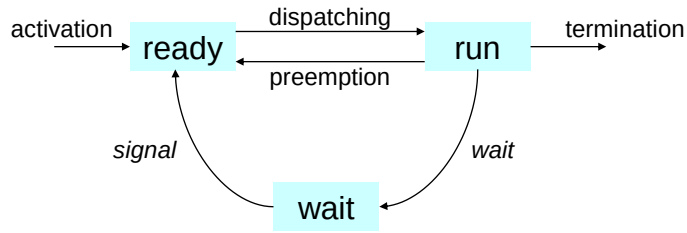
REVIEW: scheduling independent and preemptable periodic tasks

- **Earliest deadline first:**
  - Optimal solution among all dynamic-priority schedulers
  - Schedulability guaranteed if processor utilization $U \leq 1$.
- **Rate monotonic scheduling:**
  - Optimal solution among all fixed-priority schedulers
  - Schedulability of n tasks guaranteed, if processor utilization $U \leq n(2^{1/n} - 1)$.
  - Schedulability check:

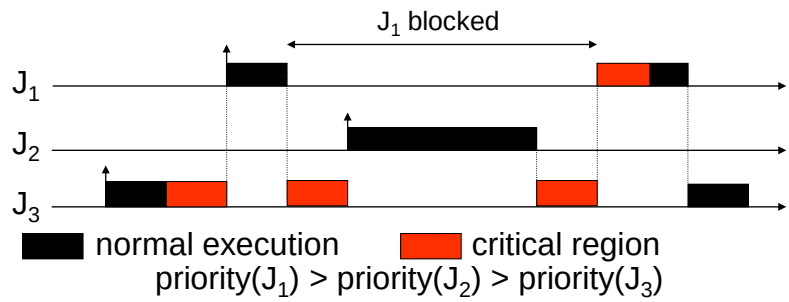
$\forall$ i: $R_i^{(0)} = C_i$
repeat
  $\forall$ i: $R_i^{(j+1)} = C_i + \sum_{k=1}^{i-1} \lceil R_i^{(j)} / T_k \rceil \cdot C_k$
until ($\exists$ i with $R_i^{(j+1)} > D_i$) **or** ($\forall$ i $R_i^{(j+1)} = R_i^{(j)}$);
if ($\forall$ i $R_i^{(j+1)} = R_i^{(j)}$) **then**
  report ("RM schedulable");

REVIEW: Wait state caused by resource constraints

- Each mutually exclusive **resource** R_i is protected by a **semaphore** S_i.
- Each critical section operating on R_i must begin with a *wait*(S_i) primitive and end with a *signal*(S_i) primitive.
- *wait* primitive on locked semaphore
  → wait state until another task executes signal primitive



REVIEW: The priority inversion problem



- Blocking time equal to length of critical section + computation time of J_2.
- Unbounded time of priority inversion, if J_3 is interrupted by tasks with priority between J_1 and J_3 during its critical region.

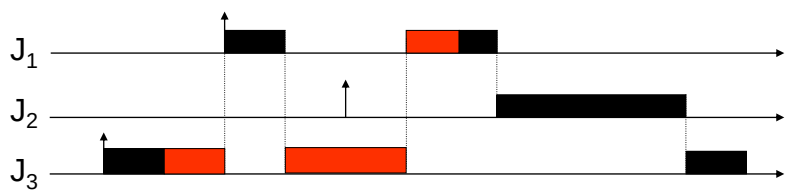
REVIEW:
The priority inheritance protocol

Idea of **priority inheritance protocol**:

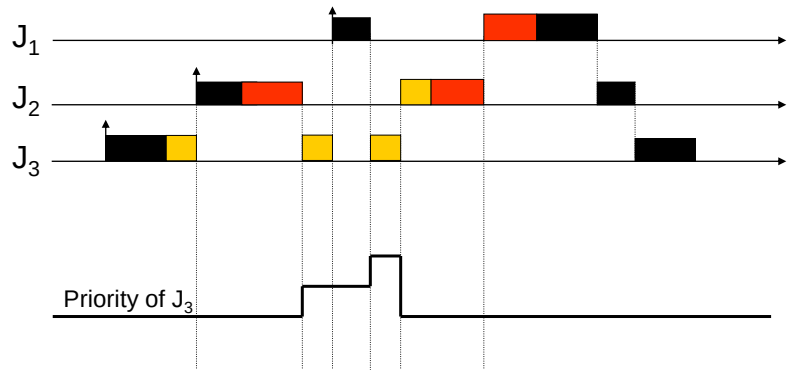
- If a task J_h blocks, since another task J_l with lower priority owns the requested resource, then J_l inherits the priority of J_h.
- When J_l releases the resource, the priority inheritance from J_h is undone.
- **Rule:** Tasks always inherit the highest priority of tasks blocked by it.

REVIEW: Direct vs. push-through blocking

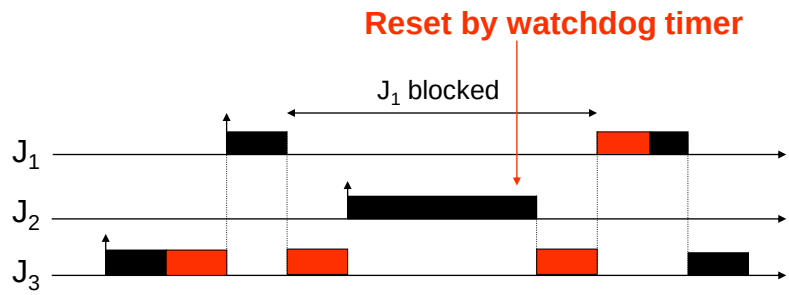
- **Direct blocking:** High-priority job tries to acquire resource already held by lower-priority job
- **Push-through blocking:** Medium-priority job is blocked by lower-priority job that has inherited a higher priority.



REVIEW: Transitive priority inheritance



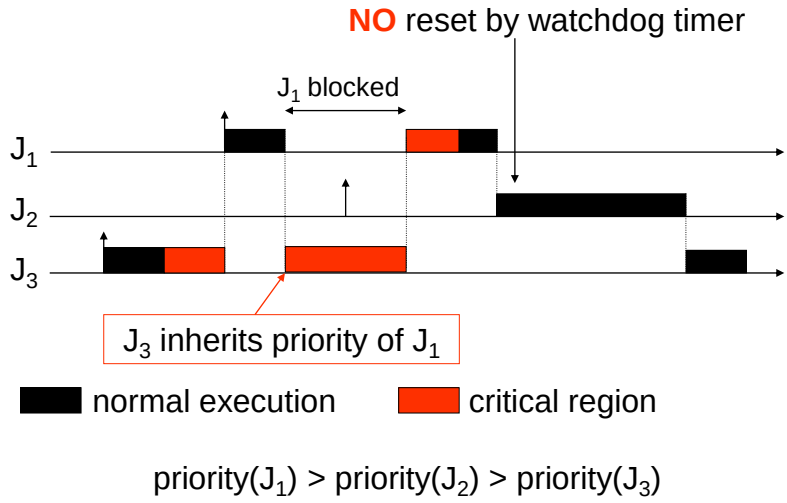
REVIEW:
The MARS Pathfinder problem



normal execution critical region

priority(J_1) > priority(J_2) > priority(J_3)

Priority inheritance for the Pathfinder example



priority(J_1) > priority(J_2) > priority(J_3)

Schedulability check

Let B_i be the maximum blocking time due to lower-priority jobs that a job J_i may experience.

$\forall$ i: $R_i^{(0)} = C_i$

repeat

  $\forall$ i: $R_i^{(j+1)} = C_i + B_i + \sum_{k=1}^{i-1} \lceil R_i^{(j)} / T_k \rceil \cdot C_k$

until ($\exists$ i with $R_i^{(j+1)} > D_i$) **or** ($\forall$ i $R_i^{(j+1)} = R_i^{(j)}$);

if ($\forall$ i $R_i^{(j+1)} = R_i^{(j)}$) **then**

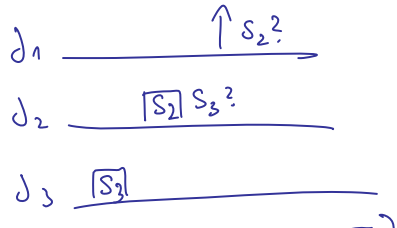
  report("RM schedulable");

Blocking Time Computation

- Precise algorithm based on exhaustive search: exponential cost

- Here: approximative solution
- Assumption: no nested critical sections

Lemma: Transitive priority inheritance can only occur in the presence of nested critical sections.



Suppose J2 blocks J1,
J3 blocks J2;
J2 holds S2
J3 holds S3
J3 attempted to lock S3 region inside the critical protected by S2.
=> J2

Blocking Time

priority ceiling C(S)=priority of the highest-priority job that can lock S

Theorem: In the absence of nested critical sections, a critical section of job J guarded by semaphore S can only block job J‘ if priority(J) < priority(J‘) ≤ C(S).

priority (J) < priority (J'):
otherwise J' cannot preempt J
priority (J') ≤ C(S)
any job that uses the S cannot have a priority > C(S)

Blocking Time

- $D_{j,k}$: duration of longest critical section of task τ_j, guarded by semaphore S_k
- **Blocking Time**
  - $B_i \leq \sum^{n}_{j=i+1} \max_k[D_{j,k} : C(S_k) \geq P_i]$
  - $B_i \leq \sum^{m}_{k=1} \max_{j>i}[D_{j,k} : C(S_k) \geq P_i]$

  where the task set consists of n periodic tasks that use m distinct semaphores.

Example

D_{ik}	S_a	S_b	S_c
τ_1	1	1	*
τ_2	*	8	2
τ_3	7	6	*
τ_4	5	4	3

D_{ik} = * : task τ_l does not use semaphore S_k

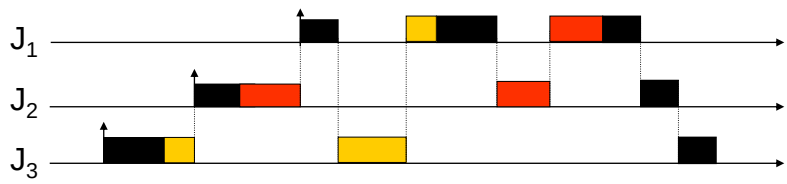
$B_1 \leq 8 + 7 + 5 = 20$
$B_1 \leq 7 + 8 = 15$
} $B_1 \leq 15$

$B_2 \leq 7 + 5 = 12$
$B_2 \leq 7 + 6 + 3 = 16$

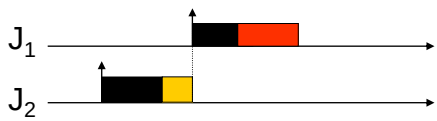
$B_3 \leq 5$
$B_3 \leq 5 + 4 + 3 = 12$
} 5

$B_4 = 0$

Problem: Chained Blocking



Problem: Deadlock

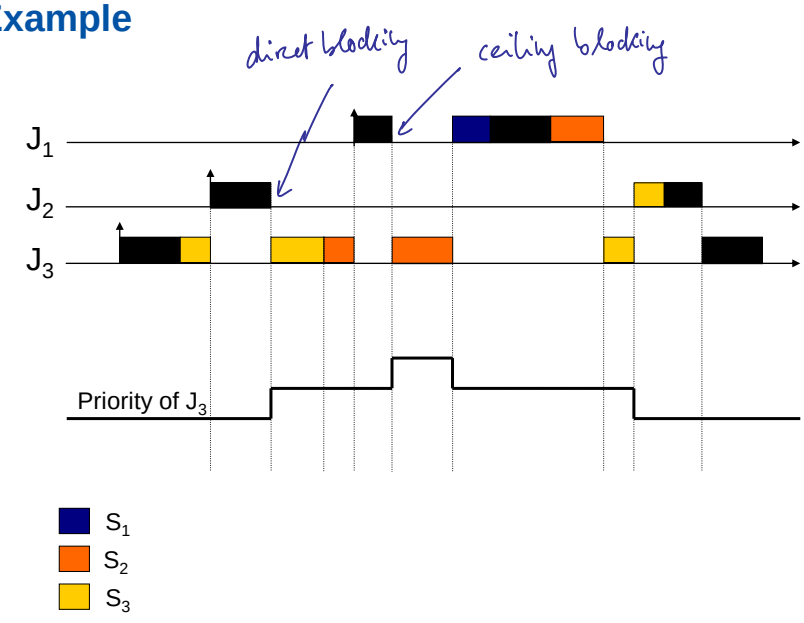


J_1:	J_2:
wait(S_a)	wait(S_b)
wait(S_b)	wait(S_a)
signal(S_b)	signal(S_a)
signal(S_a)	signal(S_b)

Priority Ceiling Protocol

- The processor is assigned to a ready job J with highest priority.
- To **enter** a critical section, J needs priority > C(S*),
  where S* is the currently locked semaphore with max C.
  → otherwise J „blocks on semaphore“ and
  priority of J is inherited by job J‘ holding S*.
- When J‘ **exits** critical section, its priority is updated to the highest priority of some job that is blocked by J‘ (or to the nominal priority if no such job exists).

Example



Priority Ceiling Protocol

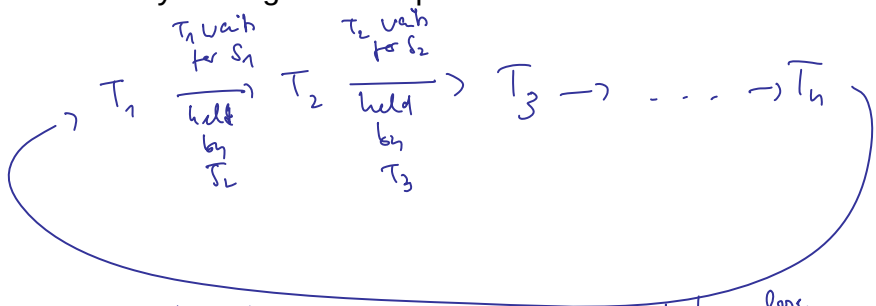
Theorem (Sha/Rajkumar/Lehoczky): Under the Priority Ceiling Protocol, a job can be blocked for at most the duration of one critical section.

- Suppose T_i is blocked by T_1, T_2 with lower priority.
- Let T_1 be the task that enter the critical region first.
- Let $C_1^* = \max_{S \text{ locked by } T_1} C(S)$
- When T_2 enters the critical section $P_2 > C_1^*$
- If T_i is blocked by T_1, $P_i \leq C_1^*$

Hence, $P_i \leq C_1^* < P_2$ ↯

Priority Ceiling Protocol

The Priority Ceiling Protocol prevents deadlocks.



Let S_j be the last semaphor acquired to close the circle

⇒ task $J_{j\oplus 1}$ must have priority $> C(S_k)$ $\forall k = 1 \ldots j-1, j+1, \ldots n$ ↯ Impossible, because $J_{j\oplus 1}$ can lock $S_{j\oplus 1}$

Incorporating aperiodic tasks

- In real systems, not all tasks are periodic
  - Environmental events to be processed
  - Exceptions raised
  - Background tasks running whenever CPU time budget permits
- Thus, real systems tend to be a **combination** of
  - periodic and
  - aperiodic tasks

  and of
  - hard real-time and
  - soft real-time tasks.

Aperiodic and periodic tasks together (1)

- **Aperiodic and periodic tasks together**
  - can be handled by dynamic-priority schedulers like EDF
- **Problem:**
  - Off-line guarantees can not be given without assumptions on aperiodic tasks.
  - If deadlines for aperiodic tasks are hard, aperiodic tasks need to be characterized by a minimum interarrival time between consecutive instances
    $\Rightarrow$ bounds on the aperiodic load
  - Aperiodic tasks with maximum arrival rate may be modeled as periodic tasks with this rate

    $\Rightarrow$ periodic scheduling
  - Aperiodic tasks with maximum arrival rate are called sporadic tasks.

Aperiodic and periodic tasks together (2)

- Other solutions for the case that periodic tasks have hard deadlines, aperiodic tasks have soft deadlines.
  - Simplest solution: **Background scheduling**
    - Aperiodic tasks are only executed when no periodic task is ready
    - Guarantees for periodic tasks do not change
    - Only applicable when load is not too high
  - **Other solutions:**
    - Define new periodic tasks, a so-called **server**
    - Aperiodic tasks are executed during “execution time” of server process
    - Independent scheduling strategies possible for periodic tasks and aperiodic tasks “inside the server”

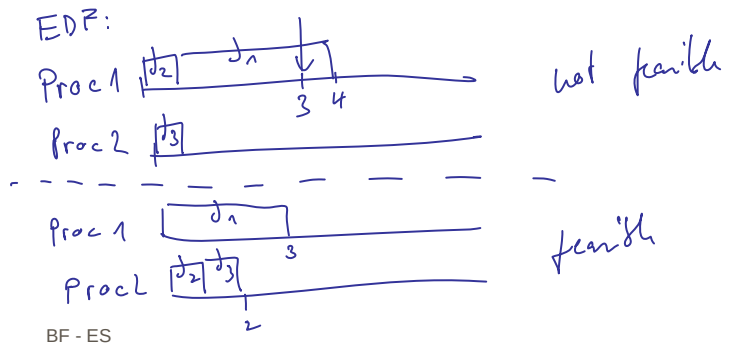
Multiprocessor scheduling

EDF with multiple processors?

$C_1 = 3, d_1 = 3$

$C_2 = 1, d_2 = 2$

$C_3 = 1, d_3 = 2$



Multiprocessor Scheduling

Given

- n equivalent processors,
- a finite set M of aperiodic/periodic tasks

find a schedule such that each task always meets its deadline.

Assumptions:

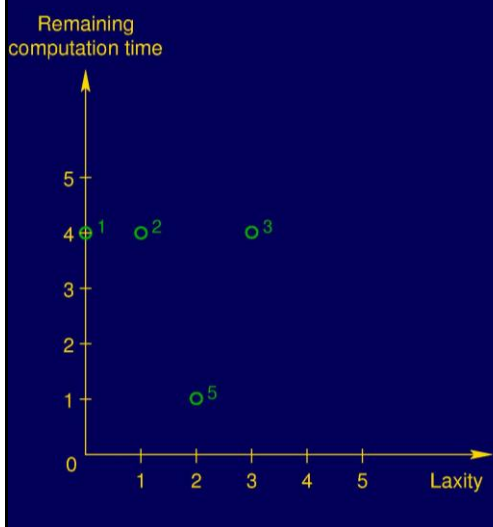
- Tasks can freely be migrated between processors
  - at any integer time instant, without overhead
  - however: no task may run on two processors simultaneously
- All tasks are preemptable
  - at any integer time instant, without overhead

Game-theoretic problem formulation

- Associate possible **states of the system** with positions on a **game board.**
- Associate **choices one can influence** in order to solve the problem with **own moves** **on the game board.**
- Associate **choices one cannot influence** with **opponent's moves.**
- Identify **feasible solutions** with **winning positions**.

Problem solution: find a winning strategy

Game-board representation



When tasks are released, they are inserted into the game board according to their WCET and laxity (= deadline – remain. comp. time).

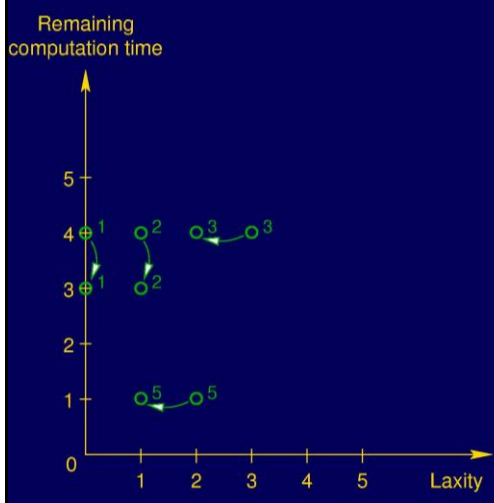
In every time scheduling step / turn of the game:
– at most n nodes go down by 1
– the rest moves 1 to the left

Nodes reaching the x–axis have been allocated all the computation time they need and are thus removed from the game board, as they don't represent scheduling constraints any longer.

The game is lost (i.e., the schedule is infeasible) if some node reaches the second quadrant.

It is won if no node remains on the board.

Game-board representation



When tasks are released, they are inserted into the game board according to their WCET and laxity (= deadline – remain. comp. time).

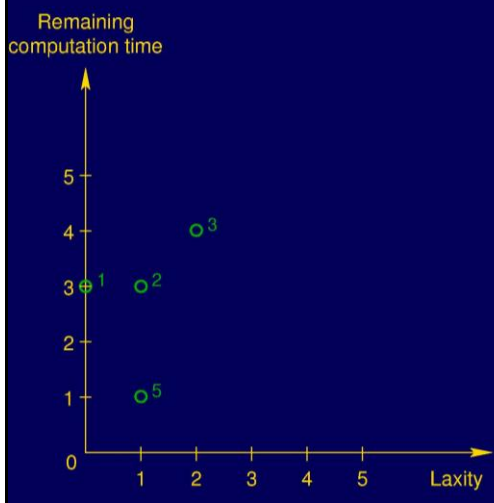
In every time scheduling step / turn of the game:
– at most n nodes go down by 1
– the rest moves 1 to the left

Nodes reaching the x–axis have been allocated all the computation time they need and are thus removed from the game board, as they don't represent scheduling constraints any longer.

The game is lost (i.e., the schedule is infeasible) if some node reaches the second quadrant.

It is won if no node remains on the board.

Game-board representation



When tasks are released, they are inserted into the game board according to their WCET and laxity (= deadline – remain. comp. time).

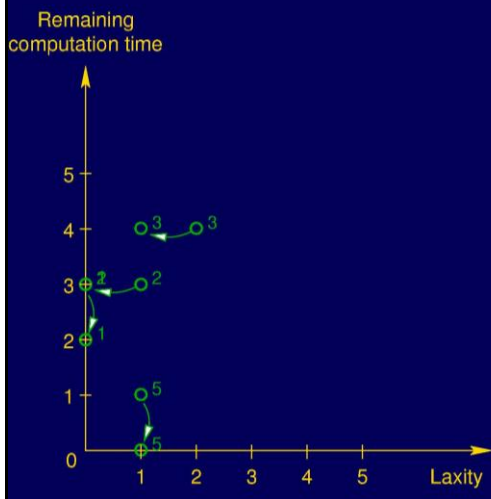
In every time scheduling step / turn of the game:
– at most n nodes go down by 1
– the rest moves 1 to the left

Nodes reaching the x–axis have been allocated all the computation time they need and are thus removed from the game board, as they don't represent scheduling constraints any longer.

The game is lost (i.e., the schedule is infeasible) if some node reaches the second quadrant.

It is won if no node remains on the board.

Game-board representation



When tasks are released, they are inserted into the game board according to their WCET and laxity (= deadline – remain. comp. time).

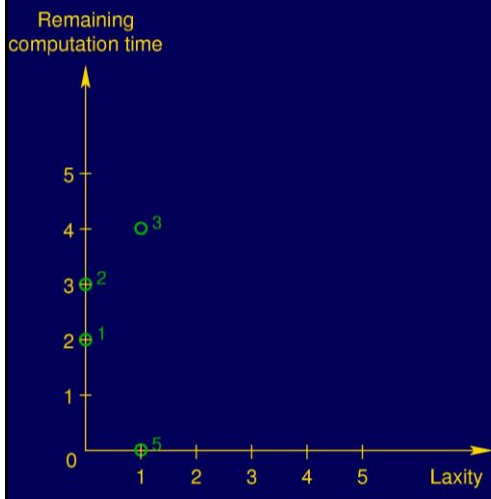
In every time scheduling step / turn of the game:
– at most n nodes go down by 1
– the rest moves 1 to the left

Nodes reaching the x–axis have been allocated all the computation time they need and are thus removed from the game board, as they don't represent scheduling constraints any longer.

The game is lost (i.e., the schedule is infeasible) if some node reaches the second quadrant.

It is won if no node remains on the board.

Game-board representation



When tasks are released, they are inserted into the game board according to their WCET and laxity (= deadline – remain. comp. time).

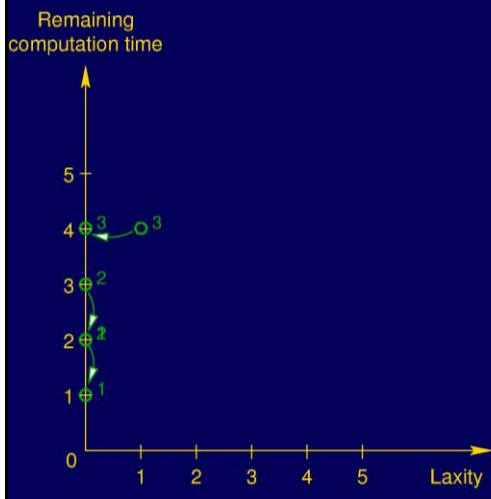
In every time scheduling step / turn of the game:
– at most n nodes go down by 1
– the rest moves 1 to the left

Nodes reaching the x–axis have been allocated all the computation time they need and are thus removed from the game board, as they don't represent scheduling constraints any longer.

The game is lost (i.e., the schedule is infeasible) if some node reaches the second quadrant.

It is won if no node remains on the board.

Game-board representation



When tasks are released, they are inserted into the game board according to their WCET and laxity (= deadline – remain. comp. time).

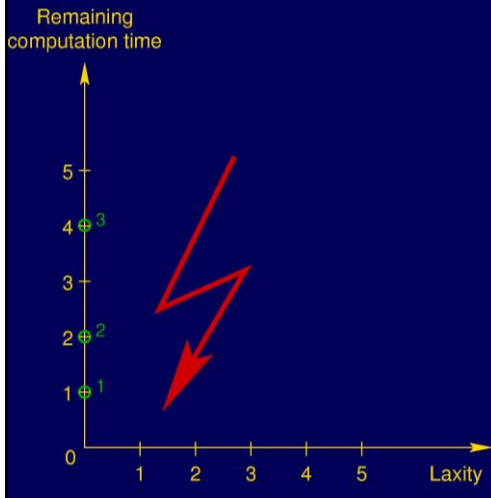
In every time scheduling step / turn of the game:
– at most n nodes go down by 1
– the rest moves 1 to the left

Nodes reaching the x–axis have been allocated all the computation time they need and are thus removed from the game board, as they don't represent scheduling constraints any longer.

The game is lost (i.e., the schedule is infeasible) if some node reaches the second quadrant.

It is won if no node remains on the board.

Game-board representation



When tasks are released, they are inserted into the game board according to their WCET and laxity (= deadline – remain. comp. time).

In every time scheduling step / turn of the game:
– at most n nodes go down by 1
– the rest moves 1 to the left

Nodes reaching the x–axis have been allocated all the computation time they need and are thus removed from the game board, as they don't represent scheduling constraints any longer.

The game is lost (i.e., the schedule is infeasible) if some node reaches the second quadrant.

It is won if no node remains on the board.

Extensions

- **Resource conflicts:** restricted move rules
- **Precedence constraints:** restricted move rules
- **Periodic tasks:** opponent's moves insert new nodes; game won if no task ever reaches second quadrant

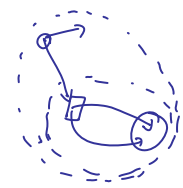
Game-theoretic solution

Theorem: In games with

- **finitely many positions** on the game board, and
- **complete** information

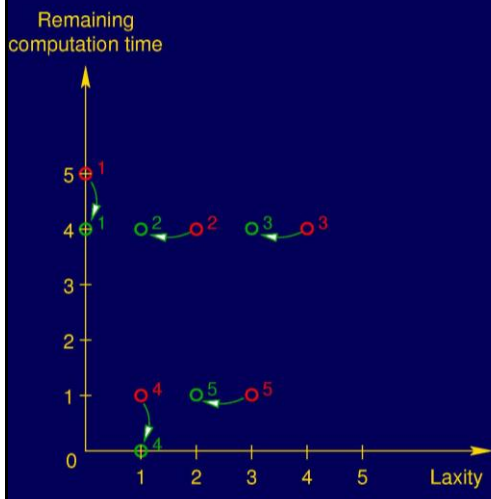
there is a always a winning strategy for one of the two players; it can be constructed effectively.

- Start with the losing positions
- Add all positions where we cannot avoid moving into this set
- Add all positions where the opponent can move into this set
- repeat until no more change



However: high complexity ⇒ predefined strategies preferred.

LLF (Least Laxity First)

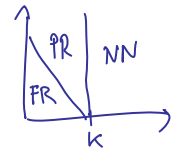


When tasks are released, they are inserted into the game board according to their WCET and laxity (= deadline – remain. comp. time).

In every time scheduling step / turn of the game:
– at most n nodes go down by 1
– the rest moves 1 to the left

LLF is optimal.

Schedulability



Within a set M of aperiodic tasks, we identify three classes with respect to the next k time units starting at time t:

1. Tasks that have to be fully run within the next k time units:

$$FR(t,k) = \{i \in M \mid D_i(t) \leq k\}$$

2. Tasks that have to be partially run within the next k time units:

$$PR(t,k) = \{i \in M \mid L_i(t) \leq k \wedge D_i(t) > k\}$$

3. Tasks that need not be run within the next k time units:

$$NN(t,k) = \{i \in M \mid L_i(t) > k\}$$

Surplus computing power

$$SCP(t,k) = \underbrace{kn}_{\text{avail. computing power}} - \underbrace{\sum_{i \in FR(t,k)} C_i(t)}_{\text{needed for full runs}} - \underbrace{\sum_{i \in PR(t,k)} (k - L_i(t))}_{\text{needed for partial runs}}$$

Lemma: SCP(0,k)≥0 for all k>0 is a **necessary** condition for schedulability.

- Executing $FR(0,k)$ requires $\sum_{i \in FR(0,k)} C_i$
- Executing $RR(0,k)$ requires $\sum_{i \in PR(0,k)} (k - L_i(0))$
- kn total computing power in k steps

horizontal means downward during test