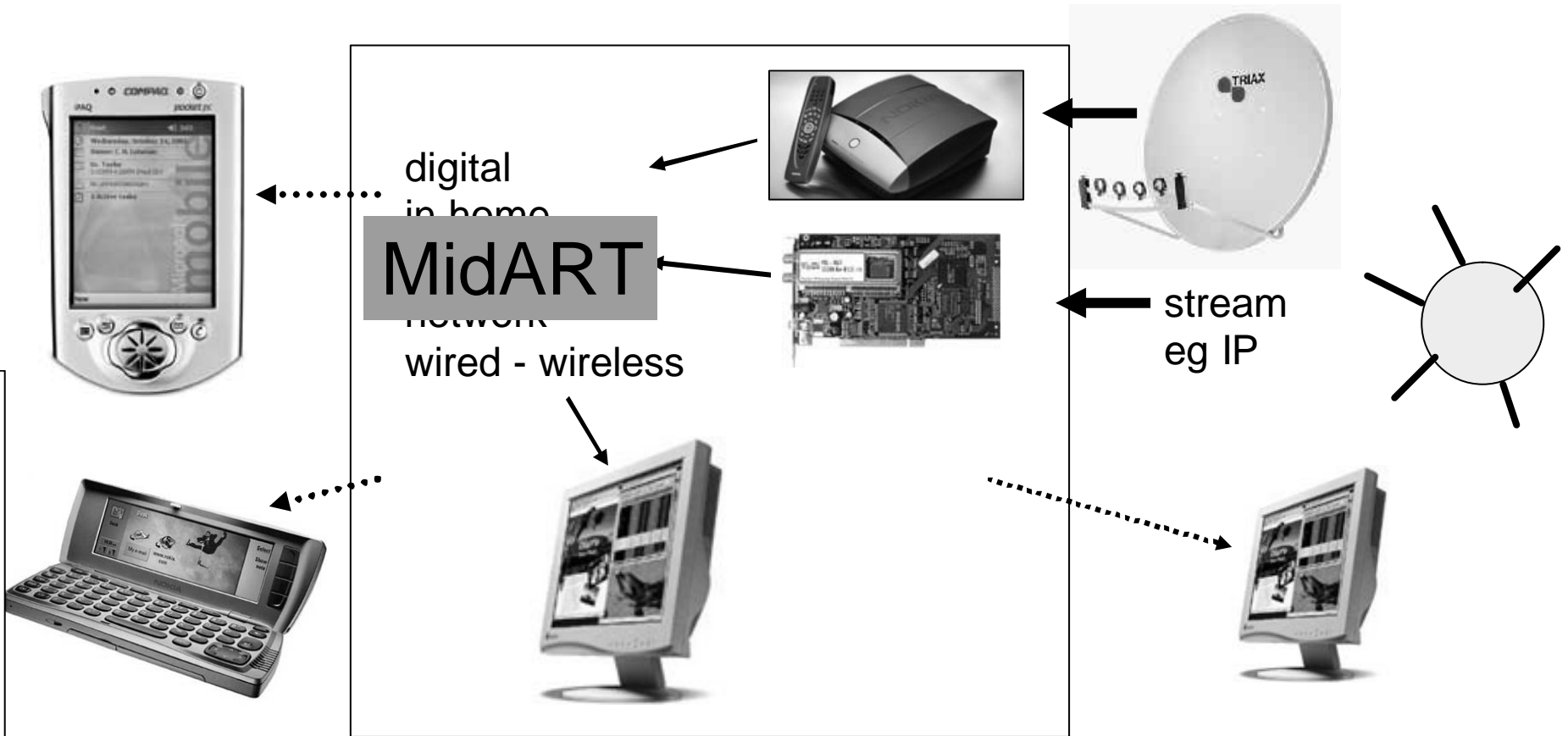
Characteristics

mix of streams and demands

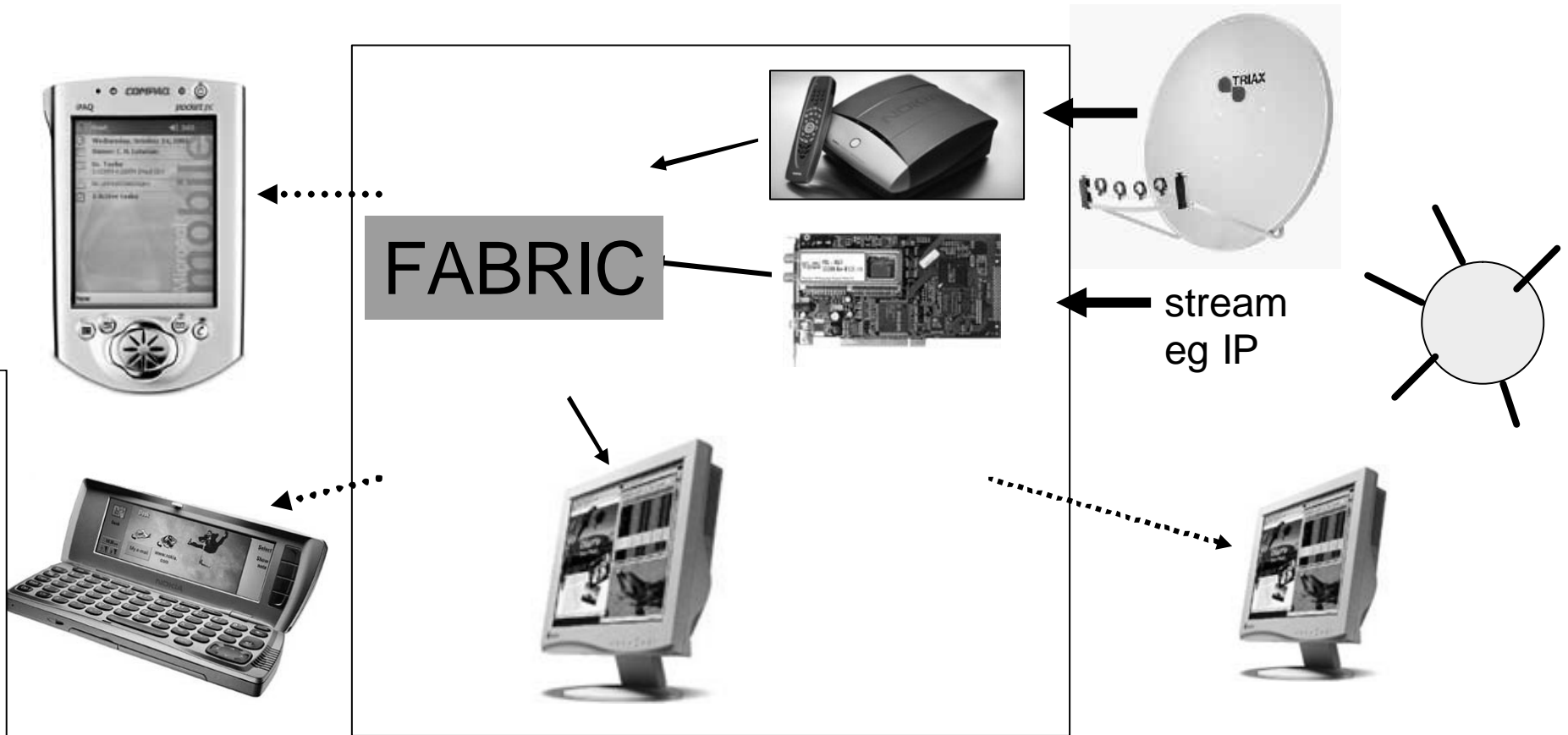
- high quality video for consumer terminals, eg, TV sets, movies
 - strict *real-time* behavior
 - frame rate
 - continuity
- mobile devices, eg, for news, sports
 - less strict demands, but good adaptive quality
- IP
 - internet, www
 - ok performance, not interfere with “quality streams”
- variations in MPEG stream demands and network availability

-
- real-time methods to determine resource demands of groups of pictures
 - offline “skeleton” of reserved resources for minimum quality of service of guaranteed streams
 - online handling of additional frames
 - acceptance tests to
 - select “best” group of frames
 - guaranteed to be decoded – no loss due to partial frames
 - various streams for various devices, networks, quality

- Mitsubishi Research Labs
- real-time communication middleware – IP
- NT - CE



- EU – IST Project
- Philips coordinator
- IP, bluetooth, firewire, etc



Summary

- predictable flexibility
configure amount of flexibility for each task individually
- system with mix of activities and demands
select appropriate methods and costs
- combine event triggered and time triggered activation schemes
- transform between offline, EDF, FPS scheduling schemes
less scheduling centric design
- applied to applications with mixed demands
 - control
 - in home entertainment networks

THE END

Pre Run-time Scheduling – Flexibility Integration Offline - Online

Real-Time Systems - ESSES

Gerhard Fohler

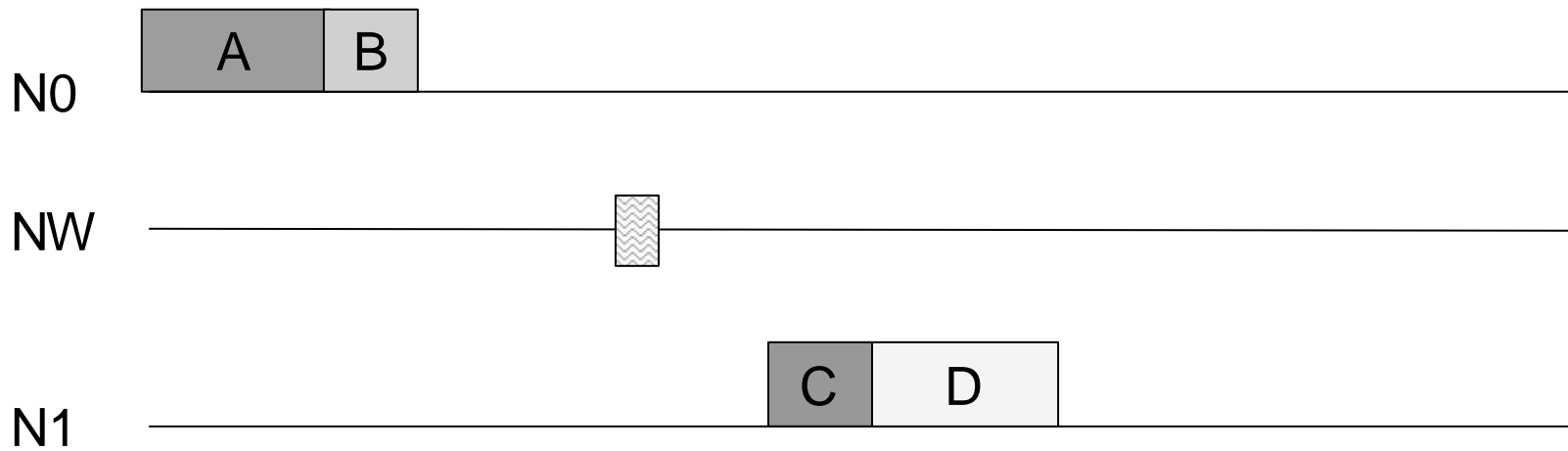
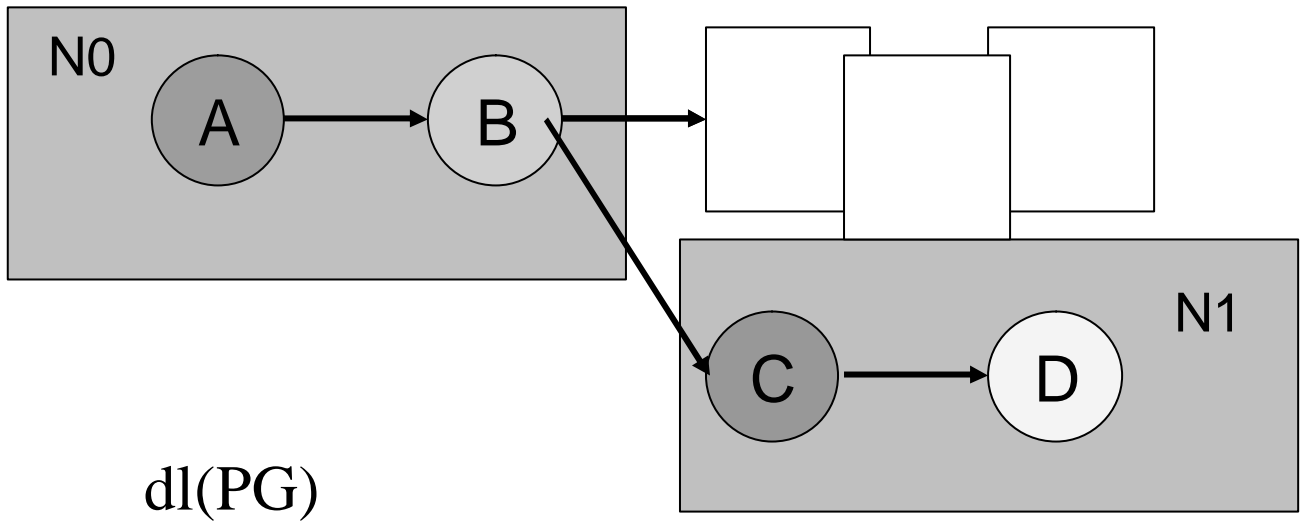
Mälardalen University, Sweden

gerhard.fohler@mdh.se

Offline schedules

- general timing constraints
- offline scheduler
 - resolves constraints
 - constructs one solution which meets all constraints
- fixed (blind) runtime execution
- no flexibility

- how can we
 - increase flexibility
 - add dynamic tasks
 - integrate with online scheduling methods



Slot Shifting...

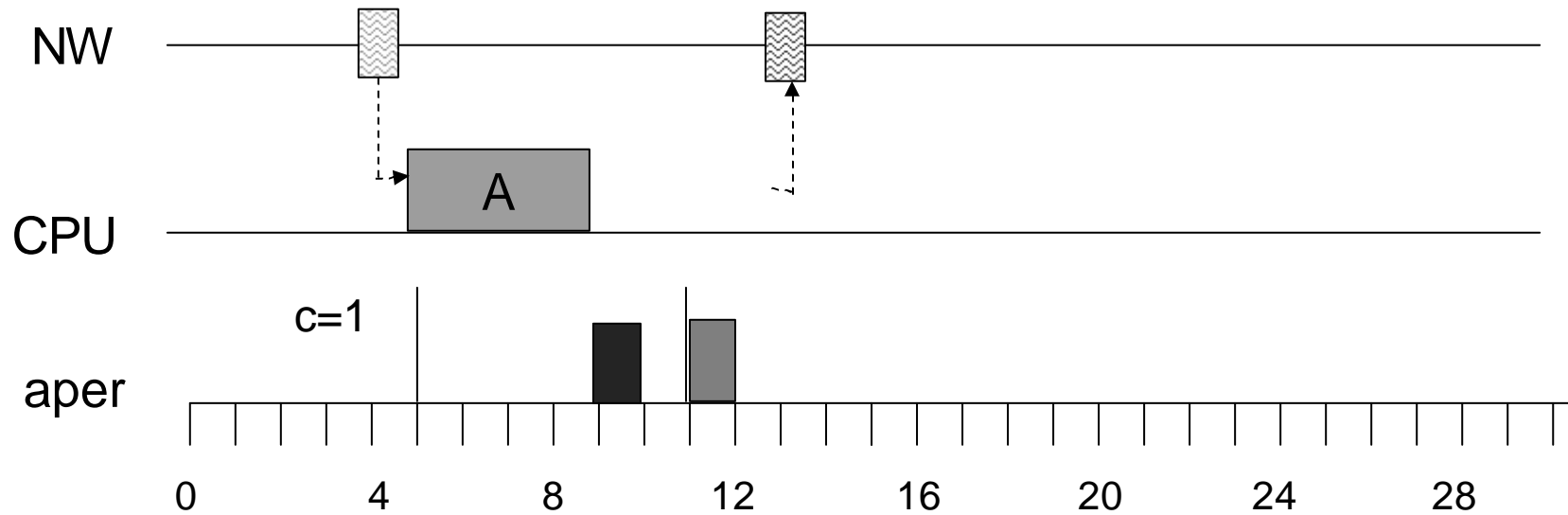
Offline

- timing constraints ✓
- offline schedule ✓

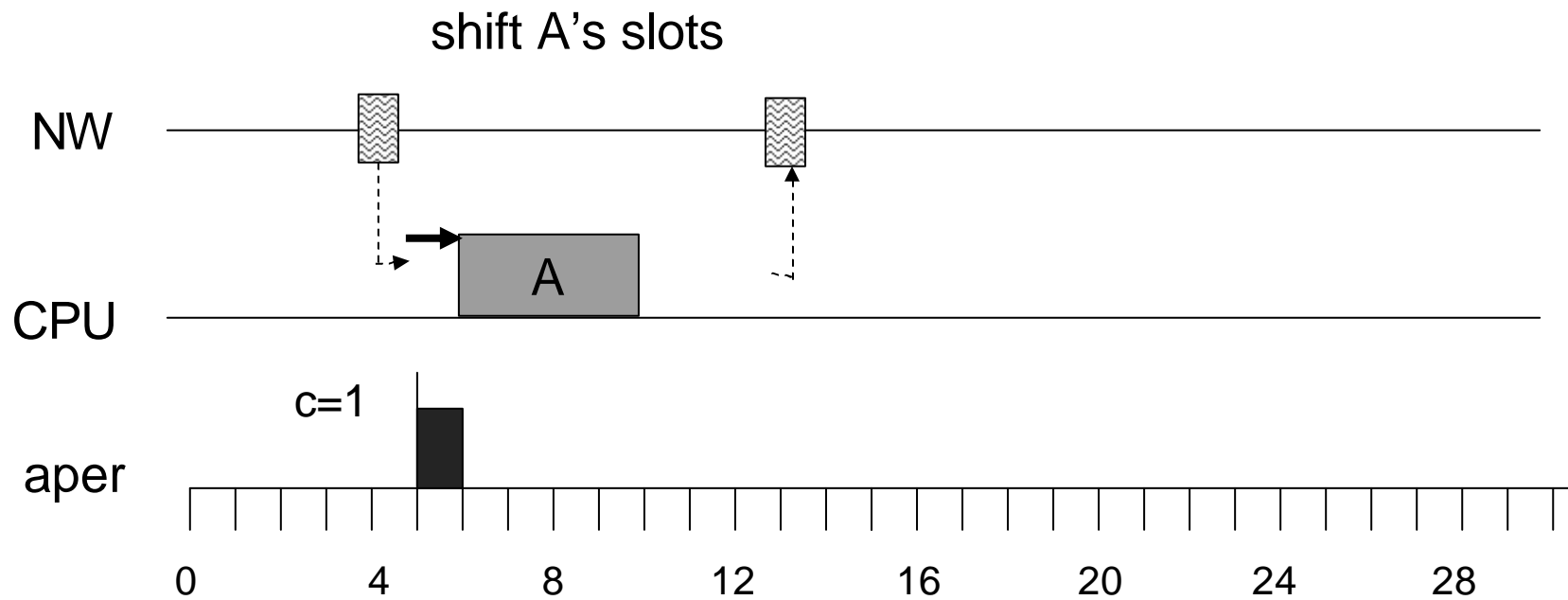
- we have
 - offline constructed schedule
- we want
 - include dynamic tasks
 - schedule them online
- what can we do?
 - include in offline schedule (e.g., pseudo periodic)
⇒ inefficient
 - fit into empty slots ⇒ no guarantees
 - we can do better!

“Background Service”

fixed pre runtime schedule:



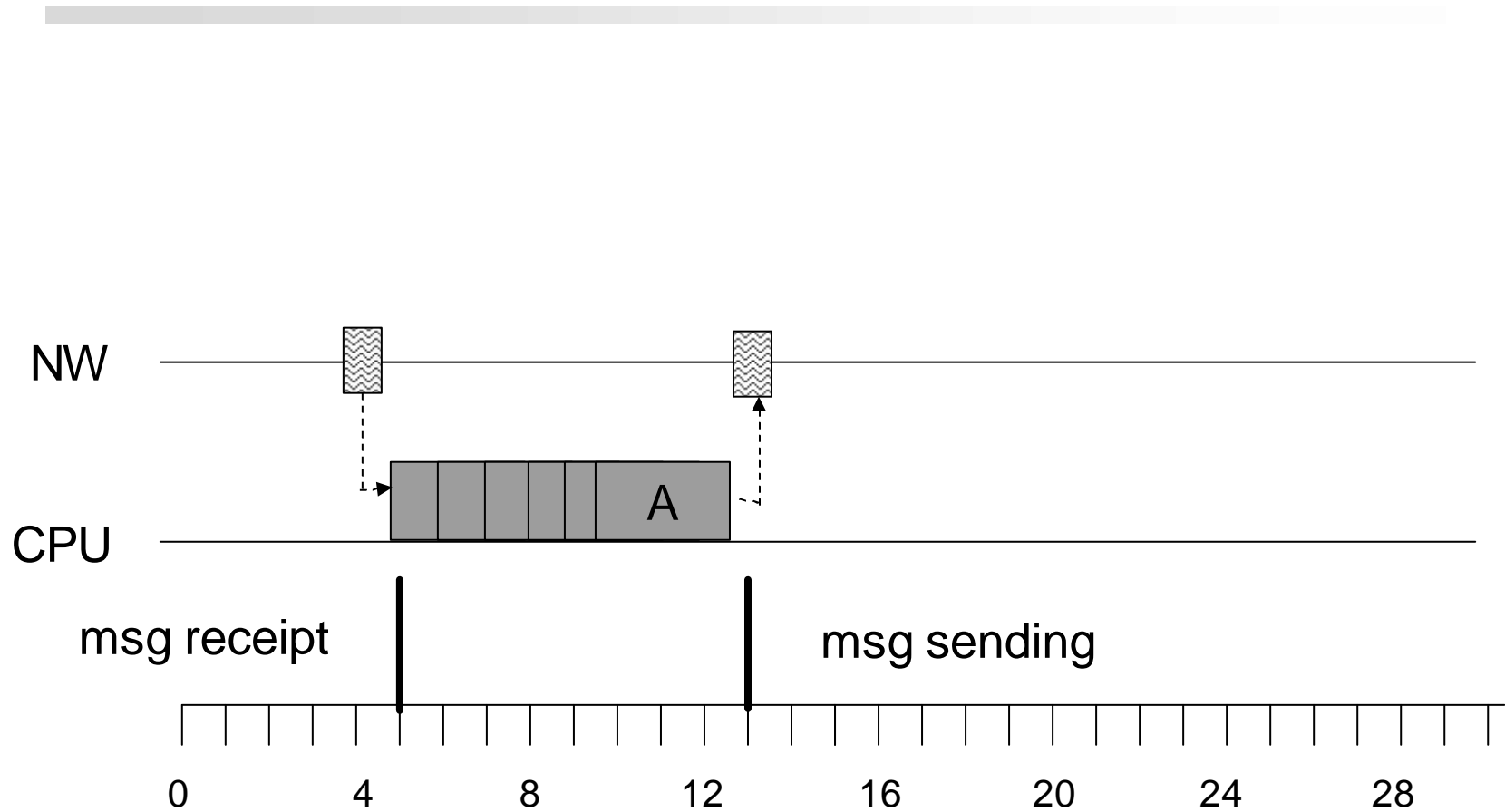
Basic Idea



Shifting pre-runtime tasks

- pre runtime schedule assigns fixed times for execution
 - although different times possible
 - *overconstrains* schedule
 - we have to select *one out of several possible* times
 - ...for the *sake of algorithm* only
- we know, that we can shift A
 - execute the aperiodic task at once
 - feasibility of tasks not violated
 - how much and where can we shift?
 - what are boundaries?

Shifting tasks



Limitations on Shifting

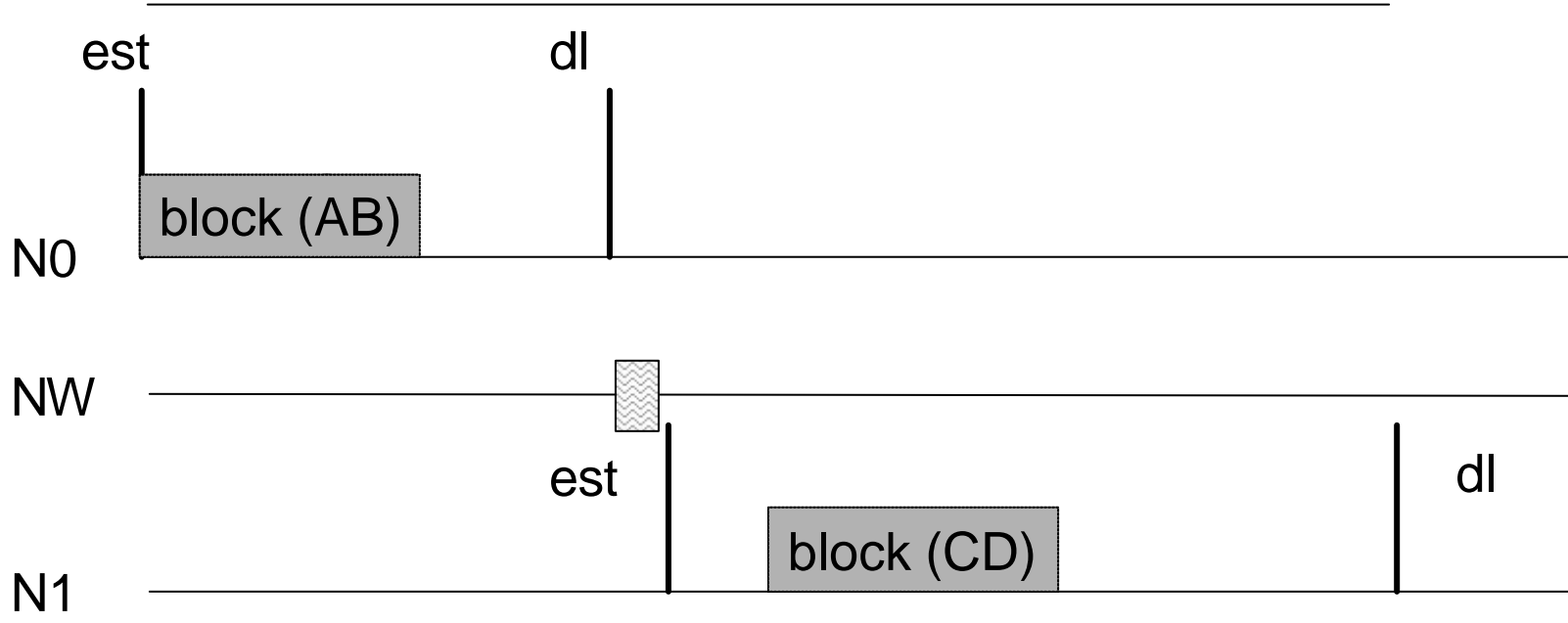
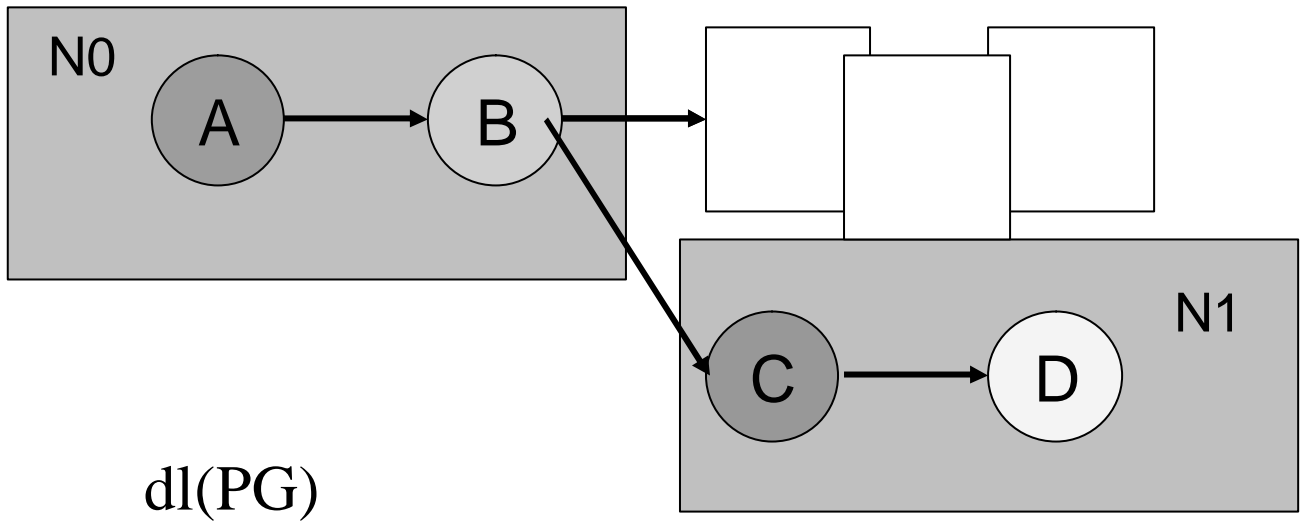
we can shift tasks

limitations

- receipt of message
- sending of message
- earliest start time of precedence graph, end-to-end constraints, task chain
- deadline of -"-

calculate start time, deadline pairs for tasks

- expresses flexibility of task
- reduces overconstraining
- fit in aperiodic task by shifting as long as these constraints met



These tasks are assigned fixed *starttimes* or *deadlines*.
(Subgraphs of precedence graphs allocated to nodes combined.)

⇒ independent tasks with starttimes, deadlines on single nodes

simple EDF runtime scheduling

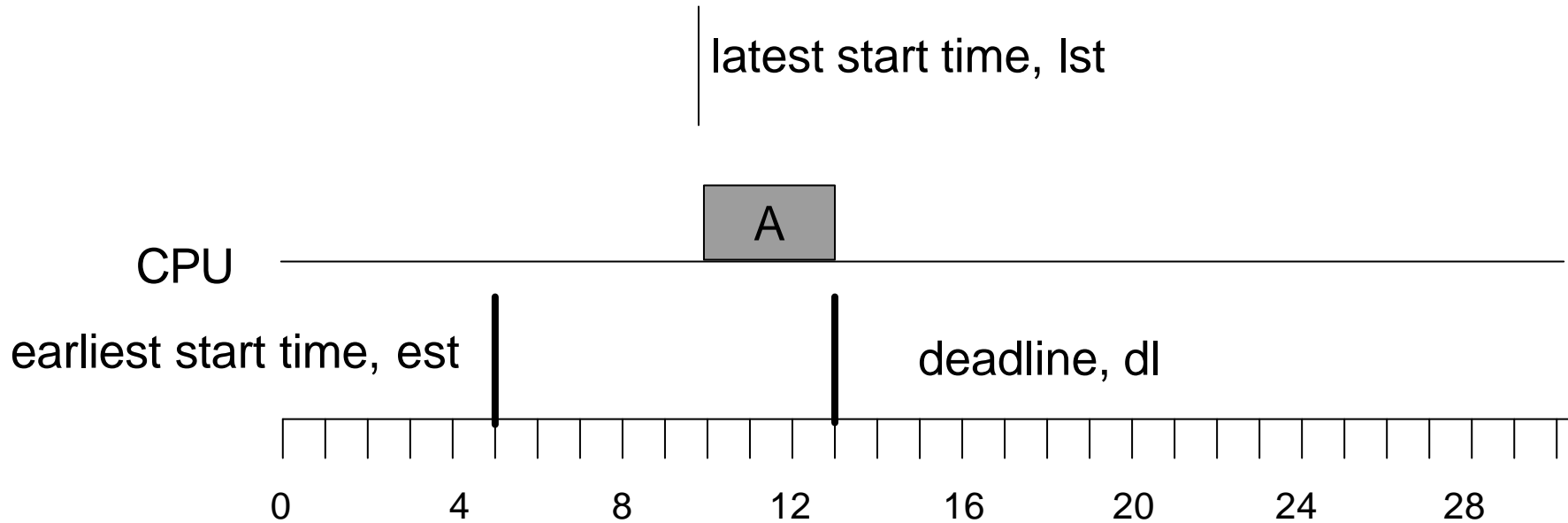
Slot Shifting ...

Offline

- timing constraints
- offline schedule
- earliest start times, deadlines ✓

How much shifting?

- know what is
 - earliest time to start task
 - latest time to finish
- aperiodic arrives: how far can we shift static task?



- latest start time
start no later or violate deadline
- have to ensure when executing aperiodics
how?
- more complex dispatching
still next task, but check for constraints
- more memory - 3 integers per task

Slot Shifting

Offline

- timing constraints
- offline schedule
- earliest start times, deadlines
- latest start times ✓

Insert how much? Where?

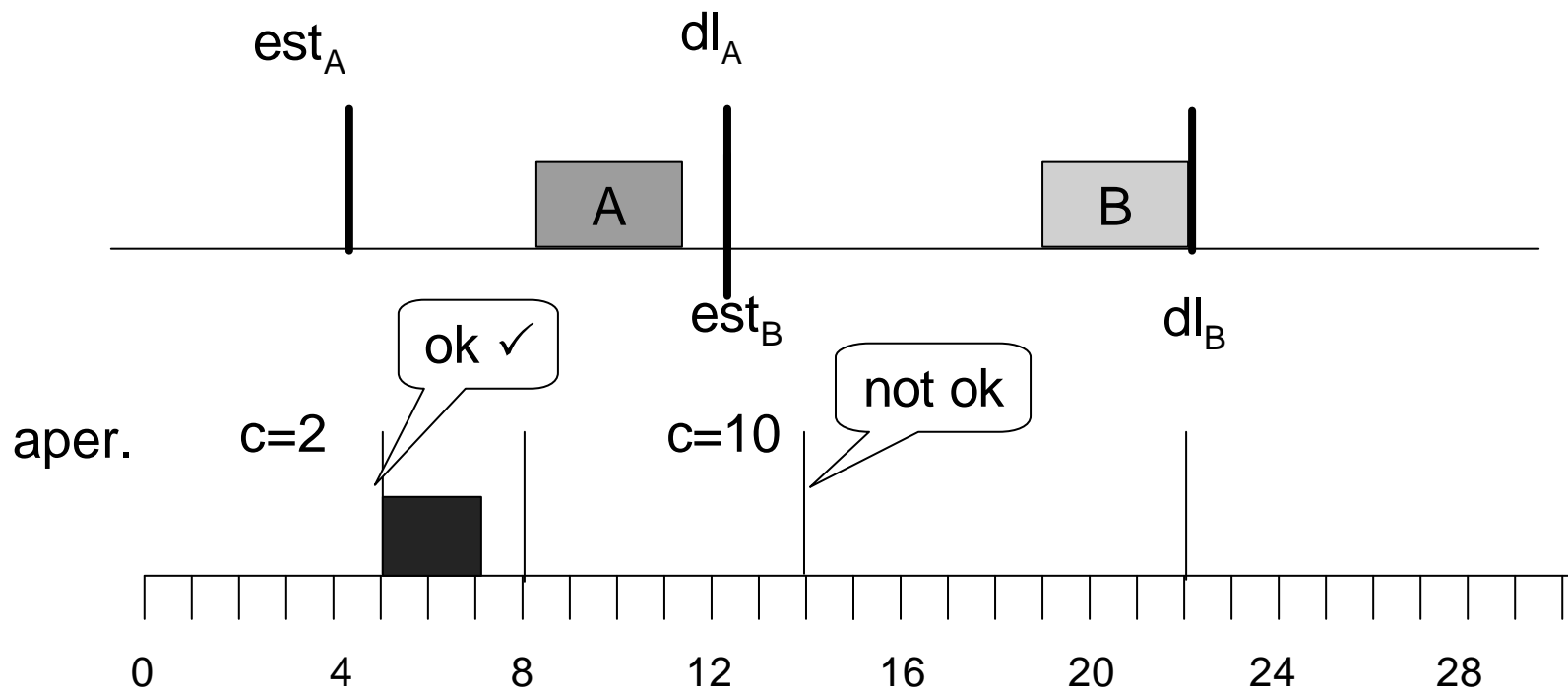
so far soft aperiodics

can we give guarantees for firm aperiodics?

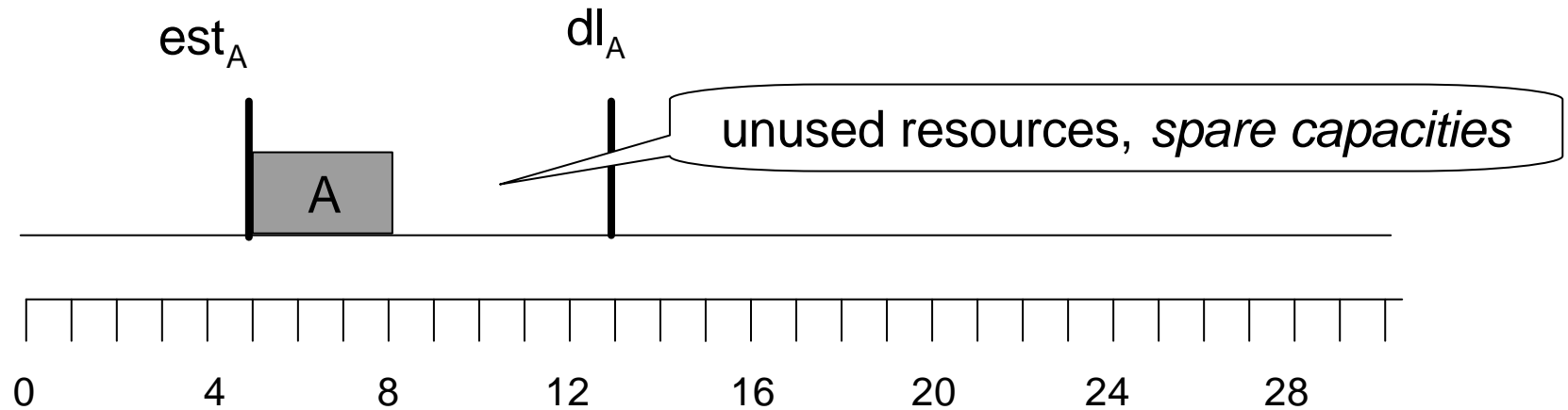
- worst case execution time
- deadline
- before start, want to guarantee that we can complete them

how can we decide?

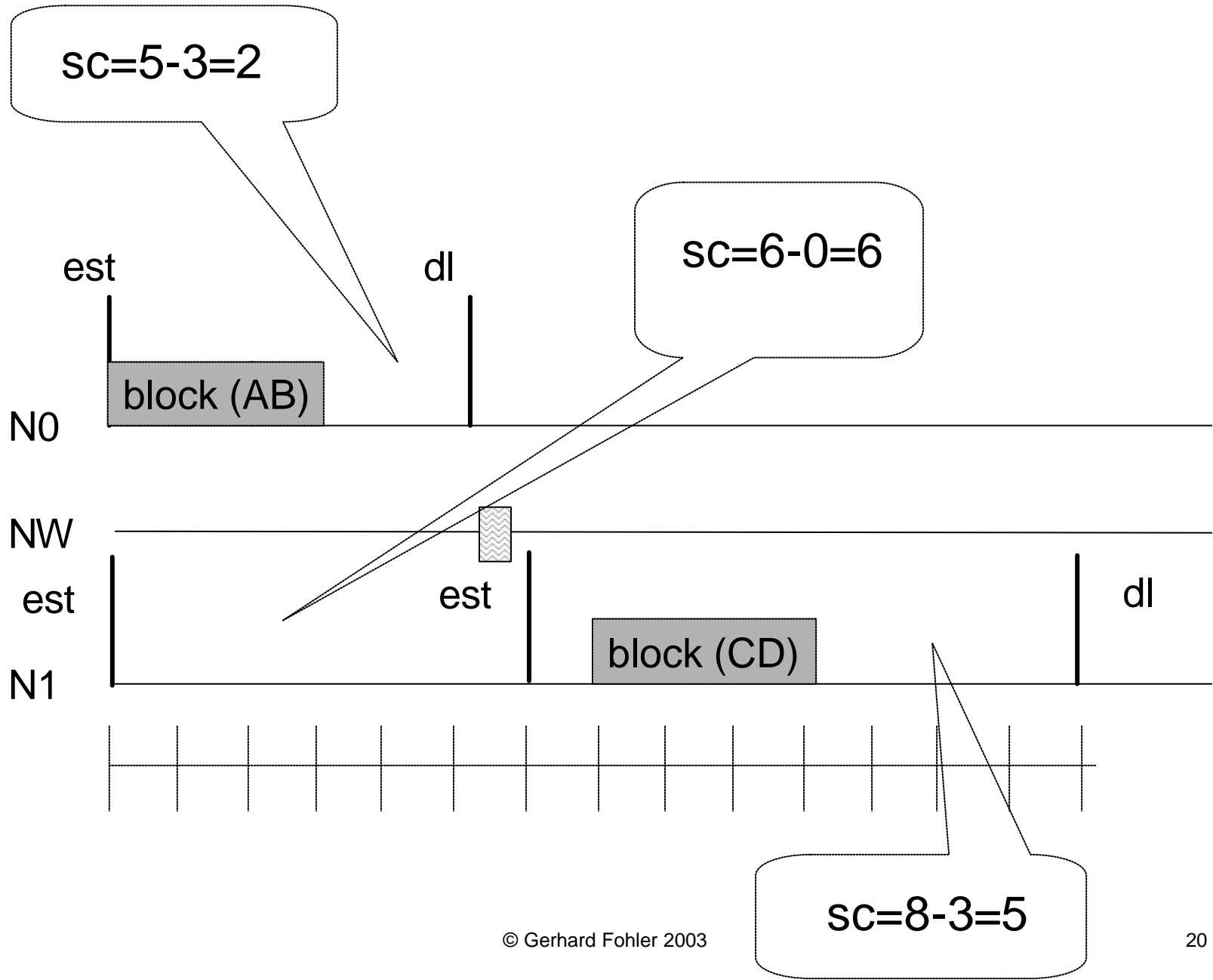
- need idle resources for aperiodics
- before deadline of aperiodic
- which resources can we use?



Spare Capacities



- spare capacities, $sc = \text{length of execution interval} - \text{execution times}$
- available for aperiodic tasks
- know amount and location from schedule!



Slot Shifting

Offline

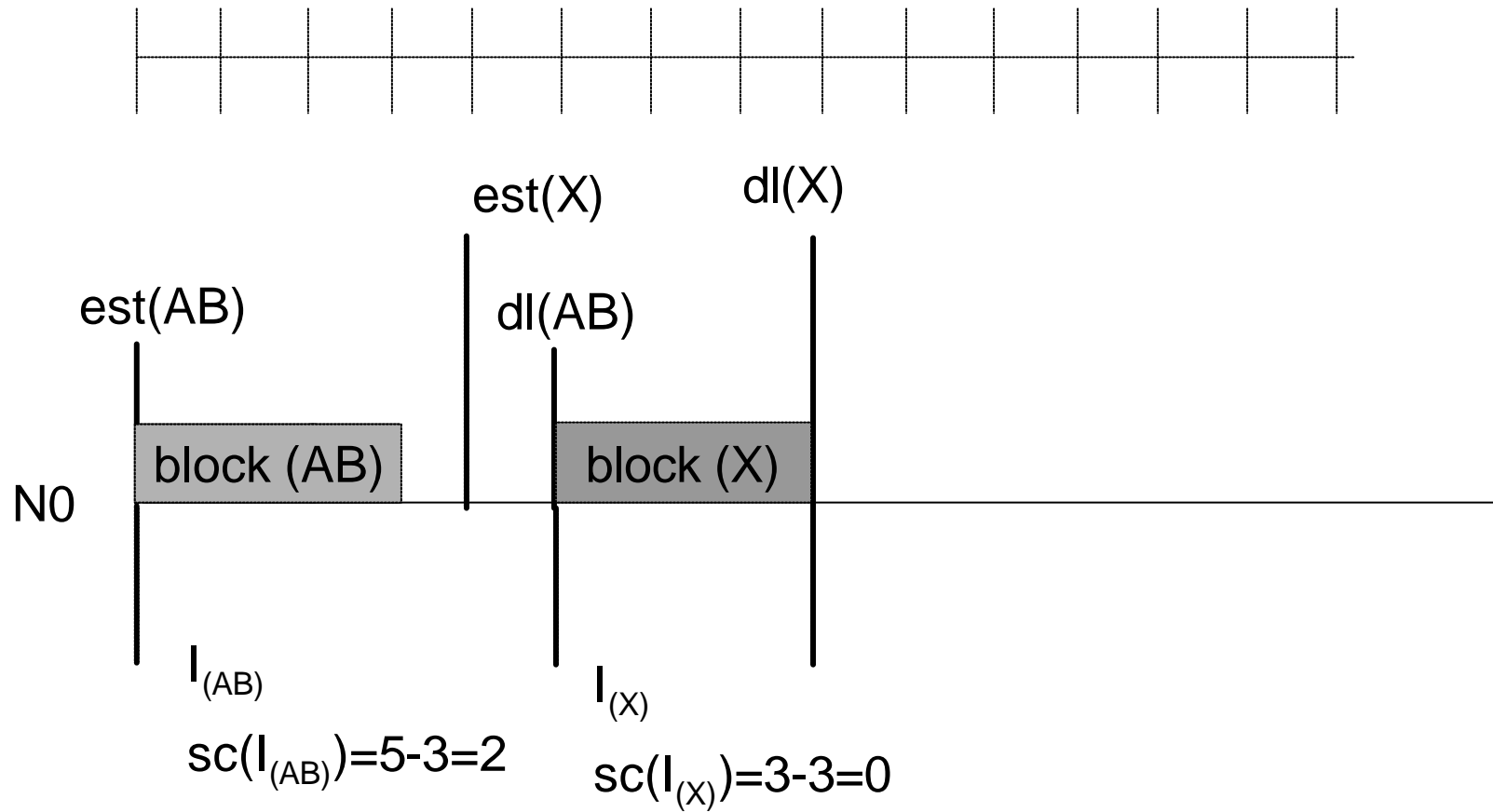
- timing constraints
- offline schedule
- earliest start times, deadlines
- latest start times
- ~~spare capacities~~ ✓ not yet...

Intervals

sort deadlines \Rightarrow *disjoint intervals*:

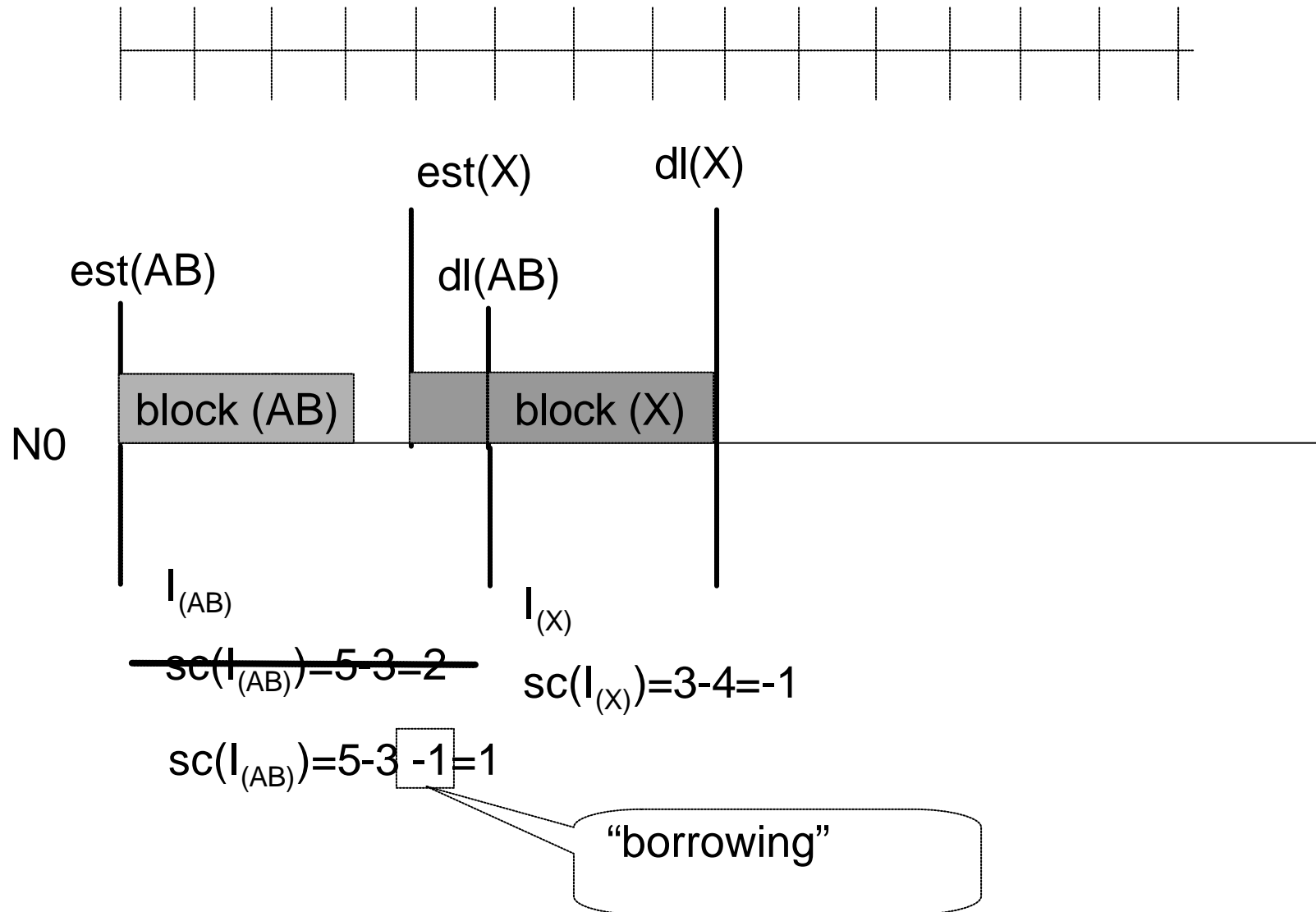
- *end*: deadline of task(s)
- tasks with that deadline
- *spare capacity, sc*
the amount of idle resources in that *interval*
- *start*: max of est of task(s) and end previous interval

- empty intervals:
 - $\text{end}(I_{i-1}) < \text{start}(I_i)$
 - $\text{wcet} = 0$



$$sc(I) = |I| - \sum_{T \in I} wce(T) \quad \text{..almost the truth...}$$

- intervals \neq execution intervals!



$$sc(I_i) = |I_i| - \sum_{T \in I_i} wcet(T) + \min(sc(I_{i+1}), 0)$$

borrowing mechanism:

- if tasks in subsequent interval need more resources than available in it:
execute in other interval, use resources from there “borrow”
- run-time mechanisms resolve negative spare capacity
- only for calculation and flexibility
- start of interval can be \neq earliest start time
- earliest start time checked separately

Slot Shifting

Offline

- timing constraints
- offline schedule
- earliest start times, deadlines
- latest start times
- intervals ✓
- spare capacities ✓

Online Mechanisms- Scheduling

online scheduler invoked at each node after each slot

- check for new aperiodic tasks
- guarantee algorithm
- take scheduling decision
- update spare capacities
- execute scheduling decision

earliest deadline first

- after each slot, scheduling decision taken locally at each node
 - no ready task:
CPU idle
 - $sc(I_c) > 0$, \exists soft aperiodic task A:
execute A
 - $sc(I_c) = 0$:
an offline or guaranteed task has to be executed or
deadlines are missed
takes care that no latest start time is missed!
no other mechanism needed, eg, watchdog, etc
implicit invocation, no extra memory needed
 - $sc(I_c) > 0$, $\neg \exists$ soft aperiodic task:
offline or guaranteed task executed

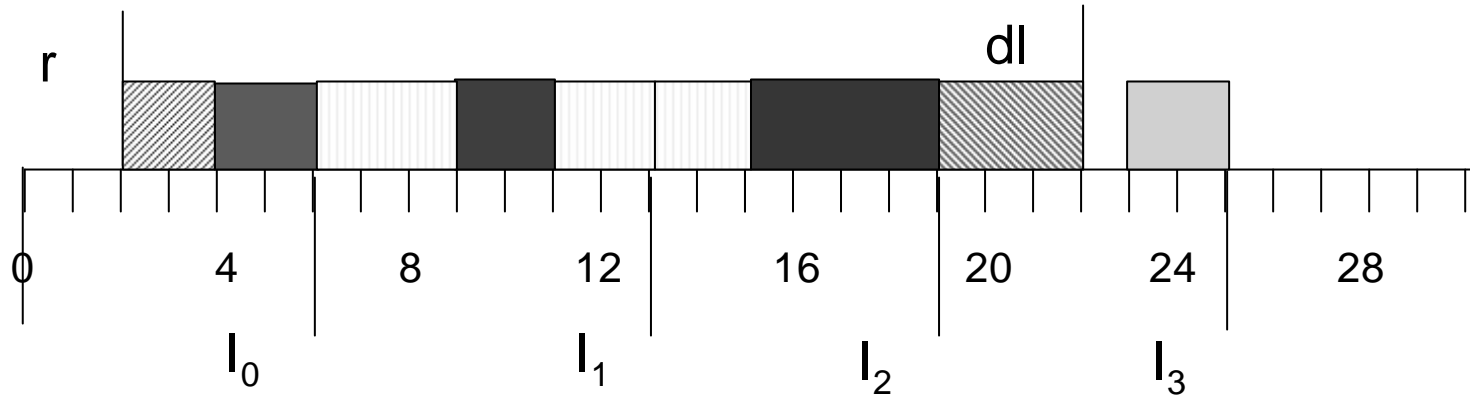
Acceptance of aperiodics




- aperiodics (without deadline):
sc > 0: one slot can be given to it
- firm aperiodics (wcet and deadline):
want them executed either completely or not at all

⇒ *guarantee algorithm*

O(N)

- aperiodic task A ($r, wcet, dl$)
- three parts of spare capacities available



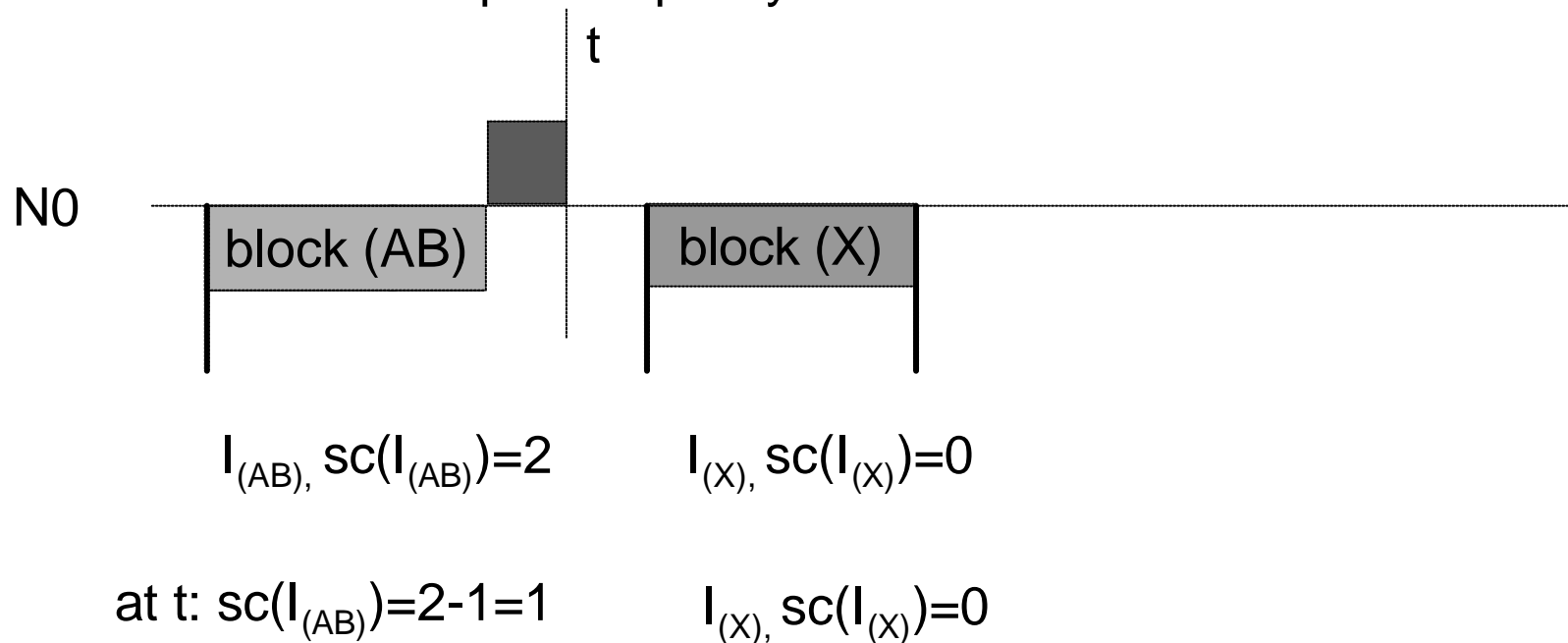
-  $sc(I_c)$: remaining sc in current interval
-  $sc(I_i)$: $sc(I_i) > 0$, $c < i \leq I$, $end(I_i) \leq dl(A)$, $end(I_{i+1}) > dl(A)$, sc in all *full* intervals between r and dl
-  $\min(sc(I_{i+1}), dl(A) - dl(I))$, minimum spare capacities of last interval or up to the deadline of aperiodic in last interval

Guarantee

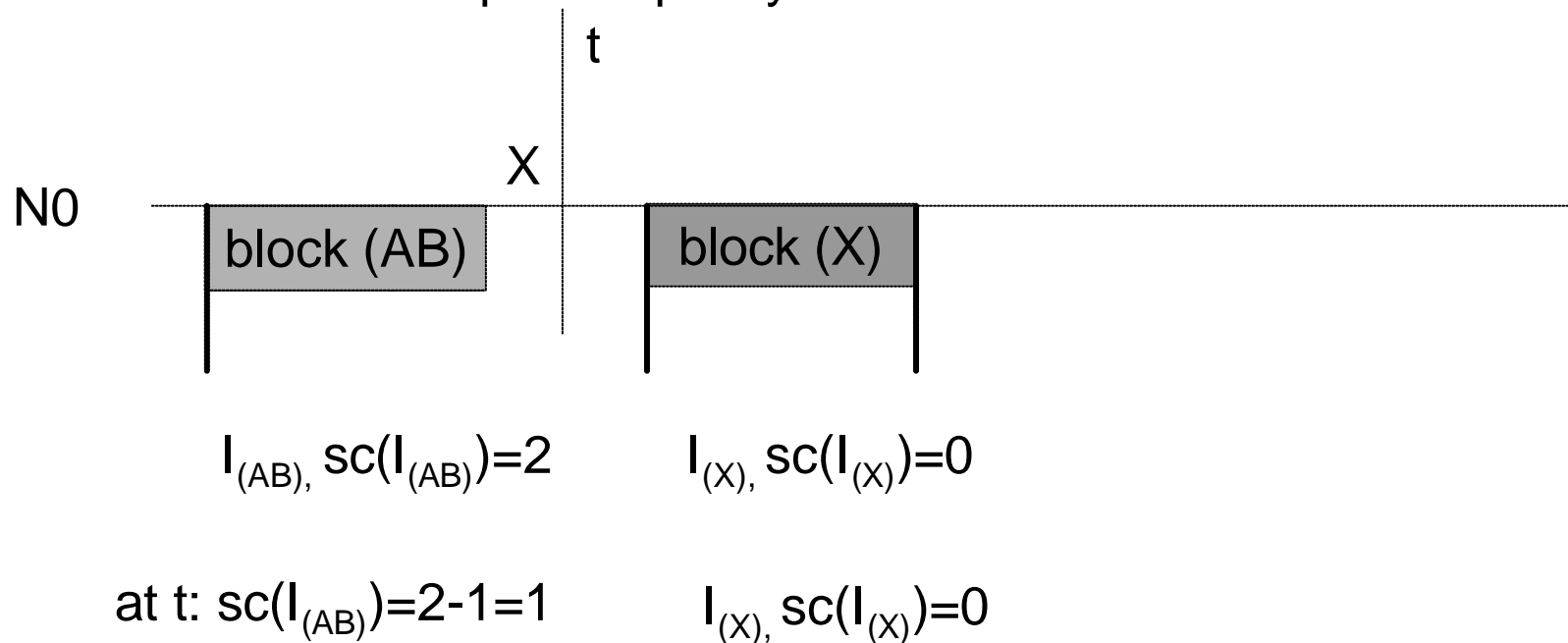
- if sum of total sc between dl and r are larger or equal wcet, guarantee
- need to ensure guarantees resources are not used otherwise
- after guarantee:
 - update interval l
 - update interval l-1
 - ...
 - update interval c

Spare capacities at runtime

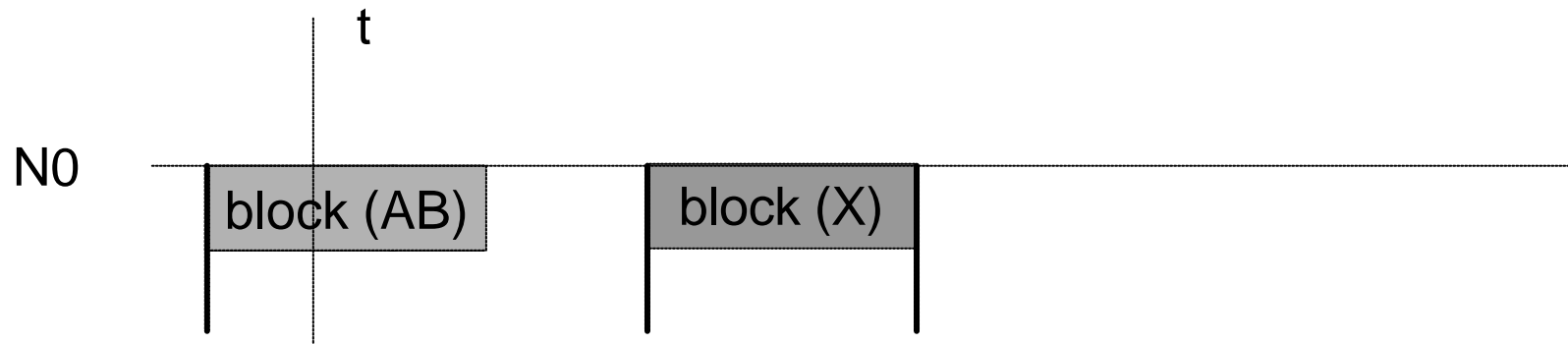
- aperiodic execution
 - decrease spare capacity of current interval



- no execution
 - decrease spare capacity of current interval



- execution of offline task T
 - $T \in$ current interval I_c
 spare capacity stays the same



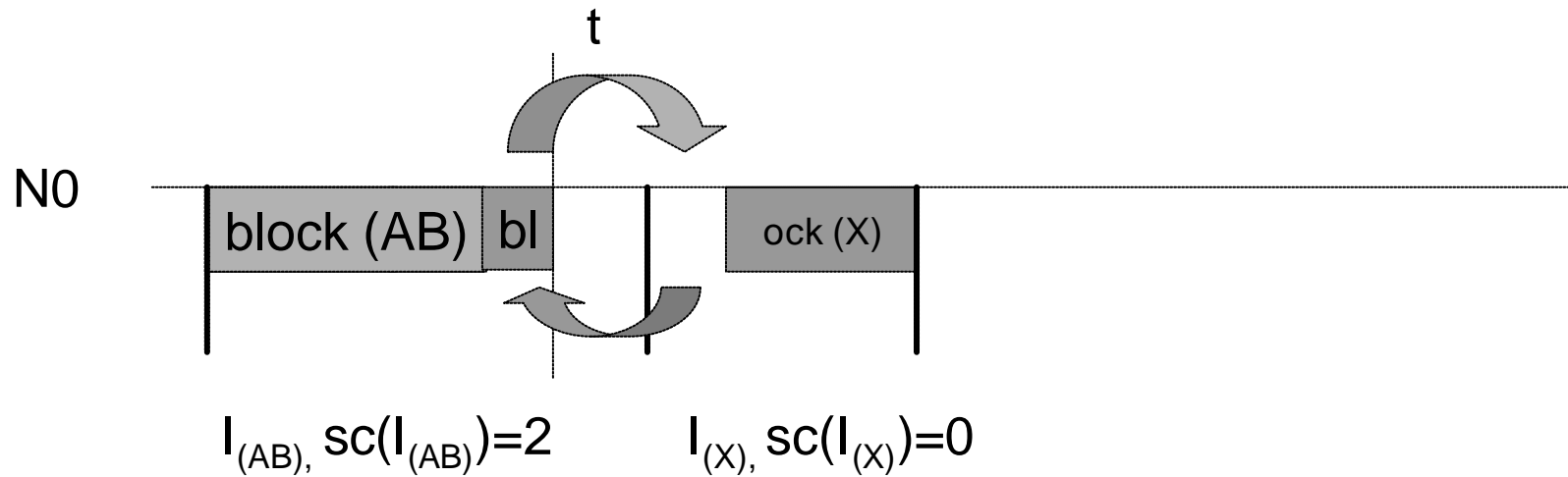
$$I_{(AB)}, sc(I_{(AB)})=2$$

$$I_{(X)}, sc(I_{(X)})=0$$

$$\text{at } t: sc(I_{(AB)})=2$$

$$I_{(X)}, sc(I_{(X)})=0$$

- execution of offline task T
 - $T \in$ future interval I_f
 - spare capacity I_c decreased
 - spare capacity I_f increased



at t: $sc(I_{(AB)})=2-1=1$

$I_{(X)}, sc(I_{(X)})=0+1=1$

- update capacity of I_f
 - if ≥ 0 ...done
 - if < 0 ... need to update previous interval I_{f-1}
- $sc(I_{f-1})$
 - if ≥ 0 ...done
 - if < 0 ... need to update previous interval I_{f-2}
-
- until $sc \geq 0$ or I_c

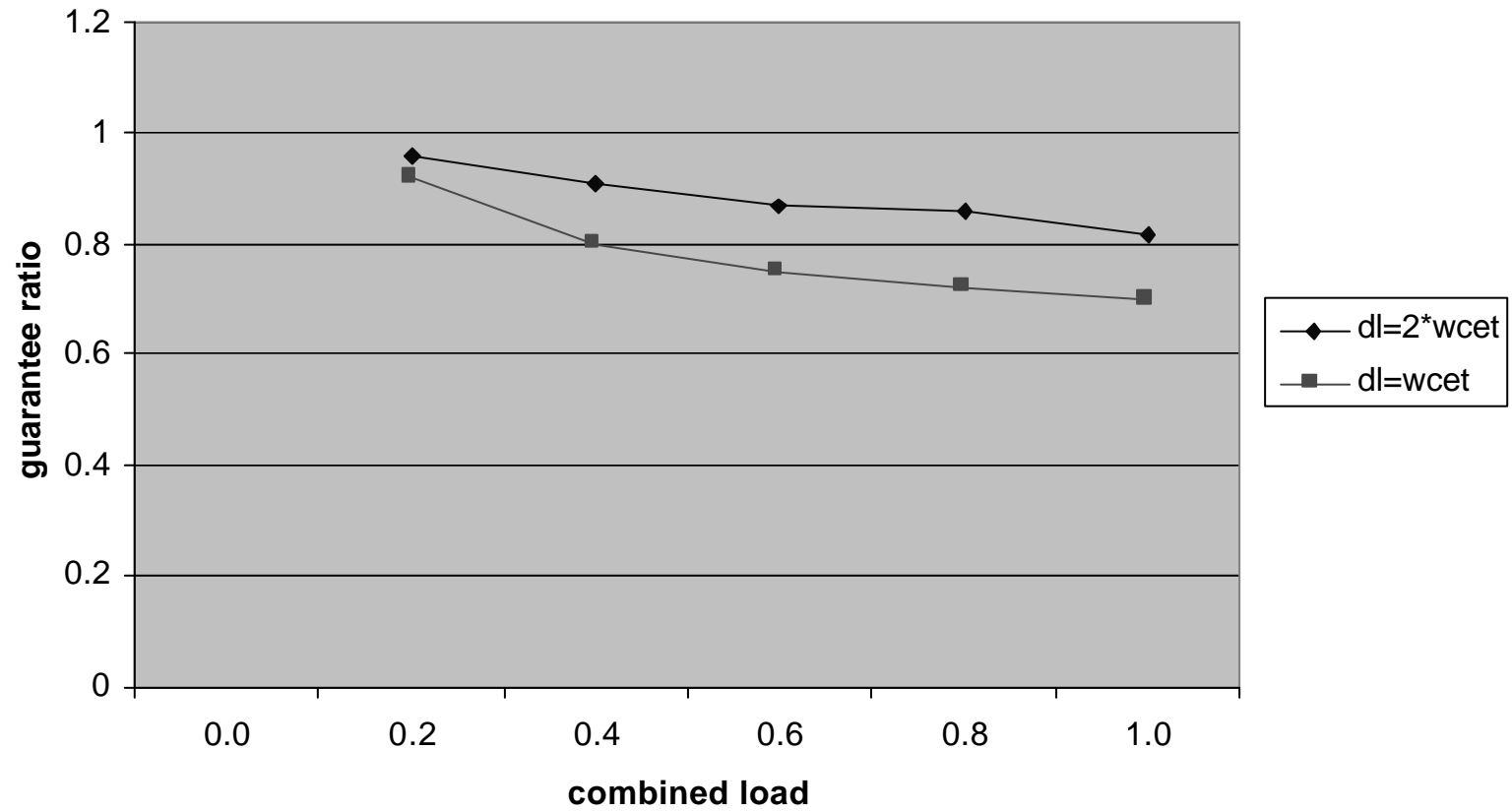
Shifting Messages

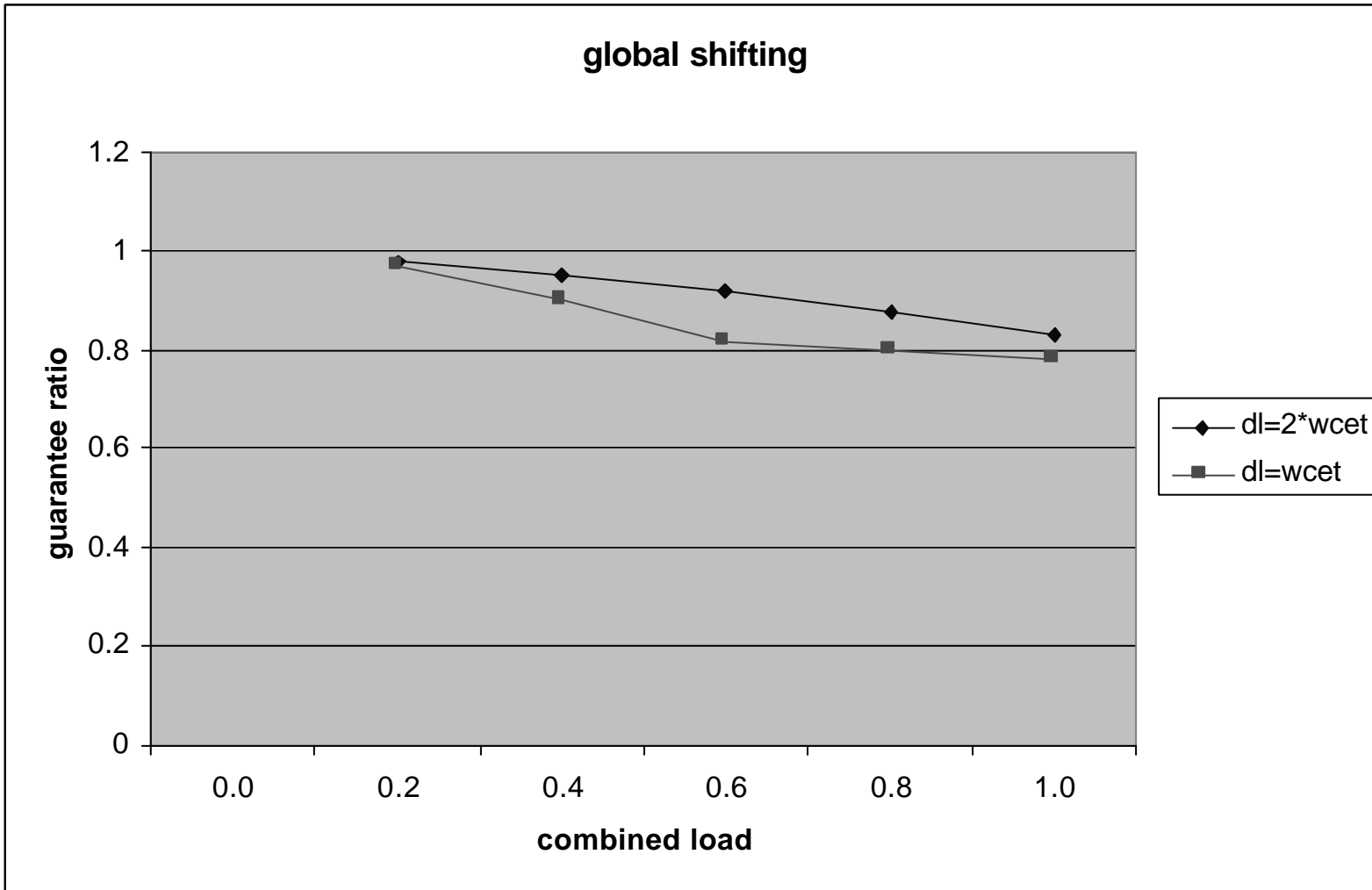
- communication medium resource like CPU from scheduling perspective
- shift messages as well
- restriction to sending messages *earlier*
 - no receiver synchronization necessary
 - may increase spare capacities at receiver
 - when message received - spare capacities updated
 - else same

Analysis

- MARS
- 4 CPUs
- TDMA network
- ~1600 task sets generated and pre runtime scheduled
- randomly generated aperiodic tasks
- each point in plots 700-1000 task sets
- 0.95 confidence intervals < 5%

local shifting





“Slot shifting nouveau”

- further acceptance test
- integration with TBS
-

| | | Periodic with constraints | | Sporadic | Aperiodic | |
|---------|------|---------------------------|---|--|--|--|
| | | Simple | Complex | | Firm | Soft |
| | | | <ul style="list-style-type: none"> • Periods • Deadlines • Start times | <ul style="list-style-type: none"> • End-to-end dl • Inst. separation • Distribution • Jitter etc. | <ul style="list-style-type: none"> • Minimum separation between instances | <ul style="list-style-type: none"> • Deadlines • Guarantee |
| Offline | Sch | X | X | | | |
| | Test | | | X | | |
| Online | Sch | X | X | X | X | X |
| | Test | | | | X | |

Slot Shifting - Summary

- handle online tasks while maintaining feasibility of offline scheduled tasks
- offline reduction of complexity
- simple runtime handling
- “interface” for integration of offline and online scheduling

- offline scheduled system for critical activities
- restrict amount of shifting
- flexibility for rest

predictable flexibility

Articles

- Gerhard Fohler
Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems
Proc. of the 16th IEEE Real-Time Systems Symposium, Pisa, Italy, December 1995.
- Damir Isovich, Gerhard Fohler
Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints
Proc. of the 21st IEEE Real-Time Systems Symposium, Orlando, Florida, USA , November 2000

Pre Run-time Scheduling - Mode Changes

Real-Time Systems - ESSES
Gerhard Fohler 2003
Mälardalen University, Sweden
gerhard.fohler@mdh.se

Mode Changes

- systems undergo a number of mutually exclusive mode during operation
e.g., air craft ground, take off, flight, landing
 - different system activities
 - different attributes of activities
altitude not critical on ground
 - system configuration
- difficult to handle in single schedule
- *provide separate modes plus transitions*
- context of offline scheduling
- important in aeronautics

What is different in modes?

- selection of control loops
- Timing requirements
- attributes of activities (“critical”, “hard”,...)
- system configuration
- reliability

how to deal with pre runtime scheduling?

put all activities of all modes into a single schedule

What is affected by this approach?

- solutions:
High overhead of incorporating all resource needs, even mutually exclusive ones.
- design and understandability
 - violates modularization and separation of concerns
 - large number of design items
 - difficult to recognize coherent activities
- testing
input space larger than really required

better approach

separate modes as well as mode changes

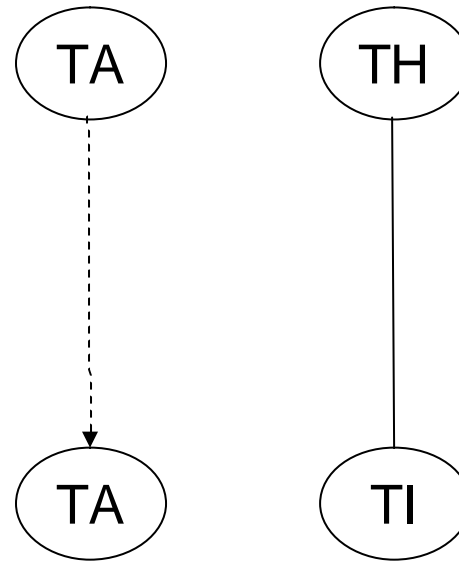
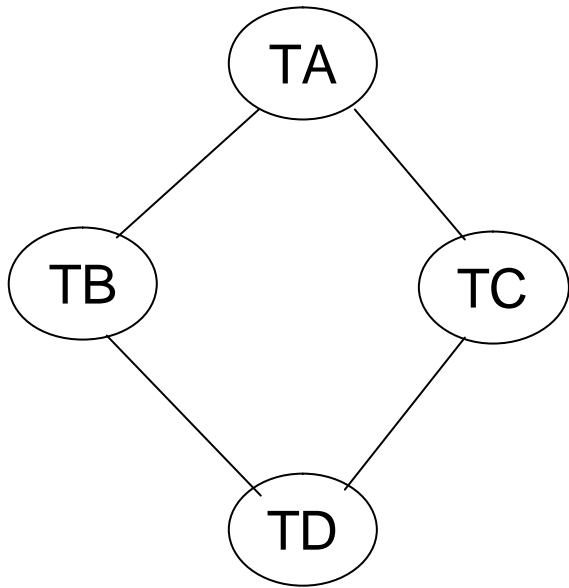
Requirements

- deterministic temporal behavior
- specification
 - timing constraints
 - also for mode changes, transitions
 - adhere to design principles of single modes
 - not new methods
 - consistent design approach
- retain continuous system operation during mode changes!
 - tasks executing in old, new mode, and transitions not impaired by mode change
 - e.g., don't shutdown engines during transition in midair

Design/Specification Issues

Mode:

- single operational phase, performed by a single static schedule (-ing table)
 - specified as precedence graphs (transactions, execution chains)
 - one mode:
 - set of precedence graphs - all activities in one mode
 - two modes:
 - set of precedence graphs for each mode separately
 - allow tasks in both modes, label
 - second dimension: modes
- two dimensional precedence constraints*



M0

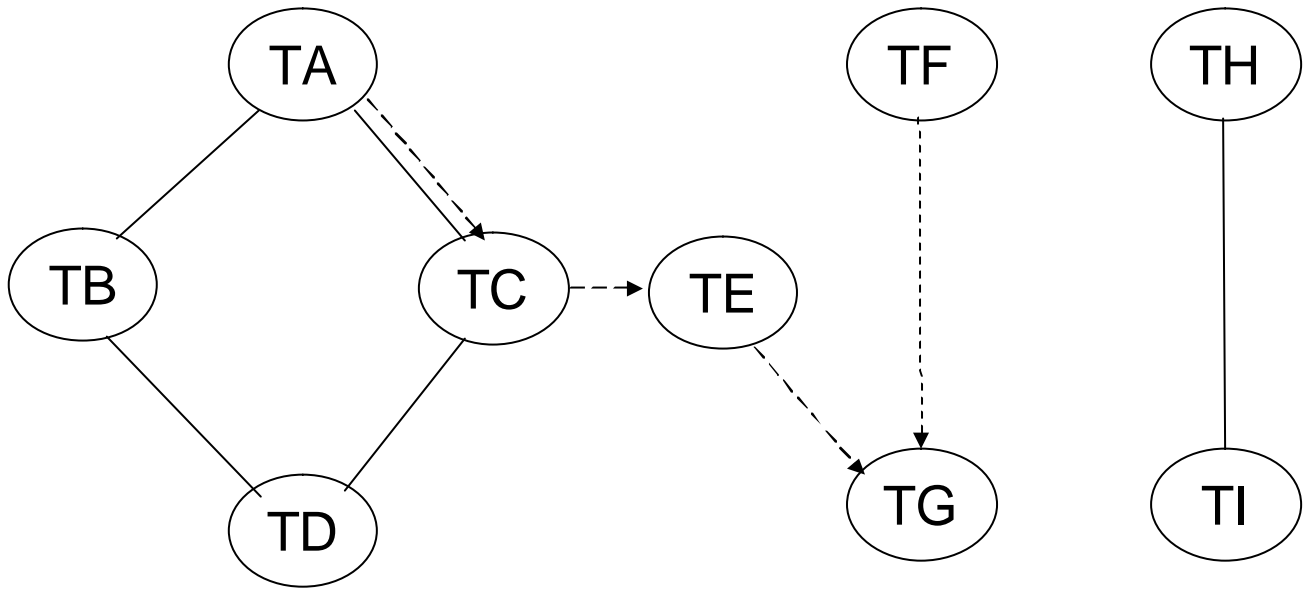
MF

Transition Precedence Graph

transition:

tasks of

- old mode
- new mode
- plus (optional) additional ones
 - complete old activities
 - prepare for new mode
 - intermediate actions
- viewed as mode itself, but not executed continuously
- same design method as for single mode
- ongoing activities part of transition schedule



MO ——— MT - - - - -

MF - - - - -

Runtime handling

- mode change requested
 - switch to transition mode, schedule when feasible
 - execute transition mode until all activities in it completed
 - switch to new mode when feasible
- switching directly into mode schedule may cause problems, e.g., inconsistencies, aborted tasks, etc.
- agreement on which mode should be changed two if more than one request - offline resolution

Design Issues

Semantic constraints

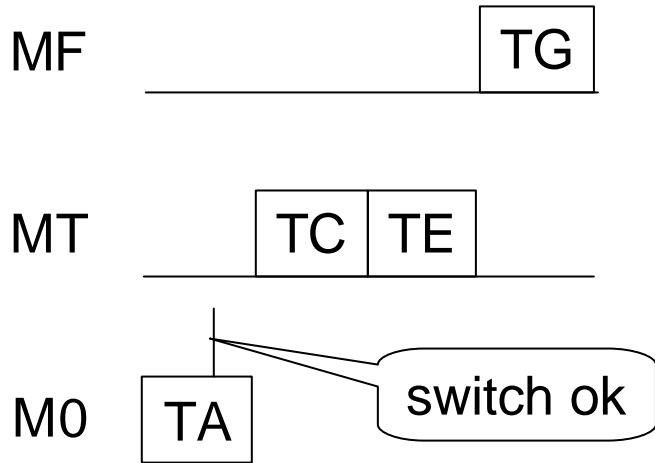
what do we need to be able to express?

- *immediate change, aborting current activities*
empty transition graph
- *completing all current activities before changing*
transition graph identical to graph of old mode
- *completing some of the current activities*
transition graph comprised of part of activities of old mode and new mode
- *additional activities*
old, new mode activities, plus new ones

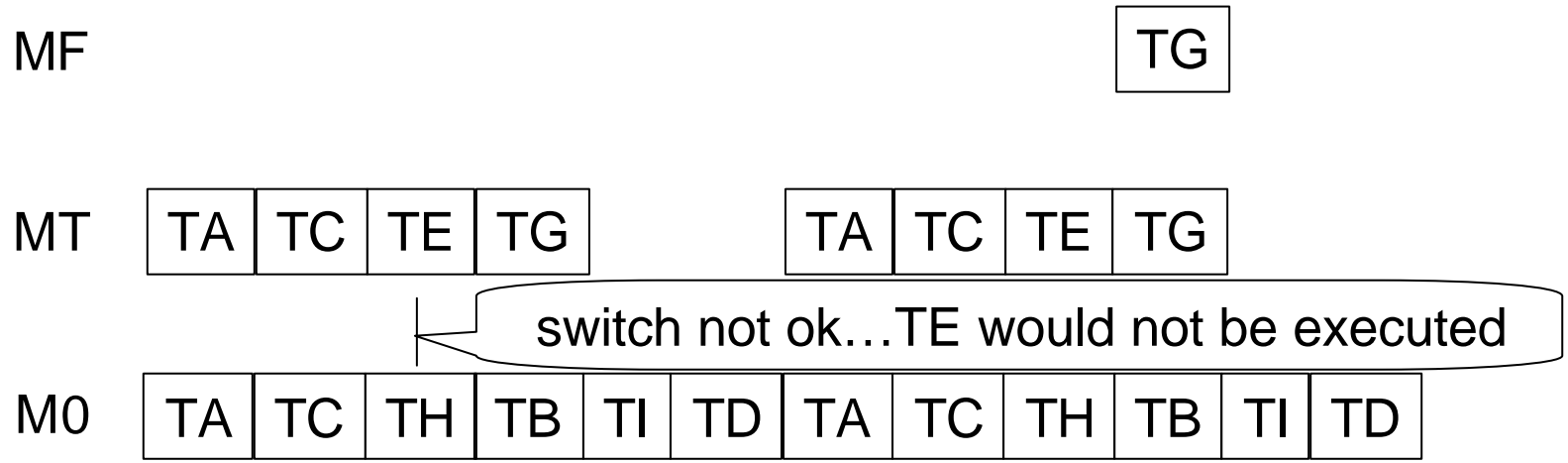
Mode change conditions

e.g., $t > 50$

- consistency check
- conflict resolution
- who initiates mode change request
- design via automaton
- global consistent view



| mode change request



Mode Change Schedule Construction

Construct a schedule such, that

- timing constraints of individual modes met
- timing constraints for transitions met
- deterministic behavior
- flexible and fast reaction times

single mode scheduling NP hard..how about that?

Trick:

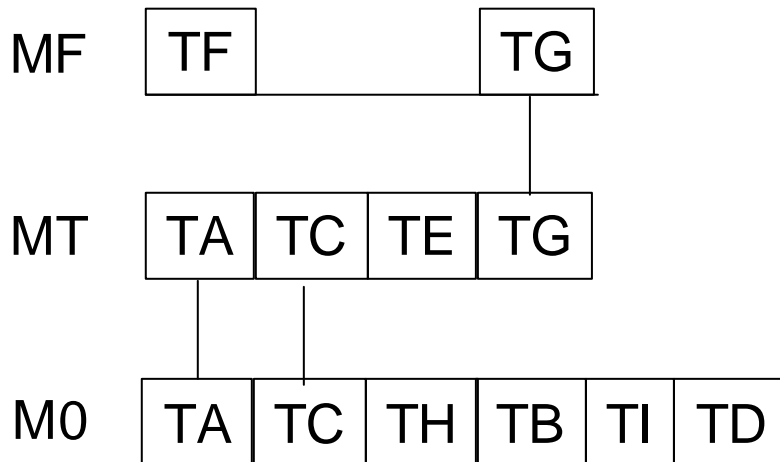
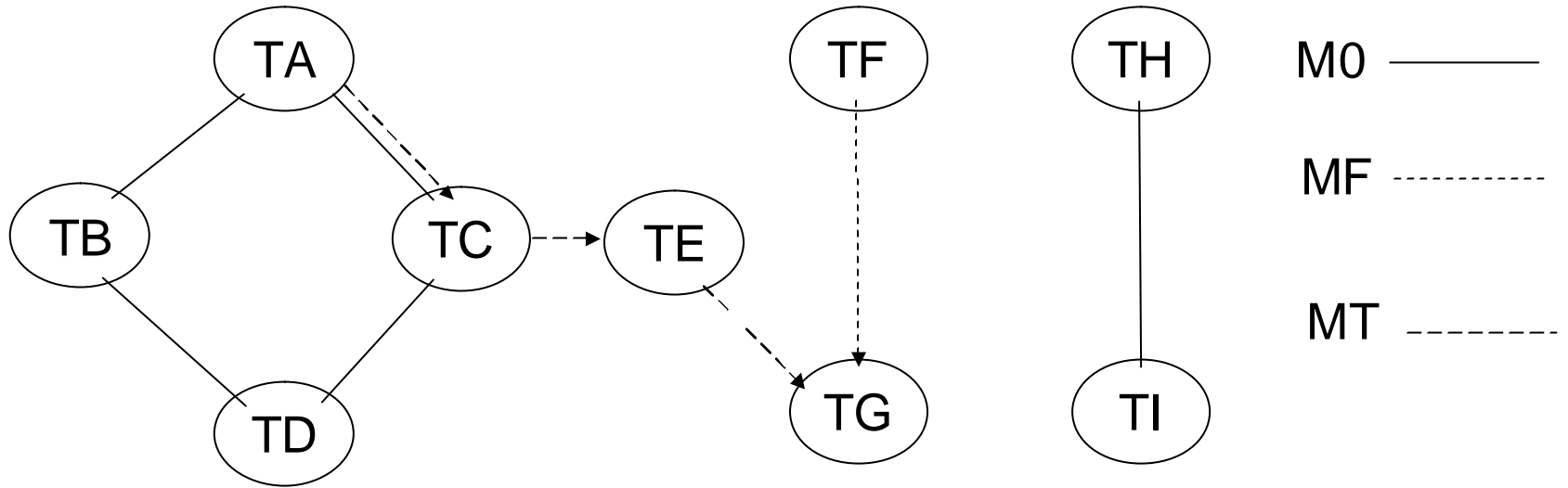
switch through requirement

enforce consistency by scheduling tasks at same times in all modes

then, when a task is executing which is also in another, can probably switch immediately (still need to check consistency precedence etc.)

how?

- apply (single-mode) selection strategy to all modes simultaneously
- predictable
- can specify and guarantee transition deadlines
- simple runtime handling

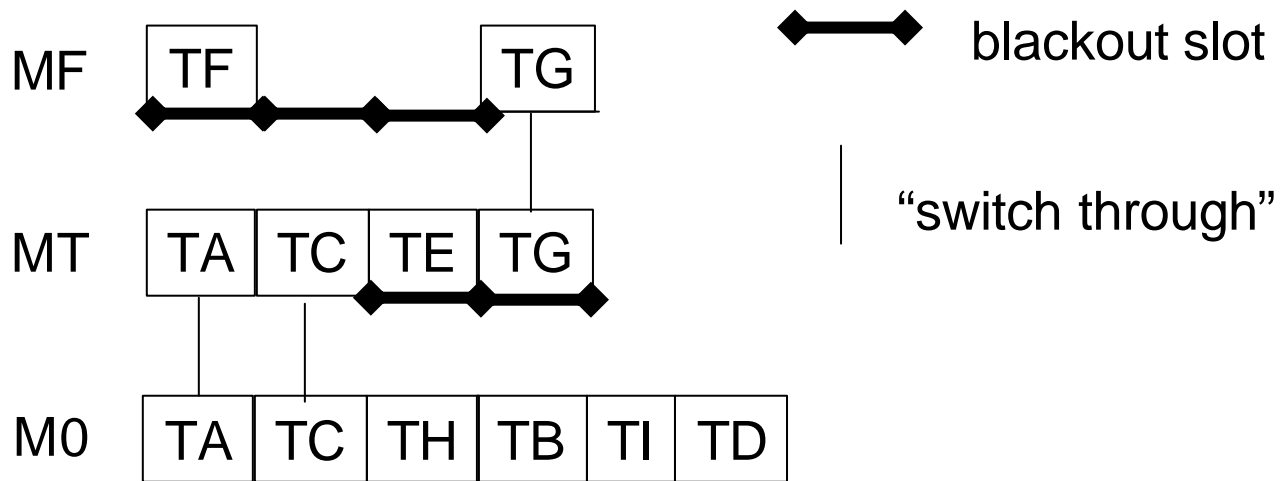


“switch through”

Runtime handling - II

How can I feasible switch schedules?

- check all requirements before switch - intractable
- resolve all that during schedule construction
- efficient representation in runtime dispatching
- *black out slots*
 - flag at each slot in destination mode
 - when set, switch not feasible
 - wait till next slot without blackout
 - else switch right away
- very memory and time efficient!!!



mode change methods for offline schedules presented

- constructs schedules for modes and transitions
- “switches” between scheduling tables in specified, feasible way
- given time, schedule, and mode change request
⇒ known sequence of activities to execute transition
- slot level determinism

Off-line Scheduling - Methods and Assumptions

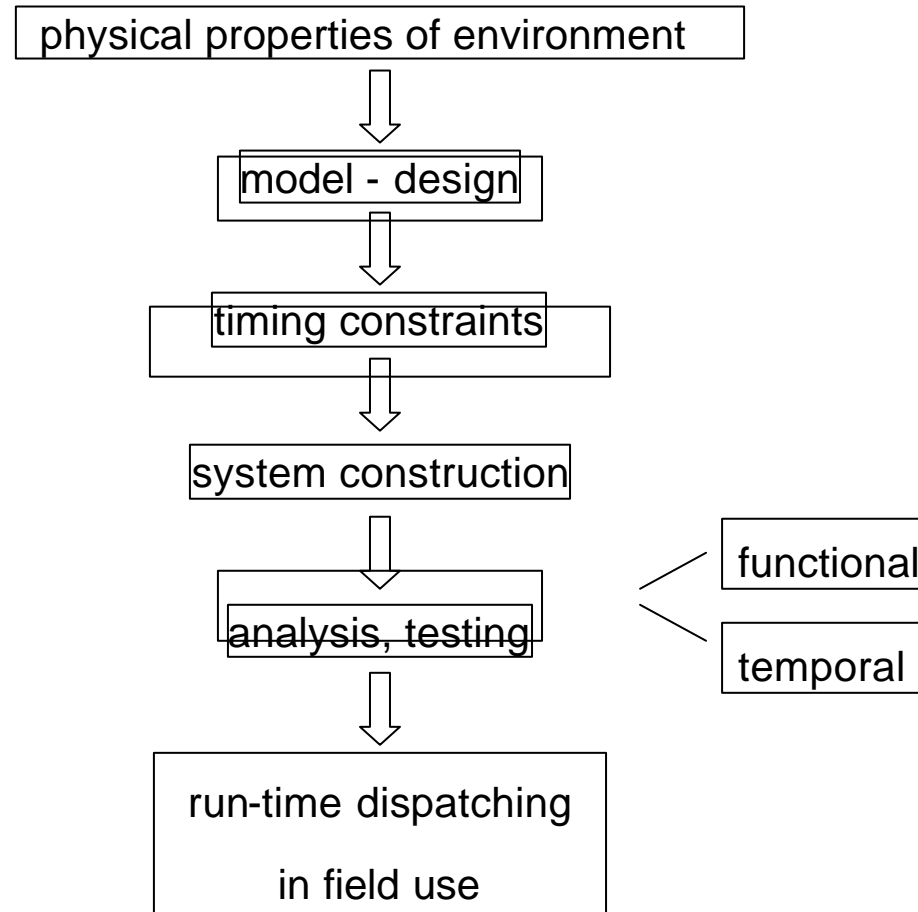
Real-Time Systems - ESSES

Gerhard Fohler 2003

Mälardalen University, Sweden

gerhard.fohler@mdh.se

Real-time scheduling - making the right decisions to guarantee time



Who is doing the scheduling? And when?

Run-time dispatcher controls *which activities* are performed at *which time*. It controls access to the CPU by *tasks*.

Part of *real-time kernel*.

- Keeps track of the system state, e.g., time, resource accesses, book keeping information, e.g., priorities, deadlines.
- Tasks execute until completion or may be interrupted: *non-preemptive* or *preemptive*.

Non-preemptive dispatching is in general simpler:

- only one task (and stack etc.) active at a time.
- resource access - contention resolved

- Run-time dispatching is performed according to a set of rules.
- Off-line analysis and testing has to ensure that the provided rules for the run-time dispatcher are correct:
 - when the dispatcher takes scheduling decisions according to the given rules, all timing constraints are kept.
 - ***off-line guarantees***

How long?

- standard OS schedulers work on strategies without guarantees
 - handle “task transition graph” waiting - ready - executing...
 - select one out of the ready tasks to execute
 - perhaps prevent deadlocks etc.
 - go on until shutdown or system lock/crash, e.g., windows
- off-line guarantees: before, for entire *mission lifetime*
 - minutes
 - hours, days, more
 - need to guarantee every one of them
 - combinatorial explosion

shorten analyzed lifetime

- analyze only single, selected part of lifetime
 - worst case proofs
 - need to ensure assume worst case is worst case
 - restrict complete freedom of task parameters
 - periods
- analyze repeating patterns during lifetime
 - typically periods
 - if harmonic, enough to analyze for duration of longest period
 - if not, *least common multiple LCM* of all involved periods
 - can be large
 - execute repeatedly

Guarantees

- System designer selects *scheduling strategy* and *algorithm*
Constructs a set of rules for the *run-time dispatcher* from specification and timing constraints. These rules range from complete schedules to priority strategies, etc.
- During *analysis/testing*, the designer determines, whether the rules provided will guarantee the temporal behavior, if applied by the run-time dispatcher.
If no rules can be found or testing gives a negative result, a redesign has to be done.
- Depending on whether these rules determine most scheduling decision before run-time or or leave part of the decisions to the run-time system, the scheduling is called *offline (pre run-time, static)* or *online (run-time, dynamic)*.

Pre run-time vs. run-time scheduling

Pre run-time scheduling constructs complete schedules that are feasible before the system is used in-field.

This is a *proof-by-construction* of feasibility.

Run-time dispatching only executes the decision, does not take any by itself.

- ☺ Very simple for run-time system, e.g., list or table lookup.
- ☹ Inflexible, can only handle fully specified events and tasks, requires *complete* knowledge.

Run-time scheduling constructs a set of rules for run-time dispatching and a *proof (schedulability test)* of feasibility when the rules are kept, before the system is used.

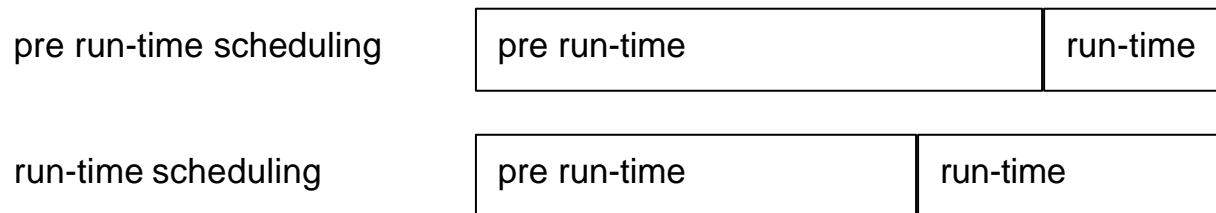
Run-time dispatching can take decisions on its own, as long as rules are kept.

- ☺ Flexible, can handle only partially known events and tasks.
- ☹ High cost at run-time (book keeping, calculations)
Difficult to predict *exact* behavior at run-time.

Run-time scheduling can provide more flexibility, but

no magic:

What is not exactly known before run-time cannot be guaranteed then, independent of the used scheduling strategy. Only events for which a task has been specified, i.e., code is available, can be handled.



work

- Run-time data structures and handling can be engineering problem, e.g., priority inheritance - paper by Victor Yodaiken
- Micro kernel with system threads, e.g., message handling tricky with run-time scheduling

Recently, algorithms have been presented to *integrate pre run-time and run-time scheduling – slot shifting*.
Benefits from pre run-time, but more flexibility.
→ lecture “integrated offline – online”

How to schedule within LCM?

- *Cyclic scheduling*
 - tasks in period classes
 - schedule tasks within classes
 - group task class schedules
 - ...until all tasks scheduled
- easy to handle, historically popular

very different from offline scheduling!

less powerful, more restrictive, etc

often mixed up

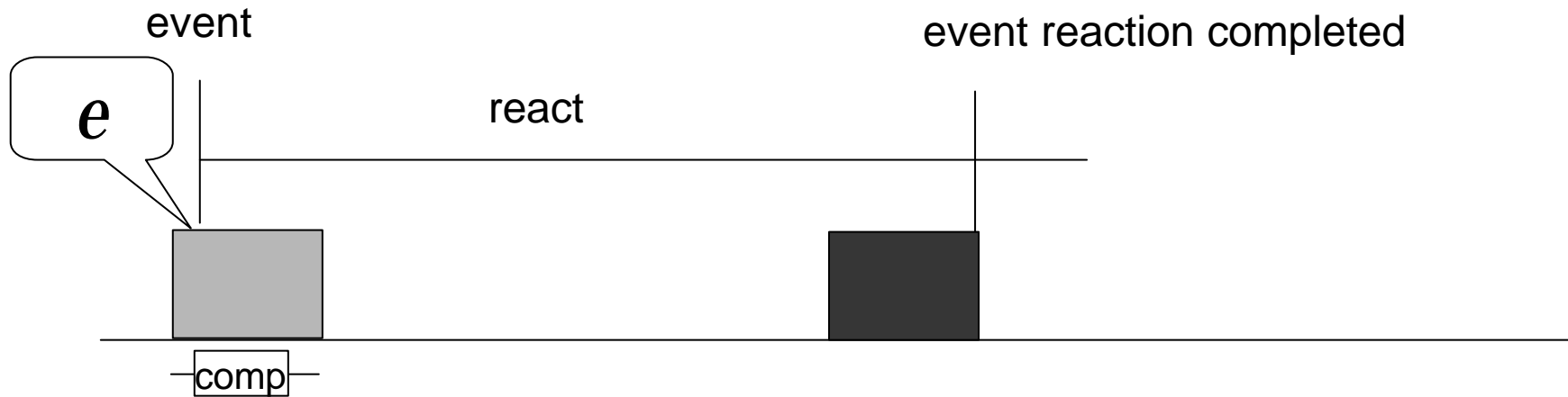
- *off-line scheduling*
static, pre run-time
 - construct schedule of length LCM
 - apply smart method
 - fulfill all constraints
 - not limited to “period concatenation”

Off-line Schedule Construction

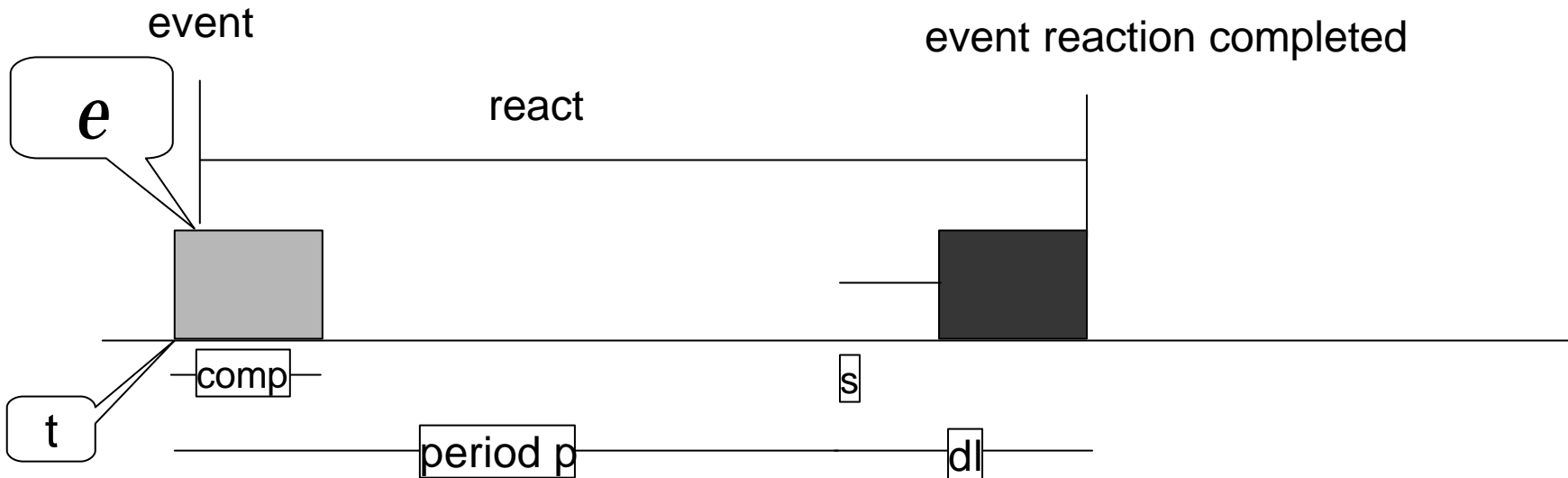
- time triggered
- totally pre-planned
- global time base
- cars, airplanes
- periodic “world”
- some say all “real tasks of real applications” are periodic
- true for some applications
- generally not!

Making a periodic world

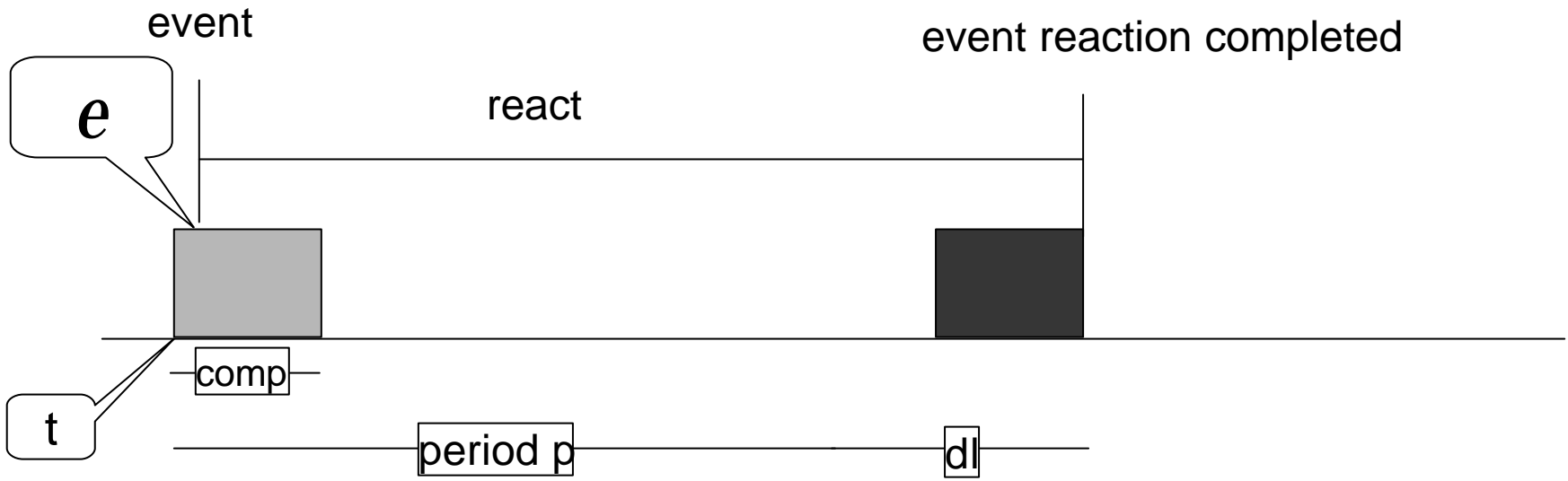
- “naturally periodic”, e.g., control, sampling
- aperiodic tasks, i.e., without any restriction on arrival
no way
- sporadics
transform into *pseudo periodic tasks*
assumptions about *events*
 - maximum rate of change, *minimum inter arrival interval, mint*
 - maximum delay of reaction, *react*
 - computation time, *comp*
- determine period and deadline
- have to ensure that
 1. reaction is not late
 2. no event missed



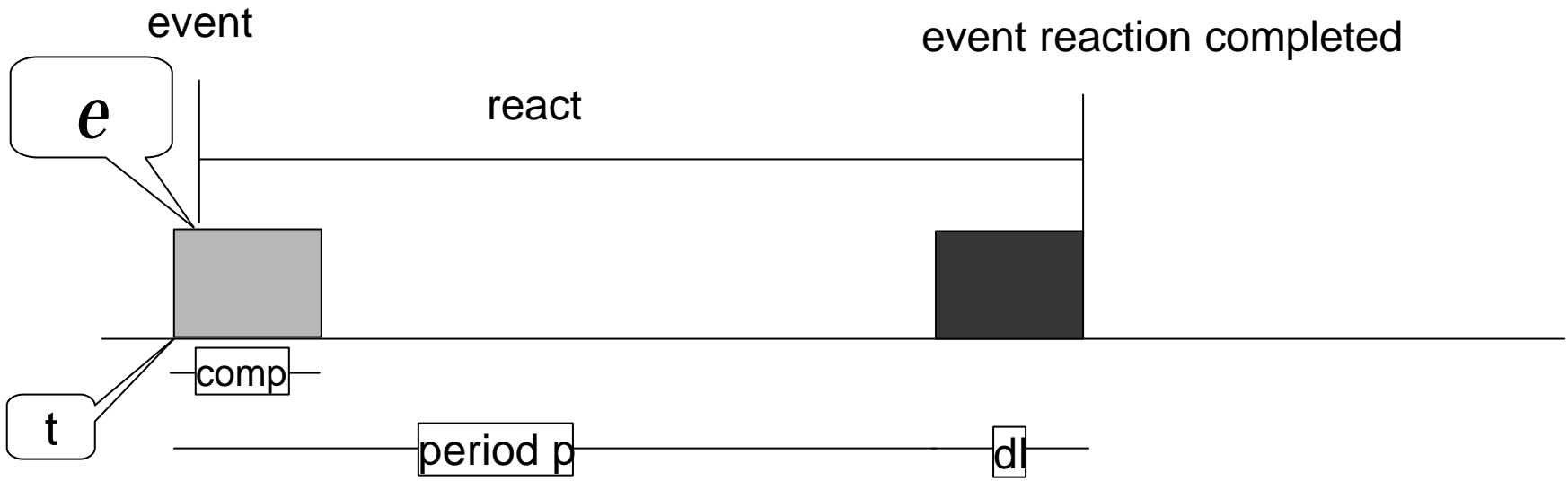
- worst case:
event happens right after task start - misses data just by e
event gets reacted by task only at next instance invocation



- deadline
 $dl = comp + s, s \geq 0$
- next instance completes no later than *react* after event
 - event starts at $t + e$
 - reaction finishes at $t + p + dl$
 - $t + p + dl - t - e \leq react$
 - $p + dl \leq react + e$ or $p + comp + s \leq react + e$



- maximum value for p - not react too late
 $p < \text{react} + e - dl$ or $p < \text{react} + e - \text{comp} - s$
- maximum value for p - not miss event
 $p < \text{mint}$



- assume $d_l = \text{comp}$; $s = 0$

$$p < \begin{cases} \text{react} - \text{comp} + e \\ \text{mint} \end{cases}$$

- Utilization:

$$U_0 = \frac{comp}{p} = \frac{comp}{react - comp + e}$$

- assume $dl = comp + s$; $s > 0$

$$U_s = \frac{comp}{p} = \frac{comp}{react - comp - s + e}$$

- $U_0 < U_s$!

- period and deadline dependent on each other
- tradeoff
 - large period:
 - low utilization demand
 - tight deadline - schedulability problems
 - small period:
 - relaxed deadline
 - high utilization demand
- change for individual instances
e.g., collision, relax deadline
- *flexible timing constraints* new project

- if events are *rare*, but urgent when they occur transformation inefficient, high utilization demands

e.g.,

mint=1000*comp; react=2*comp:

$p < \text{react} + e - \text{comp} = \text{comp} + e$

$$U = \frac{\text{comp}}{\text{comp} + e} \approx 1$$

- monopolization of CPU
- actual need to handle event without pseudo periodic transformation

$$U = \frac{\text{comp}}{1000 * \text{comp}} = 0.001$$

Why use it?

- number of - particular - critical application have periodic nature
- predictable behavior - know exactly what is going on
- testing, certification much easier
- simple fault-tolerance, replica determinism
- receiver based error detection
- non temporal constraints, e.g., cost
- explicit flow control, synchronization
- “proof by construction”
- very simple dispatching
- micro kernel synchronization of system threads
- high resource utilization

Off-line Scheduling Methods

What do we want to achieve?

- we want to find solutions
 - NP hard in more than trivial cases
 - can take very long time
 - have to optimize search to find solutions fast
- but
- once we find solution, we are done
 - likely that first try will not work, maybe solution does not exist
 - what if we don't find one/does not exist?
 - total time spent in schedule design:
time of not (finding * #failures) + (1*time of finding)
→ not finding at least as important as finding

we need

- algorithm for
 - fast detection of no solution/not finding
 - fast finding of feasible solution
- strategy to
 - select tradeoffs
 - choose time spent
 - allow for detection of why no solution found (difficult)
 - good redesign for next schedule attempt
- designer support

most current algorithms concentrate on finding solution only

Directions

How to construct a schedule?

- simple solution: use online scheduling, e.g., EDF
 - still better than online - can backtrack or redesign
 - better utilization because resource conflicts are known, don't need to assume worst case
 - testing
 - etc.
- search
 - popular
 - easy to change constraints
 - easy algorithm
 - problems with feedback problem - source in search tree

- genetic algorithms
e.g., simulated annealing
 - simple
 - does not get stuck easily with hard sub problems
 - can handle large task sets
 - difficulties with complex constraints
 - good for allocation of tasks to nodes in distributed system
- “by hand”
 - sometimes really fully by hand
 - with support
 - resolve difficult parts by hands
 - extend existing schedules
 - place some tasks by hand

- safety critical automotive application
 - specification of tasks (in place A)
 - scheduling (in place B)
 - transfer of schedule to chips by engineer (in place C)
 - “don’t like these tasks here, they should be separated”
 - engineering practice
 - cannot be scheduled, because cannot be expressed
- intelligent scheduling editor
 - display schedule
 - allow engineer to modify
 - provide info about constraints
 - allow rescheduling of selected tasks
 - current project - SALSART toolsuit
 - distributed cooperative schedule design

- incremental scheduling
 - want to modify existing schedule
 - upgrades
 - new versions
 - etc.
 - existing schedule trusted, tested, certified - spent high effort
 - rescheduling - completely new schedule
 - efforts again
 - better to keep existing schedule as much as possible
 - select “unmovable tasks”
 - interactive graphical tool with scheduling support
 - research - tool to be implemented

- networked based
 - distributed system
 - nodes under control of different suppliers
 - not knowledge about internals of other nodes
 - neutral designer
 - schedules communication
 - distribution bandwidth
 - specifies timing constraints (“windows”) to nodes
 - distributed tool - web based (possible SALSART application)
- also non-cooperative scheduling
 - auctioning of time

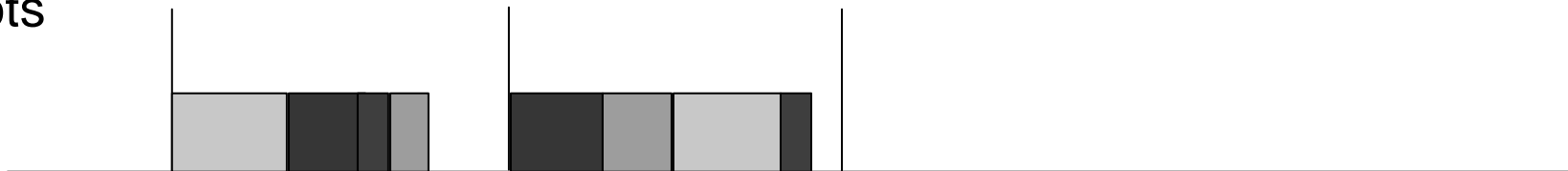
Off-line Scheduling and the Real World

- Many algorithms assume tasks, messages, slots, constant operating system overhead
- real-world demands
 - interrupts
 - threads, chains
 - micro kernel OS
 - system threads
 - task ensembles for tasks, e.g., message transmission
 - depending on scheduling and allocation
 - dynamic creation of threads
- do not fit into off-line schedule in straightforward way

Threads

- threads are shorter than granularity of slots
- better utilization of slots
- scheduling/dispatching happens not only at slot boundaries

slots

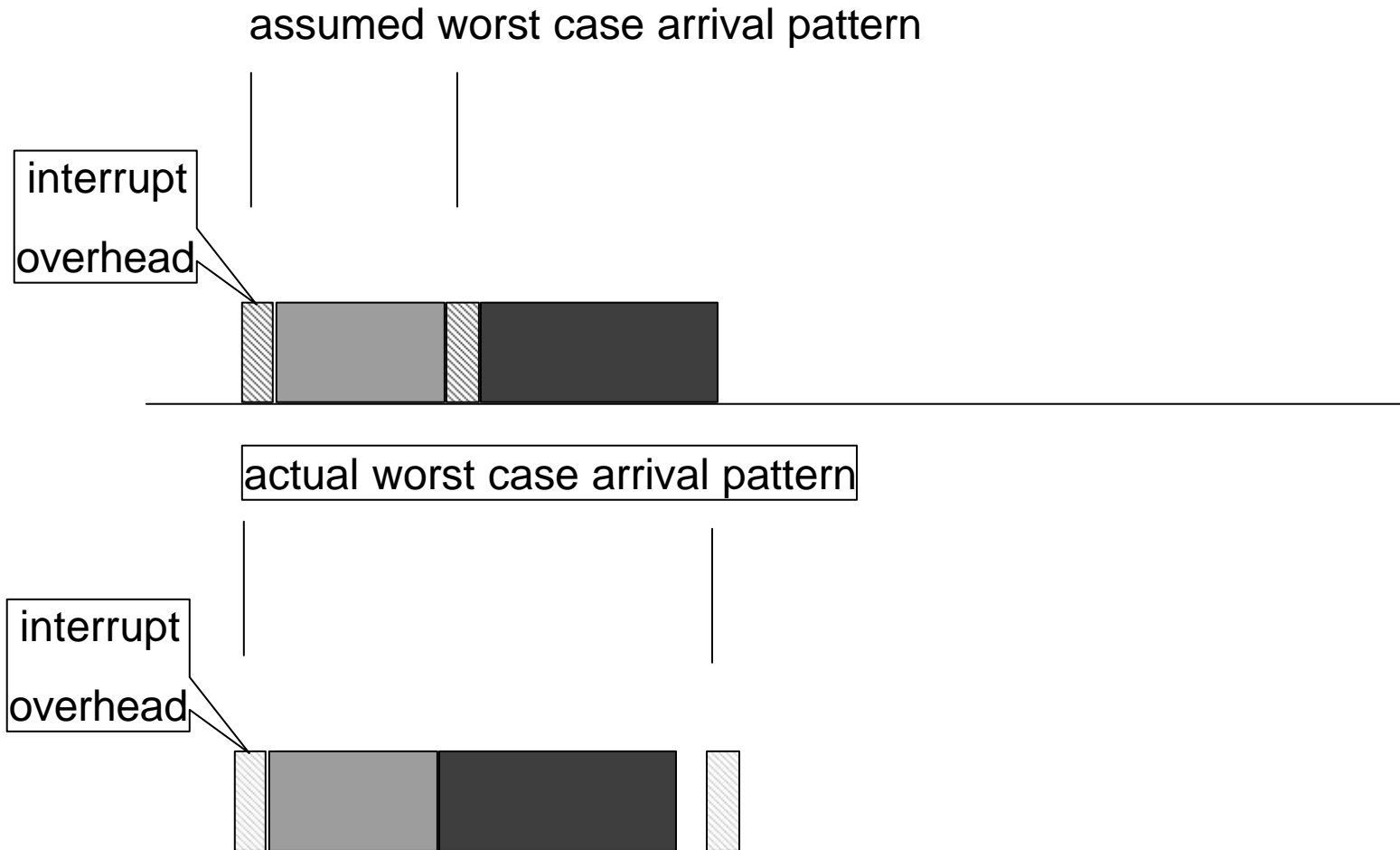


- scheduler needs to construct chains as well
- offline scheduler does “micro scheduling”, e.g., thread cumulating within slot
- backtracking, heuristic etc only at slot boundaries
- not optimal, but tractable

Interrupts

- interrupts have to be considered
- cannot
 - ignore them - too much time demand
 - handle them as tasks/threads -
too high overhead, too long response times
 - have to account for in analysis during schedule construction
 - minimum inter arrival time - maximum overhead
- naïve approach
 - assume each task can be hit by a worst case arrival of interrupts
 - ala exact analysis
 - very high overhead

- if task is shorter than minimum inter arrival time
interrupt overhead is considered too often for two consecutive tasks



- sophisticated analysis algorithms
taking into account successors, precedence relations, etc.
- used for analysis only and consideration during schedule construction
- online scheduled without further provisions

Off-line Scheduling - Search

precedence graph structure well suited for

(heuristic) search through search tree

- nodes represent (partial) schedule
- edges represent scheduling decisions
- heuristic function used to guide search through search tree

search strategies examples for distributed systems

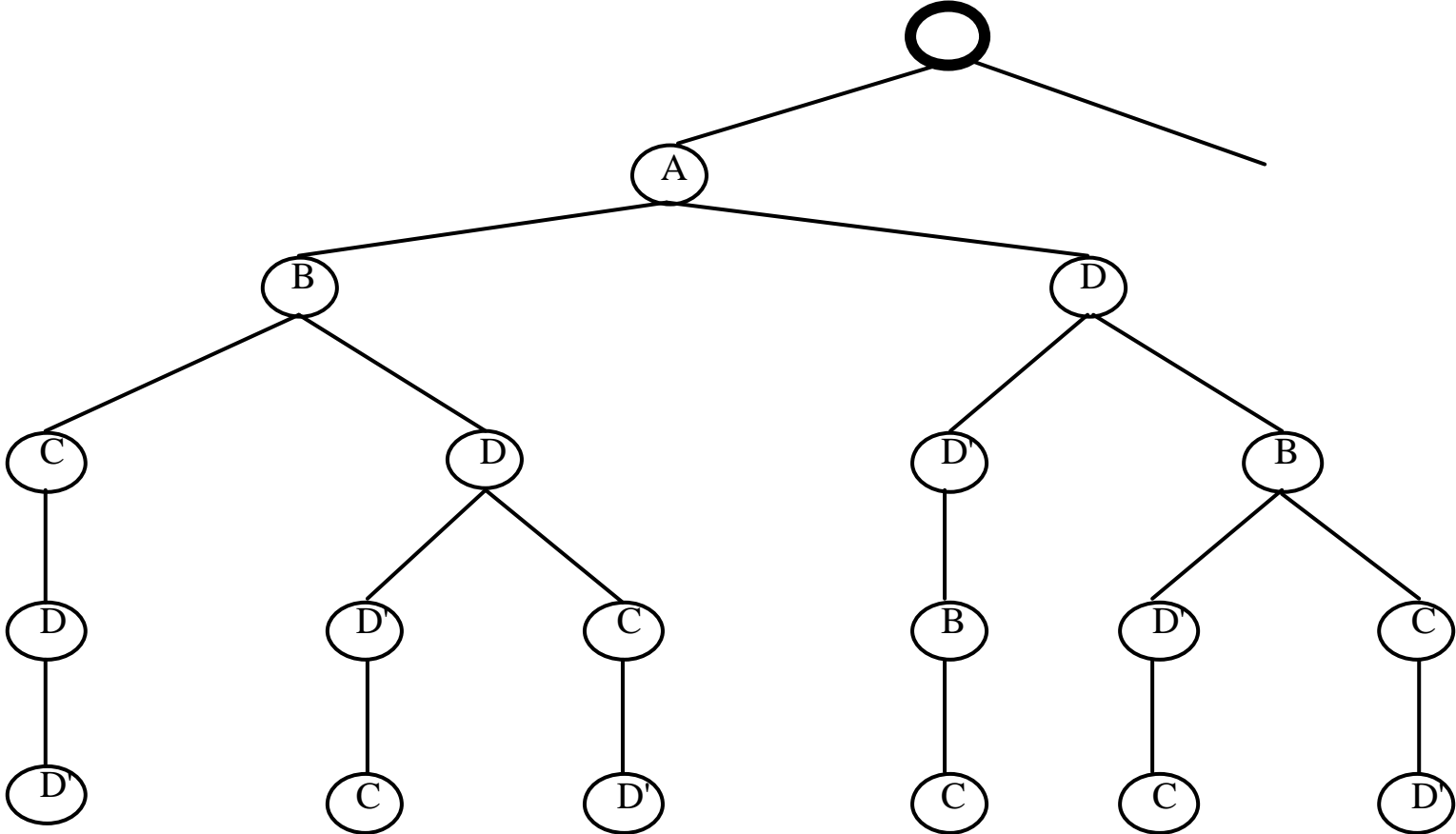
- A*, IDA*: Fohler 1989, 1991
- branch-and-bound: Ramamritham 1991
- “meta”, two stage branch-and-bound for pipelining
Fohler, Ramamritham 1997
- resulting schedule is a set of schedules for each node in the distributed system

Example Taskmodel for Pre Run-time Scheduling

- Precedence graphs
- period, starttime, deadline for entire precedence graph (end-to-end)
- release time and deadline for selected tasks
- Precedence constraints with communication (synchronized data flow) or without (synchronization only)
- preemptive tasks
- simple tasks (black boxes): read input - compute - write output
- communication time over bus bounded
- *slots* dispatcher runs with granularity, creating slots

- precedence graphs have different periods:
 - different number of instances in schedule
 - schedule length: least common multiple, lcm , of all periods
 - each precedence graph with period P_i has lcm / P_i instances in schedule
 - construct graph with correct number of instances
- **comprehensive graph**
(only for deadlines < periods)
- generate search tree
- traverse it for solution

Search tree



- each path in the search from the root represents a (partial) schedule
e.g., the second one to the left: ABCDD
e.g., the rightmost: ADBCD
- *branching factor*: number of edges from node
determines size of search tree
- *non preemption factor*: minimum size of “chunk of execution time”
determines size of search tree

Off-line Scheduling Strategies

- how to minimize the overall time to find schedule
- *search parameters*
 - determine size of expanded search tree
 - small tree:
 - easy solution can be found fast; but lower chances
 - no solution found is detected fast
 - larger search tree:
 - more time spent to find solution
 - long time spent to detect no solution
 - can be set by designer
- allows to start with small tree (easy solution fast) and increase as desired and tolerated

Analysis (simplified answers)

success ratio: how many solutions found in number of searches

- start depth does not influence success ratio
- larger branching factor (BF) increases success ratio
flattens out fast
- cost for finding solution
 - with minimum start depth higher
 - with larger branching factor higher
- cost for no solution
 - with minimum start depth higher
 - increase with BF, higher at lows of non preemption

Conclusions

- start with high start depth
- search with small BF first, don't increase too much
- use high non preemption factor, lower not too much

A*, IDA* Search

- developed by Korf 1984, derivative of A*, Nilson 1982
- heuristic search strategy
- uses heuristic function to guide search

```
ITERATION( )
{
while(DEEPEN(rootnode)not done)
{
threshold=threshold +min_exceed;
min_exceed = infinity;
}
}
```

```

DEEPEN(node)
{apply(node); // apply sched decision, update data
if(feasible(node) == true)
    { if(solution_found(node)) done;

successors= create all successors of node;
// collect tasks, message ready, create sched decisions
calculate f(n) for all nodes in successors;
best_nodes = sort all nodes in succ. by f(n)

for(i=0;i<BRANCHINGFACTOR; i++)
    {if((f(best_nodes[i] < threshold)
        DEEPEN(best_nodes[i];
    else if(f(best_nodes[i] - threshold < min_exceed)
        min_exceed = f(best_nodes[i] - threshold;
    }
} // if feasible
}

```

- task data accessed very often
- elaborate data structures
 - on purpose redundancy
 - areas instead of pointers
- IDA* linear with search depth in memory need
 - search tree represented in area
 - size known at program start

Heuristic Function

- Search tree can be very large
a complete search will take too long
- select “promising” paths in the search tree, e.g., by use of a heuristic function
- some heuristic search strategies, e.g., A^* , explicitly handle heuristic functions and provide guarantees for finding solutions based on their quality
- ad hoc heuristic functions, e.g., next deadline first, can be used as well, but don’t provide guarantees for solutions
- $f(n) = g(n) + h(n)$
 - $g(n)$ real cost so far
 - $h(n)$ estimated cost for rest

- example heuristic function *TUR* - time until response
 - sum of execution times of remaining tasks
 - distributed precedence graph - tricky problem
 - sum of remaining communication times estimation
 - idle times
0
- tradeoff
 - very elaborate heuristic function finds solution fast
 - but is expensive to calculate - invoked often
e.g., feasible schedule is good heuristic!
- problems if solution does not exist - expands large parts of search tree

RM vs. EDF

Giorgio Buttazzo

Department of Computer Science
University of Pavia
E-mail: buttazzo@unipv.it

Basic results

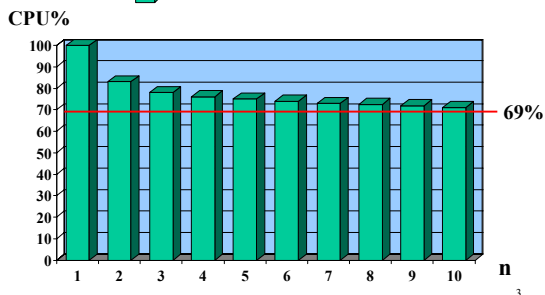
Assumptions: $\left\{ \begin{array}{l} \text{Independent tasks} \\ \Phi_i = 0 \quad D_i = T_i \end{array} \right.$

A set of n periodic tasks can be feasibly scheduled

$\left\{ \begin{array}{l} \text{under RM} \quad \text{if} \quad \sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \\ \text{under EDF} \quad \text{if and only if} \quad \sum_{i=1}^n \frac{C_i}{T_i} \leq 1 \end{array} \right.$

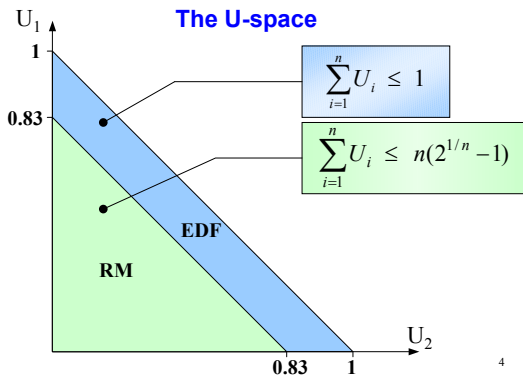
Schedulability bound

 RM  EDF



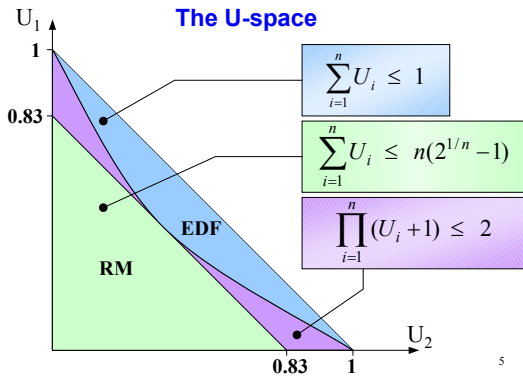
Schedulability region

The U-space



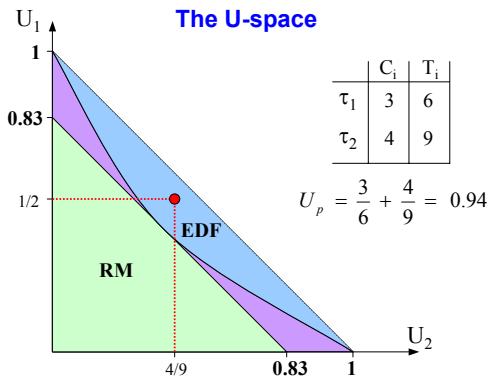
Schedulability region

The U-space

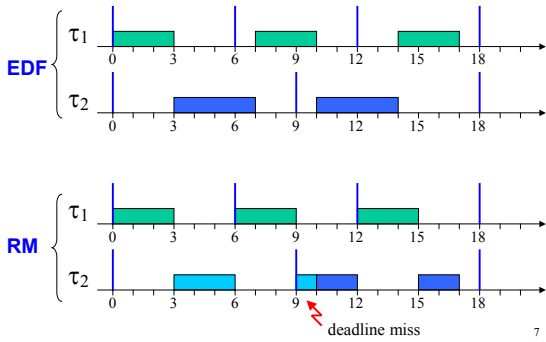


Schedulability region

The U-space



Schedule



Questions

- If EDF is more efficient than RM, why commercial RT systems are still based on RM?
- Why is RM preferred to EDF?
- What are the limitations of EDF that prevent its use?

After 30 years of work on scheduling, there are still a lot of misconceptions

8

Typical misconceptions

They tend to favor RM more than EDF:

- RM is easier to implement and analyze;
- RM introduces less runtime overhead;
- RM is more predictable during overloads;
- RM causes less jitter.

9

Objectives of this work

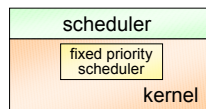
1. Address the misconceptions
2. Compare the algs w.r.t. different metrics
 - ⇒ Implementation complexity
 - ⇒ Runtime overhead
 - ⇒ Schedulability analysis
 - ⇒ Robustness during overloads
 - ⇒ Jitter
 - ⇒ Aperiodic task handling

10

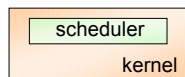
Implementation complexity

We have to distinguish two cases:

1. Implementation on top of a fixed priority kernel



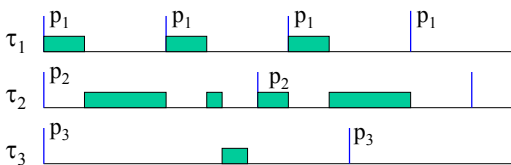
2. Implementation from scratch



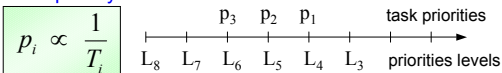
11

On top of a FP-kernel

RM is straightforward to implement



fixed priority

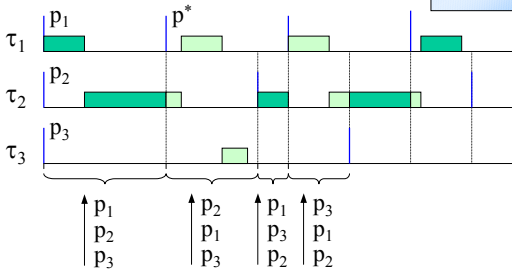


12

On top of a FP-kernel

EDF requires dynamic priorities

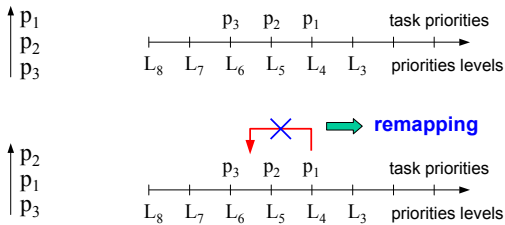
$$p_i \propto \frac{1}{d_i}$$



13

On top of a FP-kernel

EDF sometimes remapping is required:



14

As a basic kernel mechanism

Both RM and EDF require the same complexity for queue management:

Task Control Block

| |
|----------------------|
| periodic / aperiodic |
| criticality |
| WCET |
| Minimum Inter. Time |
| Relative Deadline |
| Absolute Deadline |
| Utilization Factor |
| ⋮ |
| ⋮ |

Under EDF

the absolute deadline must be updated at each job release:

$$d_i = r_i + T_i$$

(negligible overhead)

15

Existing EDF kernels

- **SPRING** (Stankovic-Ramamritham 87)
- **YARTOS** (Jeffay 92)
- **HARTIK** (Buttazzo-Lamastra-Lipari 93)
- **SHARK** (Gai-Buttazzo 99)
- **MARTE-OS** (Gonzalez 01)
- **ERIKA** (Gai 01)
- **MCU-OS** (Carlini-Buttazzo 01)

16

Runtime overhead

Two different types of overhead are considered:

1. Overhead for job release

⇒ EDF has more than RM, because the absolute deadline must be updated at each job activation

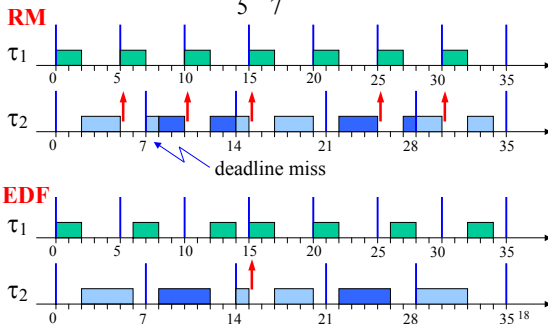
2. Overhead for context switch

⇒ RM has more than EDF because of the higher number of preemptions

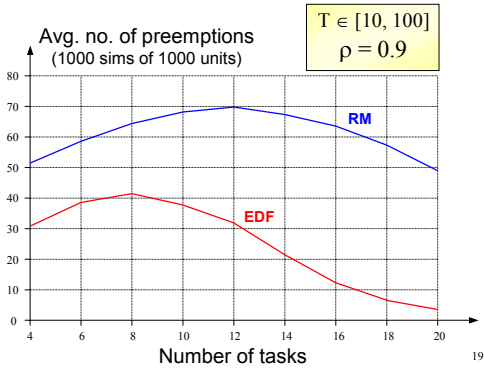
17

Preemptions

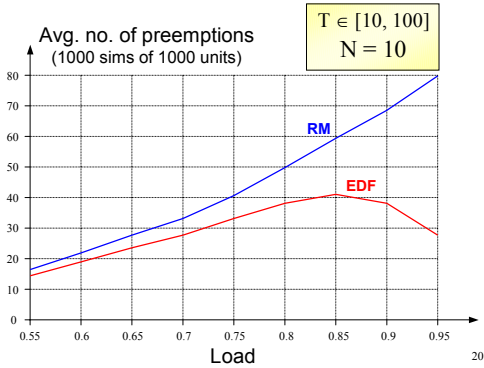
$$U = \frac{2}{5} + \frac{4}{7} \cong 0.97$$



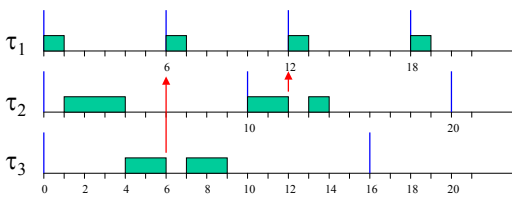
Preemptions



Preemptions

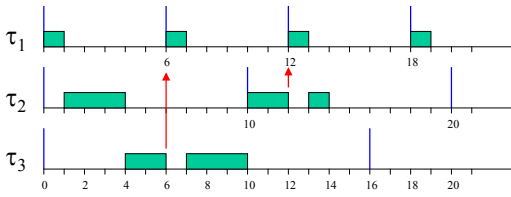


Example with RM



Under RM, preemptions increase as computation times increase

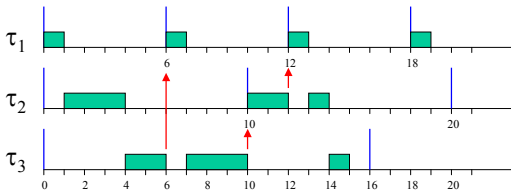
Example with RM



Under **RM**, preemptions increase as computation times increase

22

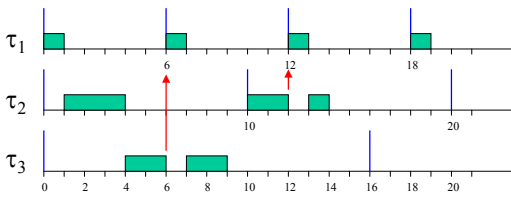
Example with RM



Under **RM**, preemptions increase as computation times increase

23

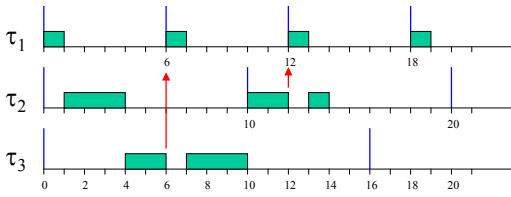
Example with EDF



Under **EDF**, preemptions may decrease as computation times increase

24

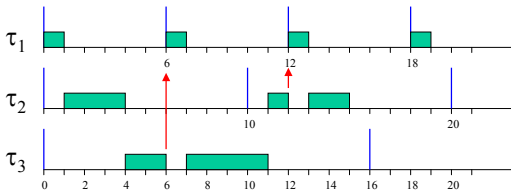
Example with EDF



Under EDF, preemptions may decrease as computation times increase

25

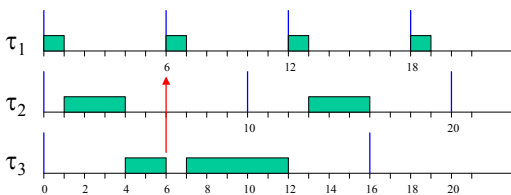
Example with EDF



Under EDF, preemptions may decrease as computation times increase

26

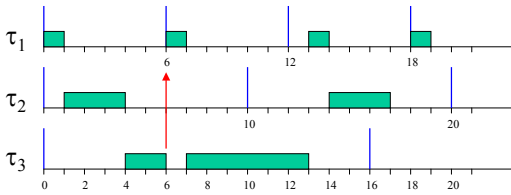
Example with EDF



Under EDF, preemptions may decrease as computation times increase

27

Example with EDF



Under **EDF**, preemptions may decrease as computation times increase

28

Schedulability Analysis

| | $D_i = T_i$ | $D_i \leq T_i$ |
|------------|--|---|
| RM | <p><i>Suff.:</i> polynomial $O(n)$</p> <p>LL: $\sum U_i \leq n(2^{1/n} - 1)$</p> <p>HB: $\prod(U_i + 1) \leq 2$</p> <p><i>Exact</i> pseudo-polynomial RTA</p> | <p><i>pseudo-polynomial</i> Response Time Analysis</p> <p>$\forall i \quad R_i \leq D_i$</p> <p>$R_i = C_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$</p> |
| EDF | <p><i>polynomial:</i> $O(n)$</p> <p>$\sum U_i \leq 1$</p> | <p><i>pseudo-polynomial</i> Processor Demand Analysis</p> <p>$\forall L > 0, \quad g(0, L) \leq L$</p> |

29

RM: harmonic periods

Harmonic task sets are schedulable by RM
if and only if $U \leq 1$.

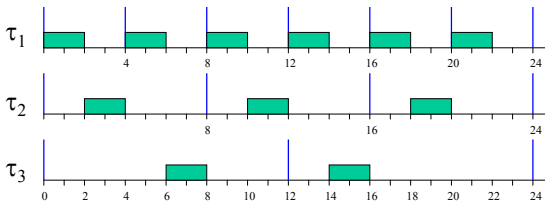
A set of tasks is **harmonic** if **every** pair of periods are in harmonic relation.

A common misconception

The RM schedulability bound is 1 if every period is multiple of the shortest period.

30

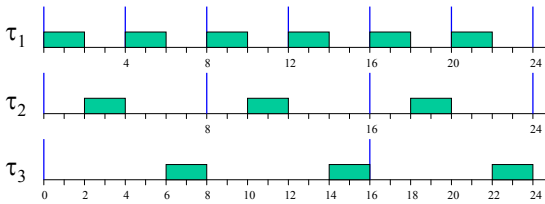
Non harmonic periods



$$U = \frac{2}{4} + \frac{2}{8} + \frac{2}{12} \cong 0.917$$

Any increase in the C_i 's makes the system unschedulable

Harmonic task set



$$U = \frac{2}{4} + \frac{2}{8} + \frac{4}{16} = 1$$

32

Robustness under overloads

Two situations are considered:

1. Permanent overload

⇒ This occurs when $U > 1$

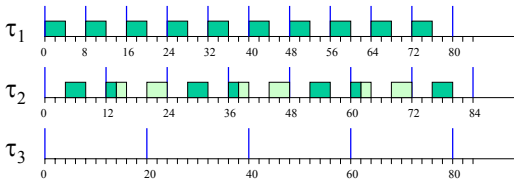
2. Transient overload

⇒ This occurs when some job executes more than expected

33

RM under permanent overload

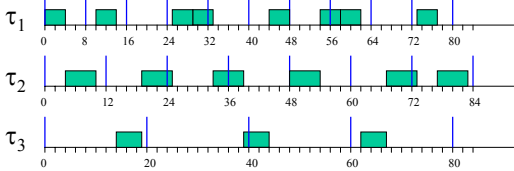
$$U = \frac{4}{8} + \frac{6}{12} + \frac{5}{20} = 1.25$$



- High priority tasks execute at the proper rate
- Low priority tasks are completely blocked

EDF under permanent overload

$$U = \frac{4}{8} + \frac{6}{12} + \frac{5}{20} = 1.25$$



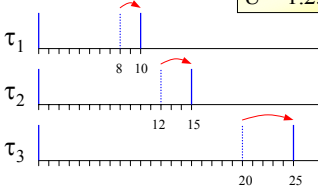
- All tasks execute at a slower rate
- No task is blocked

EDF is predictable in overloads

Theorem (Cervin '03)

If $U > 1$, EDF executes tasks with an average period $T'_i = T_i U$.

$$U = 1.25$$



| | T_i | T'_i |
|----------|-------|--------|
| τ_1 | 8 | 10 |
| τ_2 | 12 | 15 |
| τ_3 | 20 | 25 |

Big misconceptions

EDF is not predictable during overloads because we don't know which tasks are going to miss their deadlines.

RM is predictable during overloads because the tasks that miss their deadlines are low priority tasks.

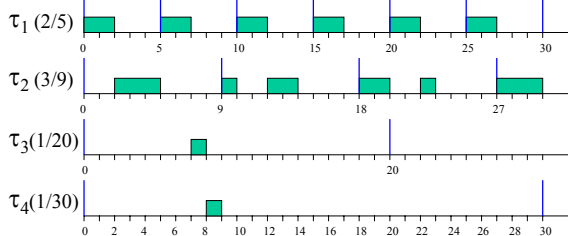


We now show that this is not true

37

RM during transient overruns

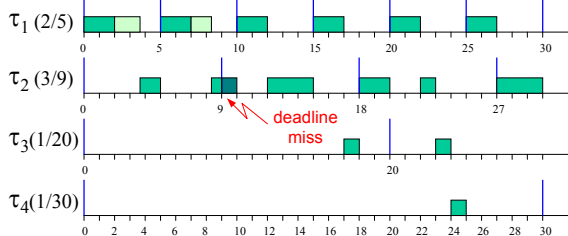
$$U_{avg} = 0.817 \quad C_{l_{avg}} = 2, \quad C_{l_{max}} = 4$$



38

RM during transient overruns

$$U_{avg} = 0.817 \quad C_{l_{avg}} = 2, \quad C_{l_{max}} = 4$$



Who is missing its deadline is not the lowest priority task

Jitter

Jitter for an event

The maximum time variation in the occurrence of a particular event in two consecutive jobs of a task.

Another misconception

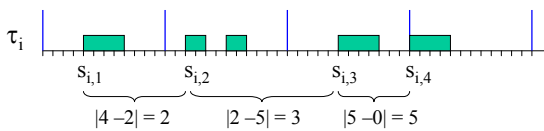
RM reduces jitter during task execution more than EDF

40

Types of Jitter

Start Time Jitter

$$STJ_i = \max_k |s_{i,k} - s_{i,k+1}|$$



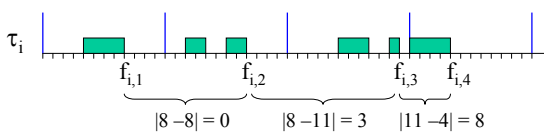
$$STJ_i = 5$$

41

Types of Jitter

Response Time Jitter

$$RTJ_i = \max_k |R_{i,k} - R_{i,k+1}|$$



$$RTJ_i = 8$$

42

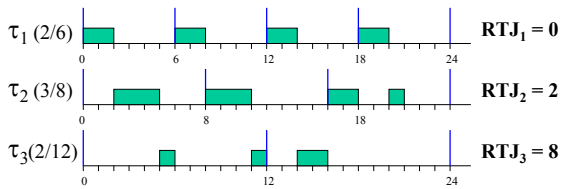
Effects of Jitter

- In some control application, jitter is tolerated by the inertial nature of the system
- In some other applications, jitter can cause instability or jerky behavior

We compare the performance of RM and EDF in terms of RTJ

43

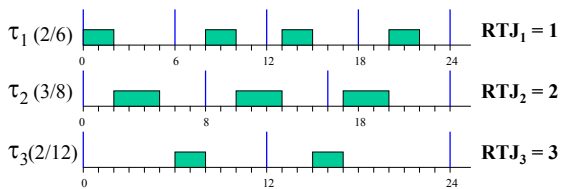
Jitter under RM



τ_3 experiences a very high jitter

44

Jitter under EDF

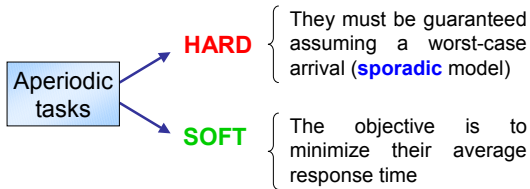


For a little increase of RTJ_1 ,
 RTJ_3 is decreased a lot

45

Aperiodic task handling

Most RT applications require the execution of **periodic** (time driven) and **aperiodic** (event driven) activities.



Important results (1)

Theorem 1 (Tia-Liu-Shankar '96)

Under fixed priority scheduling it is not possible to minimize the response time of every aperiodic job.

Theorem 2 (Tia-Liu-Shankar '96)

Under fixed priority scheduling no on-line algorithm can minimize the average response time aperiodic requests.

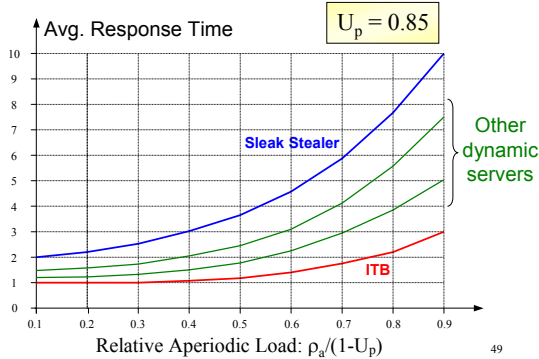
Important results (2)

Under dynamic priority scheduling there are optimal algorithms that minimize the response time of aperiodic jobs.

Improved Total Bandwidth Server (ITB) (Buttazzo-Sensini '97)

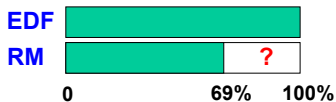
It minimizes response times by scheduling each aperiodic job with the minimum deadline that preserves the periodic guarantee.

Aperiodic responsiveness



Conclusions (1)

1. RM and EDF have same implem. complexity
A small additional overhead is needed in EDF to update the absolute deadline at each job release
2. Runtime overhead is smaller in EDF
Due to the smaller number of context switches
3. EDF achieves full processor utilization, whereas RM only guarantees 69%



Conclusions (2)

4. EDF is simpler to analyze if $D_i = T_i$
This is important for reducing admission control overhead in small embedded systems
5. EDF is more flexible in overload conditions
EDF automatically expand periods, whereas RM causes a complete block of low priority tasks
6. EDF is fair in reducing jitter, whereas RM only reduces the jitter of the highest priority tasks
7. EDF is more efficient than RM for handling aperiodic tasks

Conclusions (3)

The only real advantage of RM is that it can be easily implemented on top of fixed priority kernels.

Challenge

Develop EDF kernels to exploit all the advantages of dynamic scheduling without paying additional overhead.

Overload management

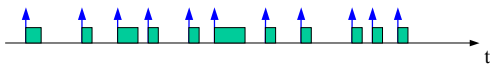
Outline

- **Definitions:** load, overload, overrun
- **Overload handling methods:**
 - ⇒ Admission control
 - ⇒ Resource Reservation
 - ⇒ Imprecise Computation
 - ⇒ Job Skipping
 - ⇒ Elastic Scheduling

2

Load definitions

- **For non real-time systems:**



$$\rho = \lambda \bar{C}$$

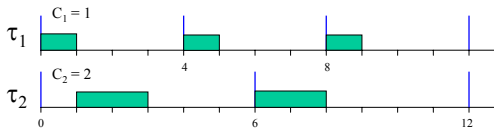
- ρ = load
- λ = average arrival rate
- \bar{C} = average execution time

3

Load definitions

- For hard real-time periodic tasks:

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



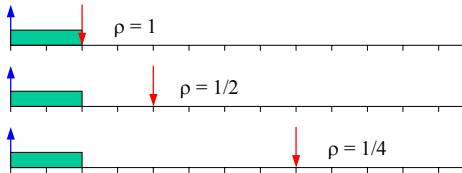
$$U = \frac{1}{4} + \frac{1}{3} = \frac{7}{12}$$

4

Load definitions

- For real-time aperiodic tasks:

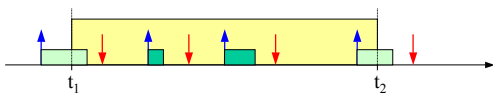
The load also depends on the deadline



5

Computing the load

In general, the load in an interval is computed using the processor demand in that interval:

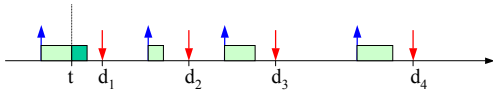


$$\rho = \frac{g(t_1, t_2)}{t_2 - t_1} = \frac{\sum_{r_i \geq t_1, d_i \leq t_2} C_i}{t_2 - t_1}$$

6

Instantaneous load $\rho(t)$

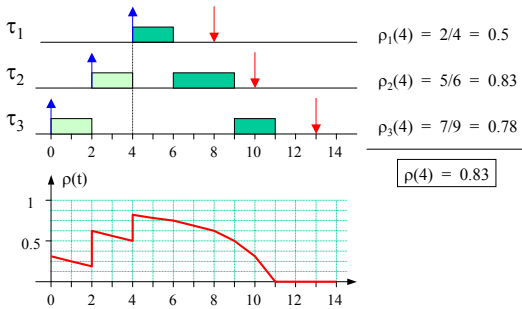
Maximum processor demand from the current time and the deadlines of all active tasks.



$$\rho(t) = \max_k \frac{g(t, d_k)}{d_k - t} = \max_k \frac{\sum_{r_i \leq t, d_i \leq d_k} c_i(t)}{d_k - t}$$

7

Example



8

Transient vs. permanent overload conditions

Transient overload: $\rho_{avg} < 1, \rho_{max} > 1$

Possible causes

- ⇒ Arrival of aperiodic activities
- ⇒ Exceptions raised by the kernel
- ⇒ Malfunctioning of input devices
- ⇒ Task with variable execution
- ⇒ Sporadic overruns

9

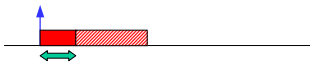
Types of overruns

- A task is said to be in overrun if the time demanded for execution exceeds the expected value according to which the task has been guaranteed.

- There are two types of overrun:

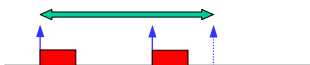
Execution overrun

A job executes more than expected



Activation overrun

A job arrives before the time it is expected



10

Transient vs. permanent overload conditions

Permanent overload: $\rho_{avg} > 1$

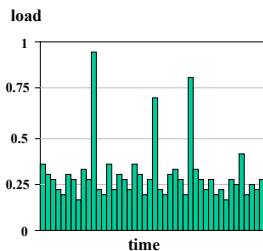
Possible causes

- ⇒ Activation of a new periodic task
- ⇒ Increase in the task frequencies
- ⇒ Increase in the task quality (execution times)
- ⇒ Changes in the environment
- ⇒ Bad system design

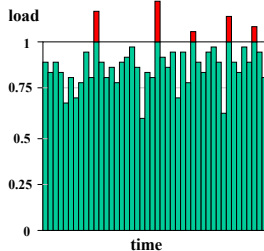
11

Examples of load

System designed under worst-case assumptions



System designed under average-case assumptions



12

Predictability vs. efficiency

Pessimistic assumptions lead to

- high predictability
- low efficiency → **high cost**
only justified for critical systems

Average-case design leads to

- high efficiency
- low predictability → necessary to handle and tolerate overloads

13

Overload management

Overload can be handled using different approaches:

- **Value-based scheduling**
 - tasks are assigned values and executed accordingly
- **Resource Reservation**
 - Resources are reserved to tasks and cannot be used
- **Admission control**
 - least importance tasks are rejected
 - important tasks receive full service
- **Performance degradation**
 - all tasks are executed
 - but with reduced requirements

14

Value-based scheduling

- If $\rho > 1$, no all tasks can finish within their deadline.
- To avoid domino effects, the load is reduced by rejecting the least important tasks.
- To do that, the system must be able to handle tasks with both timing constraints and importance values.

15

Deadline and Value

- Under RM and EDF, the value of a task is implicitly encoded in its period or deadline.
- However, in a chemical plant controller, a task reading the steam temperature every 10 seconds is more important than a task which updates the clock icon every second.

16

How to assign values

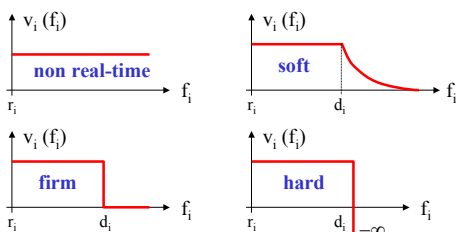
A task τ_i can be assigned a value v_i according to different criteria. Those most common are:

| | |
|-----------------|--------------------|
| $v_i = V_i$ | arbitrary constant |
| $v_i = C_i$ | computation time |
| $v_i = V_i/C_i$ | value density |

17

Value as a function of time

In a real-time system, the value of a task depends on its completion time and criticality:



18

Performance evaluation

- The performance of a scheduling algorithm A on a task set T can be evaluated through its *Cumulative Value*:

$$\Gamma_A(\mathbb{T}) = \sum_{i=1}^n v_i(f_i)$$

- Note that: $\Gamma_A(\mathbb{T}) < \Gamma_{\max}(\mathbb{T}) = \sum_{i=1}^n V_i$

19

Optimality under overloads

$$\Gamma^*(\mathbb{T}) = \max_A \Gamma_A(\mathbb{T})$$

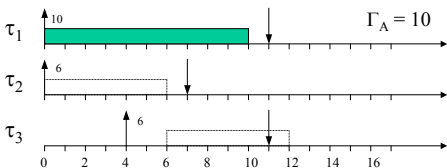
The performance of an algorithm can be evaluated with respect to Γ^* .

In overload conditions, there are no optimal **on-line** algorithms able to guarantee a cumulative value equal to Γ^* .

20

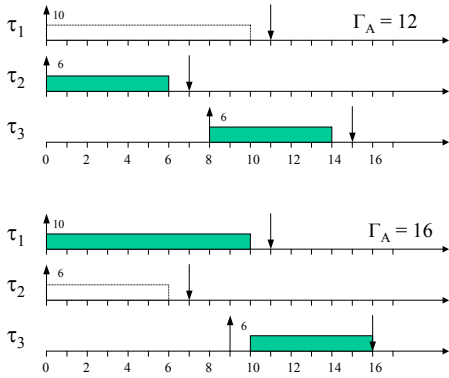
Proof (assume: $V_i = C_i$)

To maximize Γ_A we should know the future.



If at time $t = 0$ r_3 is not known, we cannot select the task that maximizes the cumulative value.

21



22

Competitive Factor

- Let Γ^* the maximum cumulative value achievable by an optimal clairvoyant algorithm.
- An algorithm A has a competitive factor φ_A , if it is guaranteed that, for any task set, it achieves:

$$\Gamma_A \geq \varphi_A \Gamma^*$$

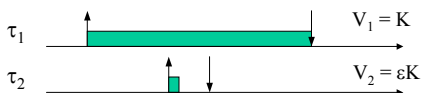
- Hence, $\varphi_A \in [0,1]$ and can be computed as:

$$\varphi_A = \min_{\tau} \frac{\Gamma_A(\tau)}{\Gamma^*(\tau)}$$

23

Competitive factor of EDF

- It is easy to show that $\varphi_{\text{EDF}} = 0$:



In such a situation, $\Gamma_{\text{EDF}} = V_2$ and $\Gamma^* = V_1$,
 hence $\Gamma_{\text{EDF}} / \Gamma^* = V_2 / V_1 \rightarrow 0$ for $V_2 \gg V_1$

24

A theoretical upper bound

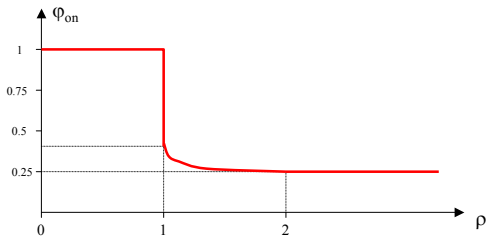
[Baruah et al., 91]

If $\rho > 2$ and task value is proportional to computation time, then no on-line algorithm can have a competitive factor greater than 0.25.

$$\text{That is: } \max_A \varphi_A \leq 0.25$$

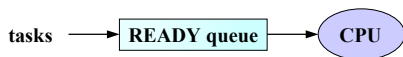
25

In general, the upper bound of the competitive factor is a function of the load and varies as follows:



26

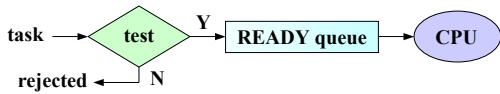
Best-effort scheduling



- Tasks are always accepted in the system.
- Performance is controlled through a suitable (value-based) priority assignment.
- **Problem:** domino effect.

27

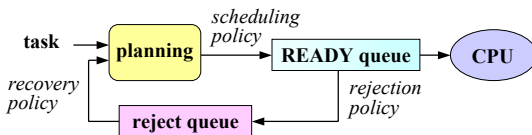
Admission control



- Every task is subject to an acceptance test which keeps the load ≤ 1 .
- It prevents domino effects, but does not take values into account.
- Low efficiency due to the worst-case guarantee (tasks may be unnecessarily rejected).

28

Robust scheduling



- Task scheduling and task rejection are controlled by two separate policies.
- Tasks are scheduled by deadline, rejected by value.
- In case of early completions, rejected tasks can be recovered by a reclaiming mechanism.

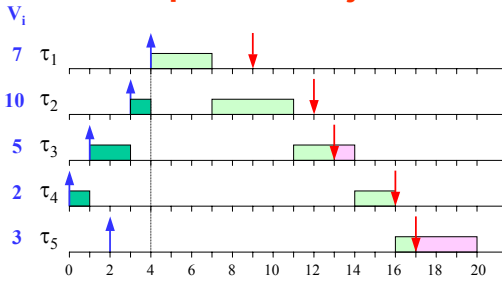
29

Robust EDF

- **Scheduling Policy** \Rightarrow **EDF**
- **Rejection policy**
when an overload is detected, reject the least value task which can bring the load below 1.
- **Recovery policy**
 - keep rejected tasks by decreasing values;
 - when there is enough spare time, re-accept the highest value task which is still feasible.

30

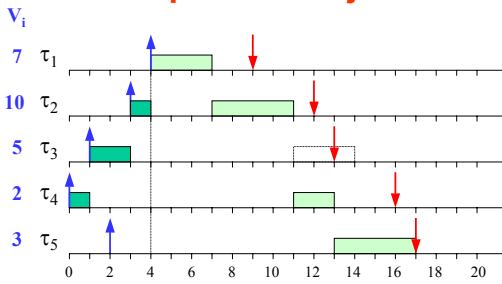
Example: task rejection



at time $t = 4 \Rightarrow \tau_3$ rejected

31

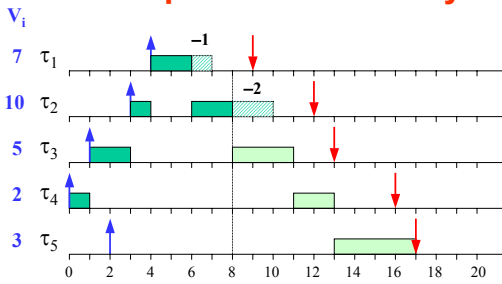
Example: task rejection



at time $t = 4 \Rightarrow \tau_3$ rejected

32

Example: task recovery



at time $t = 8 \Rightarrow \tau_3$ can be recovered

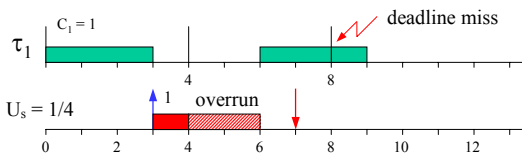
33

Resource Reservation

Handling sporadic overruns

Problems with overruns

- Without a budget management, there is no protection against execution overruns.
- If a job executes more than expected, hard tasks could miss their deadlines.



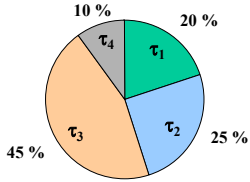
Solution: Temporal Isolation

- The execution of a task should not affect the guarantee performed on the other tasks.
- Each task τ_i receives a fraction U_i of the processor (its *bandwidth*) and behaves as it were executing alone on a slower processor of speed U_i .

Temporal isolation → { bandwidth reservation
bandwidth enforcement

Bandwidth reservation

- Ideally, each task should be assigned a given bandwidth and never demand more.



- However, tasks are subject to **overruns** or the reserved bandwidth can be **insufficient** for the task.

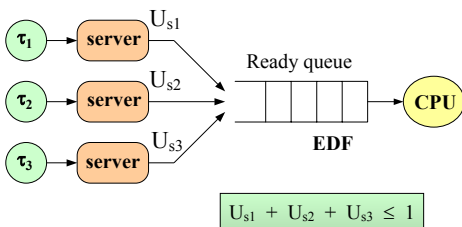
37

Bandwidth enforcement

- It is a mechanism needed for degrading the QoS when a task demands more than the reserved bandwidth.
- If a task executes more than expected, its priority should be decreased (i.e., its deadline postponed).
- When a task experiences an overrun, only that task is delayed, so that the guarantee performed on the other tasks is preserved.

38

Implementation



39

Constant Bandwidth Server (CBS)

- It assigns deadlines to tasks as the TBS, but keeps track of job executions through a budget mechanism.
- When the budget is exhausted it is immediately replenished, but the deadline is postponed to keep the demand constant.

40

CBS parameters

Given by the user

- Maximum budget: Q_s
- Server period: T_s

$$U_s = Q_s / T_s \text{ (server bandwidth)}$$

Maintained by the server

- Current budget: c_s (initialized to 0)
- Server deadline: d_s (initialized to 0)

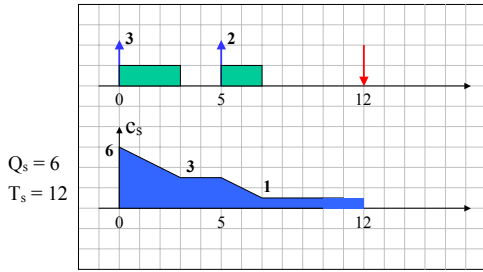
41

Basic CBS rules

- **Arrival of job J_k \Rightarrow assign d_s**
if $(r_k + c_s / U_s \leq d_s)$ then recycle d_s
else $\begin{cases} d_s = r_k + T_s \\ c_s = Q_s \end{cases}$
- **Budget exhausted \Rightarrow postpone d_s**
 $\begin{cases} d_s = d_s + T_s \\ c_s = Q_s \end{cases}$

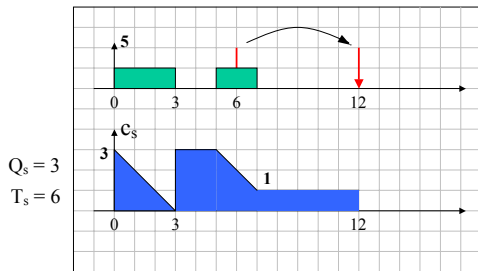
42

Deadline assignment



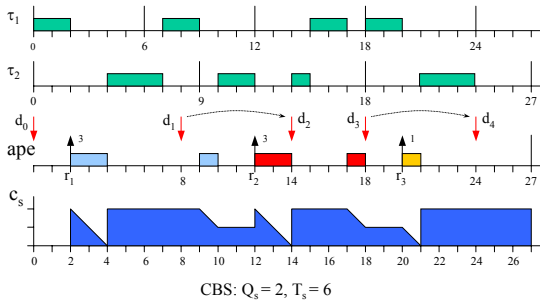
43

Budget exhausted



44

EDF + CBS schedule



45

CBS properties

- **Bandwidth Isolation**

If a task τ_i is served by a CBS with bandwidth U_s then, in any interval Δt , τ_i will never demand more than $U_s \Delta t$.

- **Hard schedulability**

A hard task $\tau_i (C_i, T_i)$ is schedulable by a CBS with $Q_s = C_i$ and $T_s = T_i$, iff τ_i is schedulable by EDF.

46

Remarks on the CBS

- It can be used as a safe server for handling aperiodic tasks under EDF.
- It can be used as a bandwidth reservation mechanism to achieve task isolation.
- It allows to guarantee a minimum performance to SOFT tasks, based on the assigned bandwidth.

47

Handling permanent overload

Performance Degradation

The load can be decreased not only by rejecting tasks, but also by reducing their performance requirements.

This can be done by:

- reducing precision of results
- skipping some jobs;
- relaxing timing constraints.

49

Reducing precision

In many applications, computation can be performed at different level of precision: the higher the precision, the longer the computation. Examples are:

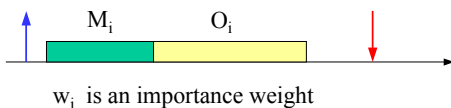
- binary search algorithms
- image processing and computer graphics
- neural learning

50

Imprecise computation

In this model, each task $\tau_i (C_i, D_i, w_i)$ is divided in two portions:

- a **mandatory** part: $\tau_i^m (M_i, D_i)$
- an **optional** part: $\tau_i^o (O_i, D_i)$



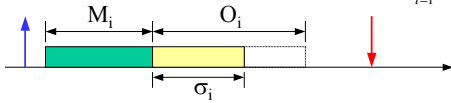
51

Imprecise computation

In this model, a schedule is said to be:

- **feasible**, if all mandatory parts complete in D_i
- **precise**, if also the optional parts are completed.

error: $\varepsilon_i = O_i - \sigma_i$ **average error:** $\varepsilon_a = \sum_{i=1}^n w_i \varepsilon_i$



GOAL: minimize the average error

52

Job skipping

Periodic load can also be reduced by skipping some jobs, once in a while.

Many systems tolerate skips, if they do not occur too often:

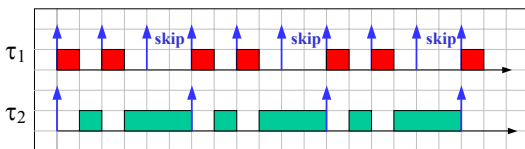
- multimedia systems (video reproduction)
- inertial systems (robots)
- monitoring systems (sporadic data loss)

53

Example

The system is overloaded, but tasks can be schedulable if τ_1 skips one instance every 3:

$$U_p = \frac{1}{2} + \frac{4}{6} = 1.17 > 1$$



54

FIRM task model

- Every job can either be executed within its deadline, or completely rejected (skipped).
- A percentage of task instances must be guaranteed off line to finish in time.
- Each task τ_i is described by (C_i, T_i, D_i, S_i) :
 S_i is the minimum number of jobs that must be executed between two consecutive skips.

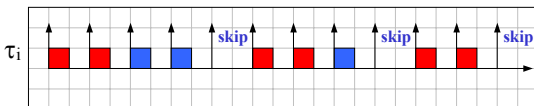
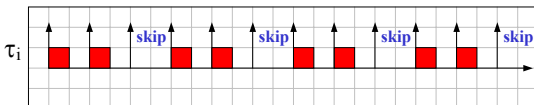
55

- Every instance can be **red** or **blue**:
 - **red** instances must finish within their deadline
 - **blue** instances can be aborted
- If a **blue** instance is aborted, the next S_i-1 instances must be **red**.
- If a **blue** instance is completed within its deadline, the next instance is still **blue**.
- The first S_i-1 instances of every task must be **red**.

56

Example

$C_i = 1$ $T_i = 2$ $D_i = 2$ $S_i = 3$



57

Equivalent utilization factor

$$U_p^* = \max_{L \geq 0} \left\{ \frac{\sum_{i=1}^n g_i(0, L)}{L} \right\}$$

$$g_i(0, L) = \left(\left\lfloor \frac{L}{T_i} \right\rfloor - \left\lfloor \frac{L}{T_i S_i} \right\rfloor \right) C_i$$

58

Schedulability Analysis

A sufficient condition

Theorem: A set of firm periodic tasks is schedulable if

$$U_p^* \leq 1$$

59

A necessary condition

Theorem: A set of firm periodic tasks is not schedulable if

$$\sum_{i=1}^n \frac{C_i (S_i - 1)}{T_i S_i} > 1$$

NOTE: the sum represents the utilization of the computation that must take place.

60

Bandwidth saving

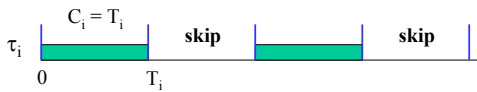
- In general, skipping jobs of periodic tasks causes a bandwidth saving:

$$\Delta U = U_p - U_p^*$$

- Such a bandwidth can be used for
 - improving aperiodic responsiveness (by increasing their reserved bandwidth);
 - accepting a larger number of periodic tasks.

61

Not always skips save bandwidth:



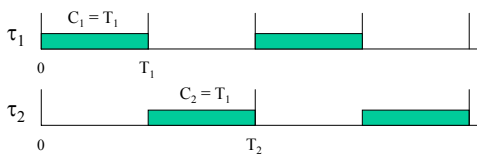
In this case: $U_p^* = 1$

In fact, for $L = T_i$ we have $g_i(0, L) = C_i = T_i$

Hence: $\frac{g_i(0, L)}{L} = \frac{T_i}{T_i} = 1$

62

However, notice that:



In this case we still have: $U_p^* = 1$

In fact: $g(0, T_1) = T_1$ e $g(0, T_2) = T_2$

Hence: $\frac{g(0, T_1)}{T_1} = \frac{g(0, T_2)}{T_2} = 1$

63

Relaxing timing constraints

- The idea is to reduce the load by increasing deadlines and/or periods.
- Each task must specify a range of values in which its period must be included.
- Periods are increased during overloads, and reduced when the overload is over.

64

Example

| task | C_i | T_{i0} | T_{\min} | T_{\max} |
|----------|-------|----------|------------|------------|
| τ_1 | 10 | 20 | 20 | 25 |
| τ_2 | 10 | 40 | 40 | 50 |
| τ_3 | 15 | 70 | 35 | 80 |

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{70} = 0.96$$

65

Load adaptation

If τ_4 arrives with: $C_4 = 5$, $T_4 = 30$ the system is not schedulable any more:

$$U_p = \frac{10}{20} + \frac{10}{40} + \frac{15}{70} + \frac{5}{30} = 1.13$$

However, there exists a feasible schedule within the specified ranges:

$$U_p = \frac{10}{23} + \frac{10}{50} + \frac{15}{80} + \frac{5}{30} = 0.99$$

66

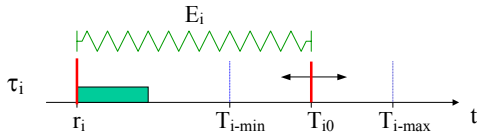
Elastic task model

- Tasks' utilizations are treated as elastic springs and can be changed by period variations.
- The resistance of a task to a period variation is controlled by an **elastic coefficient E_i** :
 - ⇒ the greater E_i the greater the elasticity

67

Elastic task model

- A periodic task τ_i is characterized by:
($C_i, T_{i0}, T_{i-min}, T_{i-max}, E_i$)
- The actual period $T_i \in [T_{i-min}, T_{i-max}]$



68

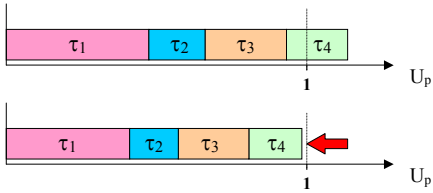
Special cases

- A task with $T_{min} = T_{max}$ is equivalent to a hard task.
- A task with $E_i = 0$ can intentionally change its period but does not allow the system to do that.

69

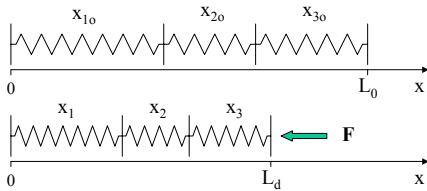
Compression algorithm

During overloads, utilizations must be compressed to bring the load below one.



70

The linear spring analogy



$$\begin{cases} F = k_1(x_{1o} - x_1) \\ F = k_2(x_{2o} - x_2) \\ F = k_3(x_{3o} - x_3) \end{cases} \quad \begin{cases} x_1 + x_2 + x_3 = L_d \\ x_{1o} + x_{2o} + x_{3o} = L_0 \end{cases}$$

71

Solution without constraints

Summing the equations, we have:

$$F \left(\frac{1}{k_1} + \frac{1}{k_2} + \frac{1}{k_3} \right) = (x_{1o} + x_{2o} + x_{3o}) - (x_1 + x_2 + x_3) = (L_0 - L_d)$$

That is:

$$F = \frac{(L_0 - L_d)}{\frac{1}{k_1} + \frac{1}{k_2} + \frac{1}{k_3}}$$

72

Solution without constraints

Substituting F in the equations, we have:

$$F = k_1(x_{1o} - x_1) = \frac{(L_0 - L_d)}{\frac{1}{k_1} + \frac{1}{k_2} + \frac{1}{k_3}}$$

That is:

$$x_1 = x_{1o} - (L_0 - L_d) \frac{1/k_1}{\frac{1}{k_1} + \frac{1}{k_2} + \frac{1}{k_3}}$$

73

Solution without constraints

$$x_i = x_{io} - (L_0 - L_d) \frac{K_{//}}{k_i} \quad K_{//} = \frac{1}{\sum_{i=1}^n \frac{1}{k_i}}$$

And defining: $E_i = 1/k_i$

$$x_i = x_{io} - (L_0 - L_d) \frac{E_i}{E_s} \quad E_s = \sum_{i=1}^n E_i$$

74

Period computation

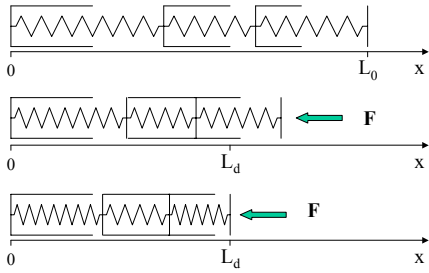
$$U_i = U_{io} - (U_0 - U_d) \frac{E_i}{E_s}$$

And then: $T_i = \frac{C_i}{U_i}$

75

Solution with constraints

Iterative solution:



76

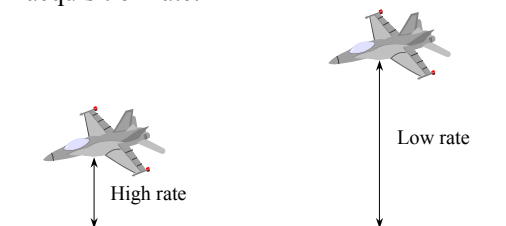
Other use of elastic tasks

- Increase frequencies to fully utilize the processor.
- Quickly find new period configurations during negotiation.
- On line period variations in control applications.

77

Examples: altimeter reading

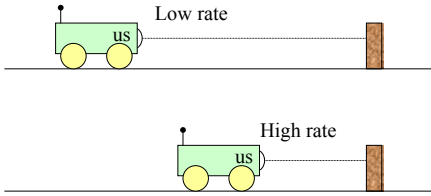
- The smaller the altitude, the higher the acquisition rate:



78

Obstacle avoidance

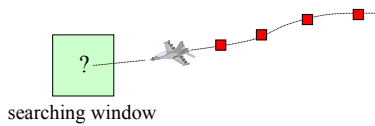
- The closer the obstacle, the higher the acquisition rate:



79

Visual tracking

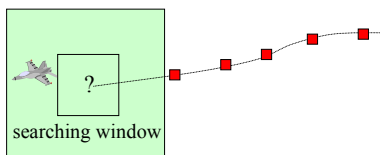
- The smaller the searching window, the higher the acquisition rate:



80

Visual tracking

- The smaller the searching window, the higher the acquisition rate:



81

Engine control

- Some tasks need to be activated at specific angles of the motor axis:
 - ⇒ the higher the speed, the higher the rate.
- Guaranteeing all the tasks at the maximum rate is not efficient or may not be possible.
- Other tasks may need to be downgraded when the engine is running at high speeds.

Dynamic Task Scheduling

Giorgio Buttazzo

Department of Computer Science
University of Pavia
E-mail: buttazzo@unipv.it

Course Outline

- Some terminology
- Basic results on dynamic scheduling
- Aperiodic task handling
- Dynamic scheduling under resources constraints
- Overload and QoS management techniques
- Comparison with fixed priority scheduling

2

Terminology

Task

is a piece of code that can be executed many times with different input data:

Each instance of a task (τ_i) is called a **job** ($\tau_{i,k}$)

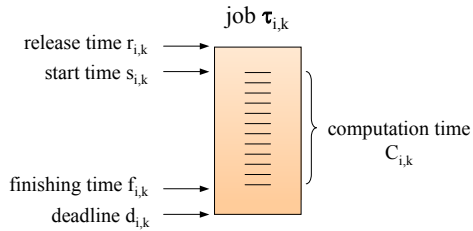
task τ_i



3

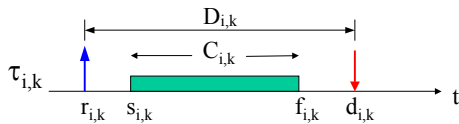
Job parameters

job $\tau_{i,k}$ is the k^{th} instance of task τ_i



4

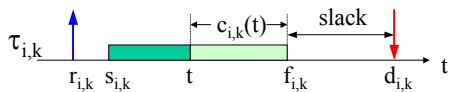
Job parameters



- $r_{i,k}$ release time (arrival time a_i)
- $s_{i,k}$ start time
- $C_{i,k}$ worst-case execution time (wcet)
- $d_{i,k}$ absolute deadline
- $D_{i,k}$ relative deadline
- $f_{i,k}$ finishing time

5

Other parameters



Residual wcet: $c_{i,k}(t)$ $c_{i,k}(r_{i,k}) = C_{i,k}$

Slack (or laxity): $d_{i,k} - t - c_{i,k}(t)$

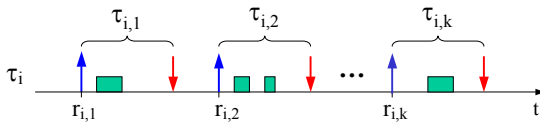
Lateness: $L_{i,k} = f_{i,k} - d_{i,k}$

Tardiness: $\max(0, -L_{i,k})$

6

Task model

A task τ_i is an infinite sequence of jobs $\tau_{i,k}$



7

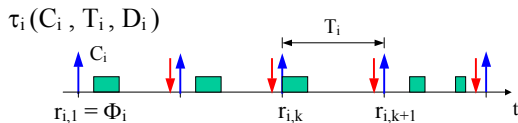
Activation modes

- **Time driven:** **periodic** tasks
the task is automatically activated by the kernel at regular intervals.
- **Event driven:** **aperiodic** tasks
the task is activated upon the arrival of an event or through an explicit call of the activation primitive.

8

Periodic task model

$$\begin{cases} r_{i1} = \Phi_i & \text{(task phase)} \\ r_{i,k+1} = r_{i,k} + T_i \end{cases}$$

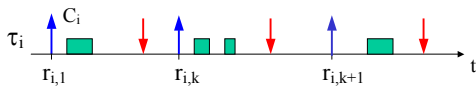


$$\begin{cases} r_{i,k} = \Phi_i + (k-1) T_i \\ d_{i,k} = r_{i,k} + D_i \end{cases} \quad \left(\begin{array}{l} \text{often} \\ \Phi_i = 0 \\ D_i = T_i \end{array} \right)$$

9

Aperiodic task model

- **Aperiodic:** $r_{i,k+1} > r_{i,k}$
- **Sporadic:** $r_{i,k+1} \geq r_{i,k} + T_i$



T_i = Minimum Interarrival Time

10

Algorithm taxonomy

- Preemptive vs. Non Preemptive
- Static vs. dynamic
- On line vs. Off line
- Optimal vs. Heuristic

11

Static vs. Dynamic

Static

scheduling decisions are taken based on **fixed parameters**, statically assigned to tasks before activation.

Dynamic

scheduling decisions are taken based on parameters that can **change with time**.

12

Off line vs. On line

Off line

all scheduling decisions are taken before task activation: the schedule is stored in a table (**table-driven scheduling**).

On line

scheduling decisions are taken at **run time** on the set of active tasks.

13

Optimal vs. Heuristic

Optimal

They generate a schedule that minimizes a cost function, defined based on an optimality criterion.

Heuristic

They generate a schedule according to a heuristic function that tries to satisfy an optimality criterion, but there is no guarantee of success.

14

Optimality criteria

- **Feasibility:** Find a feasible schedule if there exists one.
- Minimize the number of deadline miss
- Assign a value to each task, then minimize the system loss value

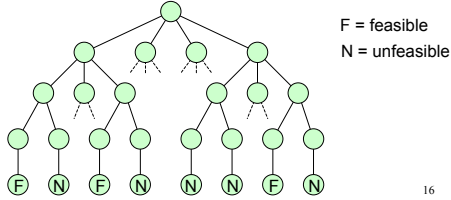
15

Example of heuristic algorithm

Spring kernel [Stankovic & Ramamritham 87]

The algorithm performs a tree search, where:

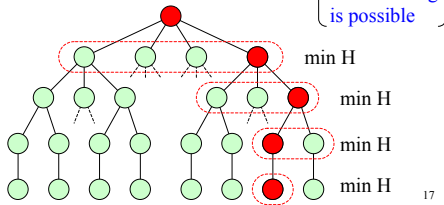
- The root node is an *empty schedule*
- Intermediate nodes are *partial schedules*
- Leaves are *complete schedules*



Example of heuristic algorithm

Spring kernel [Stankovic & Ramamritham 87]

1. The schedule for a set of N tasks is constructed in N steps
2. The search is driven by a heuristic function H
3. At each step the algorithm selects the task that minimizes the heuristic function



Example of heuristic algorithm

Spring kernel [Stankovic & Ramamritham 87]

Example of heuristic functions:

$$H = r_i \Rightarrow \text{FCFS}$$

$$H = C_i \Rightarrow \text{SJF}$$

$$H = D_i \Rightarrow \text{DM}$$

$$H = d_i \Rightarrow \text{EDF}$$

Composit heuristic functions:

$$H = w_1 r_i + w_2 D_i$$

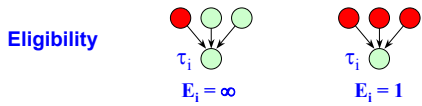
$$H = w_1 C_i + w_2 d_i$$

$$H = w_1 V_i + w_2 d_i$$

Example of heuristic algorithm

Spring kernel [Stankovic & Ramamritham 87]

Possibility to handle precedence constraints:



Heuristic functions:

$$H = E_i (w_1 r_i + w_2 D_i)$$

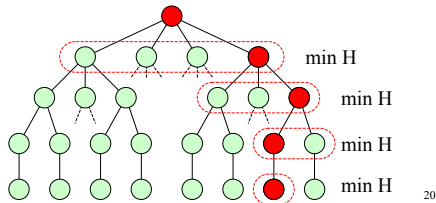
$$H = E_i (w_1 C_i + w_2 d_i)$$

19

Example of heuristic algorithm

Spring kernel [Stankovic & Ramamritham 87]

Complexity: $\left\{ \begin{array}{l} \text{Exhaustive search: } O(N!) \\ \text{Heuristic search: } O(N^2) \\ \text{Heuristic w. } k \text{ btracks: } O(kN^2) \end{array} \right.$



Examples of optimal algorithms

Rate Monotonic

$$p_i \propto \frac{1}{T_i} \quad \text{fixed priority}$$

- It is a static scheduling algorithm
- It can be preemptive or non preemptive
- It can be executed on line or off line
- It is optimal for feasibility among static algorithms

EDF

$$p_i \propto \frac{1}{d_i} \quad \text{dynamic priority}$$

- It is a dynamic scheduling algorithm
- It can be preemptive or non preemptive
- It can be executed on line or off line
- It is optimal for feasibility and minimizes L_{\max}

21

EDF Optimality

EDF is **optimal** for feasibility among all algorithms:

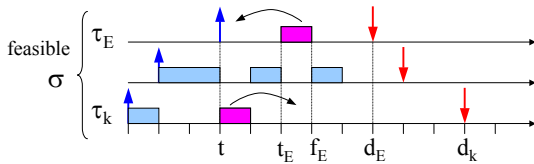
If there exists a feasible schedule for Γ , then EDF will generate a feasible schedule.



If Γ is not schedulable by EDF, then it cannot be scheduled by any algorithm.

22

EDF Optimality [Dertouzos '74]



Transforming σ in σ'

$$\begin{cases} \sigma'(t) = \sigma(t_E) \\ \sigma'(t_E) = \sigma(t) \end{cases}$$

Feasibility is preserved

$$f_k' = f_E \leq d_E \leq d_k$$

23

EDF schedulability

- In 1973, **Liu and Layland** proved that for a set of n periodic tasks:

$$U_{\text{lub}}^{EDF} = 1$$

- This means that a task set Γ is schedulable by EDF **if and only if**

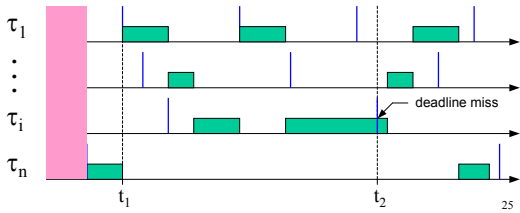
$$U_p \leq 1$$

24

Proving sufficiency

By contradiction, assume $U \leq 1$, and let t_2 be the time at which a deadline miss occurs.

Let $[t_1, t_2]$ be the longest interval of continuous utilization such that only instances with deadline $\leq t_2$ are executed:

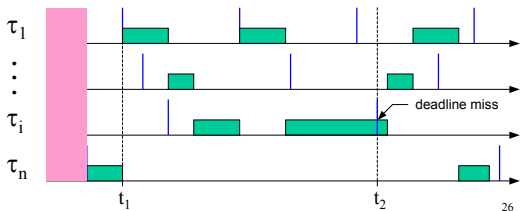


Proving sufficiency

The total computation time demanded in this interval is:

$$C_p(t_1, t_2) \leq \sum_{i=1}^n \left\lceil \frac{t_2 - t_1}{T_i} \right\rceil C_i \leq \sum_{i=1}^n \frac{t_2 - t_1}{T_i} C_i = (t_2 - t_1)U$$

deadline miss $\Rightarrow (t_2 - t_i) < C_p(t_1, t_2) \leq (t_2 - t_1)U$ **contradiction**



An alternate proof

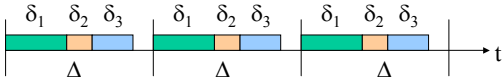
$$U_p \leq 1 \quad \Rightarrow \quad \Gamma \text{ schedulable}$$

- We find any algorithm for which the above condition holds;
- Then, for the EDF optimality, we can say that the above condition also holds for EDF.

Proving sufficiency

Consider the algorithm which schedules in every interval of length Δ a fraction of task:

$$\delta_i = U_i \Delta$$

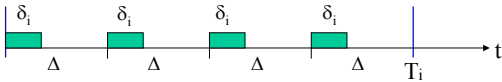


28

Proving sufficiency

With this algorithm, a task executes in each period for:

$$\frac{T_i}{\Delta} \delta_i = \frac{T_i}{\Delta} U_i \Delta = T_i U_i = C_i$$

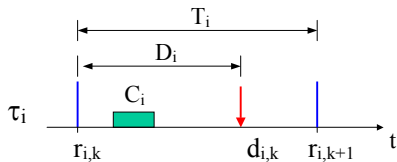


Feasibility is ensured if $\sum_{i=1}^n \delta_i \leq \Delta$ that is if

$$\sum_{i=1}^n U_i \Delta \leq \Delta \quad \rightarrow \quad U_p \leq 1$$

29

Extension to tasks with $D < T$



Scheduling algorithms

- Deadline Monotonic: $p_i \propto 1/D_i$ (static)
- Earliest Deadline First: $p_i \propto 1/d_i$ (dynamic)

30

Dynamic Priority

EDF

Schedule based on absolute deadlines

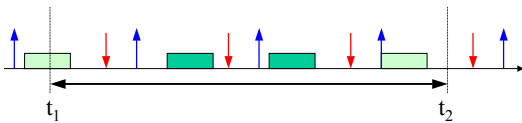
Schedulability Analysis

Processor Demand Criterion [Baruah '90]

In any interval, the computation demanded by the task set must be no greater than the available time.

31

Processor Demand

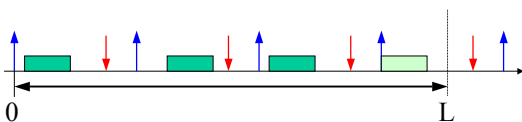


The demand in $[t_1, t_2]$ is the computation time of those jobs started at or after t_1 with deadline less than or equal to t_2 :

$$g(t_1, t_2) = \sum_{\substack{d_i \leq t_2 \\ r_i \geq t_1}} C_i$$

32

Processor Demand



Processor Demand in $[0, L]$

$$g(0, L) = \sum_{i=1}^n \left\lfloor \frac{L - D_i + T_i}{T_i} \right\rfloor C_i$$

33

Processor Demand Test

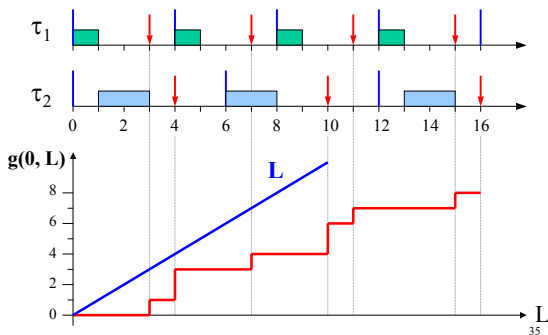
$$\forall L > 0, \quad g(0, L) \leq L$$

Question

How can we bound the number of intervals in which the test has to be performed?

34

Example



35

Bounding complexity

- Since $g(0, L)$ is a step function, we can check feasibility only at deadline points.
- If tasks are synchronous and $U_p < 1$, we can check feasibility up to the hyperperiod H :

$$H = \text{lcm}(T_1, \dots, T_n)$$

36

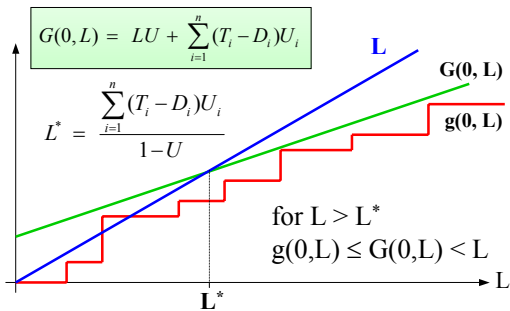
Bounding complexity

- Moreover we note that: $g(0, L) \leq G(0, L)$

$$\begin{aligned}
 G(0, L) &= \sum_{i=1}^n \left(\frac{L + T_i - D_i}{T_i} \right) C_i \\
 &= \sum_{i=1}^n L \frac{C_i}{T_i} + \sum_{i=1}^n (T_i - D_i) \frac{C_i}{T_i} \\
 &= LU + \sum_{i=1}^n (T_i - D_i) U_i
 \end{aligned}$$

37

Limiting L



38

Processor Demand Test

$$\begin{aligned}
 &U < 1 \\
 &\forall L \in D, \quad g(0, L) \leq L
 \end{aligned}$$

$$D = \{d_k \mid d_k \leq \min(H, L^*)\}$$

$$\begin{cases}
 H = \text{lcm}(T_1, \dots, T_n) \\
 L^* = \frac{\sum_{i=1}^n (T_i - D_i) U_i}{1 - U}
 \end{cases}$$

39

Handling shared resources

Problems caused by mutual exclusion

Priority Inversion

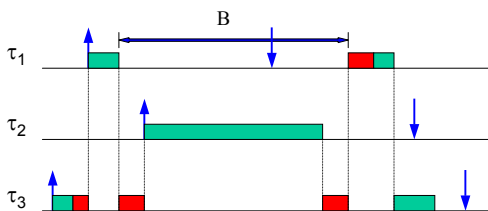
A high priority task is blocked by a lower-priority task a for an unbounded interval of time.

Deadline Inversion

A task with short deadline is blocked by a task with longer deadline a for an unbounded interval of time.

41

Conflict on a critical section



Solution

Introduce a concurrency control protocol for accessing critical sections.

42

Fixed Priority Protocols

- Non Preemptive Protocol (NPP)
- Highest Locker Priority (HLP)
- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)
- Immediate Priority Ceiling (IPC)

43

Dynamic Priority Protocols

- Dynamic Priority Inheritance (DIP)
- Dynamic Priority Ceiling (DPC)
- Stack Resource Policy (SRP)

44

Stack Resource Policy [\[Backer 90\]](#)

- It works both with fixed and dynamic priority
- It limits blocking to 1 critical section
- It prevents deadlock
- It supports multi-unit resources
- It allows stack sharing
- It is easy to implement

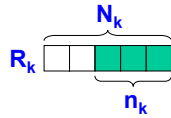
45

Stack Resource Policy [Backer 90]

- For each resource R_k :

⇒ Maximum units: N_k

⇒ Available units: n_k



- For each task τ_i the system keeps:

⇒ its resource requirements:

$$\mu_i(R_k)$$

⇒ a priority p_i :

$$RM \quad p_i \propto 1/T_i$$

$$EDF \quad p_i \propto 1/d_i$$

⇒ a static preemption level:

$$\pi_i \propto 1/D_i$$

46

Stack Resource Policy [Backer 90]

Resource ceiling

$$C_k(n_k) = \max_j \{ \pi_j : n_k < \mu_j(R_k) \}$$

System ceiling

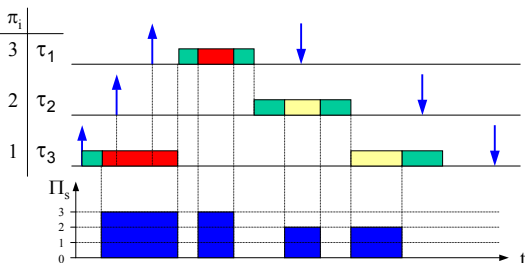
$$\Pi_s = \max_k \{ C_k(n_k) \}$$

SRP Rule

A job cannot preempt until p_i is the highest and $\pi_i > \Pi_s$

47

Example



48

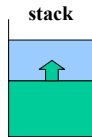
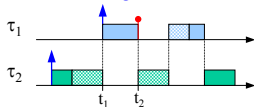
SRP: Notes

- Blocking always occurs at preemption time
- A task never blocks on a wait primitive (semaphore queuee are not needed)
- Semaphores are still needed to update the system ceiling
- Early blocking allows stack sharing

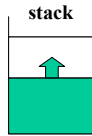
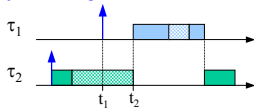
49

SRP: Stack sharing

Classical blocking



Early blocking



50

SRP: Stack sharing

- If tasks can be grouped in **M** subsets with the same preemption level, then tasks within a group cannot preempt each other.
- Then the stack size is the sum of the stack memory needed by **M** tasks.
- If we have 100 tasks with 10 preemption levels, and each task requires 10 Kb of stack, then

$$\text{Stack size} = \begin{cases} 1 \text{ Mb} & \text{without SRP} \\ 100 \text{ Kb} & \text{under SRP (90\% less)} \end{cases}$$

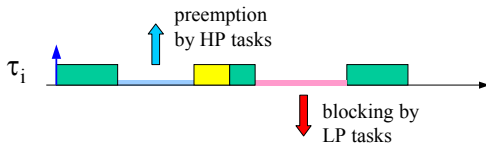
51

Guarantee with resource constraints

- Select a scheduling algorithm (e.g., EDF) and a resource access protocol (e.g., SRP).
- Compute the maximum blocking times (B_i) for each task.
- Perform the guarantee test including the blocking terms.

52

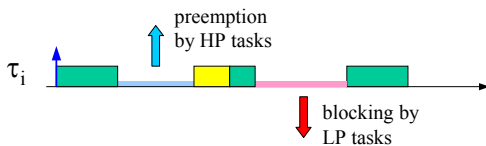
Guarantee with RM



$$\forall i \quad \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq i(2^{1/i} - 1)$$

53

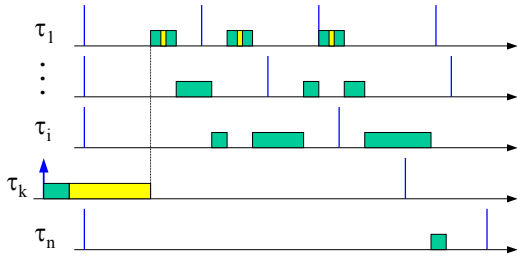
EDF Guarantee ($D_i = T_i$)



$$\forall i \quad \sum_{k=1}^{i-1} \frac{C_k}{T_k} + \frac{C_i + B_i}{T_i} \leq 1$$

54

EDF Guarantee: PD test ($D_i \leq T_i$)



55

EDF Guarantee: PD test ($D_i \leq T_i$)

$$\forall i \quad \forall L: D_i \leq L \leq \max(D_n, L_i^*)$$

$$U < 1 \quad \text{AND} \quad g_i(0, L) \leq L$$

$$\begin{cases} g_i(0, L) = B_i + \sum_{k=1}^i \left\lfloor \frac{L - D_k + T_k}{T_k} \right\rfloor C_k \\ L_i^* = \frac{B_i + \sum_{i=1}^n (T_i - D_i) U_i}{1 - U} \end{cases}$$

56

Handling Hybrid Task Sets

Periodic tasks
+
Aperiodic tasks

Handling Criticality

- Aperiodic tasks with **HARD** deadlines must be guaranteed under worst-case conditions.
- Off-line guarantee is only possible if we can bound interarrival times (**sporadic tasks**).
- Hence **sporadic tasks** can be guaranteed as periodic tasks with $C_i = WCET_i$ and $T_i = MIT_i$

$\left[\begin{array}{l} WCET = \text{Worst-Case Execution Time} \\ MIT = \text{Minimum Interarrival Time} \end{array} \right]$

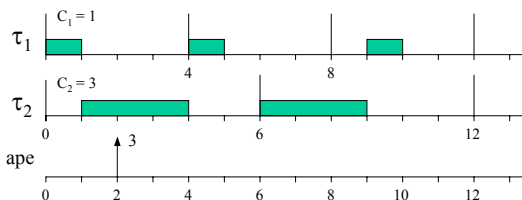
58

SOFT aperiodic tasks

- Aperiodic tasks with **SOFT** deadlines should be executed as soon as possible, but without jeopardizing HARD tasks.
- We may be interested in
 - minimizing the average response time
 - performing an on-line guarantee

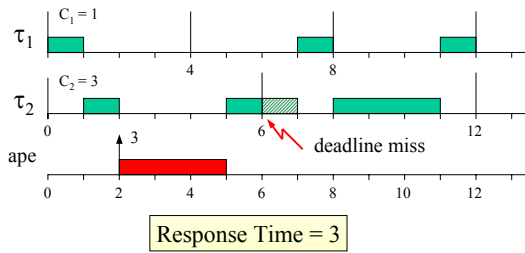
59

Periodic Scheduling (EDF)



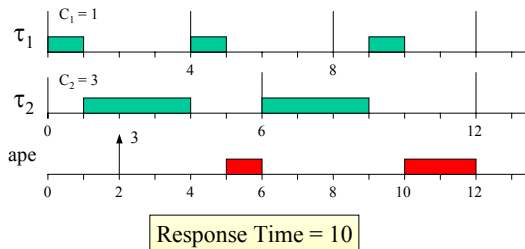
60

Immediate service



61

Background service



62

Aperiodic Servers

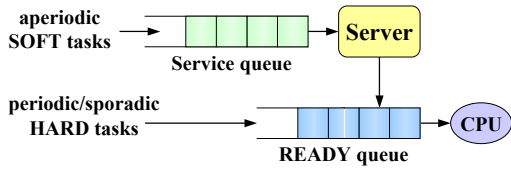
- A server is a kernel activity aimed at controlling the execution of aperiodic tasks.
- Normally, a server is a periodic task having two parameters:

$$\begin{cases} C_s & \text{capacity (or budget)} \\ T_s & \text{server period} \end{cases}$$

To preserve periodic tasks, no more than C_s units must be executed every period T_s

63

Aperiodic service queue



- The server is scheduled as any periodic task.
- Priority ties are broken in favor of the server.
- Aperiodic tasks can be selected using an arbitrary queueing discipline.

64

Fixed-priority Servers

- Polling Server
- Deferrable Server
- Sporadic Server
- Slack Stealer

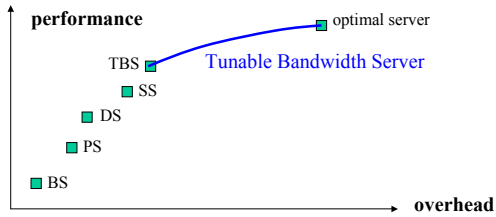
65

Dynamic-priority Servers

- Dynamic Polling Server
- Dynamic Sporadic Server
- Total Bandwidth Server
- Tunable Bandwidth Server
- Constant Bandwidth Server

66

Selecting the most suitable service mechanism



It depends on the price (overhead) we want to pay to reduce task response times

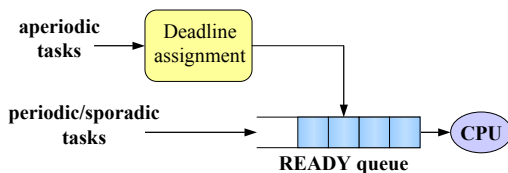
67

Total Bandwidth Server (TBS)

- It is a dynamic priority server, used along with EDF.
- Each aperiodic request is assigned a deadline so that the server demand does not exceed a given bandwidth U_s .
- Aperiodic jobs are inserted in the ready queue and scheduled together with the HARD tasks.

68

The TBS mechanism



- Deadlines ties are broken in favor of the server.
- Periodic tasks are guaranteed *if and only if*

$$U_p + U_s \leq 1$$

69

Deadline assignment rule

- Deadline has to be assigned not to jeopardize periodic tasks.
- A safe relative deadline is equal to the minimum period that can be assigned to a new periodic task with utilization U_s :

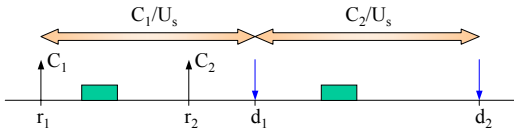
$$U_s = C_k / T_k \quad \Rightarrow \quad T_k = d_k - r_k = C_k / U_s$$

- Hence, the absolute deadline can be set as:

$$d_k = r_k + C_k / U_s$$

70

Deadline assignment rule

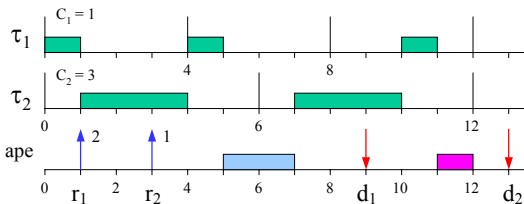


- To keep track of the bandwidth assigned to previous jobs, d_k must be computed as:

$$d_k = \max(r_k, d_{k-1}) + C_k / U_s$$

71

EDF + TBS schedule



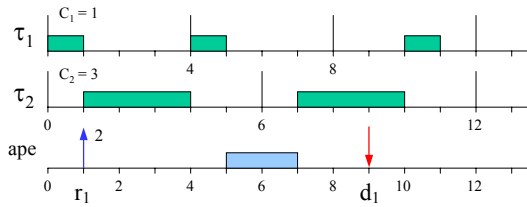
$$U_s = 1 - U_p = 1/4$$

$$\begin{cases} d_1 = r_1 + C_1 / U_s = 1 + 2 \cdot 4 = 9 \\ d_2 = \max(r_2, d_1) + C_2 / U_s = 9 + 1 \cdot 4 = 13 \end{cases}$$

72

Improving TBS

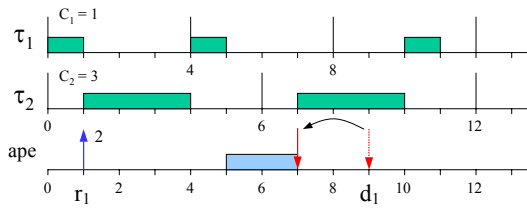
- What's the minimum deadline that can be assigned to an aperiodic job?



73

Improving TBS

- If we freeze the schedule and advance d_1 to 7, no task misses its deadline, but the schedule is not EDF:

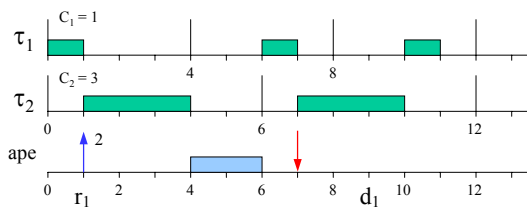


Feasible schedule \neq EDF

74

Improving TBS

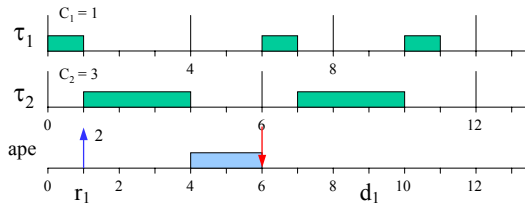
- However, since EDF is optimal, the schedule produced by EDF is also feasible:



75

Improving TBS

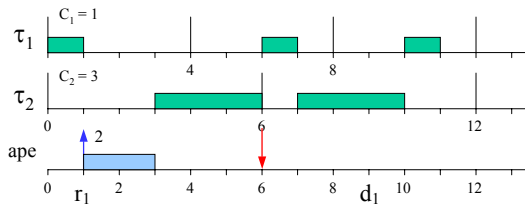
- We can now apply the same argument, and advance the deadline to $t = 6$:



76

Improving TBS

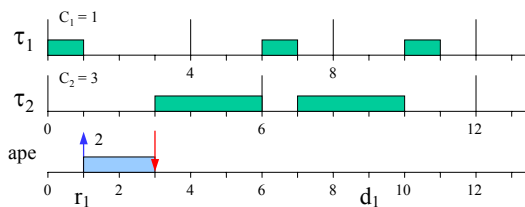
- We can now apply the same argument, and advance the deadline to $t = 6$:



77

Improving TBS

- Clearly, advancing the deadline now does not produce any enhancement in the response time:

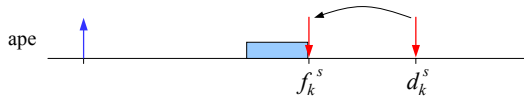


78

Computing the deadline

- In general, the new deadline has to be set to the finishing time of the current job:

$$\begin{cases} d_k^0 = \max(r_k, d_{k-1}^0) \\ d_k^{s+1} = f_k^s = f_k(d_k^s) \end{cases}$$

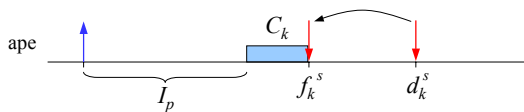


79

Computing the deadline

- Computing the actual finishing time is difficult, so we can compute an upper bound:

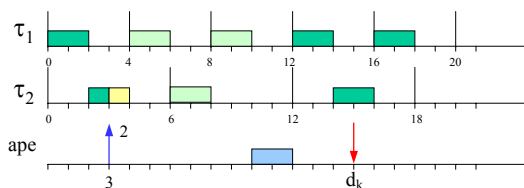
$$\begin{aligned} f_k^s &= C_k + I_p(r_k, f_k^s) \\ \tilde{f}_k^s &= C_k + I_p(r_k, d_k^s) \geq f_k^s \end{aligned}$$



80

Periodic Interference

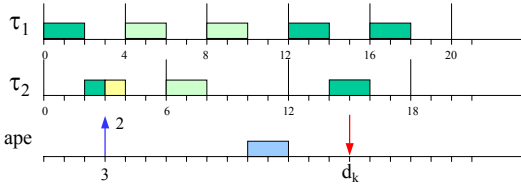
$$\begin{cases} U_p = 1/2 + 1/3 = 5/6 \\ U_s = 1 - U_p = 1/6 \end{cases} \quad \begin{cases} C_k = 2 \\ d_k = 3 + 2/U_s = 15 \end{cases}$$



$$I_p(t, d_k^s) = \underbrace{I_a(t, d_k^s)}_{\text{yellow bar}} + \underbrace{I_f(t, d_k^s)}_{\text{green bar}}$$

81

Computing interference



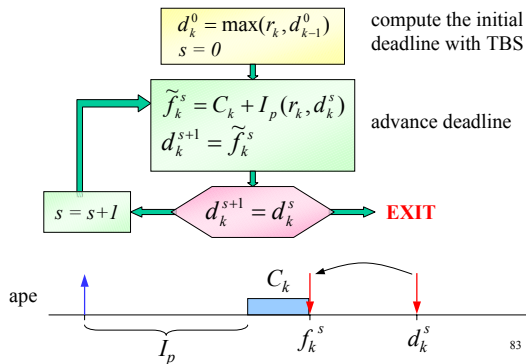
$$I_a(t, d_k^s) = \sum_{\tau_i \text{ active}} c_i(t)$$

$next_i(t)$ = next release time of task τ_i after t

$$I_f(t, d_k^s) = \sum_{i=1}^n \left\lfloor \frac{d_k^s - next_i(t)}{T_i} \right\rfloor C_i$$

82

The Optimal Server



83

Two interesting results

- If $(d_k^{s+1} = d_k^s)$ then $\tilde{f}_k^s = f_k^s$

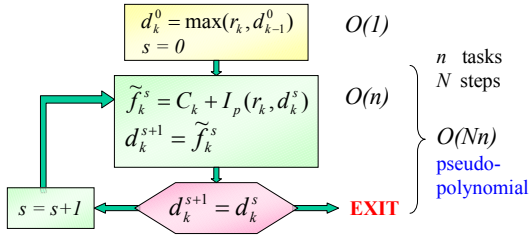
It means that the estimate is exact

- If $(d_k^{s+1} = d_k^s)$ then $\tilde{f}_k^s = f_{k \min}^s$

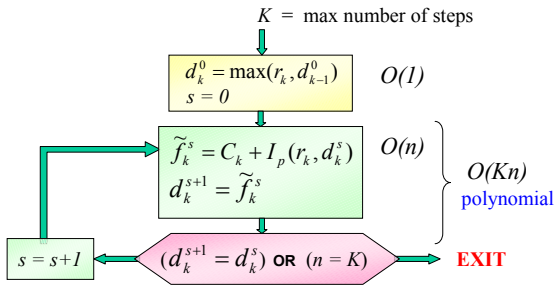
It means that the algorithm minimizes the aperiodic response time

84

Complexity



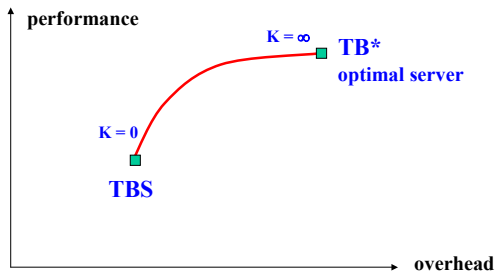
Tunable Bandwidth Server TB(K)



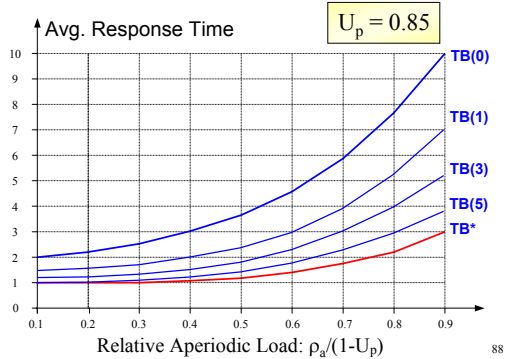
TB(0) = TBS

TB(∞) = TB*

Tuning performance vs. overhead



Aperiodic responsiveness



Lecture 1.

Preemption Threshold Scheduling: Introduction and Definition

Seongsoo Hong
Real-Time Operating Systems Lab.
Seoul National University, Korea
<http://redwood.snu.ac.kr>

Lecture Outline

- ❖ Introduce preemption threshold scheduling (PTS)
- ❖ Briefly discuss the benefits PTS offers
- ❖ Explain how to assign priorities and preemption thresholds to tasks

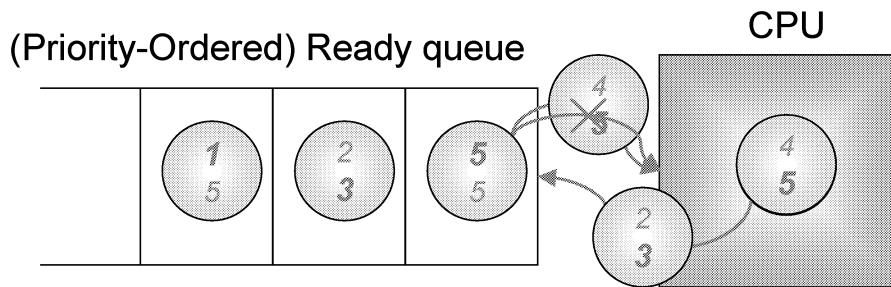
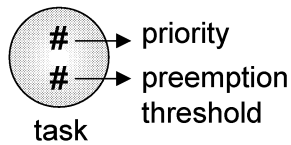
Types of Real-Time Scheduling

- ❖ Pre-runtime scheduling
 - Cyclic executive scheduling
 - Pre-runtime scheduling
- ❖ Priority Scheduling
 - Static priority scheduling
 - Based on single fixed priority per task
 - Mostly preemptive scheduling
 - Dynamic priority scheduling
 - Based on dynamically changing priority

Introduction to PTS

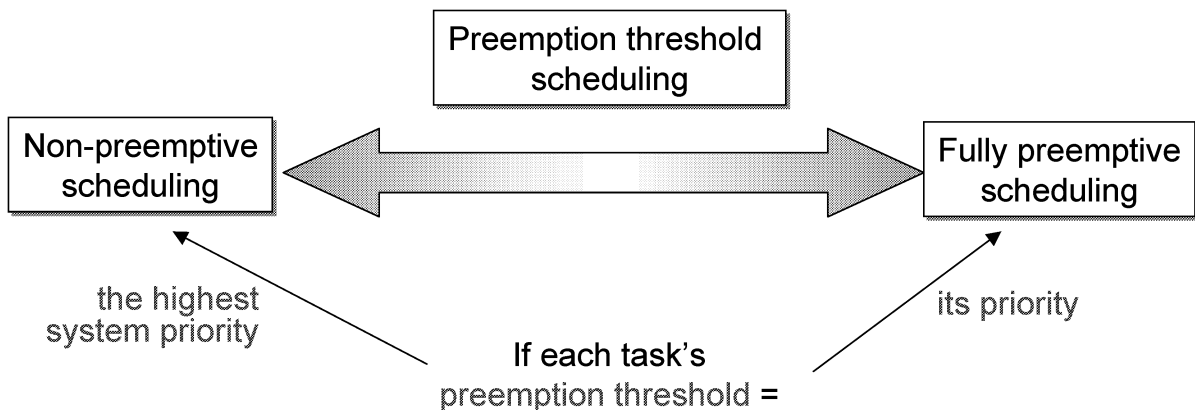
- ❖ PTS
 - Dual priority preemptive scheduling
 - Preemption threshold is just the run-time priority of a task
- ❖ PTS model
 - Extension of fixed priority scheduling
 - Each task has
 - A regular priority
 - The priority at which the task is queued and released.
 - A preemption threshold
 - Once a task gets the CPU, its priority is set to this value.

Execution Scenario



PTS Covers the Entire Spectrum

- ❖ Non-preemptive scheduling and fully preemptive scheduling are two special cases of preemption threshold scheduling.



Why PTS?

(1) Increase of schedulability

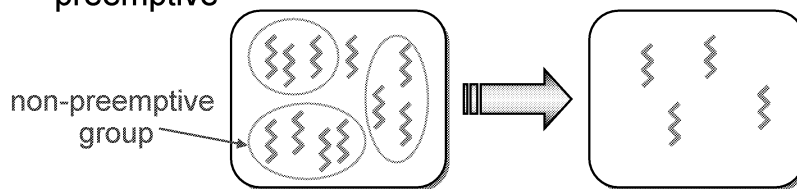
- Superior to both preemptive scheduling and non-preemptive scheduling

(2) Reduced run-time (context switch) overhead

- Due to elimination of unnecessary preemption

(3) Reduction of the number of tasks

- A non-preemptive group (will be covered in Lecture 2)
 - Consists of tasks in which every pair is mutually non-preemptive



Priority and Preemption Threshold Assignment Algorithms (1)

❖ Algorithm 1:

- Assign preemption thresholds alone
- Input:
 - An unschedulable task set where tasks have pre-assigned priorities.
- Purpose:
 - To make all tasks schedulable by assigning preemption thresholds.

Priority and Preemption Threshold Assignment Algorithms (2)

❖ Algorithm 2:

- Assign priorities alone
- Input:
 - A task set where tasks have no assigned priorities or PTs.
- Purpose:
 - To assign priorities to tasks under preemptive scheduling hoping that the task set will be schedulable by assigning PTs later

Priority and Preemption Threshold Assignment Algorithms (3)

❖ Algorithm 3:

- Assign both priorities and PTs together
- Input:
 - A task set where tasks have no assigned priorities and PTs.
- Purpose:
 - To make all tasks schedulable by assigning both PTs and priorities.
- Algorithm 2 (Assign Priorities) + Algorithm 1 (Assign PTs)

Priority and Preemption Threshold Assignment Algorithms (4)

❖ Algorithm 4:

- Assign maximum PTs
- Input:
 - A task set where tasks have assigned priorities and PTs.
- Purpose:
 - To reduce the number of context switches by assigning maximum PTs.

Notations

❖ A set of N tasks

- $T = \{ \tau_1, \tau_2, \dots, \tau_N \}$

❖ Timing attributes of τ_i

- Execution time: C_i
- Period: T_i
- Deadline: D_i

❖ Scheduling attributes of τ_i

- Priority: $\pi_i \in [1, \dots, N]$
- Preemption threshold: $\gamma_i \in [\pi_i, \dots, N]$

❖ WCRT: Worst-Case Response Time

- The calculation algorithm for this will be covered in lecture 3.

Assumption

❖ $D_i \leq T_i$

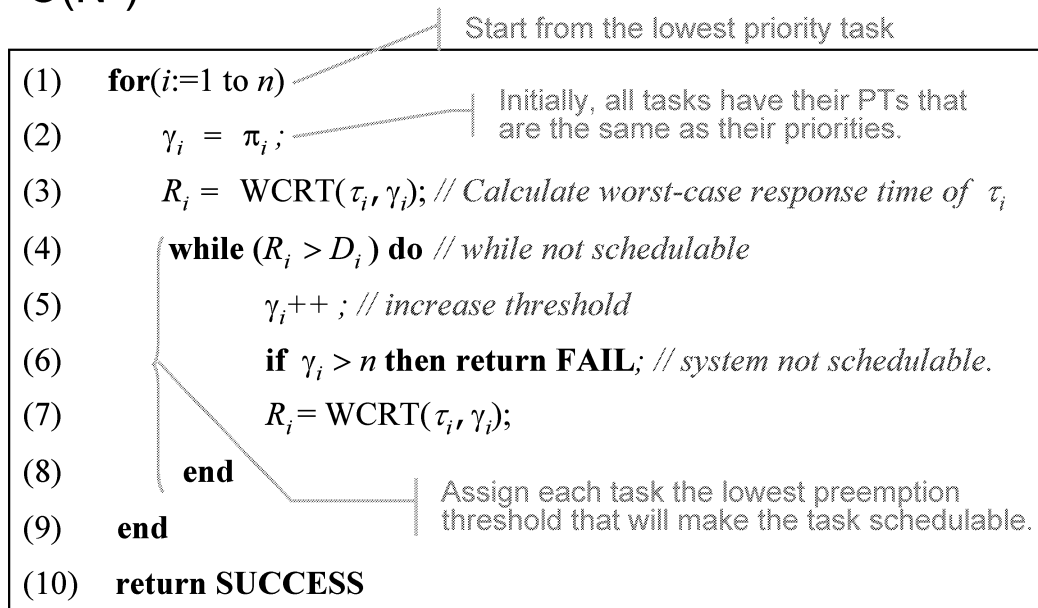
- No busy period analysis for simplicity (no loss of generality)

Algorithm 1: Assign Preemption Thresholds Alone (1)

- ❖ Optimal algorithm
- ❖ Assumes that task priorities are already known
- ❖ Algorithmic steps
 - Start from the lowest priority task
 - Initially, all tasks have preemption thresholds the same as their priorities
 - Calculate worst-case response time of task i .
 - While it is not schedulable, increase its PT value.

Algorithm 1: Assign Preemption Thresholds Alone (2)

❖ $O(N^2)$



Scout National University

RTOS Lab

15

Algorithm 2: Assign Priorities Alone (1)

❖ Extension of Audsley's algorithm

❖ Audsley's algorithm

- An optimal priority ordering algorithm
- Basic idea
 - A lower priority task's schedulability depends on higher priority task set only (not their priority ordering)
- Two parts
 - Sorted: lower n priority tasks
 - Unsorted: the remaining higher priority tasks
- Algorithmic steps: $O(N^2)$
 - Initially, all tasks are unsorted
 - Schedulability tests for tasks in the unsorted part
 - If okay, lowest priority assigned and it becomes sorted
 - If not okay, keep unsorted and test another one

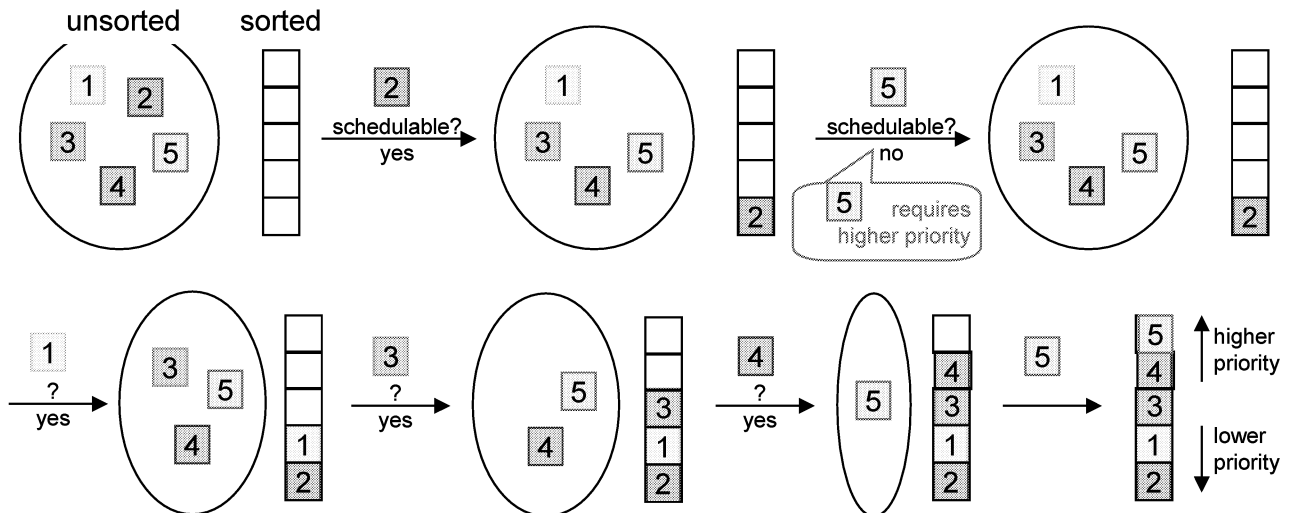
Scout National University

RTOS Lab

16

Algorithm 2: Assign Priorities Alone (2)

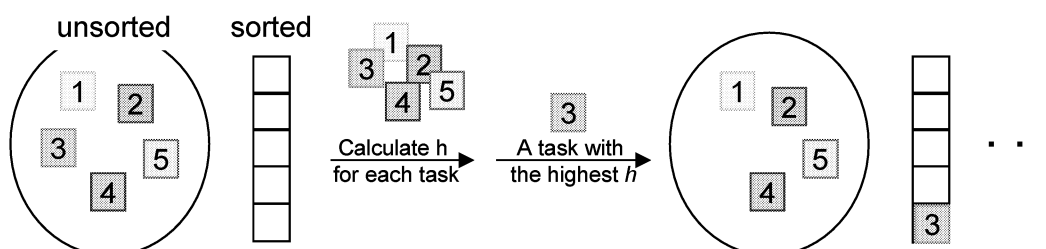
❖ Adopt Audsley's algorithm



Algorithm 2: Assign Priorities Alone (3)

❖ A heuristic and greedy approach that extends Audsley's algorithm

- Exploits a simple heuristic value h to select the next task for priority assignment
- For each candidate priority,
 - (1) Compute h_i for each task τ_i in the unsorted list and
 - (2) Move a task with the highest h to sorted list.



Algorithm 2: Assign Priorities Alone (4)

- ❖ Heuristic value h_i for each task τ_i
 - After schedulability test of task τ_i
 - If okay, h_i = the maximum blocking that a task can tolerate
 - Task τ_i may not be schedulable after PT assignment since a lower priority may cause blocking.
 - If not, $h_i = D_i - R_i$ (always a minus value)
 - It is still possible for task τ_i to be schedulable since it can be given a higher PT.
 - In case there are no tasks for the first category, we want to choose the task that needs the smallest reduction in interference from higher priority tasks.

Algorithm 2: Assign Priorities Alone (5)

❖ $O(N^2)$

```
(1)  UnSorted: = T; Sorted: = {};  
(2)  for pri: = 1 to N do  
(3)      foreach  $\tau_k \in$  UnSorted do  
(4)           $\pi_k :=$  pri; // tentative assignment  
(5)           $R_k :=$  WCRT( $\tau_k$ ); // compute response time  
(6)          if  $R_k \geq D_k$  then  $h_k := D_k - R_k$  else  $h_k :=$  GetBlockingLimit( $\tau_k$ );  
(7)           $\pi_k :=$  N; // reset, to allow computing heuristic value for other tasks  
(8)      end  
      // Select the task with the largest heuristic value next  
(9)       $\tau_k :=$  MaxHeuristicVal(UnSorted);  
(10)      $\pi_k :=$  pri; Sorted := Sorted + { $\tau_k$ }; UnSorted := UnSorted -  $\tau_k$ ;  
(11)  end
```

The maximum blocking that a task can tolerate

Algorithm 3: Assign Priorities and PTs Together

❖ Two-step algorithm

1. Assign priorities [algorithm 2]
2. And then assign preemption thresholds [algorithm 1]

❖ Non-optimal algorithm

Algorithm 4: Assign Maximum PTs (1)

❖ Considers one task at a time,

- Starts from the highest priority task
- Tries to assign it the largest PT value that will still keep the system schedulable.
 - Check the WCRT of the affected task to ensure that the system stays schedulable.

❖ Only need to go through the list of tasks once.

- By going from highest priority task to lowest priority task
 - Ensures that any change in the PT assignment in later (lower priority) tasks cannot increase the assignment of a former (higher priority) task.

Algorithm 4: Assign Maximum PTs (2)

❖ $O(N^2)$

Start from the highest priority task

```
// Assumes that task priorities are fixed, and
// a set of feasible preemption thresholds are assigned.
(1)  for(i := n down to 1)
(2)      while (schedulable == TRUE) && ( $\gamma_i < n$ )
(3)           $\gamma_i += 1$ ; // try a larger value
(4)          Let  $\tau_j$  be the task such that  $\pi_j = \gamma_i$ 
           // Calculate the worst-case response time of task j
           // and compare it with deadline
(5)           $R_j := \text{WCRT}(\tau_j)$ ;
(6)          if ( $R_j \geq D_j$ ) then schedulable := FALSE ;  $\gamma_i -= 1$ ; endif
(7)      end
(8)      schedulable := TRUE
(9)  end
```

Soof National University

RTOS Lab

23

Summary

- ❖ Preemption threshold scheduling
 - Eliminates unnecessary preemption as much as possible
- ❖ Benefits of preemption threshold scheduling
 - Increase schedulability over both preemptive and non-preemptive scheduling
 - Reduce run-time (context switch) overhead
 - Reduce the number of tasks
- ❖ Priority and preemption threshold assignment algorithms
 - Assign either priorities or preemption thresholds
 - Assign both priorities and preemption thresholds
 - Assign maximum preemption thresholds

Soof National University

RTOS Lab

24

Lecture 2. Preemption Threshold Scheduling: Application

Seongsoo Hong
Real-Time Operating Systems Lab.
Seoul National University, Korea
<http://redwood.snu.ac.kr>

Lecture Outline

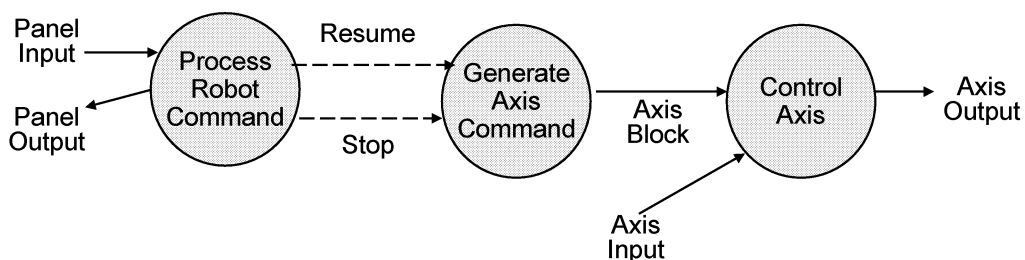
- ❖ Overview and classify SW design methodologies
- ❖ Introduce scalability problem
- ❖ Introduce scalable real-time system design with PTS

SW Design Methodologies (1)

- ❖ Have evolved since 1960's.
- ❖ Various strategies are classified into two types:
 - (1) Task-based
 - (2) Object-oriented

SW Design Methodologies (2)

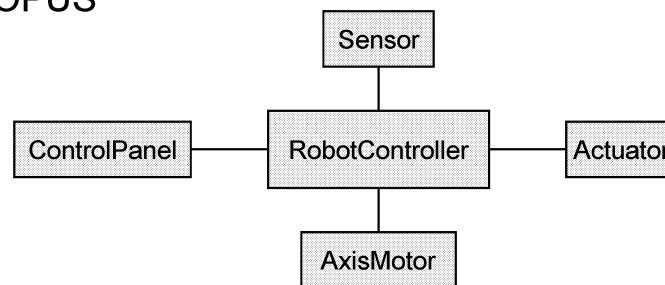
- ❖ Task-based
 - Put an emphasis on deriving tasks from the structured analysis of given requirement specifications.
 - (ex) DARTS (Design Approach for Real-Time Systems), HRT-HOOD (Hard Real-time Hierarchical Object Oriented Design)



SW Design Methodologies (3)

❖ Object-oriented

- View a real-time system as a collection of concurrent and active objects communicating with each other via messages.
- Focus on capturing the high-level abstract features of a system
- (ex) ROOM (Real-Time Object-Oriented Modeling), ROPES (Rapid Object-Oriented Process for Embedded Systems), OCTOPUS



Scalability Problem (1)

❖ Task-based SW design methodologies

- Work well only when the following conditions are met.
 1. A system is decomposed into a small number of tasks.
 2. Each task is of a relatively coarse granularity.
- Are largely focused on task cohesion criteria.

❖ Object-oriented SW design methodologies

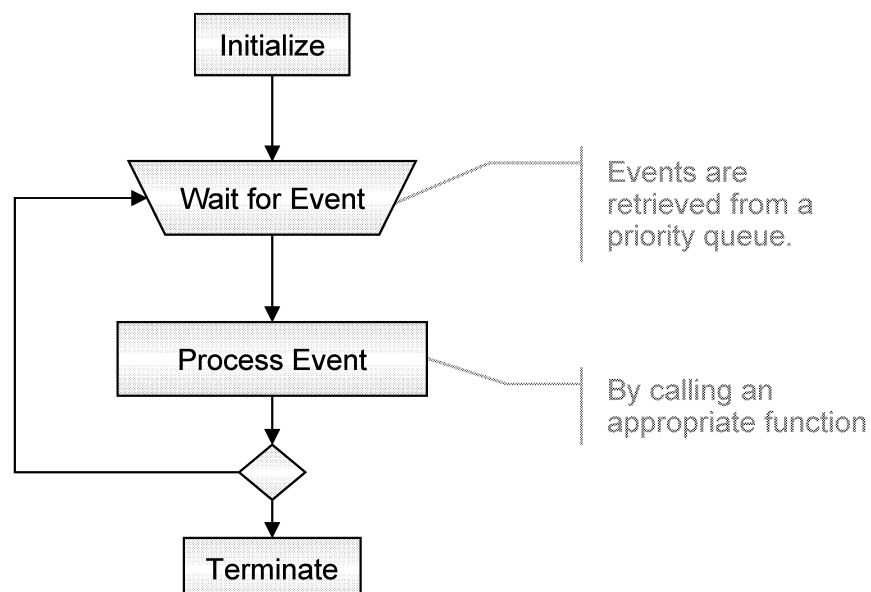
- Usually incur a large number of tasks when the implementation is automatically generated from design.
- Many commercial object-oriented CASE tools map each object to a separate task.

Scalability Problem (2)

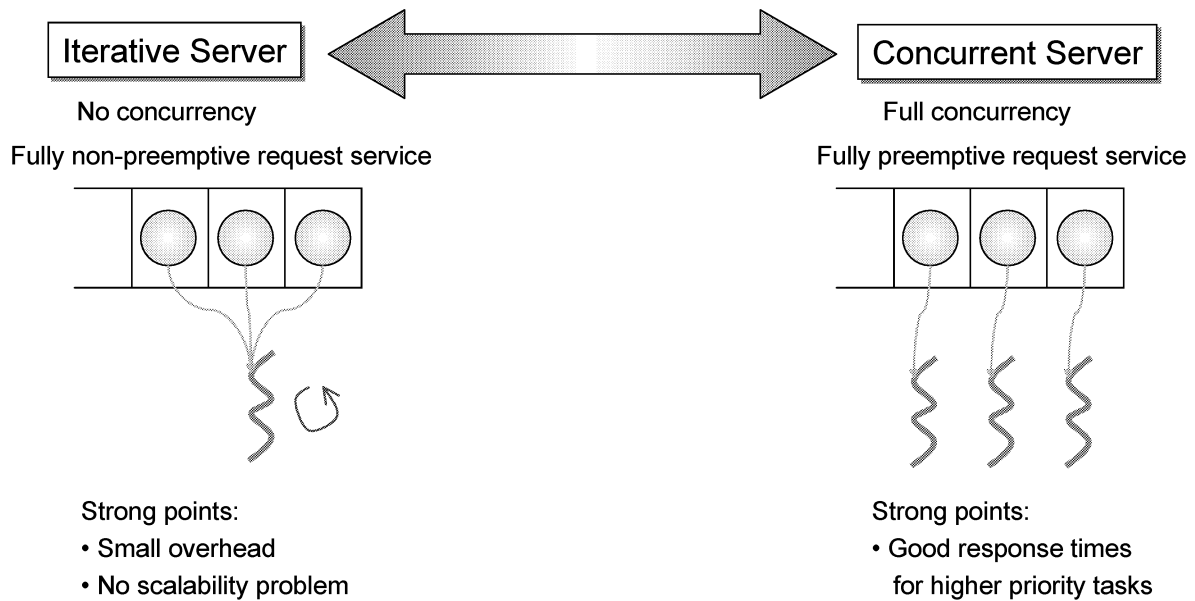
- ❖ Large number of tasks causes a scalability problem.
 - Run-time context switching overhead (performance degradation)
 - Large memory requirements (task stacks)

- ❖ Concurrent real-time system design always encounters the scalability problem.

Behavior of an Event Handling Task

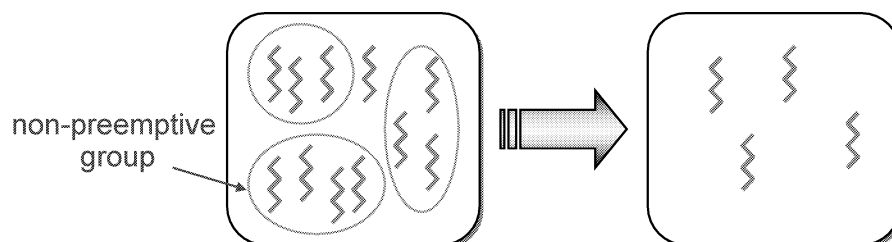


Concurrent Server vs. Iterative Server



Scalable Real-Time System Design with PTS

❖ Can reduce the number of tasks with PTS



❖ A non-preemptive group

- Consists of tasks that are mutually non-preemptive.

Mutually Non-Preemptive Relationship

❖ Notations

- A task: τ_i
- Priority of task τ_i : π_i
- Preemption threshold of task τ_i : γ_i

❖ Two tasks τ_i and τ_j are mutually non-preemptive

- If τ_i cannot preempt τ_j and τ_j cannot preempt τ_i .
- If $\pi_i \leq \gamma_j$ and $\pi_j \leq \gamma_i$.

An Optimal Algorithm for Partitioning Tasks into Non-Preemptive Groups (1)

❖ Input:

- A task set where all tasks have assigned priorities and preemption thresholds.

❖ Purpose:

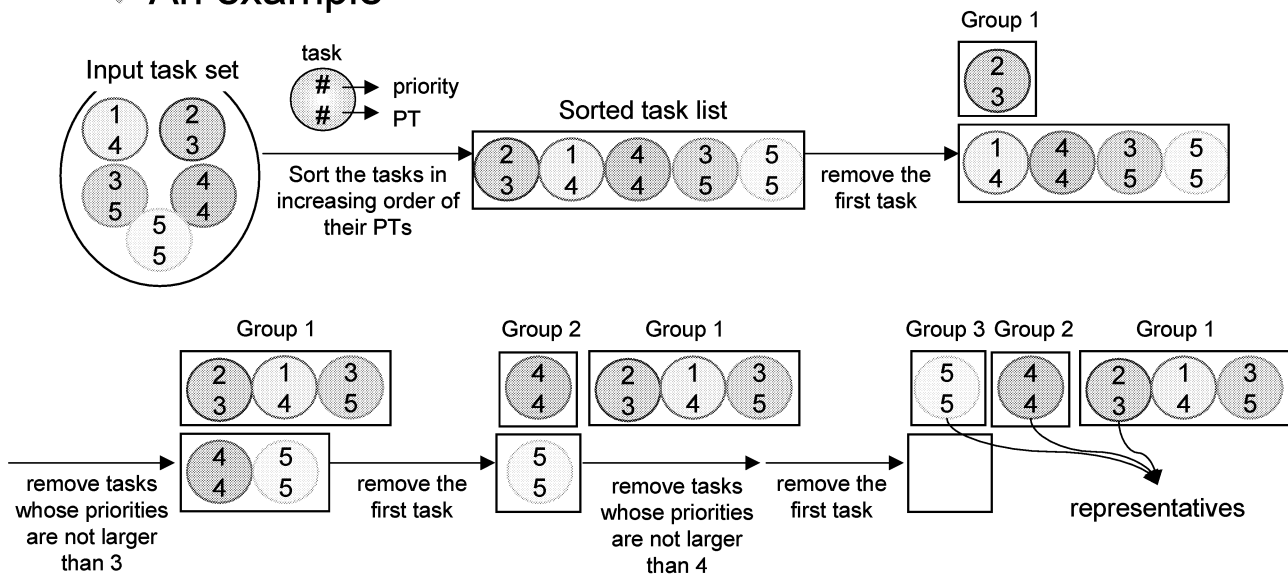
- Generate minimum number of non-preemptive groups

❖ Assumption:

- For each task τ_i , $\gamma_i \geq \pi_i$.

An Optimal Algorithm for Partitioning Tasks into Non-Preemptive Groups (2)

❖ An example



An Optimal Algorithm for Partitioning Tasks into Non-Preemptive Groups (3)

❖ $O(N^2)$

```

(1) ngroups := 0 ;
    // Sort the tasks by  $\gamma_i$  in increasing order
(2) L := SortTasksbyPreemptionThreshold(TaskSet) ;
(3) while (L != NULL) do
    // Find the task with the smallest value of  $\gamma_i$ 
(4)  $\tau_k := \text{Head}(L)$ ; G[ngroups] := { $\tau_k$ }; L := L -  $\tau_k$ ;
(5) foreach  $\tau_j \in L$  do
(6)     if ( $\pi_j \leq \gamma_k$ )
(7)     then G[ngroups] = G[ngroups] + { $\tau_j$ }; L := L -  $\tau_j$ ;
(8)     endif
(9) end
(10) ngroups := ngroups + 1;
(11) end
    
```

An Optimal Algorithm for Partitioning Tasks into Non-Preemptive Groups (4)

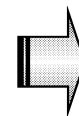
❖ Is this algorithm optimal? Yes.

- No other partitioning of non-preemptive groups can be done with a smaller number of groups.
- Proof sketch
 - Consider any two groups formed by the algorithm and their two representatives τ_i and τ_j .
 - Since a representative is compared with all remaining tasks in the sorted list, it must be the case τ_i and τ_j are not mutually non-preemptive.
 - Therefore, they must be in separate non-preemptive groups.
 - Since it is true for each pair of representative group members, it is not possible to have a solution with fewer groups.

(Ex 1) Olympus Satellite

| Task | Period | WCET | Priority | Threshold |
|------|--------|--------|----------|-----------|
| 1 | 100 | 4.08 | (3) 20 | 21 |
| 2 | 1000 | 2.06 | 11 | 21 |
| 3 | 500 | 4.12 | 18 | 21 |
| 4 | 2000 | 8.25 | 4 | 21 |
| 5 | 625 | 2.06 | 16 | 21 |
| 6 | 1870 | 2.06 | 5 | 21 |
| 7 | 1000 | 2.06 | 12 | 21 |
| 8 | 10000 | 99.32 | 2 | 19 |
| 9 | 2000 | 45.82 | 6 | 21 |
| 10 | 2000 | 425.82 | (1) 7 | 10 |
| 11 | 10000 | 58.52 | 3 | 21 |
| 12 | 1000 | 83.02 | (2) 13 | 19 |
| 13 | 100 | 24.62 | 21 | 21 |
| 14 | 1000 | 63.7 | 14 | 21 |
| 15 | 500 | 5.32 | 19 | 21 |
| 16 | 2000 | 32.02 | 8 | 21 |
| 17 | 1000 | 22.52 | 15 | 21 |
| 18 | 2000 | 32.02 | 9 | 21 |
| 19 | 1870 | 51.5 | 10 | 21 |
| 20 | 625 | 49.8 | 17 | 21 |
| 21 | 36000 | 9.42 | 1 | 21 |

- Utilization: 87.89%
- Breakdown utilization:
 - preemptive: 97.6 %
 - with threshold: 99.2 %

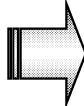


21 tasks
→ 3 tasks

(Ex 2) Generic Avionics Platform

| Task | Period | WCET | Priority | Threshold |
|------|--------|------|----------|-----------|
| 1 | 25 | 2 | 16 | 17 |
| 2 | 25 | 5 | 17 | 17 |
| 3 | 40 | 1 | 15 | 17 |
| 4 | 50 | 3 | 13 | 17 |
| 5 | 50 | 5 | 14 | 17 |
| 6 | 59 | 8 | 12 | 17 |
| 7 | 80 | 9 | 10 | 17 |
| 8 | 80 | 2 | 11 | 17 |
| 9 | 100 | 5 | 8 | 17 |
| 10 | 200 | 3 | 3 | 17 |
| 11 | 200 | 1 | 4 | 17 |
| 12 | 200 | 1 | 5 | 17 |
| 13 | 200 | 3 | 6 | 17 |
| 14 | 200 | 1 | 7 | 17 |
| 15 | 200 | 3 | 8 | 17 |
| 16 | 1000 | 1 | 1 | 17 |
| 17 | 1000 | 1 | 2 | 17 |

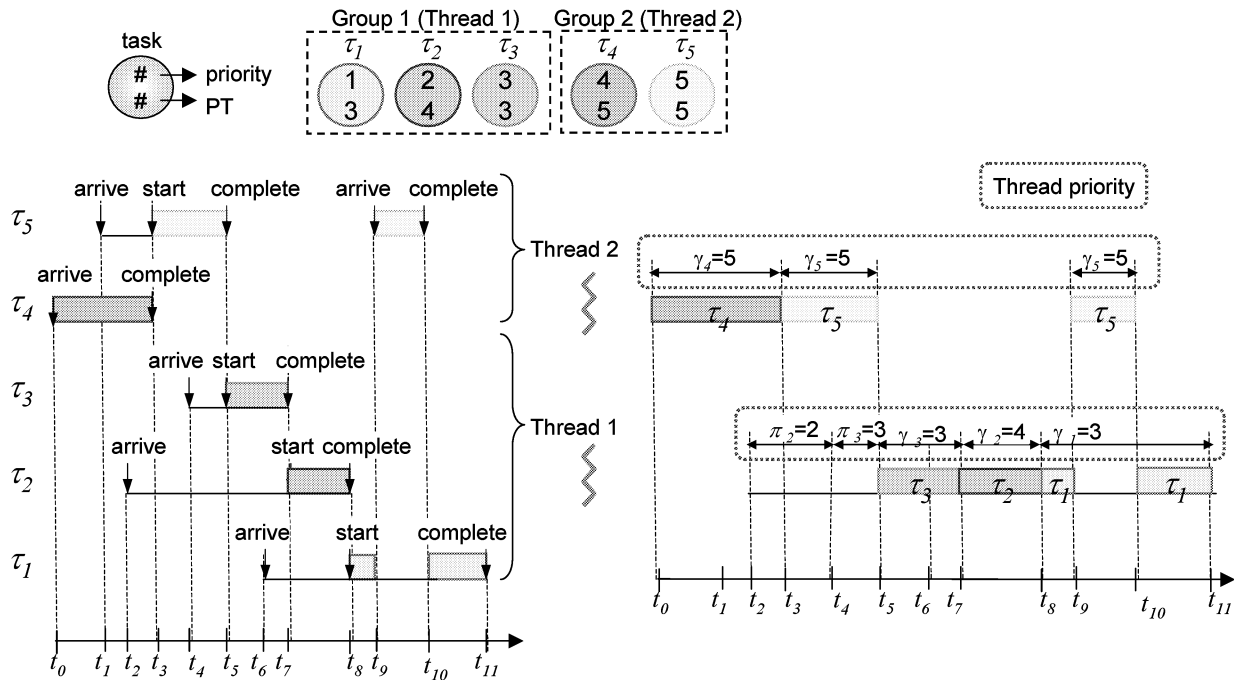
- Utilization: 85.01%
- Breakdown utilization:
 - preemptive: 94.4 %
 - non-preemptive: 93.8 %
 - with threshold: 94.4 %


 17 tasks
 → 1 task

Thread Priority Management under PTS

- ❖ Members in each non-preemptive group are mapped to a single thread.
- ❖ The priority of a thread is set to
 - (1) If the thread is not executing any tasks,
 - The maximum priority of the arrived member tasks
 - (2) Otherwise: if the thread is executing a member task,
 - PT of the executing task

Execution Scenario



Souf National University

Summary

- ❖ SW design methodologies are largely classified into two types:
 - (1) Task-based
 - Focused on task structuring.
 - (2) Object-oriented based
 - Focused on high level system structuring.
- ❖ Concurrent real-time system design always encounters the scalability problem.
- ❖ PTS is good for scalable real-time system design.
 - Partitioning tasks into non-preemptive group can significantly reduce the number of tasks.

Souf National University

Lecture 3.

Preemption Threshold Scheduling: Schedulability Analysis

Seongsoo Hong
Real-Time Operating Systems Lab.
Seoul National University, Korea
<http://redwood.snu.ac.kr>

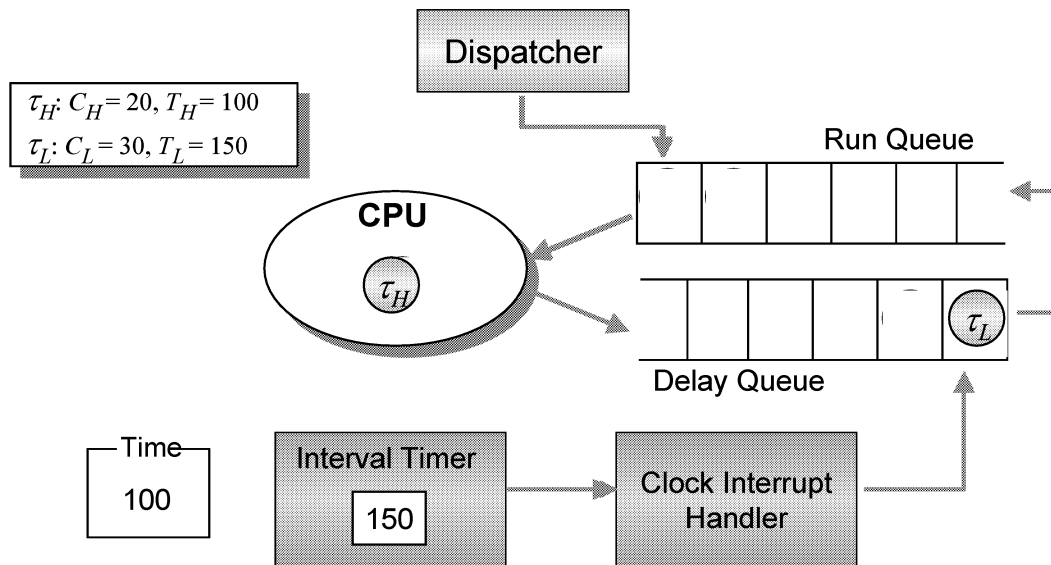
Lecture Outline

- ❖ Runtime for scheduling preemptive fixed priority periodic tasks

- ❖ Worst-case response time (WCRT) analysis with Gantt chart
 - An Example Task Set
 - Non-Preemptive Scheduling
 - Fully Preemptive Scheduling
 - Preemption Threshold Scheduling

- ❖ WCRT analysis algorithms

Periodic Task Set Scheduling



WCRT Analysis Using Gantt Chart

- ❖ An Example Task Set
- ❖ Non-Preemptive Scheduling
- ❖ Fully Preemptive Scheduling
- ❖ Preemption Threshold Scheduling

An Example Task Set for WCRT Analysis Using Gantt Chart

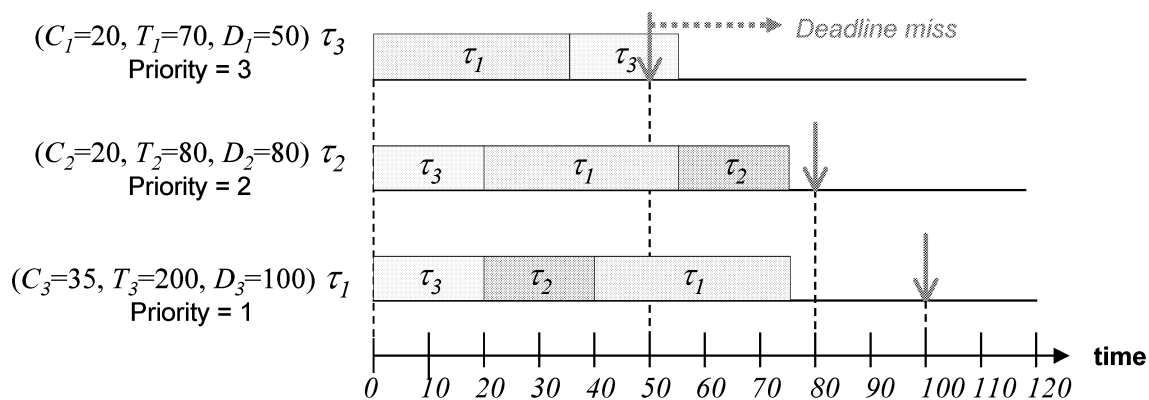
❖ A task set that is

- Not schedulable both with non-preemptive scheduling and preemptive scheduling
- But schedulable with preemption threshold scheduling

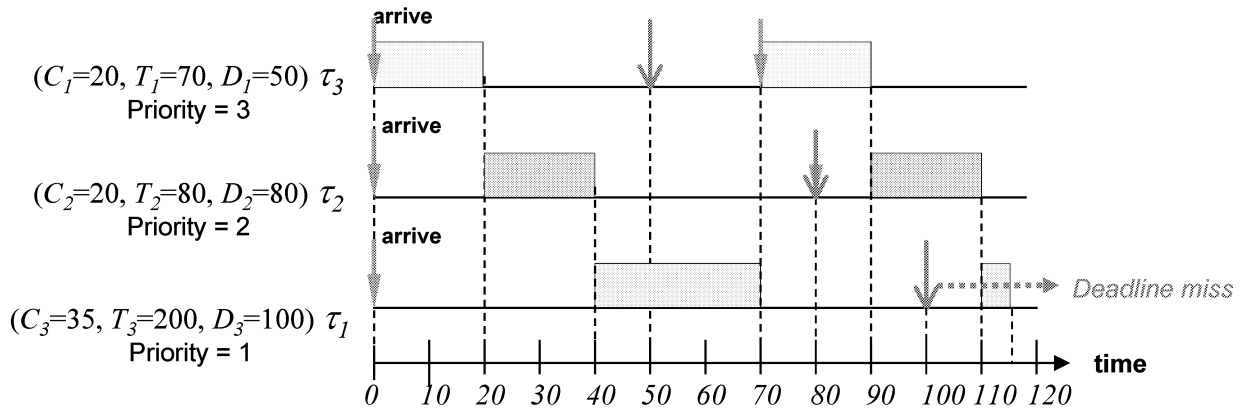
| Task | Priority | Execution Time | Period | Deadline |
|----------|----------|----------------|-------------|-------------|
| τ_3 | 3 | $C_1 = 20$ | $T_1 = 70$ | $D_1 = 50$ |
| τ_2 | 2 | $C_2 = 20$ | $T_2 = 80$ | $D_2 = 80$ |
| τ_1 | 1 | $C_3 = 35$ | $T_3 = 200$ | $D_3 = 100$ |

- Larger value denotes higher priority.

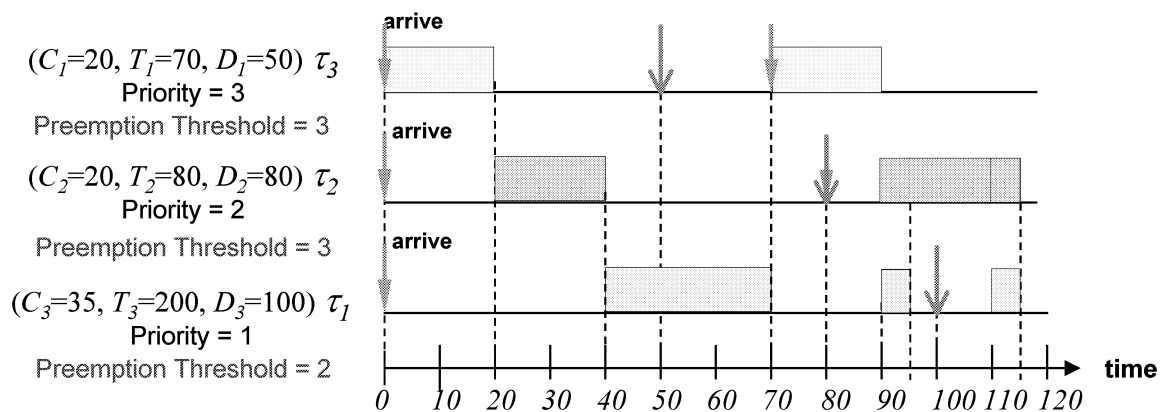
WCRT Analysis Using Gantt Chart: Non-Preemptive Scheduling



WCRT Analysis Using Gantt Chart: Fully Preemptive Scheduling



WCRT Analysis Using Gantt Chart: Preemption Threshold Scheduling



WCRT Analysis Algorithms

- ❖ Notations
- ❖ WCRT Analysis of Fully Preemptive Scheduling
- ❖ WCRT Analysis of Non-Preemptive Scheduling
- ❖ WCRT Analysis of Preemption Threshold Scheduling

Notations

- ❖ A set of N tasks
 - $T = \{\tau_1, \tau_2, \dots, \tau_N\}$
- ❖ Timing attributes of τ_i
 - Execution time: C_i
 - Period: T_i
 - Deadline: D_i
- ❖ Scheduling attributes of τ_i
 - Priority: $\pi_i \in [1, \dots, N]$
 - Preemption threshold: $\gamma_i \in [\pi_i, \dots, N]$
- ♣ Assumption
 - $D_i \leq T_i$
 - No busy period analysis for simplicity (no loss of generality)

WCRT Analysis of Fully Preemptive Scheduling

$$R_i^{n+1} = C_i + \sum_{\forall j, \pi_j > \pi_i} \left\lfloor \frac{R_i^n}{T_j} \right\rfloor \cdot C_j$$

Interference time from higher priority tasks

WCRT Analysis of Non-Preemptive Scheduling

[Blocking time] $B_i = \max_{\forall j, \pi_j < \pi_i} C_j$

[Start time] $S_i^{n+1} = B_i + \sum_{\forall j, \pi_j > \pi_i} \left(1 + \left\lfloor \frac{S_i^n}{T_j} \right\rfloor \right) \cdot C_j$ ← This equation also applies to preemptive scheduling.

[Finish time] $F_i = S_i + C_i$ ← To include task arrivals up to S_i

$R_i = F_i$ ← Once a task gets CPU, it is not preempted.

[Another representation]

$$R_i^{n+1} = B_i + C_i + \sum_{\forall j, \pi_j > \pi_i} \left(1 + \left\lfloor \frac{R_i^n - C_i}{T_j} \right\rfloor \right) \cdot C_j$$

WCRT Analysis of Preemption Threshold Scheduling

- ❖ Start time S_i and finish time F_i of τ_i should be considered separately.
 - Why?: Interfering tasks changes once τ_i gets CPU.

$$B_i = \max_{\forall j, \gamma_j \geq \pi_i > \pi_j} C_j$$

$$S_i^{n+1} = B_i + \sum_{\forall j, \pi_j > \pi_i} \left(1 + \left\lfloor \frac{S_i^n}{T_j} \right\rfloor \right) \cdot C_j$$

$$F_i^{n+1} = S_i + C_i + \sum_{\forall j, \pi_j > \gamma_i} \left(\left\lfloor \frac{F_i^n}{T_j} \right\rfloor - \left(1 + \left\lfloor \frac{S_i}{T_j} \right\rfloor \right) \right) \cdot C_j$$

$R_i = F_i$

Interference time before starting Interference time after starting

Summary

- ❖ Schedulability analysis of PTS
 - Blocking due to PTS occurs only before a task starts its execution.
 - Blocking due to PTS occurs only once by one task.
 - Blocking duration of task A due to PTS is the maximum of the worst-case execution times of such tasks
 - whose priorities are lower than the priority of task A and
 - whose preemption thresholds are higher than the priority of task A.
 - Start time and finish time should be considered separately.
 - Since the interfering task set is changed before and after a task starts its execution.

Lecture 4.

Preemption Threshold Scheduling: Real-Time Synchronization

Seongsoo Hong
Real-Time Operating Systems Lab.
Seoul National University, Korea
<http://redwood.snu.ac.kr>

Lecture Outline

- ❖ Task model
- ❖ Priority inversion problem
- ❖ Preventing the priority inversion problem
- ❖ Preventing deadlock and bounding blocking delay
- ❖ Minimizing context switches and improving blocking delay bound
- ❖ Performance evaluation
- ❖ Summary

Task Model (1)

| | |
|------------|--|
| τ_i | A task |
| T_i | The period of task τ_i |
| C_i | The worst-case execution time of task τ_i |
| π_i | The fixed-priority of task τ_i |
| γ_i | The preemption threshold of task τ_i |

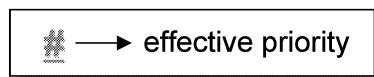
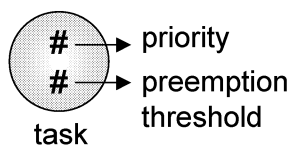
- We denote a higher priority with a larger value.
- Assumption
 - for each task τ_i , $\gamma_i \geq \pi_i$.

Task Model (2)

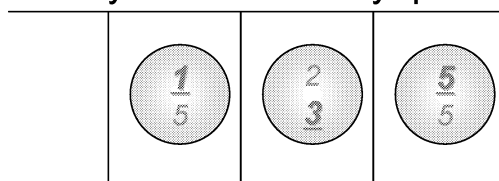
| | |
|-------|---|
| p_i | The effective priority of task τ_i |
|-------|---|

❖ Effective Priority

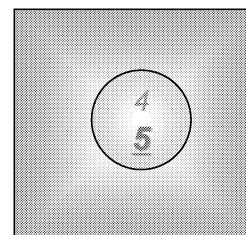
- The task priority used by the kernel scheduler for selecting a task to be dispatched.



Priority-Ordered Ready queue



CPU



Task Model (3)

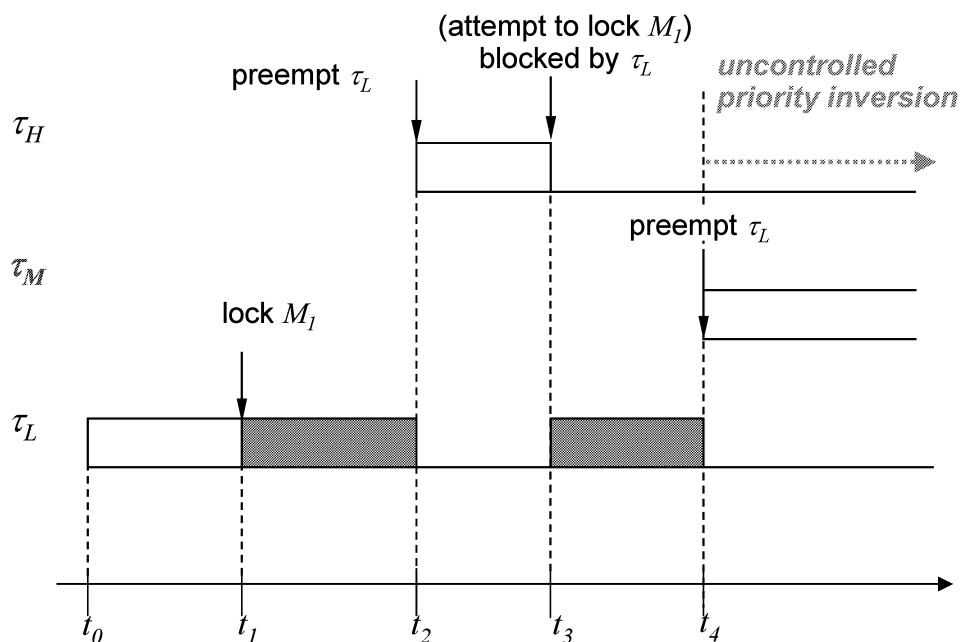
| | |
|-------|----------------------------|
| M_i | A mutex (binary semaphore) |
|-------|----------------------------|

- ❖ No voluntary blocking
 - Tasks do not suspend themselves, say for I/O operations.
- ❖ Properly nested critical sections
 - (ex)

$\tau_i = \{ \dots, P(M_1), \dots, P(M_2), \dots, V(M_2), \dots, V(M_1), \dots \}$ Properly nested semaphores

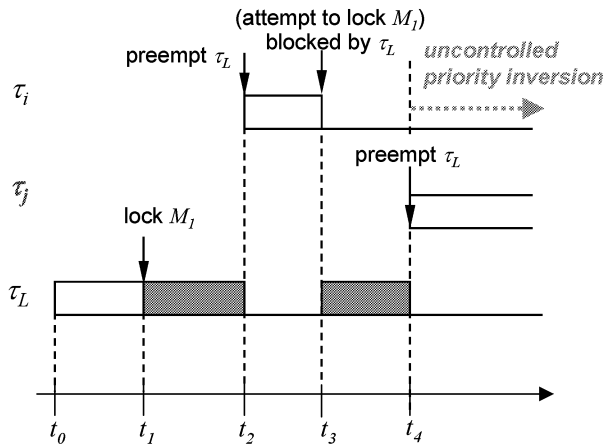
$\tau_i = \{ \dots, P(M_1), \dots, P(M_2), \dots, V(M_1), \dots, V(M_2), \dots \}$ Not properly nested semaphores

Priority Inversion Problem

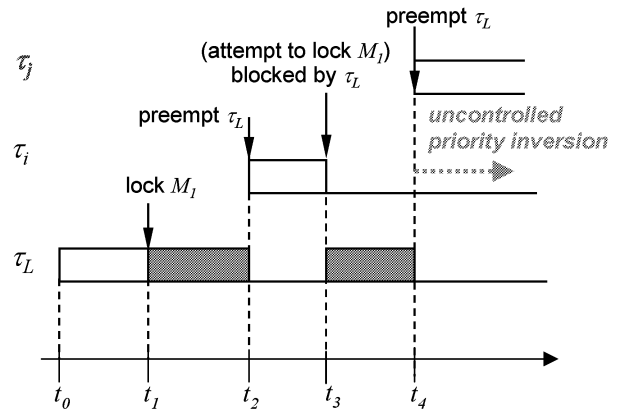


Effective Priority Inversion Problem under PTS

$$\gamma_L < \pi_j \leq \gamma_i$$



(a) Case 1.



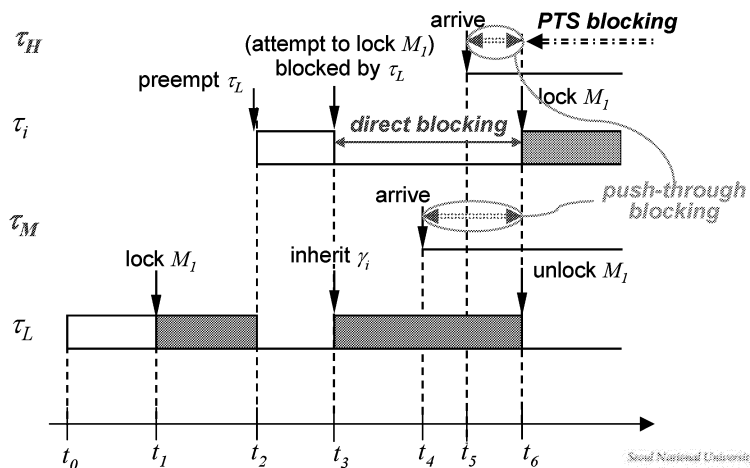
(b) Case 2.

Preventing Priority Inversion Problem under PTS

❖ Basic priority inheritance protocol (BPI) under PTS

- Basic idea: let the blocking task inherit the effective priority of the blocked task.
- Solves priority inversion problem through push-through blocking

$$\gamma_L < \pi_M < \pi_i < \pi_H \leq \gamma_i$$



BPI under PTS

❖ P1. Inheriting effective priorities.

- When task τ_L blocks task τ_H , p_L is set as p_H .
- Effective priority inheritance is transitive.
 - If task τ_L blocks τ_M and τ_M blocks τ_H , p_L is set as p_H via p_M .

❖ P2. Updating effective priorities.

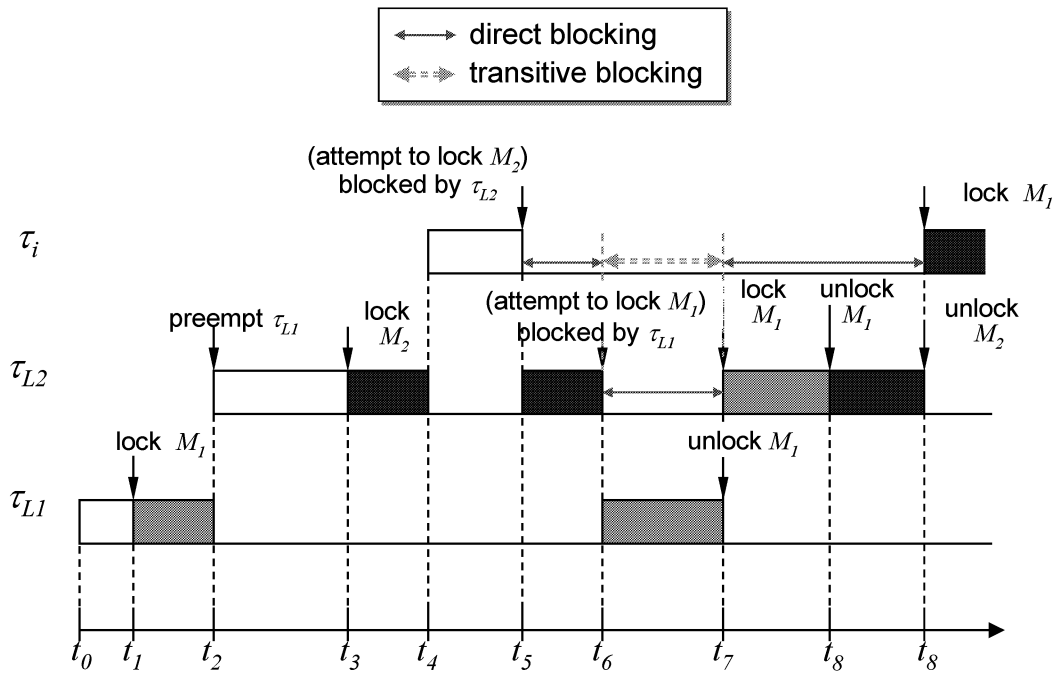
- When task τ_L exits from a critical section, it updates p_L as follows.
 - If the critical section is the outermost one, p_L is updated as γ_L .
 - Otherwise, p_L is updated as $\max(p_1, p_2, \dots, p_j)$ such that $\tau_1, \tau_2, \dots, \tau_j$ are the tasks blocked through mutexes that are still locked by task τ_L after τ_L exits from the critical section.

Blockings in BPI under PTS

❖ Four Types of blocking

- Direct blocking
 - Ensures the consistency of shared data.
- Push-through blocking
 - Prevents indefinite blocking due to priority inversion
- Transitive blocking
 - When task τ_j is blocked by task τ_M and task τ_M in turn is blocked by another task τ_L .
- PTS blocking
 - When task τ_i is blocked by τ_j where $\pi_j < \pi_i$ but $\gamma_j \geq \pi_i$.

Transitive Blocking



Scout National University

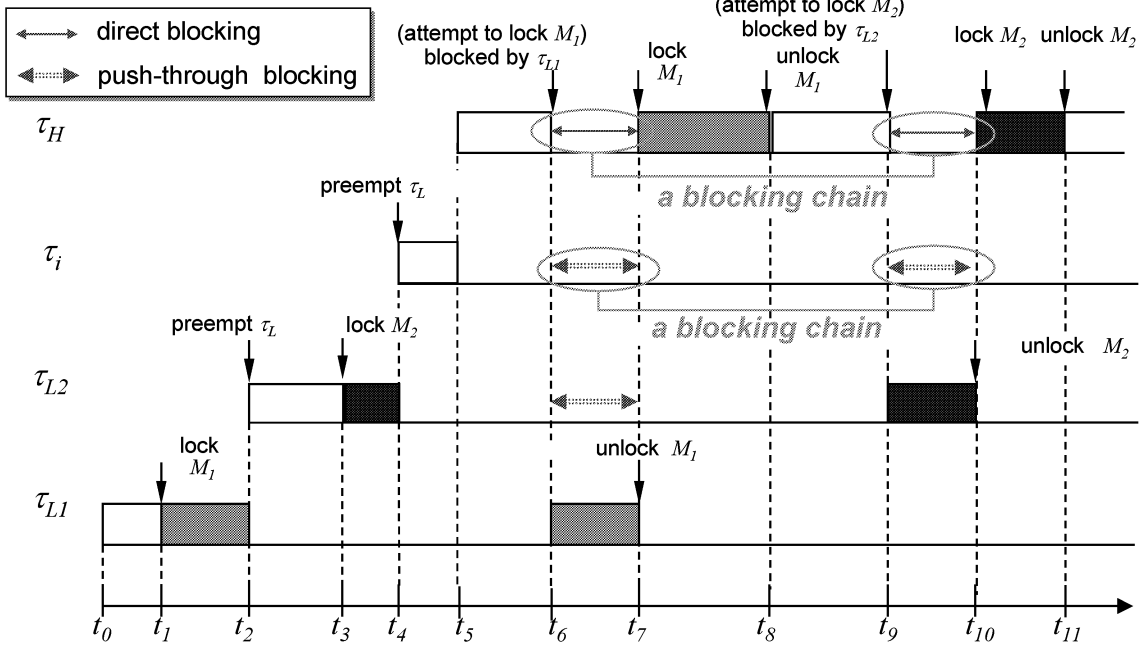
Problems of BPI

- ❖ Long blocking delay due to blocking chains (also called chained blocking)
 - Occurs when a task is repeatedly blocked by more than one task.
 - A transitive blocking always incurs a blocking chain.
 - Not a form of blocking, but refers to a situation where a task is blocked more than once.

- ❖ Deadlocks
 - Occurs when multiple tasks try to access nested mutexes in a circular manner.

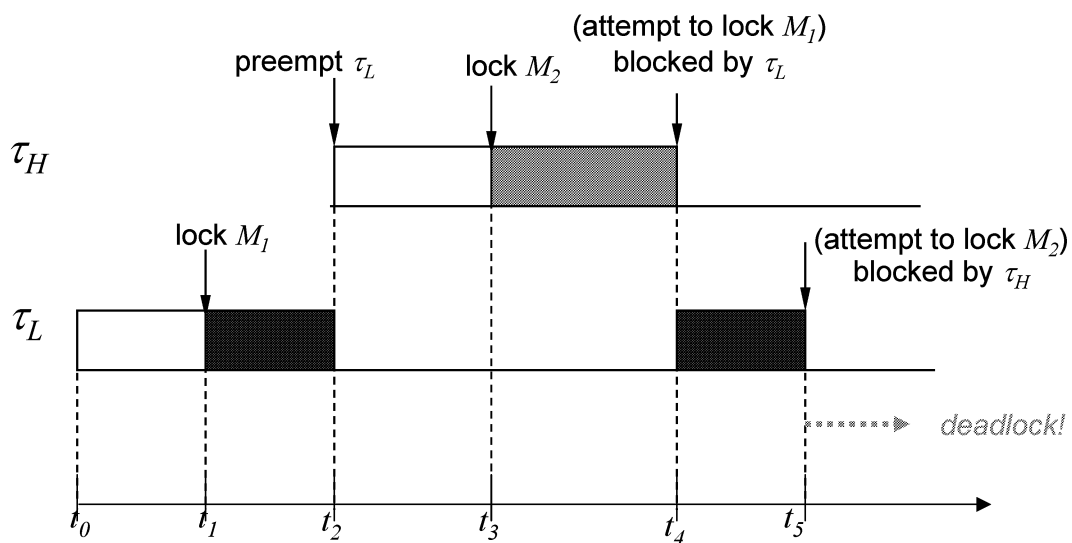
Scout National University

Chained Blocking



Scout National University

Deadlock

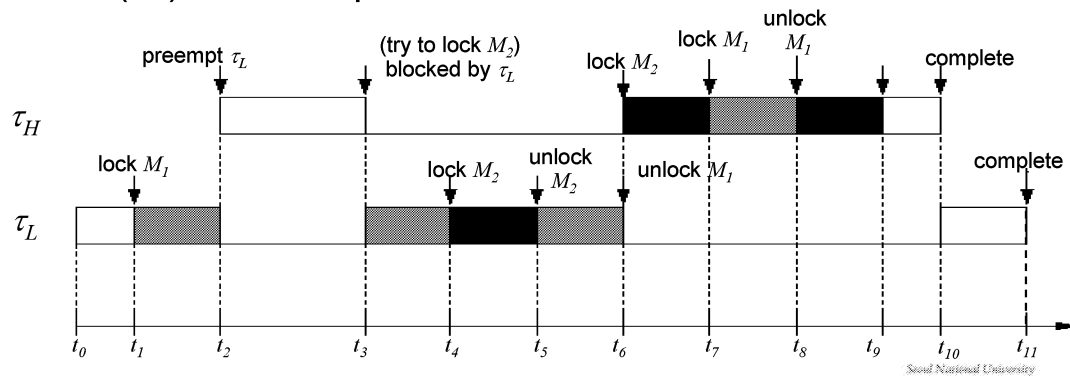


Scout National University

Preventing Deadlock and Bounding Blocking Delay

❖ Priority ceiling protocol (PCP)

- Extension of BPI
- Additional condition for allowing a task to start a new critical section: only if all mutexes that the task, as well as all higher priority tasks, may use are not locked.
- (ex) Deadlock prevention in PCP



Realization of PCP

❖ Ceiling of a mutex

- The priority of the highest priority task that may use the mutex

❖ Additional locking condition

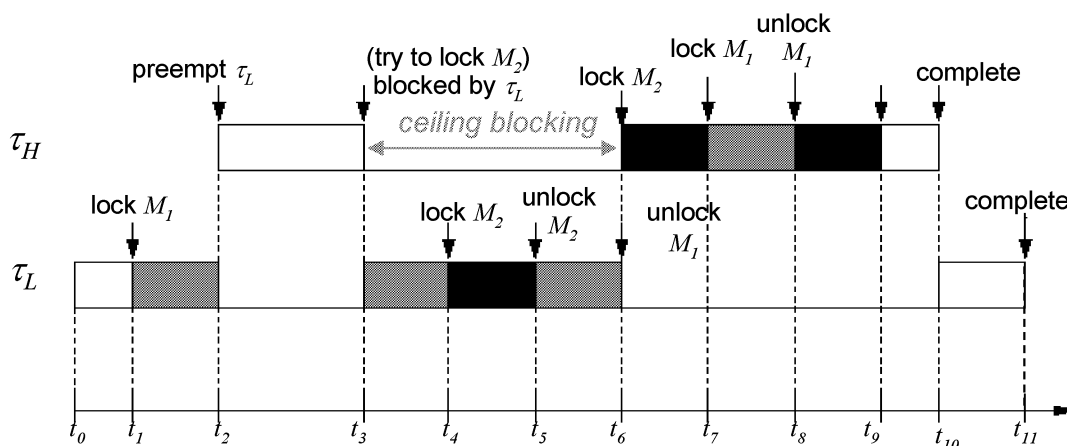
- A task τ can start a new critical section only if τ 's priority is higher than all ceilings of all the mutexes locked by tasks other than τ .

PCP under PTS

- ❖ Ceiling of a mutex
 - The priority of the highest priority task that may use the mutex
- ❖ Additional locking condition
 - A task τ can start a new critical section only if τ 's priority is higher than all ceilings of all the mutexes locked by tasks other than τ .
- ❖ Which of priority and preemption threshold should be used under PTS?
 - Both can be used!
 - { **PC-PCP** (PCP with priority ceilings) and
 - PTC-PCP** (PCP with preemption threshold ceilings)

Ceiling Blocking in PCP

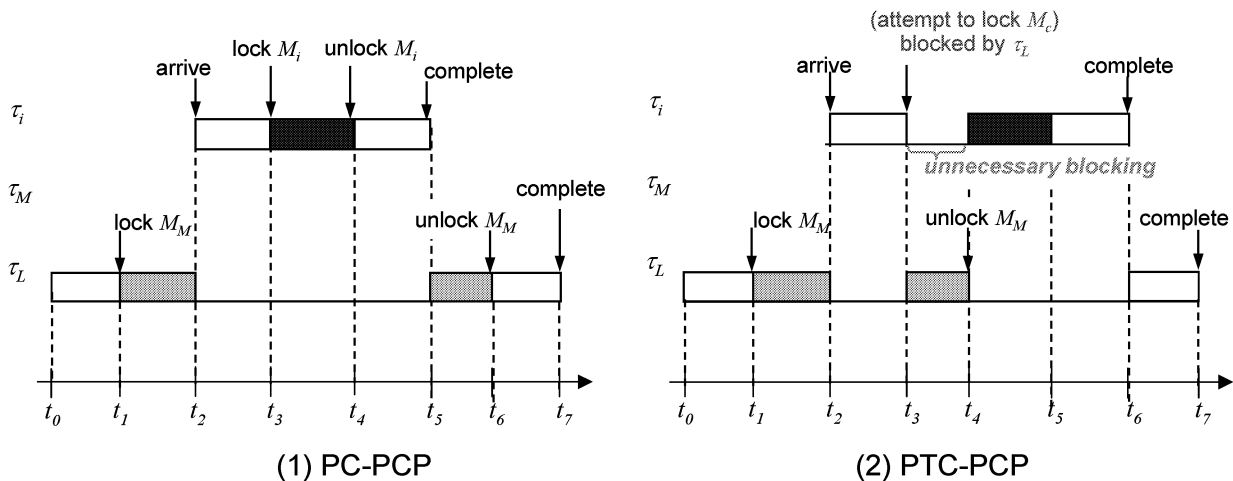
- ❖ PCP solves problems (deadlock and blocking chains) of BPI through ceiling blocking



PC-PCP vs. PTC-PCP (1/2)

❖ PTC-PCP leads unnecessary blocking

The preemption thresholds of τ_M and τ_i are equal.



Saudi National University

RTOS Lab

19

PC-PCP vs. PTC-PCP (2/2)

❖ Unnecessary blocking in PTC-PCP

- Does not contribute to the prevention of deadlock and blocking chains

❖ The policy of maximum preemption threshold assignment

- Commonly used in PTS
 - To reduce as many context switches as possible.
- Causes the unnecessary blocking of PTC-PCP to be very frequent.

→ PC-PCP is better than PTC-PCP as PCP under PTS.

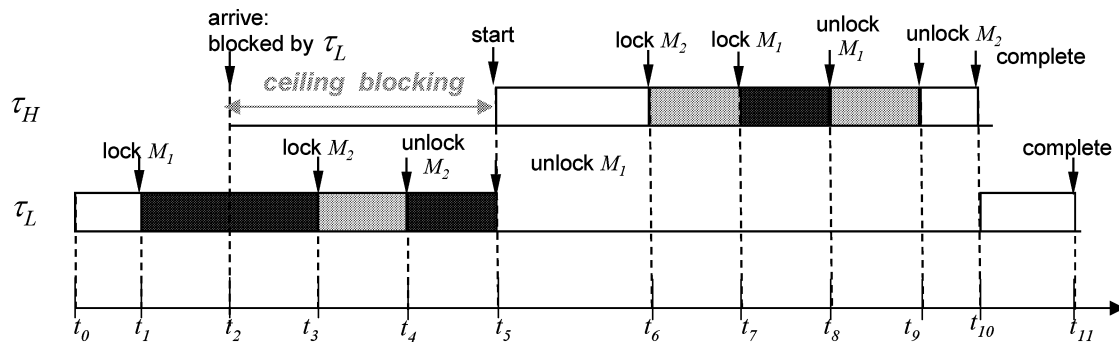
Saudi National University

RTOS Lab

20

Minimizing Context Switches

- ❖ Immediate inheritance protocol (IIP)
 - Shifting ahead the time point of applying the additional condition in PCP
 - PCP: the point of starting a new critical section →
 - IIP: the point of starting its execution
 - (Ex) Preventing deadlock in IIP



Seoul National University

Realization of IIP

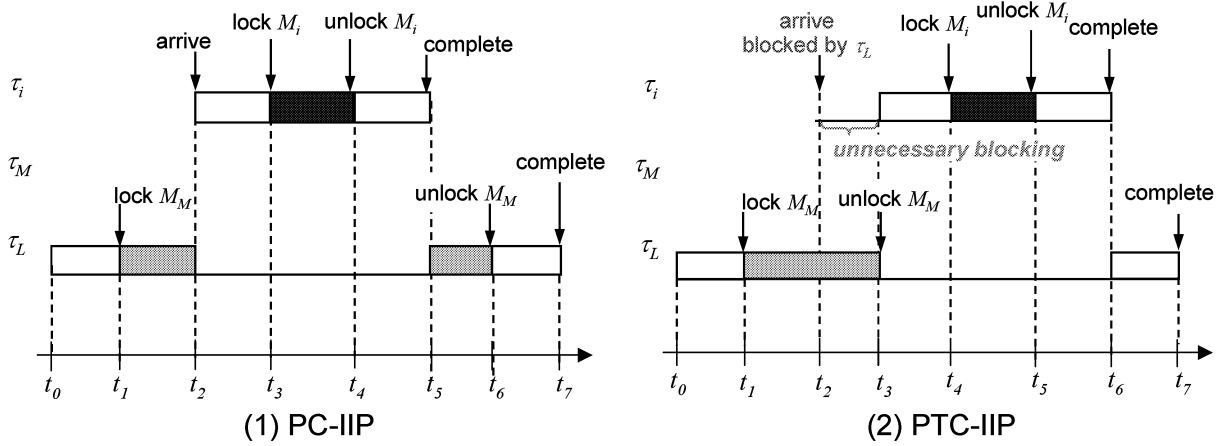
- ❖ Ceiling of a mutex:
 - The highest priority of the tasks that may use the mutex
- ❖ Immediate ceiling inheritance
 - The effective priority of task τ is set to the highest ceiling of the mutexes that τ locks currently.
- ❖ Which of priority and preemption threshold should be used under PTS?
- Both can be used:
 - PC-IIP and PTC-IIP

Seoul National University

PC-IIP vs. PTC-IIP

- ❖ PC-IIP always performs better than PTC-IIP.
 - For any task τ_i , the set of mutexes that induce ceiling blocking in PC-IIP is a subset of that in PTC-IIP.

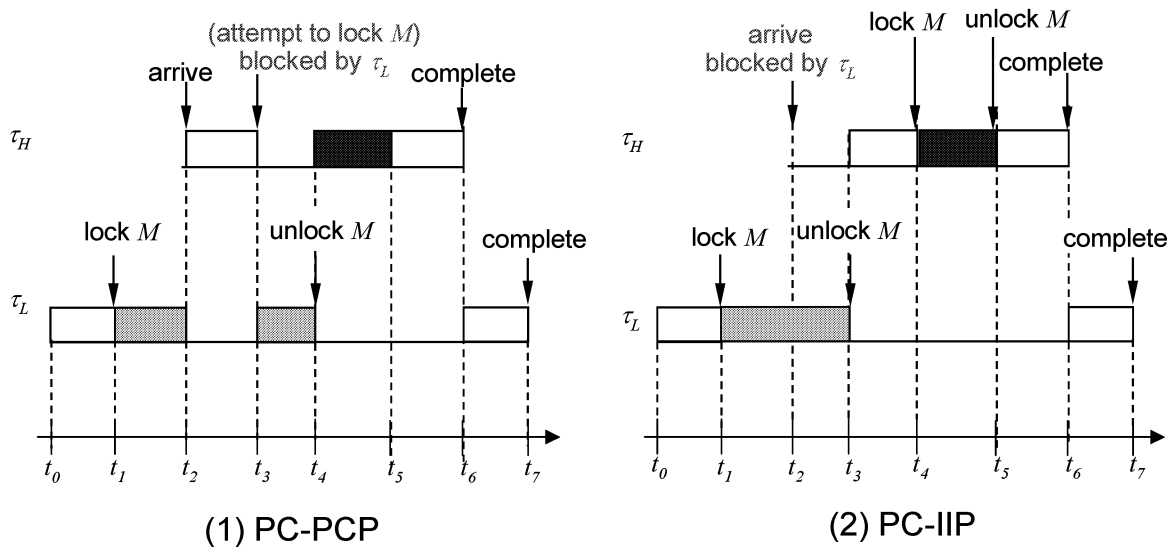
The preemption thresholds of τ_M and τ_i are equal.



Soof National University

PC-PCP vs. PC-IIP (1)

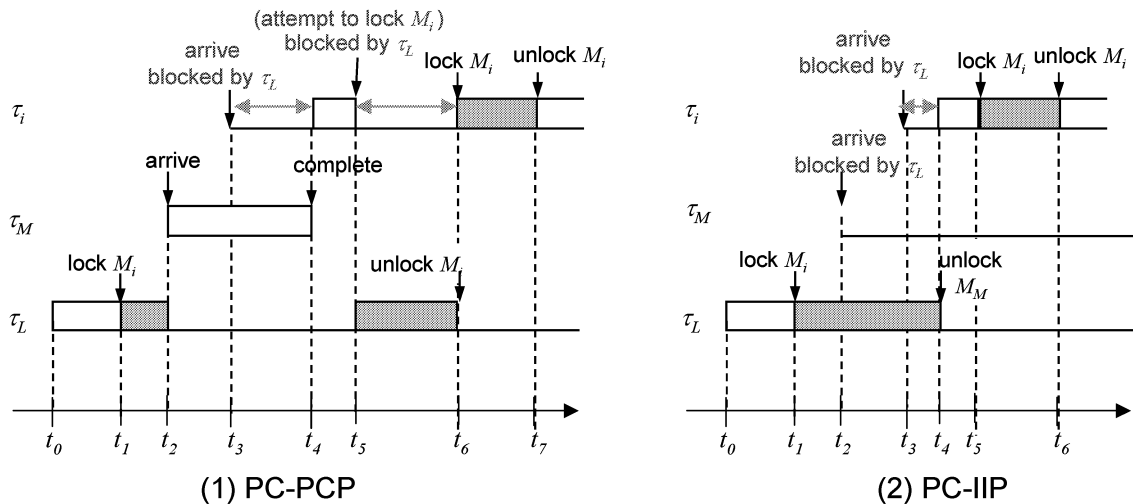
- ❖ PC-IIP leads to fewer context switches than PC-PCP in the worst case.



Soof National University

PC-PCP vs. PC-IIP (2)

- ❖ PC-IIP improves blocking delay bound over PC-PCP.
 - PC-PCP: τ_i is blocked by both τ_M and τ_L .
 - PC-IIP: τ_i is blocked by only one of τ_M and τ_L .



Scout National University

PC-PCP vs. PC-IIP (3)

- ❖ In PC-PCP, a task can be blocked twice.
 - Once before it starts its execution
 - Once after it starts its execution when it tries to lock a mutex
- ❖ In PC-IIP, a task can be blocked only once.
 - Once before it starts its execution
- ❖ The worst-case duration of blocking in PC-PCP is longer than that of PC-IIP.
 - In PC-PCP: PTS blocking duration + Ceiling blocking duration
 - In PC-IIP: Max (PTS blocking duration, Ceiling blocking duration)
- ❖ For a task that does not access any shared resource,
 - PC-IIP may cause unnecessary blocking.

Scout National University

Performance Evaluation

- ❖ Experimental setup
- ❖ Performance metrics
- ❖ Performance results
- ❖ Comparison between PC-PCP and PTC-PCP
- ❖ Comparison between PC-IIP and PTC-IIP
- ❖ Comparison between PC-PCP and PC-IIP

Experimental Setup

- ❖ Example task set: Hiker's Buddy application
- ❖ Generated a large number of mutex assignments while varying the number of mutexes up to four.
 - For each mutex assignment, assigned maximum possible preemption thresholds
- ❖ Release time variance: 10, run length = 10,000,000

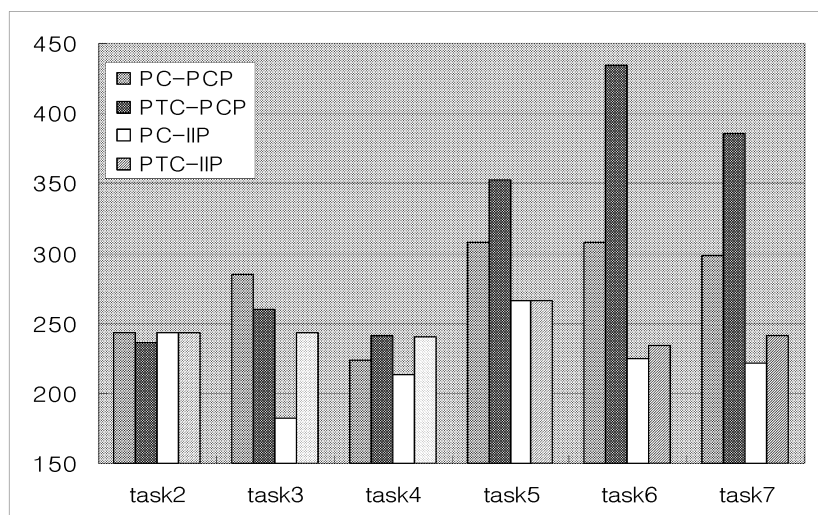
| Task Id | Period (10us) | WCET (10us) | Deadline (10us) | Prio. | Thres. | Mutexes |
|---------|---------------|-------------|-----------------|-------|--------|------------|
| 1 | 15000 | 450 | 15000 | 1 | 1 | M_1, M_3 |
| 2 | 2000 | 135 | 2000 | 2 | 7 | M_1 |
| 3 | 5000 | 270 | 5000 | 3 | 3 | M_2 |
| 4 | 3000 | 270 | 1500 | 4 | 5 | M_2 |
| 5 | 10000 | 225 | 2000 | 5 | 7 | M_3 |
| 6 | 10000 | 450 | 1000 | 6 | 6 | M_4 |
| 7 | 4000 | 315 | 700 | 7 | 7 | M_4 |

Performance Metrics

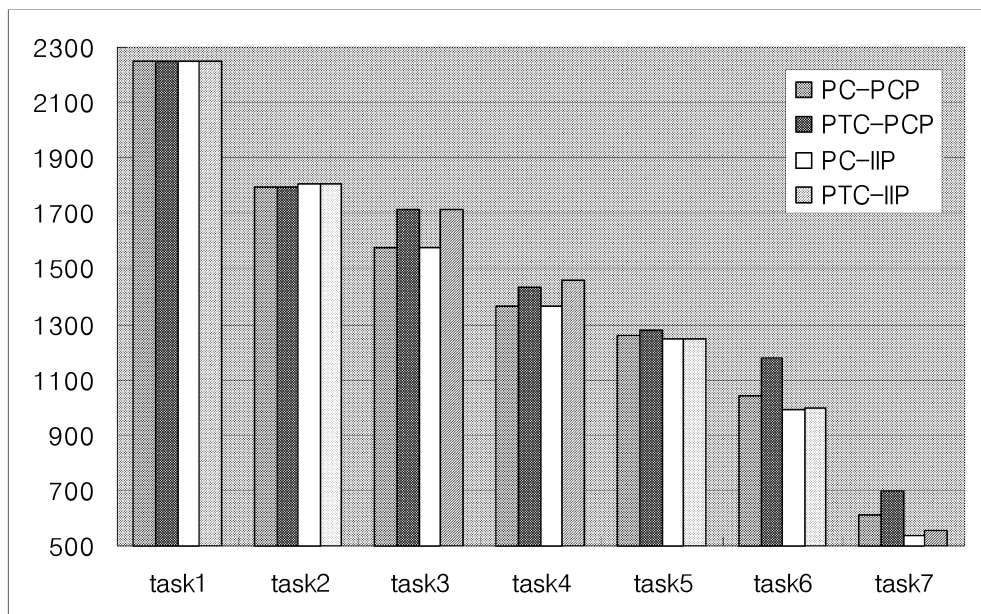
- ❖ Maximum blocking times
- ❖ Maximum response times
- ❖ Number of context switches

Performance Results (1): Maximum Blocking Times

- ❖ The blocking time of the lowest priority task (task 1) is omitted since it is always zero.

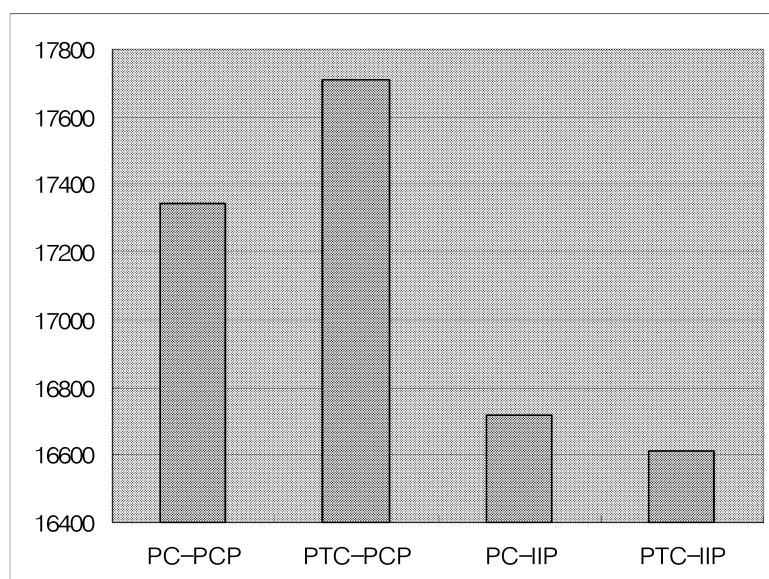


Performance Results (2): Maximum Response Times



Seoul National University

Performance Results (3): Number of Context Switches



Seoul National University

Comparison Between PC-PCP And PTC-PCP

- ❖ The maximum blocking times of high priority tasks (tasks 4, 5, 6, and 7) in PC-PCP are shorter than those in PTC-PCP.
 - For tasks 2 and 3, PTC-PCP leads shorter blocking time
 - The preemption threshold of task 2 is usually assigned the highest system priority.
 - When task 2 is blocked in PC-PCP, task 3 also can be blocked due to push-through blocking.
- ❖ The response times of all tasks in PC-PCP are not longer than those in PTC-PCP.
 - For lower priority tasks, interference times are the dominant term composing response times.
- ❖ The number of context switches in PC-PCP is smaller than that in PTC-PCP.

Comparison Between PC-IIP And PTC-IIP

- ❖ The maximum blocking times of all tasks in PC-IIP are not longer than those in PTC-IIP.
- ❖ The maximum response times of all tasks in PC-IIP are also not longer than those in PTC-IIP.
- ❖ The number of context switches in PTC-IIP is smaller than that in PC-IIP.
 - Any task that is holding a lock on a mutex is less likely to be preempted in PTC-IIP than PC-IIP since the ceiling value of each mutex in PTC-IIP is always higher than that of PC-IIP.

Comparison Between PC-PCP And PC-IIP

- ❖ The number of context switches in PC-IIP is in fact significantly smaller than that in PC-PCP.
- ❖ The blocking times of all tasks in PC-IIP are also not longer than those in PC-PCP.
- ❖ The response times of higher priority tasks (tasks 5, 6, and 7) in PC-IIP are shorter than those in PC-PCP.
 - The response times of other lower priority tasks in PC-IIP are the same as or slightly longer than those in PC-PCP.

Summary

- ❖ BPI under PTS
 - Effective priority inheritance
 - Prevents effective priority inversion problem through push-through blocking
- ❖ PCP under PTS
 - Extension of BPI with additional condition for mutex locking
 - Prevents deadlock and blocking chains through ceiling blocking
- ❖ IIP under PTS
 - Simplification of PCP with immediate ceiling inheritance
 - Minimizes context switches and improves blocking delay bound.
- ❖ Adoption of priority ceilings rather than the adoption of preemption threshold ceilings is better.

Lecture 5: Object-Oriented Design with Preemption Threshold Scheduling

Seongsoo Hong

Real-Time Operating Systems Lab.

Seoul National University, Korea

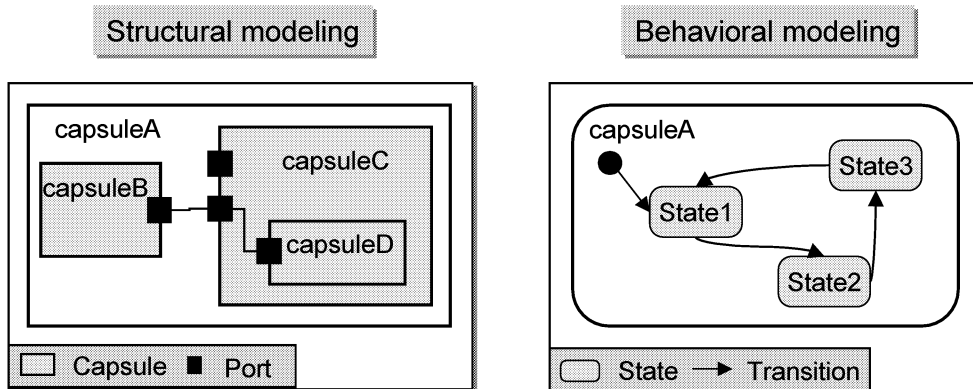
<http://redwood.snu.ac.kr>

Lecture Outline

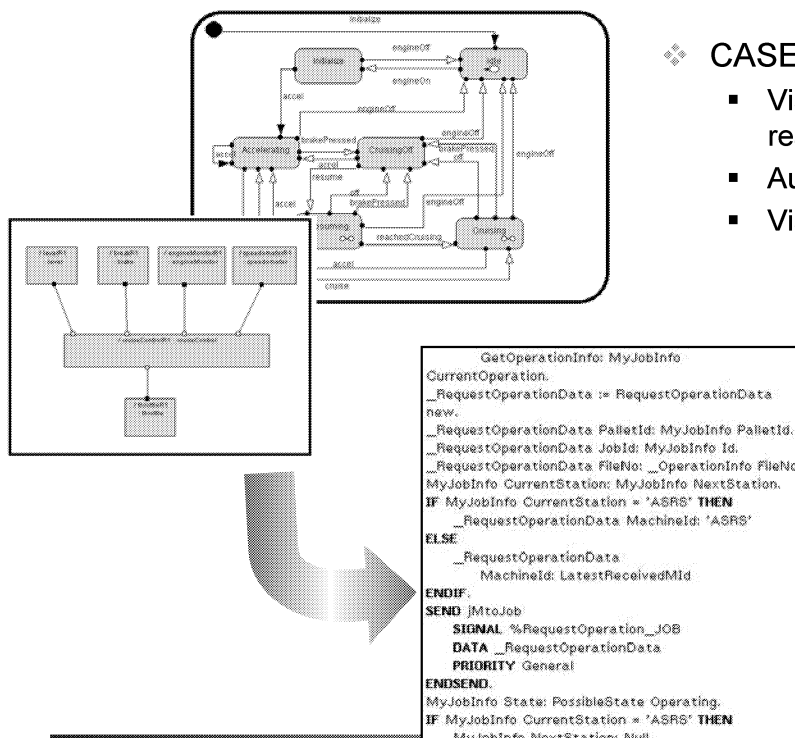
- ❖ UML-RT and its CASE tool
- ❖ OO design and RT system: the problems
- ❖ Schedulability-aware scenario-based multithreading of UML-RT models
- ❖ A case study: soccer robot system

UML-RT Modeling

- ❖ Real-time object-oriented modeling
- ❖ Capsule: object associated with ports
- ❖ Ports: abstracted communication pattern (protocol)
 - Models communication path instead of communication flow

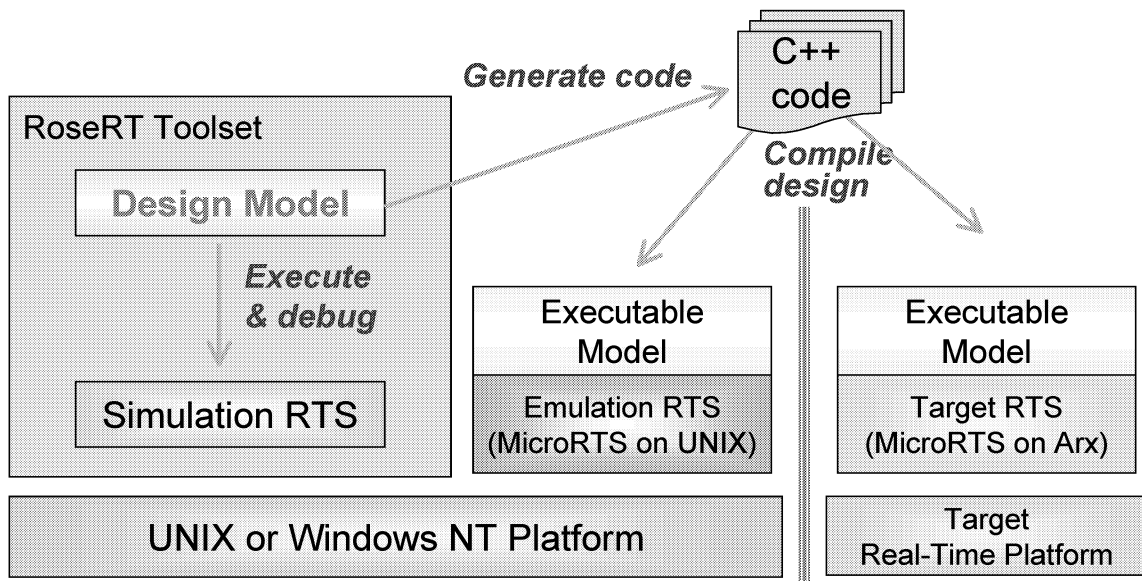


RoseRT (1)



- ❖ CASE tool based on UML-RT
 - Visual modeling of event-driven real-time system
 - Automatic generation of code
 - Visual execution of model

RoseRT (2)



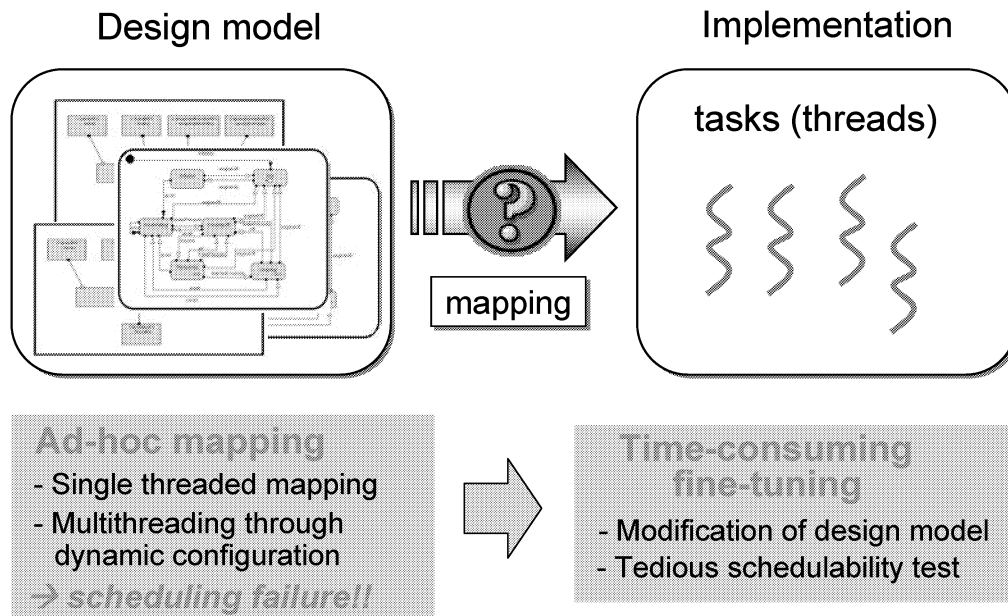
RTS: Run-Time System (Library)

OO Design and RT System

- ❖ The problem of scalability
 - Automated implementation from the OO RT design usually incurs a large number of tasks.
 - The benefits of PTS help increasing scalability.

- ❖ The problem of task identification
 - Inherent discrepancies between objects and tasks
 - It is hard to derive tasks while maximizing real-time schedulability.
 - We need a schedulability-aware mapping method.

Task Identification Problem



Traditional Multithreading

❖ Capsule-based multithreading

- Map all messages associated with an object to a single thread.
- Programmers need to assign priorities both to each message and each thread.

❖ Direct multithreading specification in application models

- Programmers should modify both structural design and behavioral design models to support multithreading.

Drawbacks of Capsule-Based Multithreading

- ❖ May degrade the performance of real-time systems by extending blocking time unnecessarily.

- ❖ Blocking due to inter-thread message passing

- Can be bounded as once for each task if IIP (Immediate Priority Inheritance Protocol) is adopted.

- ❖ Blocking due to run-to-completion semantics

- Can be neither eliminated nor bounded as once!

Drawbacks of Direct Multithreading Specification

- ❖ Application design models are blurred to support multithreading.
 - Designs and implementations are not separated.
 - It is difficult to recognize implementation models.
 - The process of multithreading specification is tedious and error-prone.
- ❖ It is hard to specify deadline and priority.
 - Deadline and priority should be specified in units of end-to-end computations.
 - The UML-RT meta-model does not contain end-to-end computations as a modeling entity.

Goals and the Approach

Schedulability-aware scenario-based multithreading of OO design models

1. A multithreaded implementation method that can minimize unnecessary blocking

Scenario-based multithreading

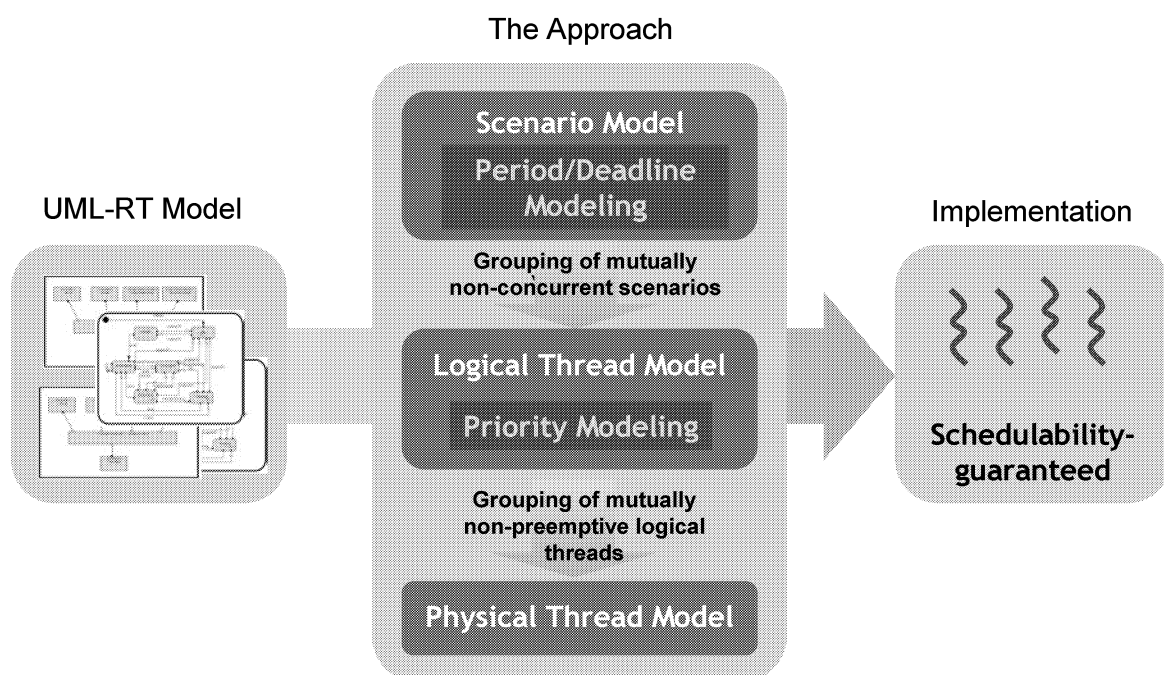
2. A model transformation method

Multithread modeling through intermediate models

3. Automated generation of schedulability-guaranteed implementations

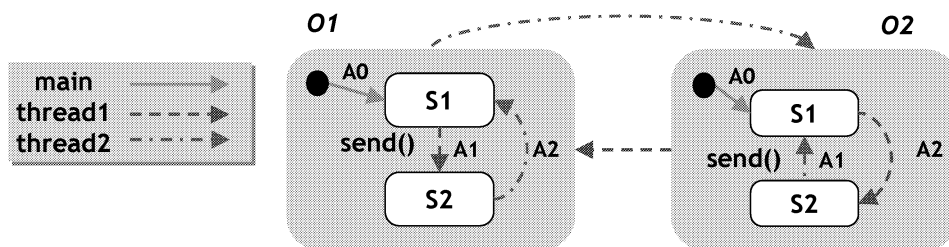
Automated assignment of priority and preemption threshold

Overview of the Approach



Scenario-Based Multithreading

- ❖ Map all events in a scenario to a single thread
- ❖ Scenario: a sequence of actions triggered by an external event
- ❖ Threading model functions orthogonal to the UML-RT model.
- ❖ Threads are permitted to traverse multiple capsules.
- ❖ The thread priorities are dynamically managed based on executing scenarios.



Why Scenario-Based Multithreading?

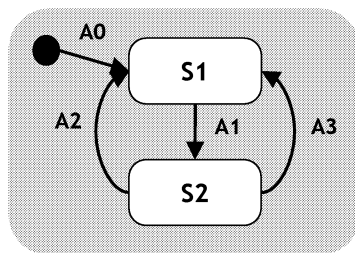
- ❖ Motivations
 - It is natural to map a unit of independent execution flow to a single thread.
 - Timing constraints should be specified to end-to-end computations.
- ❖ Benefits
 - Blocking due to inter-thread message passing can be eliminated.
 - Blocking due to run-to-completion semantics can be also bounded as once.
 - With intermediate models, programmers can explicitly specify characteristics of tasks or threads.

Guaranteeing Run-To-Completion Semantics (1)

❖ Run-to-completion semantics

- Each object with FSM has a global state.
- State transitions within the same FSM should not be interleaved, but must be synchronized with one another.

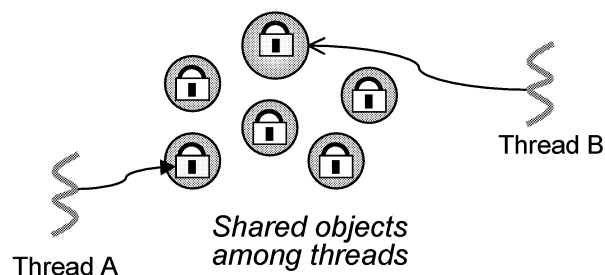
01



For example, while A1 is executing, any actions within the same object from A0 to A4 cannot execute.

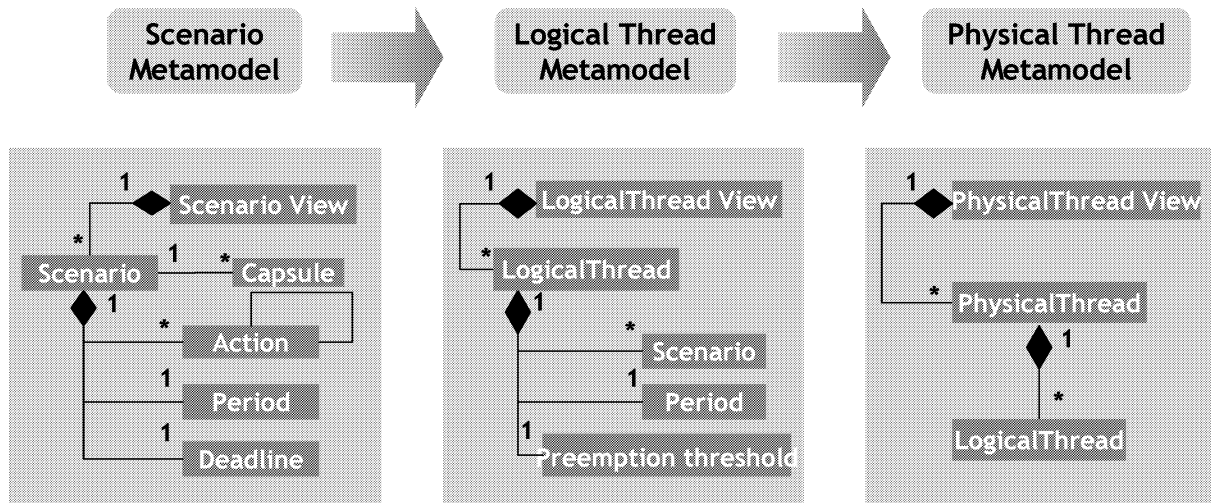
Guaranteeing Run-To-Completion Semantics (2)

- ❖ In scenario-based multithreading, multiple threads may try to make state transitions of one capsule.
- ❖ Per-capsule mutex guarding each state transition is required.
 - Class for Capsule and main loop was modified.



- ❖ PC-IIP (Immediate inheritance protocol with priority ceilings) was adopted.

Metamodels for Intermediate Models



Scenario Model (1)

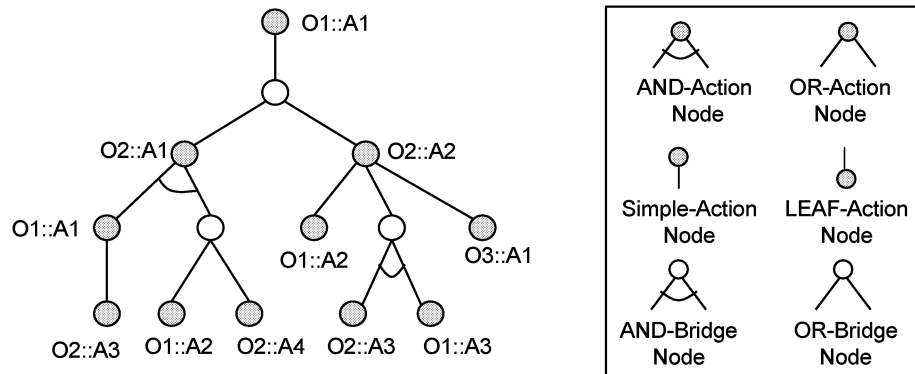
Project path:

| ID: | Capsule Role: | Port: | Signal: | WCET: | Period: | Deadline |
|-----|-----------------|----------------|----------------|-----------|------------|------------|
| 1 | AlarmController | portTiming | timeout | 131574 | 9000000 | 9000000 |
| 2 | LED | portTiming | timeout | 185377920 | 1500000000 | 1500000000 |
| 3 | KeyPad | portTiming | timeout | 36830976 | 300000000 | 300000000 |
| 4 | AlarmHardware | portHWKeyPress | key | 61985920 | 510000000 | 400000000 |
| 5 | AlarmHardware | portHWMotion | motionDetected | 18830976 | 150000000 | 50000000 |
| 6 | InputGenerator | portTiming | timeout | 38061952 | 320000000 | 140000000 |

Scenario Model (2)

❖ AND-OR action tree

- An auxiliary scenario model that shows execution path



Logical Thread Model

| ID: | Priority: | Thresh: |
|-----|-----------|---------|
| 1 | 6 | 6 |
| 2 | 4 | 5 |
| 3 | 2 | 6 |
| 4 | 1 | 4 |
| 5 | 5 | 5 |
| 6 | 3 | 4 |

❖ Non-concurrent scenarios:

- Scenarios that cannot run concurrently.
- (Ex) In a cruise control, “start a cruise mode” and “start a manual mode”
- Model transformer in default assumes that scenarios that start from the same capsule are not concurrent

Automated Priority and Preemption Threshold Assignment

❖ Two sub-steps

1. For each logical thread, assigns a feasible priority and a preemption threshold.
 - Preemption threshold
 - A run time priority of a task
 - Schedulability analysis algorithm with logical threads
2. For each logical thread, assigns possible maximum preemption threshold
 - To reduce the number of context switches

Physical Thread Model

Generate Physical Thread Model

```
numberofgroups=3
Group_idx  Members
1          2      3    4    5
2          5
3          1
```

- ❖ Mutually non-preemptive relationship
 - Tasks cannot preempt one another
- ❖ Benefits from grouping
 - To reduce the number of threads significantly
 - Thus, reduce the run-time context switching overhead and the static memory resource demands

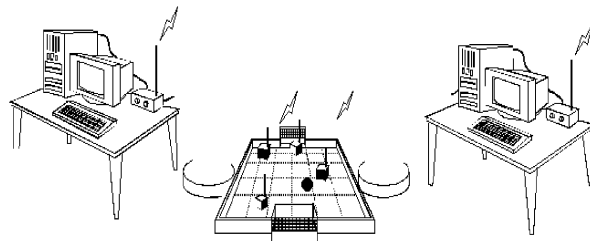
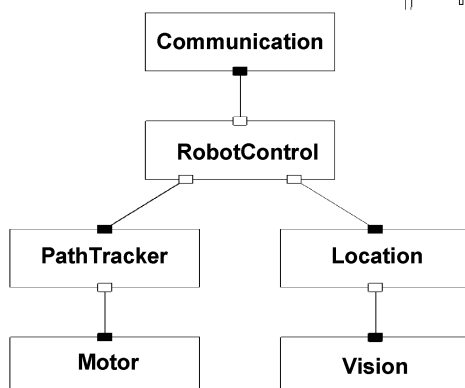
A Case Study

- ❖ Soccer robot system
- ❖ Single-thread mapping
- ❖ Capsule-based thread mapping
- ❖ Mapping using the approach

Soccer Robot System

- ❖ A robot soccer
 - A game in that two teams of autonomous robots compete with each other to get goals

- ❖ System structure



Single-Thread Mapping

- ❖ It is impossible to make the robot run:
 - Due to Run-to-completion semantics.
 - The `Motor` capsule cannot control the motors until other capsules complete the job.
- ❖ Solution
 - Re-design the model till it works again and again.
 - Break lengthy actions into small fragments.

Capsule-Based Thread Mapping

- ❖ Example mapping:

| Capsule | Thread | Priority | Period |
|---------------|------------|----------|--------|
| Communication | th_{com} | 1 | 500 ms |
| Vision | th_{vis} | 2 | 400 ms |
| Location | th_{vis} | 2 | 400 ms |
| Motor | th_{mot} | 4 | 5 ms |
| PathTracker | th_{mot} | 4 | 5 ms |
| RobotControl | th_{rot} | 3 | 100 ms |

- ❖ Response time analysis of `RobotControl` capsule's transaction initiated by `timeout` message:

$$\begin{aligned}
 R_{Robot} &= (C_{Shoot:location}^{Robot} + C_{timeout}^{Vision} + C_{Shoot:location}^{Robot} + C_{setPath}^{Path}) \\
 &\quad + (C_{Shoot:timeout}^{Robot} + C_{Shoot:Readyentry}^{Robot} + C_{requestLocation}^{Location} + C_{Shoot:location}^{Robot} + C_{setPath}^{Path}) \\
 &\quad + \left[\frac{R_{Robot}}{T_{mot}} \right] \cdot (C_{timeout}^{Motor} + C_{movement}^{Path} + C_{setPath}^{Motor}) \\
 &= 201.8 > \text{Deadline (100)}
 \end{aligned}$$

- ❖ Problems
 - Should do thread mapping and priority assignment manually.
 - No automatic scheduling test support

Mapping Using the Approach

❖ Resultant thread set and their attributes

| Physical Thread | Logical Thread | Priority | Threshold |
|-----------------------|----------------------------------|----------|-----------|
| <i>Ph₁</i> | <i>L_{Motor}</i> | 4 | 4 |
| <i>Ph₂</i> | <i>L_{RobotControl}</i> | 3 | 3 |
| | <i>L_{Communication}</i> | 1 | 3 |
| <i>Ph₃</i> | <i>L_{Vision}</i> | 2 | 2 |

- Feasible thread assignment using only three threads
- Fully automatic thread and priority assignment

Summary

❖ OO design for RT system has the problems of (1) scalability and (2) task identification

- PTS help increasing scalability.
- The proposed schedulability-aware scenario-based multithreading method automates task identification.

❖ Scenario-based multithreading

- Performs better than traditional capsule-based multithreading.
- Combined with model transformation and PTS, makes it easy to generate schedulability-guarantee executable.
- Run-to-completion semantics was maintained through the adoption of IIP with priority ceilings under PTS.

Sponsored by:

