Cyber-Physical Systems

Deadline based Scheduling

UNIVERSITY AT ALBANY
State University of New York

ICEN 553/453– Fall 2018

Prof. Dola Saha

Real-Time Systems

- The operating system, and in particular the scheduler, is perhaps the most important component

Examples:

- Control of laboratory experiments
- Process control in industrial plants
- Robotics
- Air traffic control
- Telecommunications
- Military command and control systems

- Correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced
- Tasks attempt to react to events that take place in the outside world
- These events occur in "real time" and tasks must be able to keep up with them

Hard and Soft Real-Time Tasks

- Hard
  - One that must meet its deadline
  - Otherwise it will cause unacceptable damage or a fatal error to the system
- Soft
  - Has an associated deadline that is desirable but not mandatory
  - It still makes sense to schedule and complete the task even if it has passed its deadline

Periodic and Aperiodic Tasks

- **Periodic tasks**
  - Requirement may be stated as:
    - Once per period *T*
    - Exactly *T* units apart
- **Aperiodic tasks**
  - Has a deadline by which it must finish or start
  - May have a constraint on both start and finish time

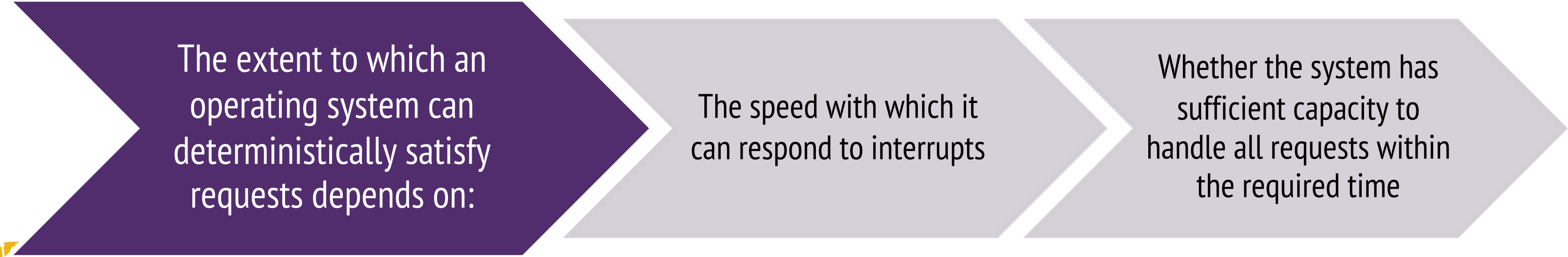
Characteristics of Real Time Systems

Real-time operating systems have requirements in five general areas:

- Determinism
- Responsiveness
- User control
- Reliability
- Fail-soft operation

Determinism

- Concerned with how long an operating system delays before acknowledging an interrupt
- Operations are performed at fixed, predetermined times or within predetermined time intervals
  - When multiple processes are competing for resources and processor time, no system will be fully deterministic



Responsiveness

- Together with determinism make up the response time to external events
  - Critical for real-time systems that must meet timing requirements imposed by individuals, devices, and data flows external to the system
- Concerned with how long, after acknowledgment, it takes an operating system to service the interrupt

Responsiveness includes:

- Amount of time required to initially handle the interrupt and begin execution of the interrupt service routine
- Amount of time required to perform the ISR
- Effect of interrupt nesting

User Control

- Generally much broader in a real-time operating system than in ordinary operating systems
- It is essential to allow the user fine-grained control over task priority
- User should be able to distinguish between hard and soft tasks and to specify relative priorities within each class
- May allow user to specify such characteristics as:

Paging or process swapping

What processes must always be resident in main memory

What disk transfer algorithms are to be used

What rights the processes in various priority bands have

Reliability

- More important for real-time systems than non-real time systems
- Real-time systems respond to and control events in real time so loss or degradation of performance may have catastrophic consequences such as:
  - Financial loss
  - Major equipment damage
  - Loss of life

Fail-Soft Operation

- A characteristic that refers to the ability of a system to fail in such a way as to preserve as much capability and data as possible
- Important aspect is stability
  - A real-time system is stable if the system will meet the deadlines of its most critical, highest-priority tasks even if some less critical task deadlines are not always met

Features common to Most RTOSs

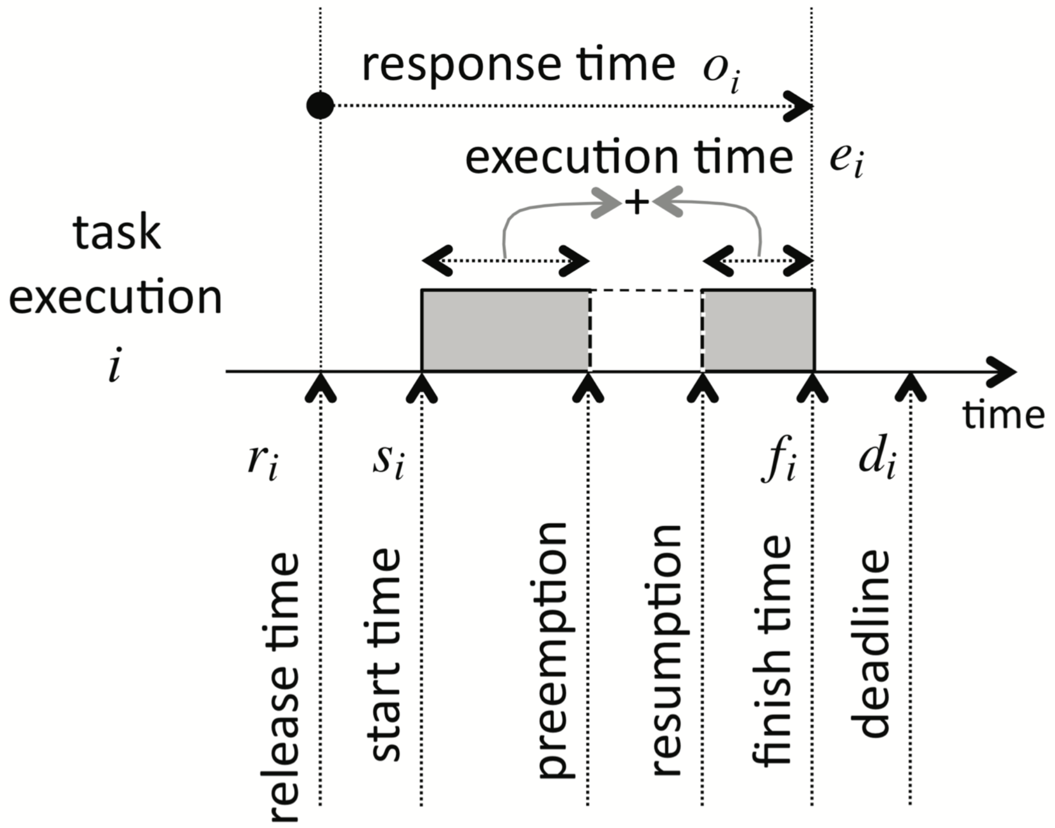
- A stricter use of priorities than in an ordinary OS, with preemptive scheduling that is designed to meet real-time requirements
- Interrupt latency is bounded and relatively short
- More precise and predictable timing characteristics than general purpose OSs

Task Model

$$s_i \geq r_i$$

$$f_i \geq s_i$$

$$o_i = f_i - r_i$$



Scheduling Strategies

- Goal: all task executions meet their deadlines

$$f_i \leq d_i$$

- A schedule that accomplishes this is called a feasible schedule.
- A scheduler that yields a feasible schedule for any task set is said to be optimal with respect to feasibility.

Criteria or Metrices

- Processor Utilization μ
- Maximum Lateness

$$L_{\max} = \max_{i \in T} (f_i - d_i)$$

- Total Completion Time or Makespan

$$M = \max_{i \in T} f_i - \min_{i \in T} r_i$$

- Average Response Time

$$\overline{t_r} = \frac{1}{n} \sum_{i=1}^{n} (f_i - a_i)$$

Rate Monotonic Scheduling

- Simple process model: n tasks invoked periodically with:
  - periods T1, … ,Tn, which equal the deadlines
  - known worst-case execution times (WCET) C1, … ,Cn
    - no mutexes, semaphores, or blocking I/O
  - independent tasks, no precedence constraints
  - fixed priorities
  - preemptive scheduling
- Rate Monotonic Scheduling (RMS): priorities ordered by period (smallest period has the highest priority)

Feasibility for RMS

- Feasibility is defined for RMS to mean that every task executes to completion once within its designated period.

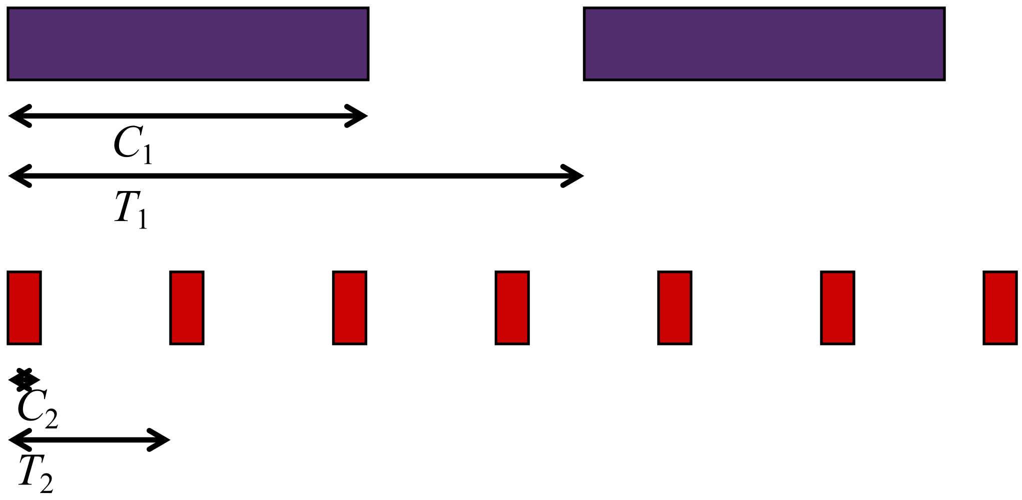
- Theorem: Under the simple process model, if any priority assignment yields a feasible schedule, then RMS also yields a feasible schedule.

- RMS is optimal in the sense of feasibility.

Liu and Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," J. ACM, 1973.

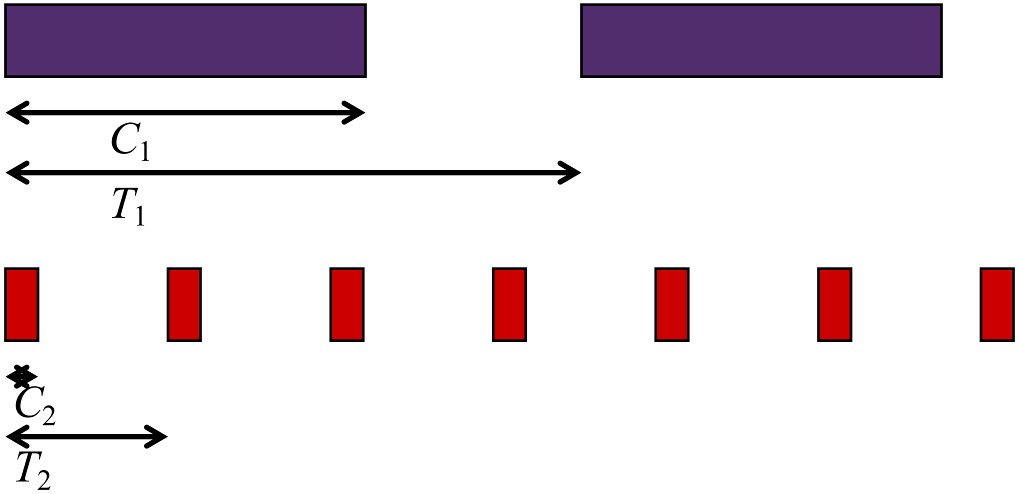
Showing Optimality of RMS:

- Consider two tasks with different periods.
- Is a non-preemptive schedule feasible?



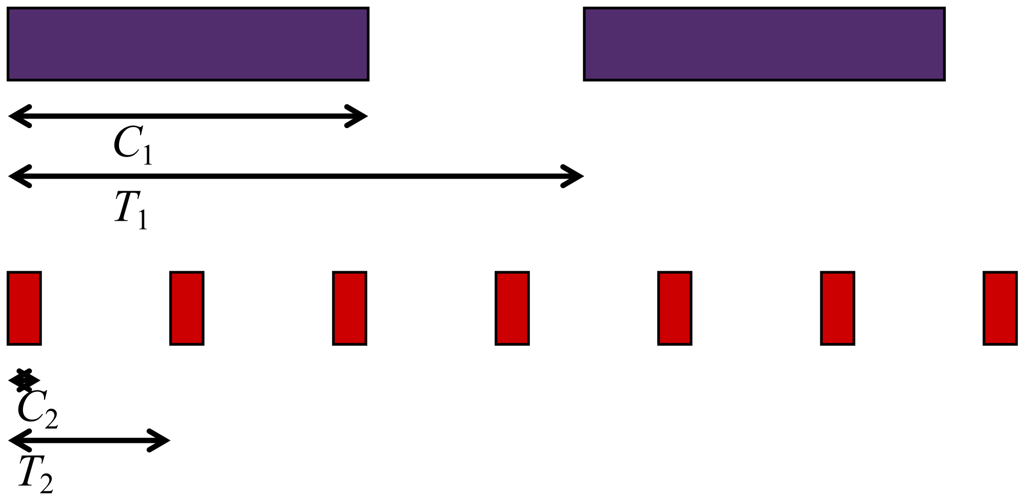
Showing Optimality of RMS:

- Non-preemptive schedule is not feasible. Some instance of the Red Task (2) will not finish within its period if we do non-preemptive scheduling.



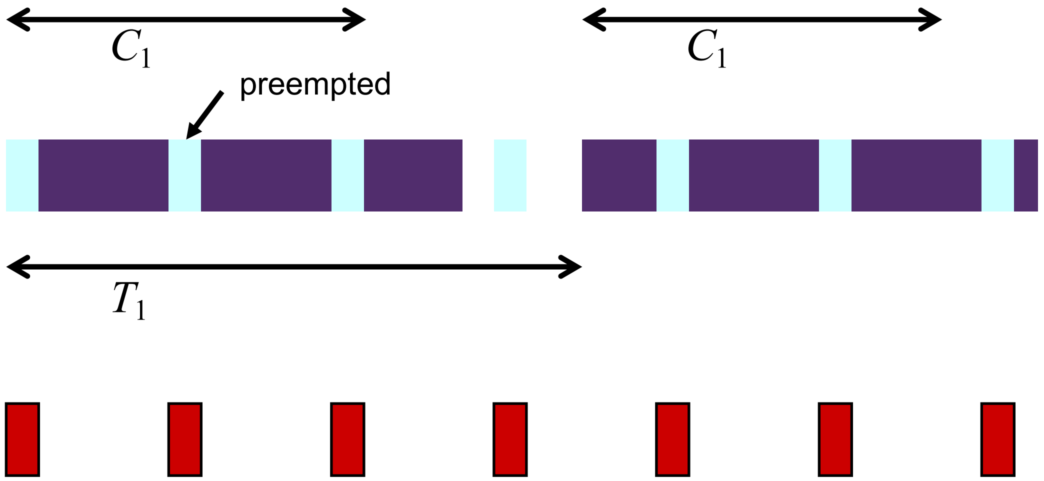
Showing Optimality of RMS:

- What if we had a preemptive scheduling with higher priority for red task?



Showing Optimality of RMS:

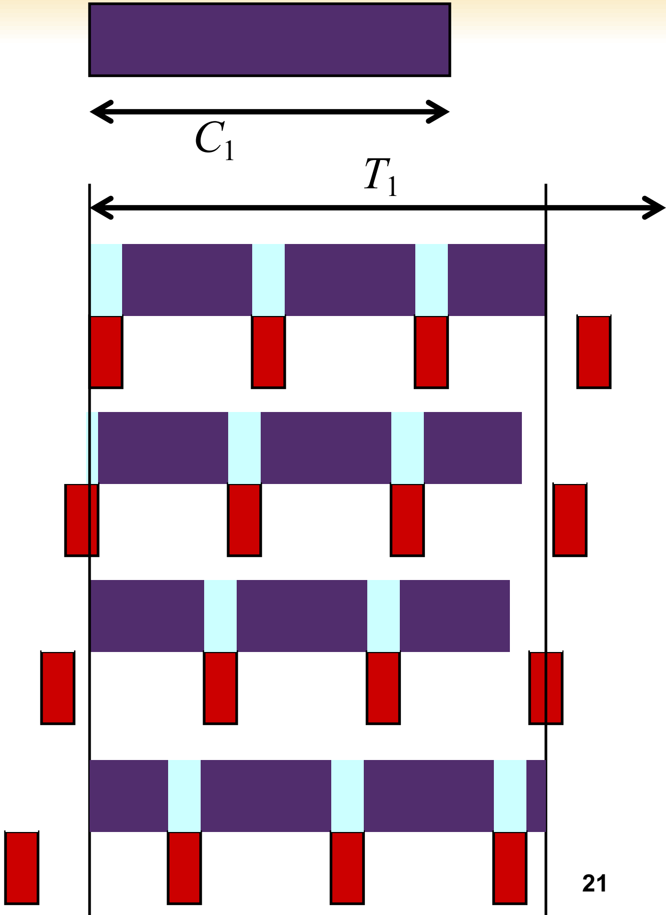
- Preemptive schedule with the red task having higher priority is feasible. Note that preemption of the purple task extends its completion time.



Alignment of tasks

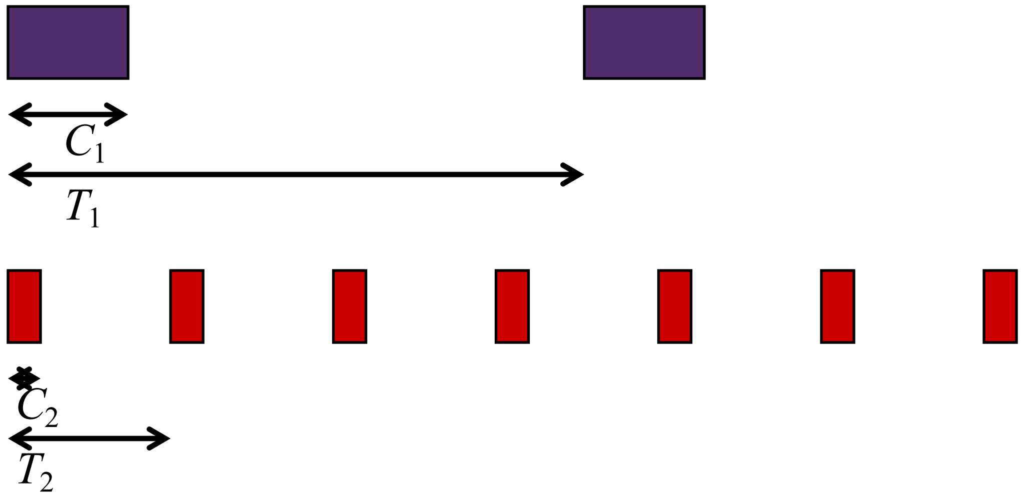
- Completion time of the lower priority task is worst when its *starting phase* matches that of higher priority tasks.

- Thus, when checking schedule feasibility, it is sufficient to consider only the worst case: All tasks start their cycles at the same time.



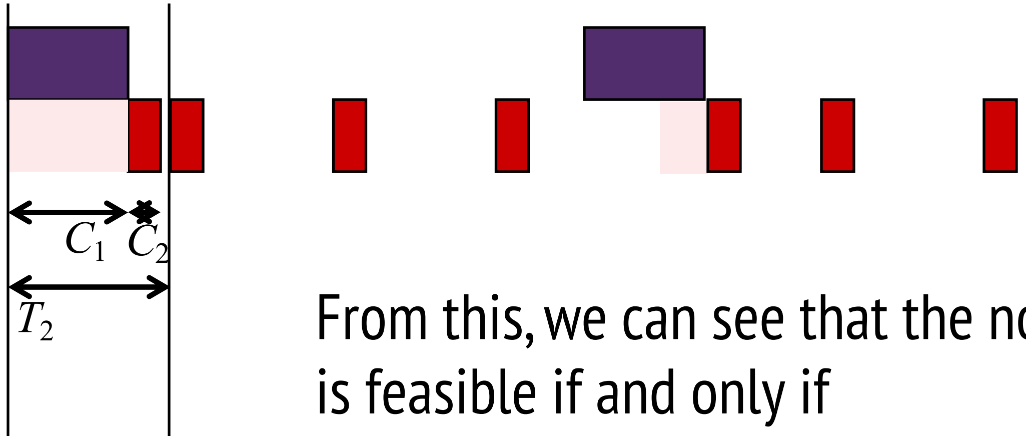
Showing Optimality of RMS: (for two tasks)

- It is sufficient to show that if a non-RMS schedule is feasible, then the RMS schedule is feasible.
- Consider two tasks as follows:



Showing Optimality of RMS: (for two tasks)

The non-RMS, fixed priority schedule looks like this:



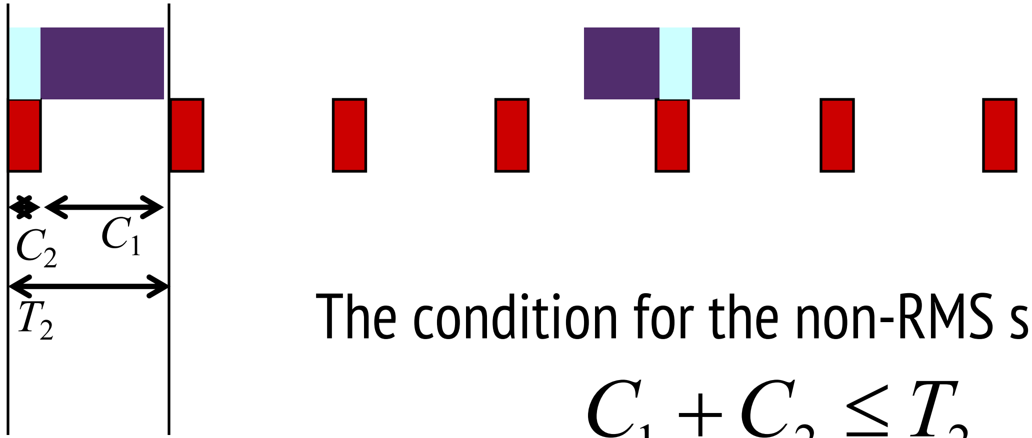
From this, we can see that the non-RMS schedule is feasible if and only if

$$C_1 + C_2 \leq T_2$$

We can then show that this condition implies that the RMS schedule is feasible.

Showing Optimality of RMS: (for two tasks)

The RMS schedule looks like this: (task with smaller period moves earlier)



The condition for the non-RMS schedule feasibility:

$$C_1 + C_2 \leq T_2$$

is clearly sufficient (though not necessary) for feasibility of the RMS schedule.

Comments

- This proof can be extended to an arbitrary number of tasks (though it gets much more tedious).
- This proof gives optimality only w.r.t. feasibility.
- Practical implementation:
  - Timer interrupt at greatest common divisor of the periods.
  - Multiple timers

RM Scheduler: Processor Utilization

$$\mu = \sum_{i=1}^{n} \frac{e_i}{p_i}$$

- If μ > 1 for any task set, then that task set has no feasible schedule
- Utilization Bound: RMS is feasible when $\mu \leq n(2^{1/n} - 1)$
- As n gets large, $\lim_{n \to \infty} n(2^{1/n} - 1) = \ln(2) \approx 0.693.$

- If a task set with any number of tasks does not attempt to use more than 69.3% of the available processor time, then the RM schedule will meet all deadlines.

Liu and Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," J. ACM, 1973.

Jackson's Algorithm: EDD (1955)

- Given n independent one-time tasks with deadlines $d_1, \ldots, d_n$, schedule them to minimize the maximum lateness, defined as

$$L_{\max} = \max_{1 \le i \le n} \{ f_i - d_i \}$$

- where f_i is the finishing time of task i. Note that this is negative iff all deadlines are met.
- Earliest Due Date (EDD) algorithm: Execute them in order of non-decreasing deadlines.
- Note that this does not require preemption.

EDD is Optimal

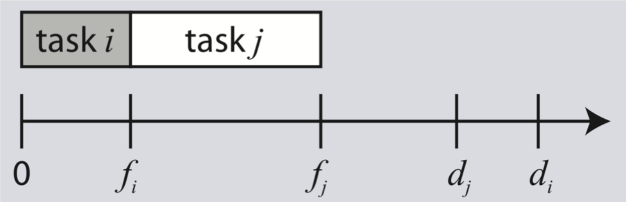
- Optimal in the Sense of Minimizing Maximum Lateness
  - To prove, use an interchange argument. Given a schedule S that is not EDD, there must be tasks a and b where a immediately precedes b in the schedule but
    $d_a > d_b$. Why?
  - We can prove that this schedule can be improved by interchanging a and b. Thus, no non-EDD schedule is achieves smaller max lateness than EDD, so the EDD schedule must be optimal.

Maximum Lateness

- First Schedule (non-EDD)

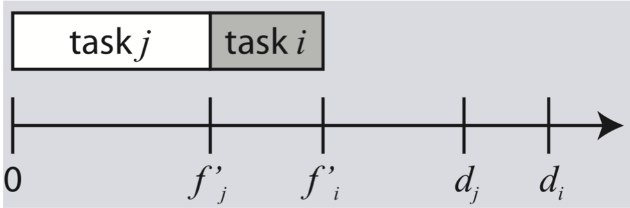
$$L_{\max} = \max(f_i - d_i, f_j - d_j) = f_j - d_j$$

  - where $f_i \leq f_j$ and $d_j < d_i$



- Second Schedule (EDD)

$$L'_{\max} = \max(f'_i - d_i, f'_j - d_j)$$



UNIVERSITY AT ALBANY
State University of New York

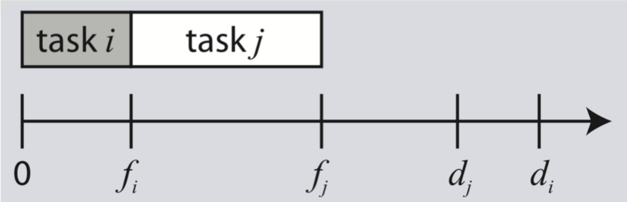
Consider Cases

Case 1: $L'_{\max} = f'_i - d_i$

$f'_i = f_j$ $\quad d_j < d_i$

$$L'_{\max} = f_j - d_i \leq f_j - d_j$$

Hence, $L'_{\max} \leq L_{\max}$

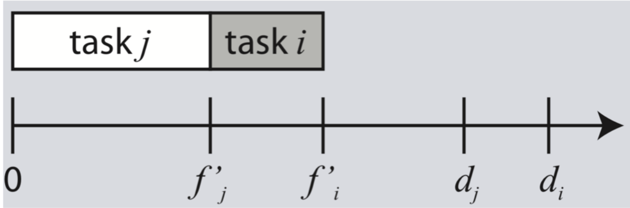


Case 2: $L'_{\max} = f'_j - d_j$

$f'_j \leq f_j$

$$L'_{\max} \leq f_j - d_j$$

Hence, $L'_{\max} \leq L_{\max}$



In both cases, the second schedule has a maximum lateness no greater than that of the first schedule.
EDD minimizes maximum lateness.

Horn’s algorithm: EDF (1974)

- Extend EDD by allowing tasks to “arrive” (become ready) at any time.
- Earliest deadline first (EDF): Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among all arrived tasks is optimal w.r.t. minimizing the maximum lateness.
- Proof uses a similar interchange argument.

Using EDF for Periodic Tasks

- The EDF algorithm can be applied to periodic tasks as well as aperiodic tasks.
  - Simplest use: Deadline is the end of the period.
  - Alternative use: Separately specify deadline (relative to the period start time) and period.

RMS vs. EDF? Which one is better?

- What are the pros and cons of each?

Comparison of EDF and RMS

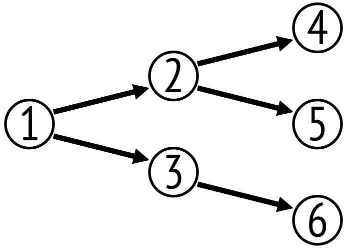
- Favoring RMS
  - Scheduling decisions are simpler (fixed priorities vs. the dynamic priorities required by EDF. EDF scheduler must maintain a list of ready tasks that is sorted by priority.)

Comparison of EDF and RMS

- Favoring EDF
  - Since EDF is optimal w.r.t. maximum lateness, it is also optimal w.r.t. feasibility. RMS is only optimal w.r.t. feasibility. For infeasible schedules, RMS completely blocks lower priority tasks, resulting in unbounded maximum lateness.
  - EDF can achieve full utilization where RMS fails to do that.
  - EDF results in fewer preemptions in practice, and hence less overhead for context switching.
  - Deadlines can be different from the period.

Precedence Constraints

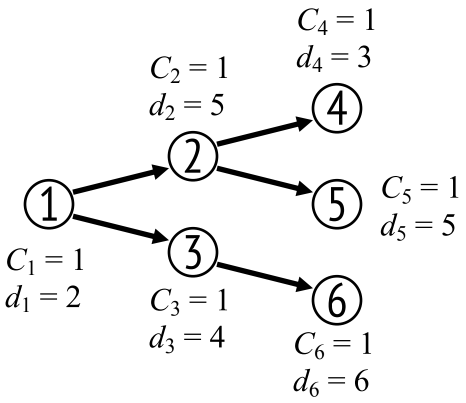
- A directed acyclic graph (DAG) shows precedences, which indicate which tasks must complete before other tasks start.

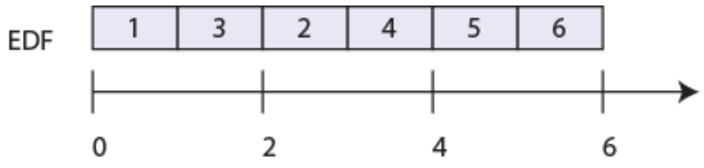


DAG, showing that task 1 must complete before tasks 2 and 3 can be started, etc.

Example: EDF Schedule

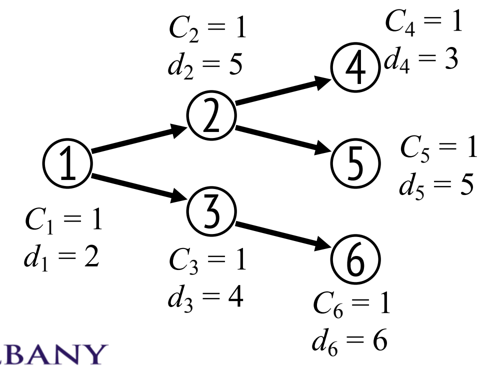
- Is this feasible?

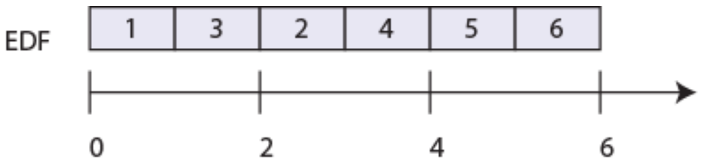




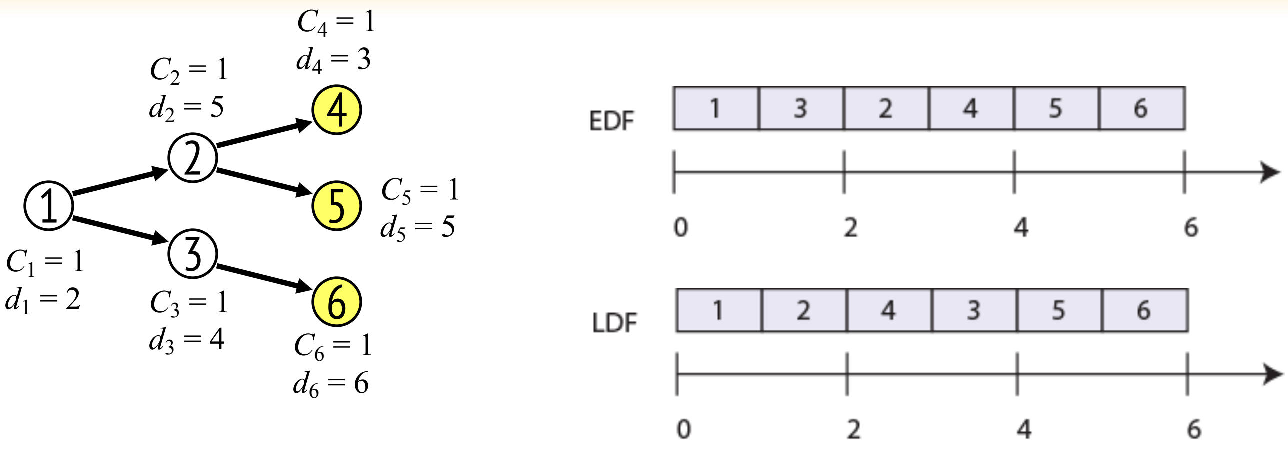
EDF is not optimal under precedence constraints

- The EDF schedule chooses task 3 at time 1 because it has an earlier deadline. This choice results in task 4 missing its deadline.



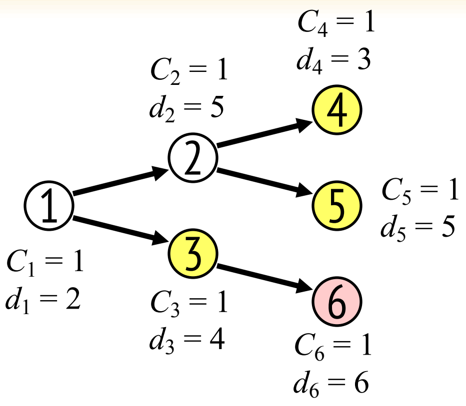


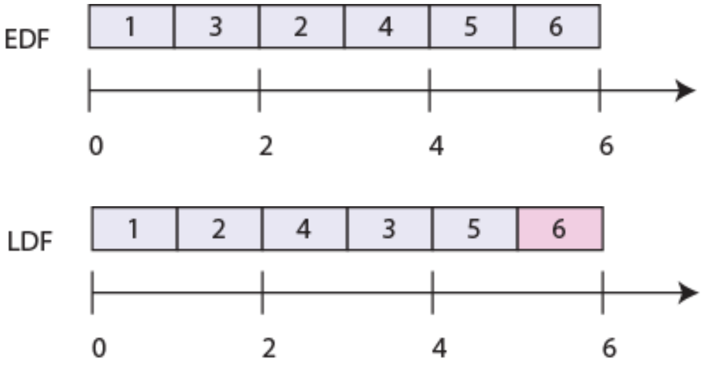
Latest Deadline First (LDF) (Lawler, 1973)



- The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

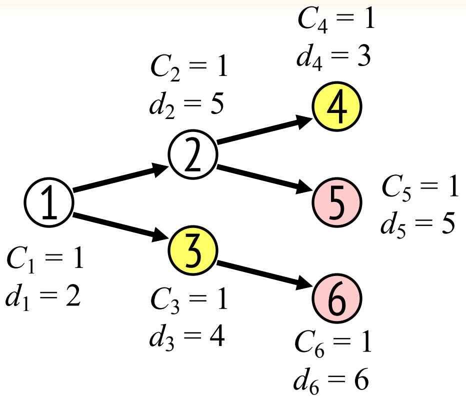
Latest Deadline First (LDF) (Lawler, 1973)

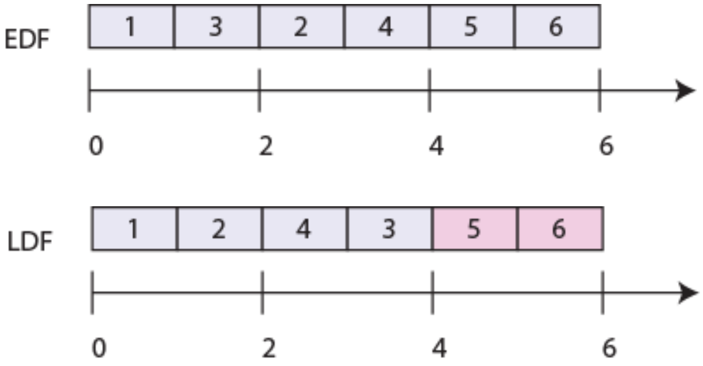




- The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

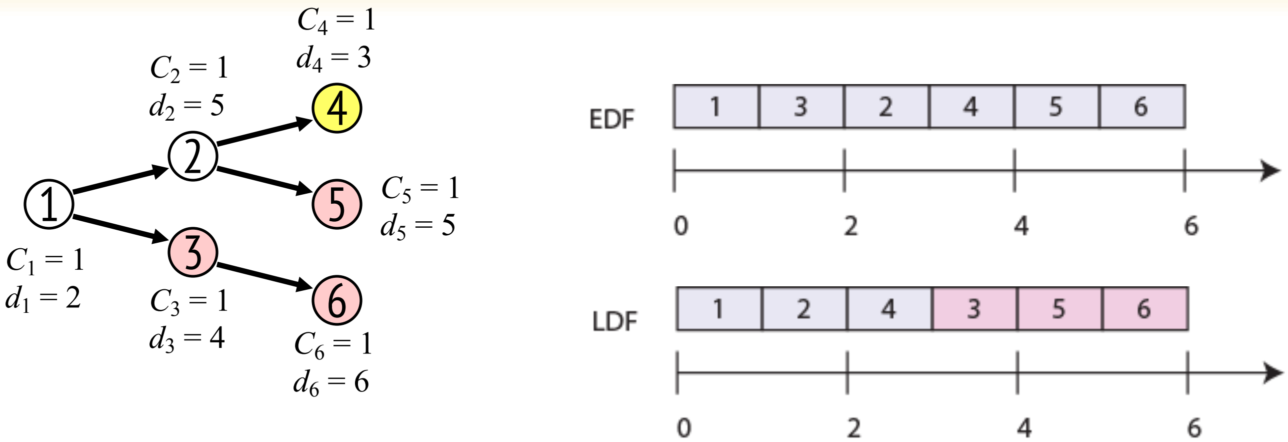
Latest Deadline First (LDF) (Lawler, 1973)





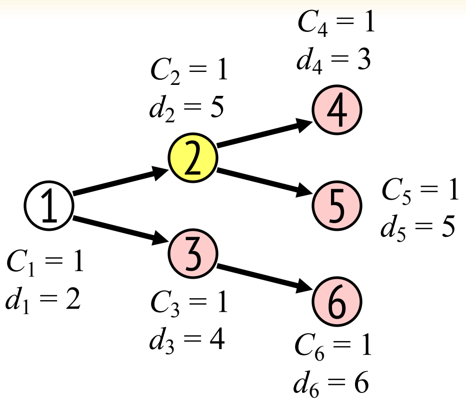
- ➢ The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

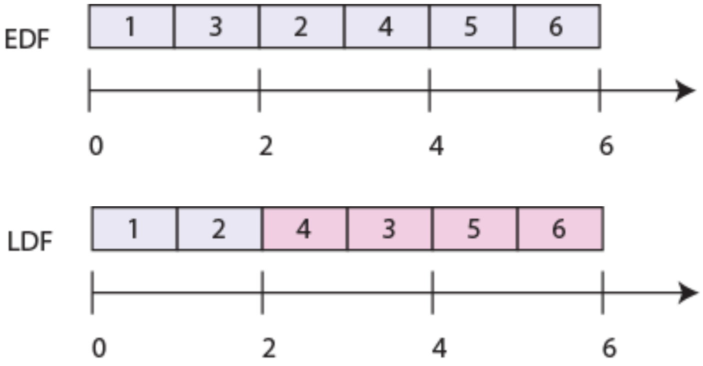
Latest Deadline First (LDF) (Lawler, 1973)



- The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

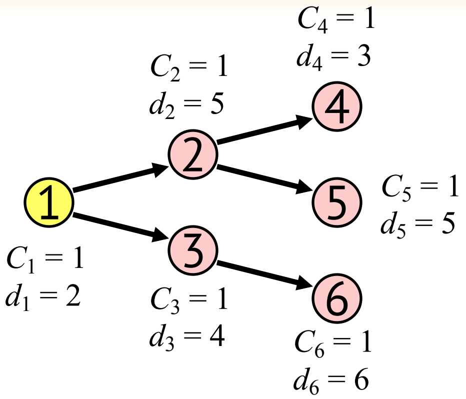
Latest Deadline First (LDF) (Lawler, 1973)

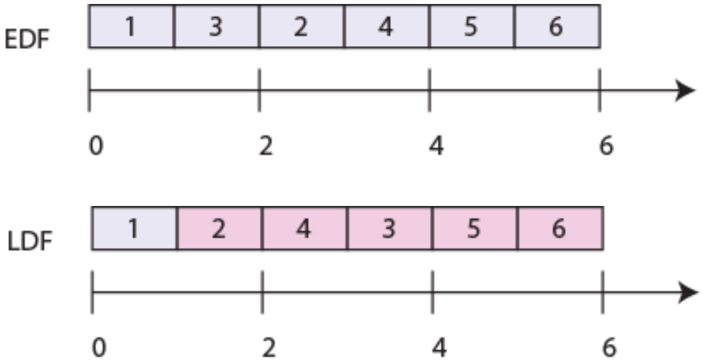




➤ The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

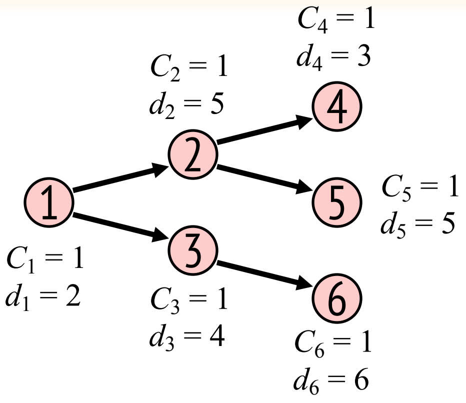
Latest Deadline First (LDF) (Lawler, 1973)

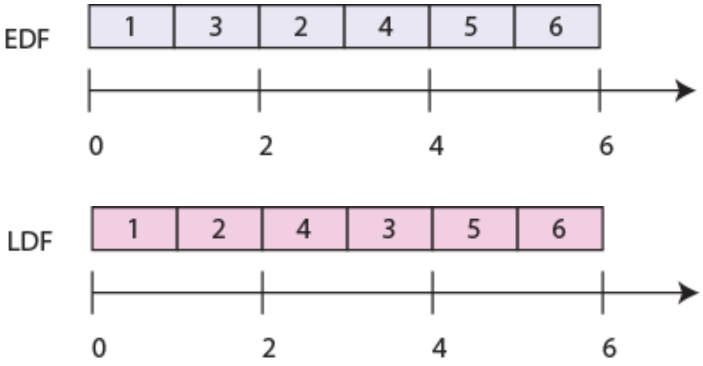




- The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

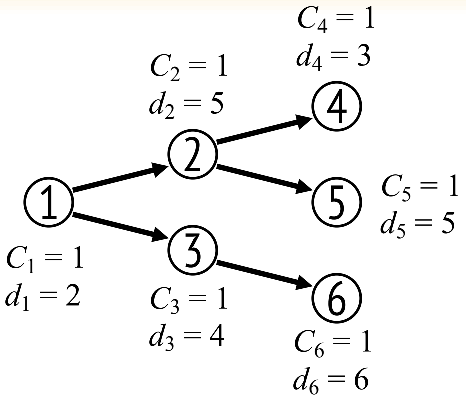
Latest Deadline First (LDF) (Lawler, 1973)

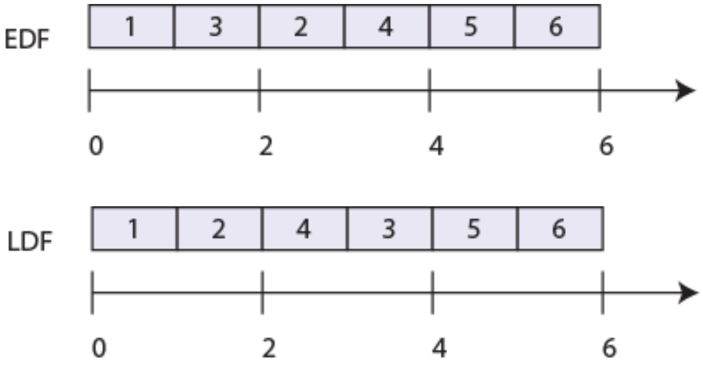




- The LDF scheduling strategy builds a schedule backwards. Given a DAG, choose the leaf node with the latest deadline to be scheduled last, and work backwards.

LDF is optimal under precedence constraints





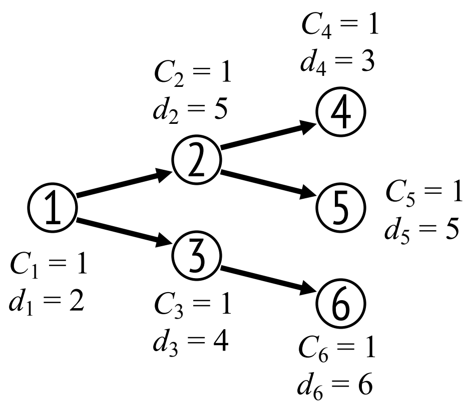
- The LDF schedule shown at the bottom respects all precedences and meets all deadlines.
- Also minimizes maximum lateness

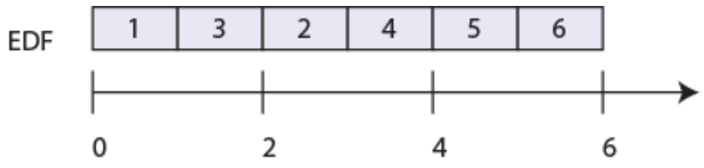
Latest Deadline First (LDF) (Lawler, 1973)

- LDF is optimal in the sense that it minimizes the maximum lateness.
- It does not require preemption. (We'll see that EDF can be made to work with preemption.)
- However, it requires that all tasks be available and their precedences known before any task is executed.

EDF with Precedences

- With a preemptive scheduler, EDF can be modified to account for precedences and to allow tasks to arrive at arbitrary times. Simply adjust the deadlines and arrival times according to the precedences.

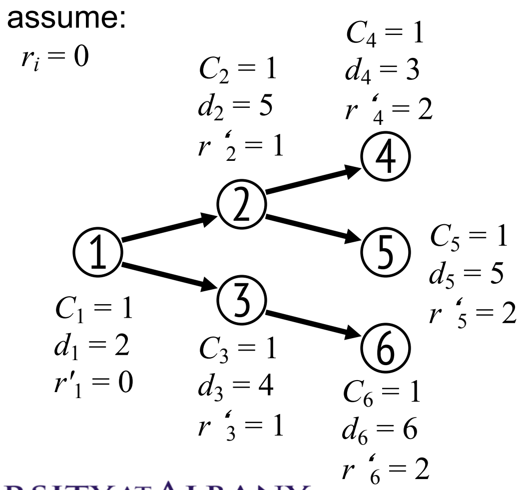




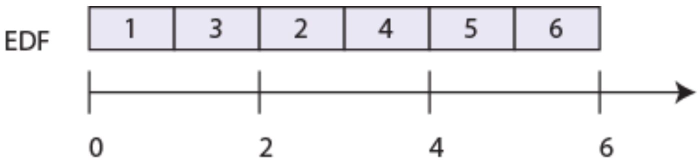
Recall that for the tasks at the left, EDF yields the schedule above, where task 4 misses its deadline.

EDF with Precedences Modifying release times

- Given n tasks with precedences and release times r_i, if task i immediately precedes task j, then modify the release times as follows:



$$r'_j = \max(r_j, r_i + C_i)$$



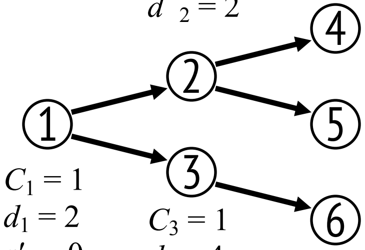
EDF with Precedences Modifying deadlines

- Given n tasks with precedences and deadlines d_i, if task i immediately precedes task j, then modify the deadlines as follows:

assume:
$r_i = 0$

$C_2 = 1$
$d_2 = 5$
$r'_2 = 1$
$d'_2 = 2$

$C_4 = 1$
$d_4 = 3$
$r'_4 = 2$
$d'_4 = 3$



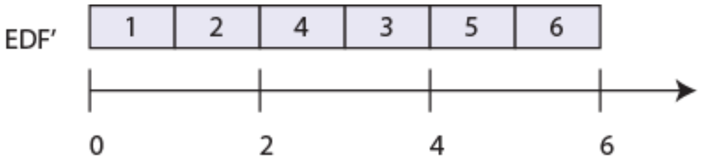
$C_1 = 1$
$d_1 = 2$
$r'_1 = 0$
$d'_2 = 1$

$C_3 = 1$
$d_3 = 4$
$r'_3 = 1$
$d'_3 = 4$

$C_6 = 1$
$d_6 = 6$
$r'_6 = 2$
$d'_6 = 6$

$C_5 = 1$
$d_5 = 5$
$r'_5 = 2$
$d'_5 = 5$

$$d'_i = \min(d_i, d'_j - C_j)$$



Using the revised release times and deadlines, the above EDF schedule is optimal and meets all deadlines.

Optimality

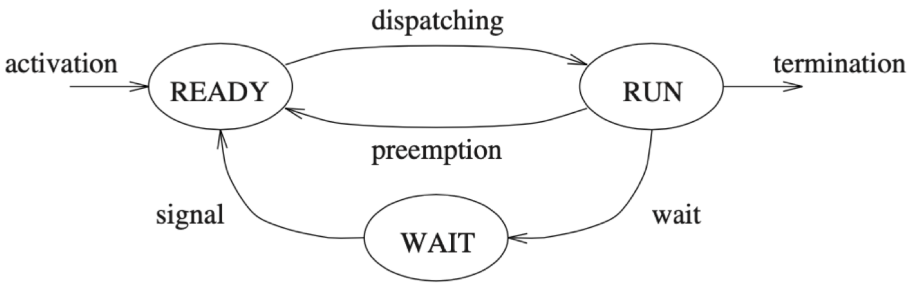
- Generalized modified deadline

$$d'_i = \min(d_i, \min_{j \in D(i)} (d'_j - e_j))$$

- EDF with precedences is optimal in the sense of minimizing the maximum lateness.

Scheduling in Shared Resource

- concurrent tasks use shared resources in exclusive mode
- Recall: critical section and mutexes/semaphores

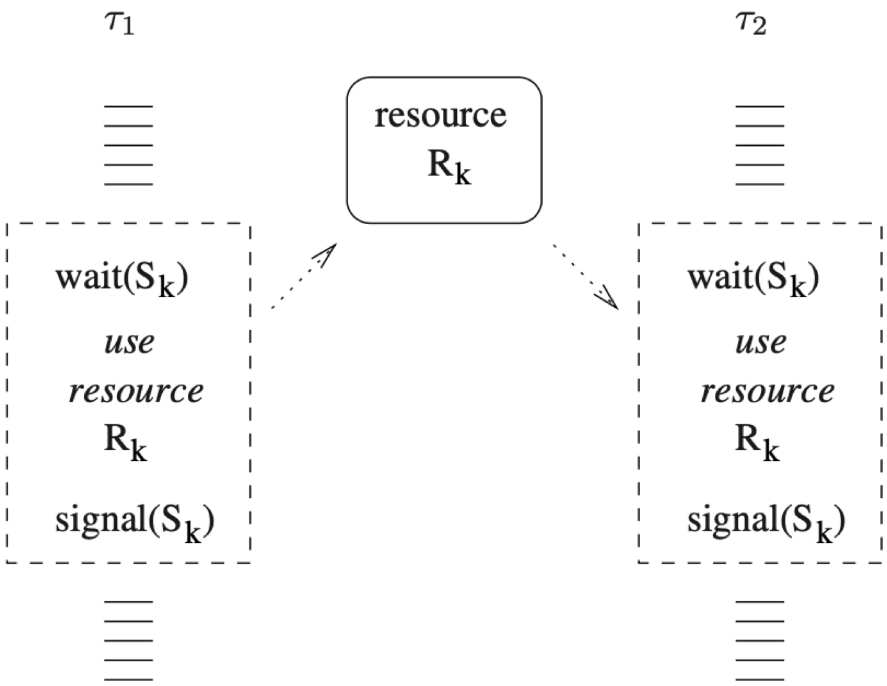


A task waiting for an exclusive resource is said to be *blocked* on that resource

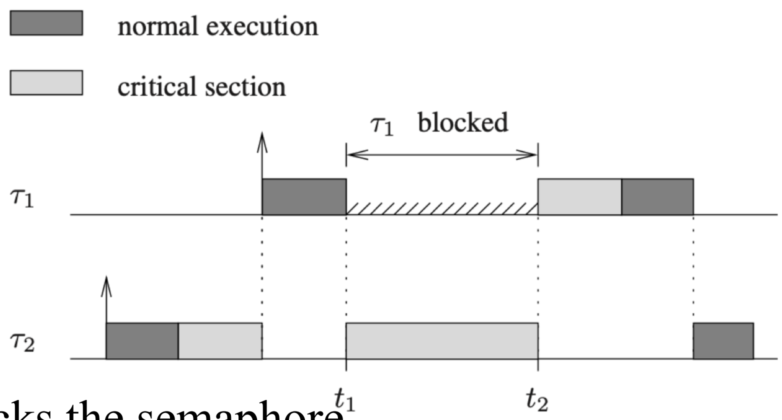
Giorgio C. Buttazzo, *Hard Real-Time Computing Systems,* Springer, 2004.

Two tasks sharing exclusive resources

```
#include <pthread.h>
...
pthread_mutex_t lock;

void* addListener(notify listener) {
  pthread_mutex_lock(&lock);
  ...
  pthread_mutex_unlock(&lock);
}

void* update(int newValue) {
  pthread_mutex_lock(&lock);
  value = newValue;
  elementType* element = head;
  while (element != 0) {
    (*(element->listener))(newValue);
    element = element->next;
  }
  pthread_mutex_unlock(&lock);
}

int main(void) {
  pthread_mutex_init(&lock, NULL);
  ...
}
```

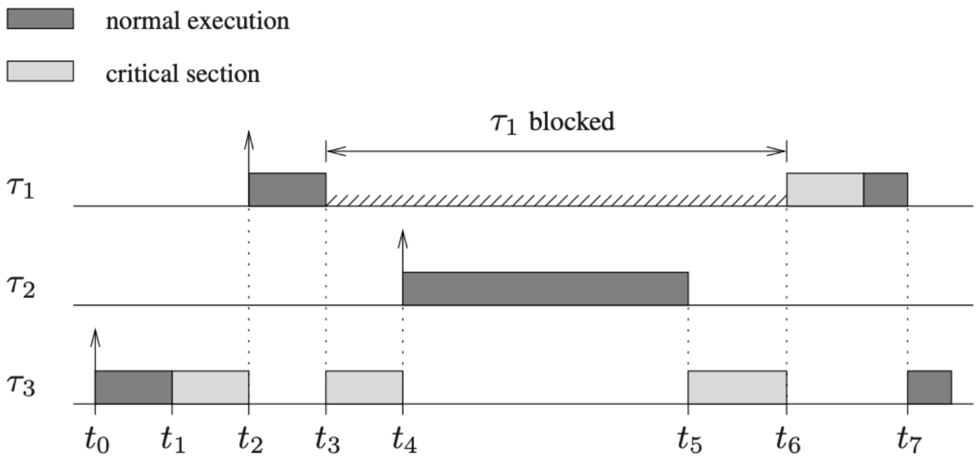


Blocking on critical section



- τ_1 has a higher priority than τ_2
- τ_2 is activated first
  - after a while, it enters the critical section and locks the semaphore.
- While τ_2 is executing the critical section
  - task τ_1 arrives, and it preempts τ_2 and starts executing.
- At t_1, τ_1 is blocked on the semaphore, so τ_2 resumes
- At t_2, τ_2 releases the critical section
- Maximum blocking time of τ_1 is equal to the time needed by τ_2 to execute its critical section.

Priority Inversion with Mutex



- A *priority inversion* is said to occur in the interval $[t_3, t_6]$, since the highest-priority task τ_1 waits for the execution of lower- priority tasks (τ_2 and τ_3).

Priority Inversion: Why is it a problem?

- Maximum blocking time of τ_1 depends on
  - the length of the critical section executed by τ_3
  - the worst-case execution time of τ_2
- Can lead to uncontrolled blocking (with multiple medium priority tasks)
  - can cause critical deadlines to be missed
- The duration of priority inversion is unbounded

Resource Access Protocols

- Non-Preemptive Protocol (NPP)
- Highest Locker Priority (HLP) or Immediate Priority Ceiling (IPC)
- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)
- Stack Resource Policy (SRP)

Terminology

- n periodic tasks, $\tau_1,\tau_2,\ldots,\tau_n$
- m shared resources, $R_1,R_2,\ldots,R_m$
- Each task is characterized by
  - a period T_i
  - a worst-case computation time C_i
- Each resource R_k is guarded by a distinct semaphore S_k
- each task is characterized by
  - a fixed *nominal* priority P_i (assigned by the algorithm) and
  - an *active* priority p_i ($p_i \geq P_i$), which is dynamic and initially set to P_i

Terminology

B_i denotes the maximum blocking time task τ_i can experience.

$z_{i,k}$ denotes a generic critical section of task τ_i guarded by semaphore S_k.

$Z_{i,k}$ denotes the longest critical section of task τ_i guarded by semaphore S_k.

$\delta_{i,k}$ denotes the duration of $Z_{i,k}$.

$z_{i,h} \subset z_{i,k}$ indicates that $z_{i,h}$ is entirely contained in $z_{i,k}$.

σ_i denotes the set of semaphores used by τ_i.

$\sigma_{i,j}$ denotes the set of semaphores that can block τ_i, used by the lower-priority task τ_j.

Terminology

$\gamma_{i,j}$ denotes the set of the longest critical sections that can block τ_i, accessed by the lower priority task τ_j. That is,

$$\gamma_{i,j} = \{Z_{j,k} \mid (P_j < P_i) \text{ and } (S_k \in \sigma_{i,j})\} \tag{7.1}$$

γ_i denotes the set of all the longest critical sections that can block τ_i. That is,

$$\gamma_i = \bigcup_{j:P_j<P_i} \gamma_{i,j} \tag{7.2}$$

Assumptions

- Priorities:
  - Tasks τ_1 , τ_2 , . . . , τ_n have different priorities
  - They are listed in descending order of nominal priority
  - τ_1 has the highest nominal priority
- Tasks do not suspend themselves on I/O
- The critical sections used by any task are *properly* nested
  - given any pair $z_{i,h}$ and $z_{i,k}$

    either $z_{i,h} \subset z_{i,k}$, $z_{i,k} \subset z_{i,h}$, or $z_{i,h} \cap z_{i,k} = \emptyset$.
- Critical sections are guarded by binary semaphores

Non-Preemptive Protocol

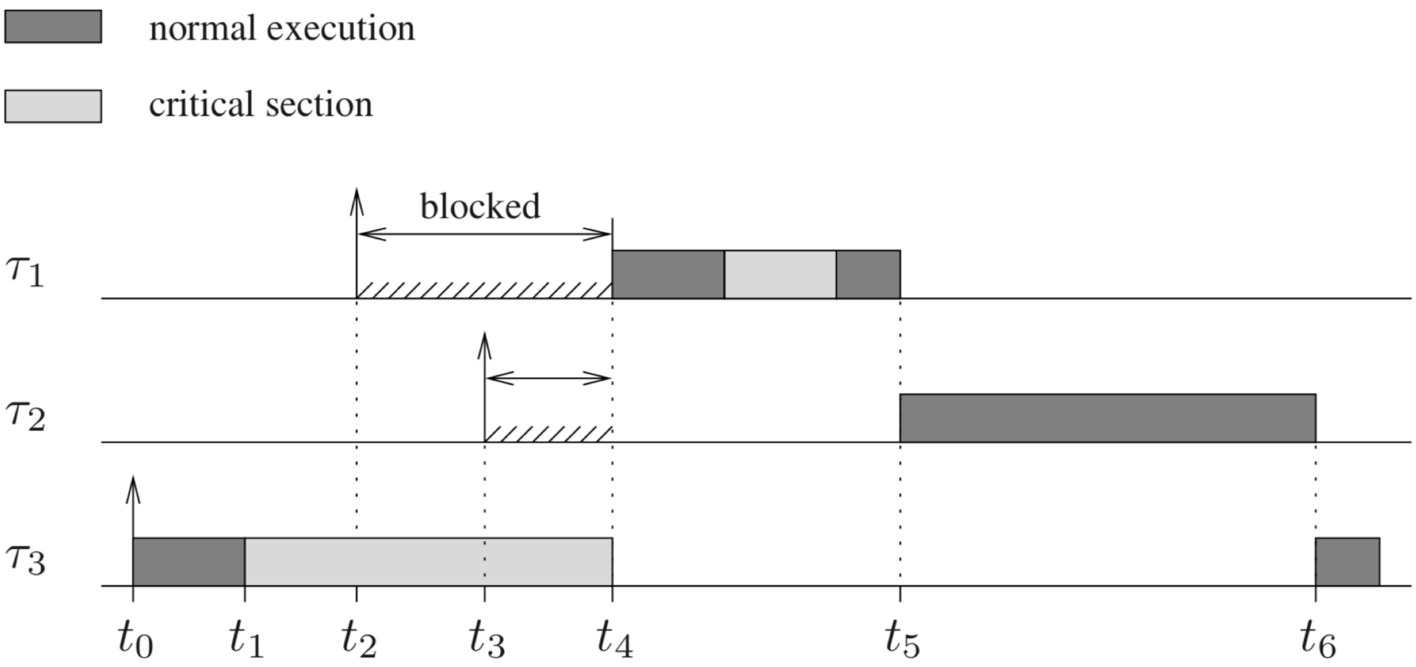
- Raise the priority of a task to the highest priority level whenever it enters a shared resource

- as a task τ_i enters a resource R_k, its dynamic priority is raised to the level:

$$p_i(R_k) = \max_h\{P_h\}.$$

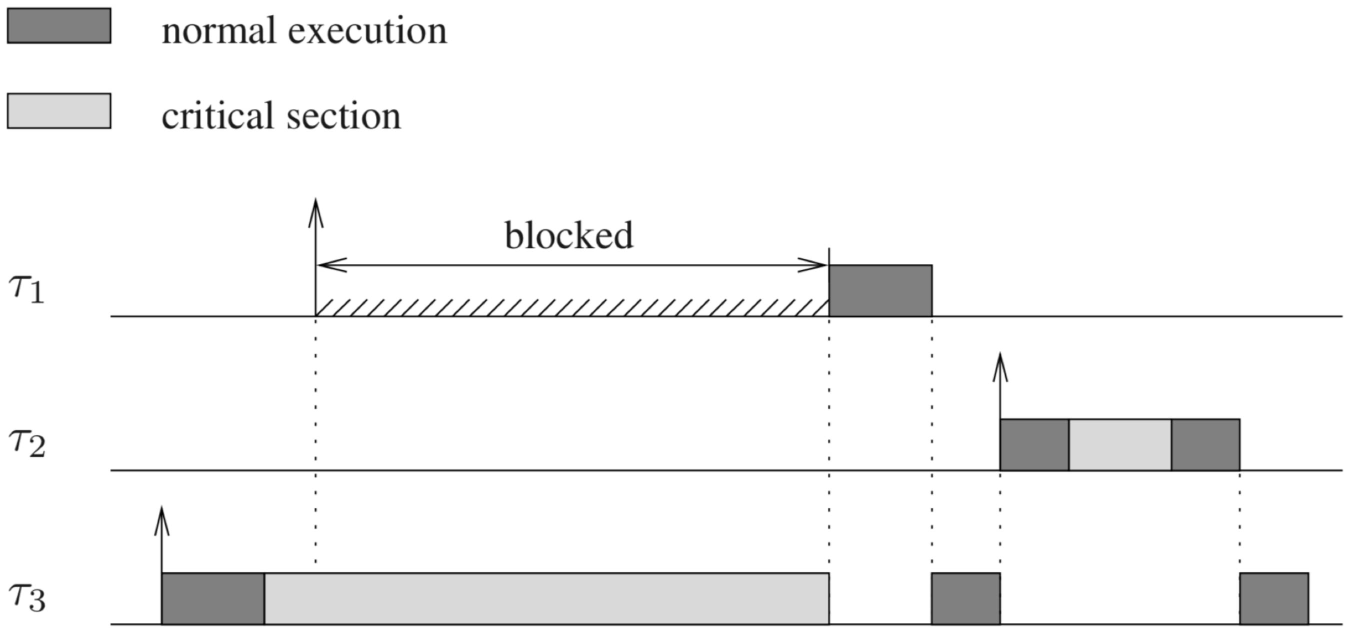
- The dynamic priority is then reset to the nominal value P_i when the task exits the critical section

Example (NPP preventing priority inversion)



NPP causes unnecessary blocking

τ_1 is the highest-priority task that does not use any resource



Blocking Time Computation (NPP)

- task τ_i cannot preempt a lower priority task τ_j if τ_j is inside a critical section

$$\gamma_i = \{Z_{j,k} \mid P_j < P_i,\ k = 1, \ldots, m\}$$

- a task inside a resource *R* cannot be preempted, only one resource can be locked at any time *t*
- a task τ_i can be blocked at most for the length of a single critical section belonging to lower priority tasks
- maximum blocking time τ_i is the duration of the longest critical section of lower priority tasks

$$B_i = \max_{j,k}\{\delta_{j,k} - 1 \mid Z_{j,k} \in \gamma_i\}$$

- one unit of time is subtracted from $\delta_{j,k}$ since $Z_{j,k}$ must start before the arrival of τ_i to block it

Highest Locker Priority (HLP)

- Raises the priority of a task that enters a resource R_k to the highest priority among the tasks sharing that resource
- as soon as a task τ_i enters a resource R_k, its dynamic priority is raised to the level

$$p_i(R_k) = \max_h \{P_h \mid \tau_h \text{ uses } R_k\}$$

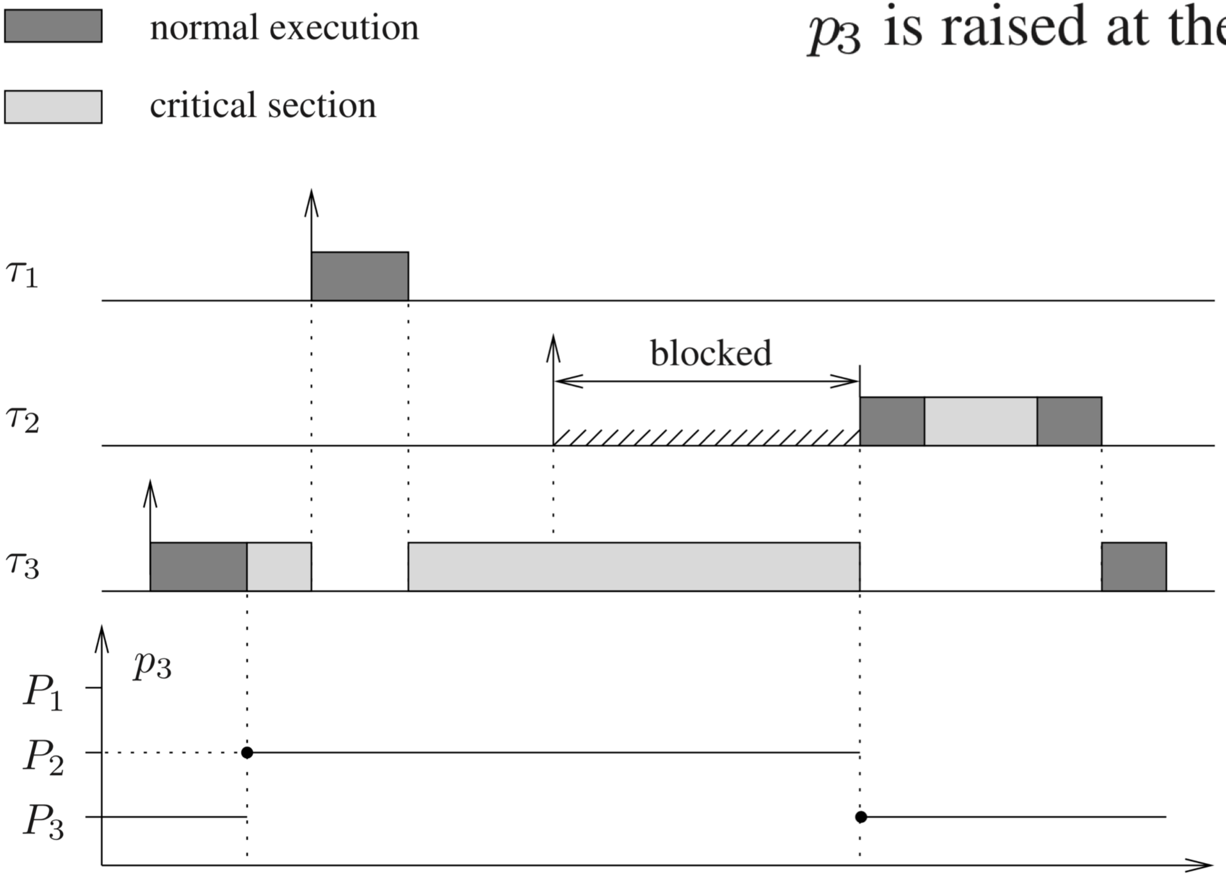
- each resource R_k is assigned a priority ceiling $C(R_k)$ (computed off-line) equal to the maximum priority of the tasks sharing R_k

$$C(R_k) \stackrel{\text{def}}{=} \max_h \{P_h \mid \tau_h \text{ uses } R_k\}$$

- Also termed Immediate Priority Ceiling

HLP Example

p_3 is raised at the level $C(R) = P_2$



Blocking Time (HLP)

- a task τ_i can only be blocked by critical sections belonging to lower priority tasks with a resource ceiling higher than or equal to P_i
- a task can be blocked at most once (Proof in the book)
- the maximum blocking time of τ_i is given by the duration of the longest critical section among those that can block τ_i

$$B_i = \max_{j,k}\{\delta_{j,k} - 1 \mid Z_{j,k} \in \gamma_i\}$$

Priority Inheritance Protocol (PIP)

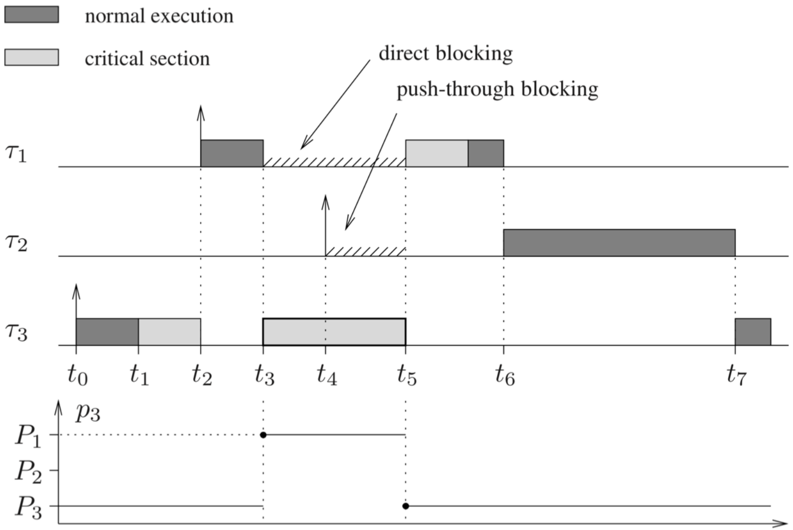
- When a task τ_i blocks one or more higher-priority tasks, it temporarily assumes (*inherits*) the highest priority of the blocked tasks
- When a task τ_i is blocked on a semaphore, it transmits its active priority to the task τ_j, that holds that semaphore
- τ_j executes the rest of its critical section with a priority $p_j = p_i$.

$$p_j(R_k) = \max\{P_j, \max_h\{P_h | \tau_h \text{ is blocked on } R_k\}\}$$

- When τ_j exits a critical section the active priority of τ_j is updated
  - if no other tasks are blocked by τ_j, p_j is set to P_j
  - otherwise it is set to the highest priority of the tasks blocked by τ_j
- Priority inheritance is transitive
  - if a task τ_3 blocks a task τ_2, and τ_2 blocks a task τ_1, then τ_3 inherits the priority of τ_1 via τ_2

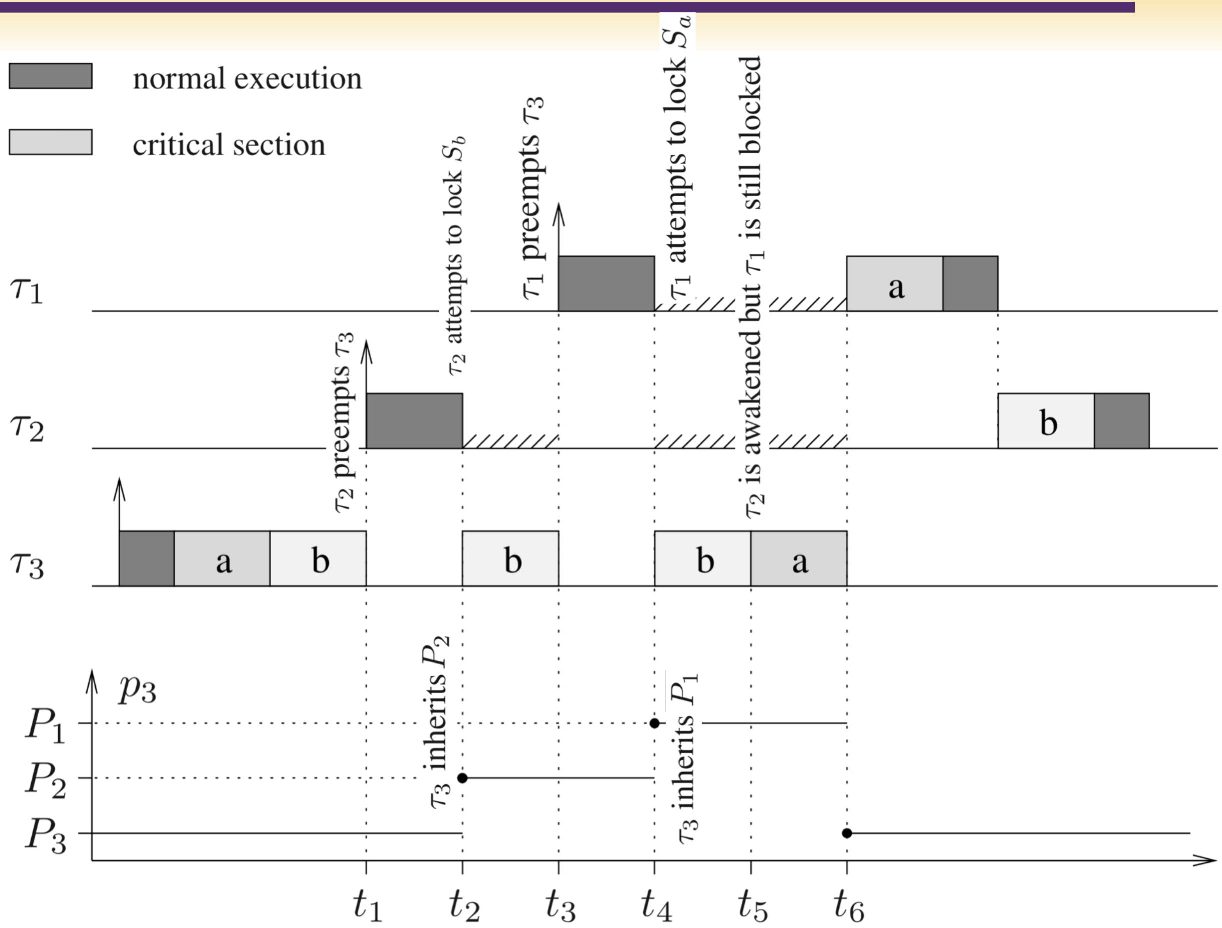
Types of Blocking in PIP

- Direct
  - a higher-priority task tries to acquire a resource held by a lower-priority task
  - Required to ensure consistency of shared resource
- Push-through
  - a medium-priority task is blocked by a low-priority task that has inherited a higher priority from a task it directly blocks
  - Required to void unbounded priority inversion



Nested Critical Section (PIP)

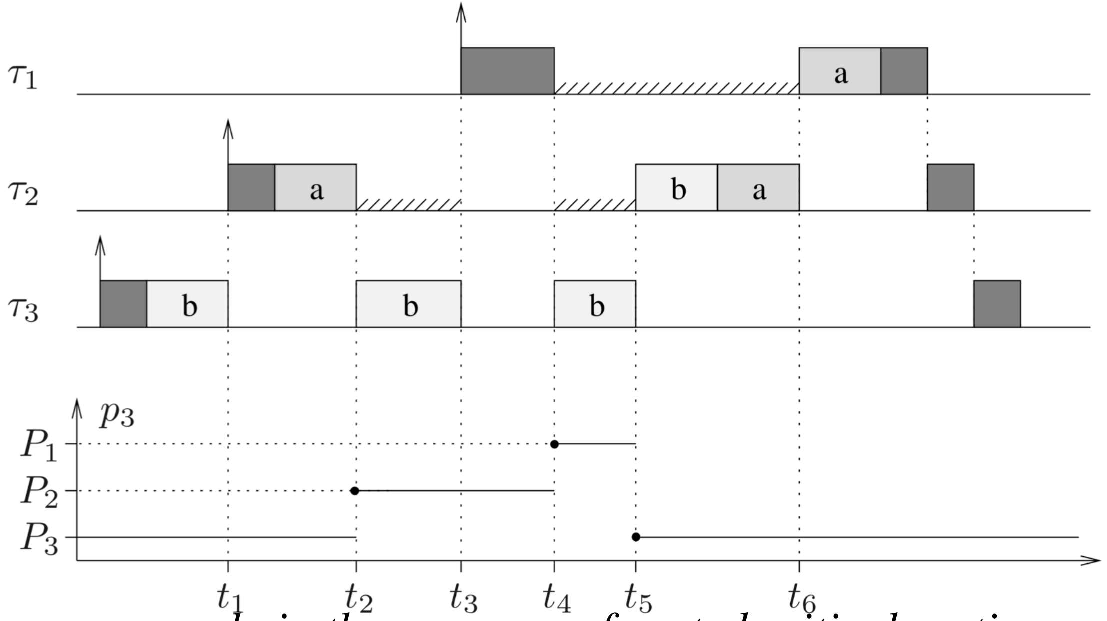
- task τ_1 uses a resource R_a guarded by a semaphore S_a,
- task τ_2 uses a resource R_b guarded by a semaphore S_b
- task τ_3 uses both resources in a nested fashion (S_a is locked first)



Transitive Priority Inheritance

- task τ_1 uses a resource R_a guarded by a semaphore S_a
- task τ_3 uses a resource R_b guarded by a semaphore S_b
- task τ_2 uses both resources in a nested fashion (S_a protects the external critical section and S_b the internal one)

A transitive inheritance occurs when a high-priority task τ_H is blocked by a medium-priority task τ_M, which in turn is blocked by a low-priority task τ_L



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

Blocking Time (PIP)

- a task τ_i can be blocked at most once for each of the l_i lower priority tasks. Hence, for each lower priority task τ_j that can block τ_i, sum the duration of the longest critical section among those that can block τ_i

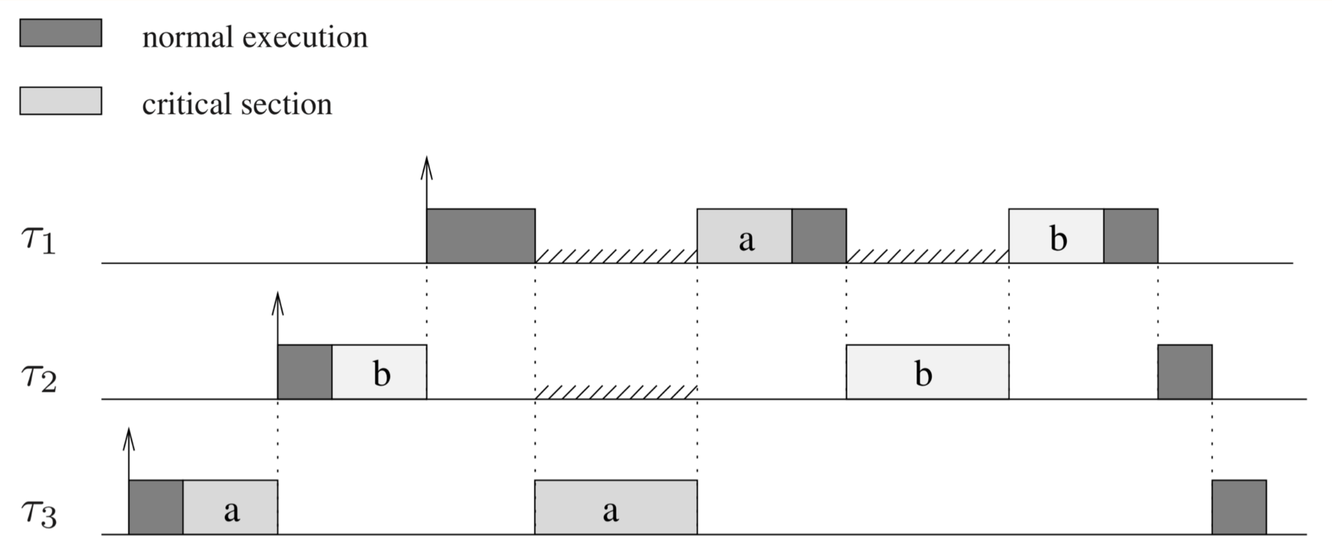
$$B_i^l = \sum_{j:P_j<P_i} \max_k \{\delta_{j,k} - 1 \mid Z_{j,k} \in \gamma_i\}$$

- a task τ_i can be blocked at most once for each of the s_i semaphores that can block τ_i. Hence, for each semaphore S_k that can block τ_i, sum the duration of the longest critical section among those that can block τ_i

$$B_i^s = \sum_{k=1}^{m} \max_j \{\delta_{j,k} - 1 \mid Z_{j,k} \in \gamma_i\}$$

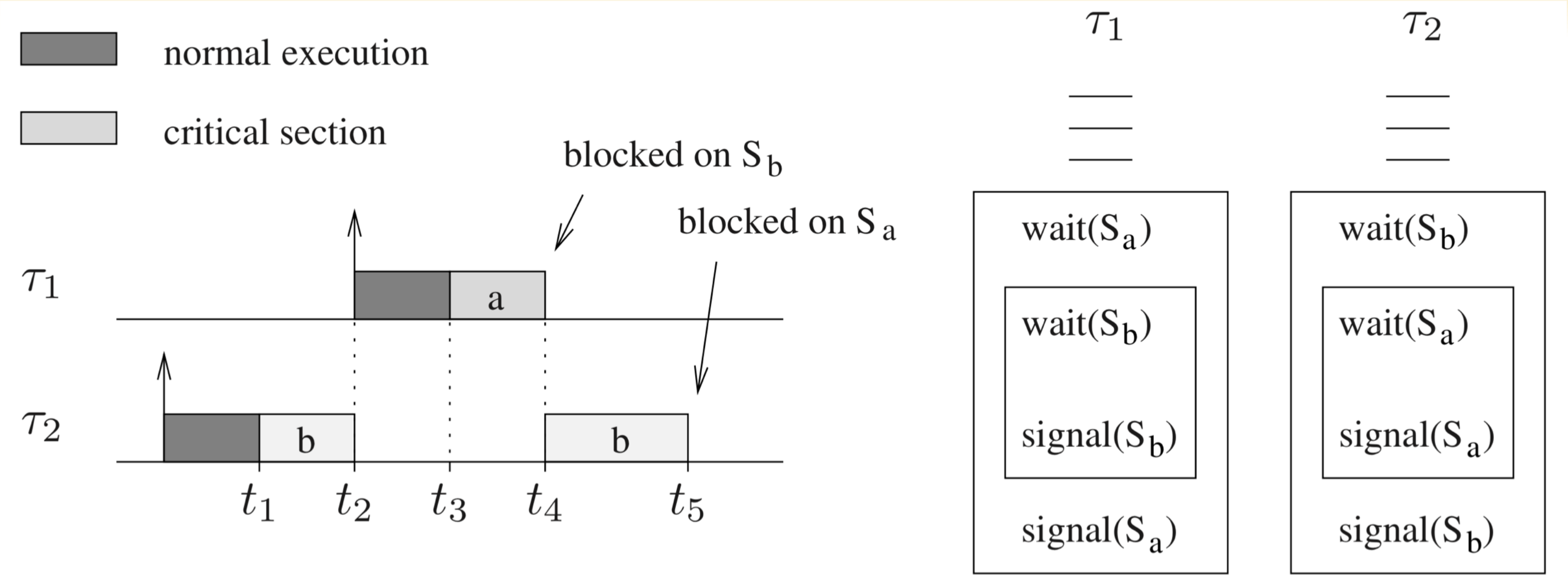
- a task τ_i can be blocked for minimum of the critical sections $B_i = \min(B_i^l, B_i^s)$

Chained Blocking



- τ_1 is blocked for the duration of two critical sections, once to wait for τ_3 to release S_a and then to wait for τ_2 to release S_b
- In the worst case, if τ_1 accesses n distinct semaphores that have been locked by n lower-priority tasks, τ_1 will be blocked for the duration of n critical sections.

Deadlock

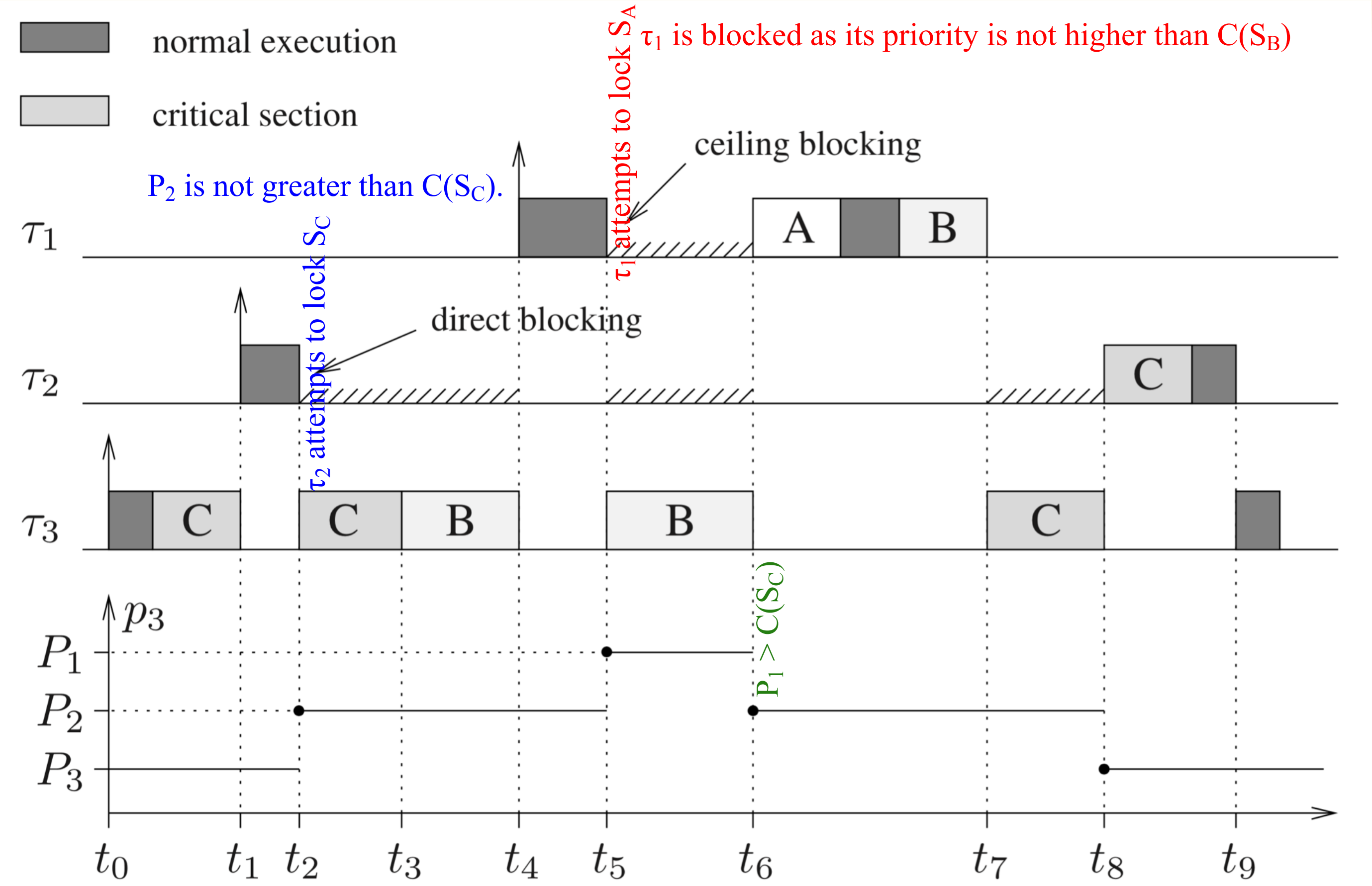


- the deadlock does not depend on the Priority Inheritance Protocol but is caused by an erroneous use of semaphores

Priority Ceiling Protocol (PCP)

- The Priority Ceiling Protocol (PCP)
  - bound the priority inversion phenomenon
  - prevent the formation of deadlocks and chained blocking
- Once a task enters its first critical section, it can never be blocked by lower-priority tasks until its completion
- Each semaphore is assigned a *priority ceiling* equal to the highest priority of the tasks that can lock it

Example Priority Ceiling Protocol



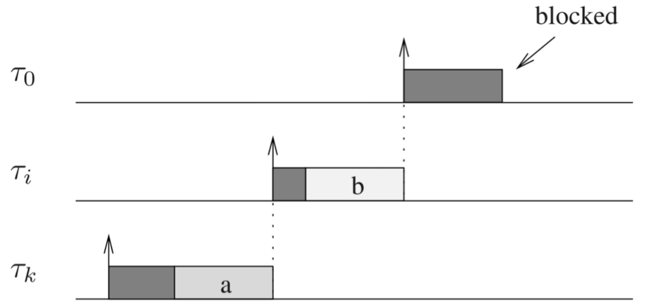
$$\begin{cases} C(S_A) = P_1 \\ C(S_B) = P_1 \\ C(S_C) = P_2. \end{cases}$$

Ceiling Blocking is necessary for avoiding deadlock and chained blocking

Lemma and Proof

If a task τ_k is preempted within a critical section Z_a by a task τ_i that enters a critical section Z_b, then, under the Priority Ceiling Protocol, τ_k cannot inherit a priority higher than or equal to that of task τ_i until τ_i completes.

- If τ_k inherits a priority higher than or equal to that of task τ_i before τ_i completes, there must exist a task τ_0 blocked by τ_k, such that $P_0 \geq P_i$.
- This leads to the contradiction that τ_0 cannot be blocked by τ_k.
- Since τ_i enters its critical section, its priority must be higher than the maximum ceiling C^* of the semaphores currently locked by all lower-priority tasks.
- Hence, $P_0 \geq P_i > C^*$.
- But since $P_0 > C^*$, τ_0 cannot be blocked by τ_k



Lemma and Proof

The Priority Ceiling Protocol prevents transitive blocking

- Suppose that a transitive block occurs
  - that is, there exist three tasks τ_1, τ_2, and τ_3, with decreasing priorities, such that τ_3 blocks τ_2 and τ_2 blocks τ_1.

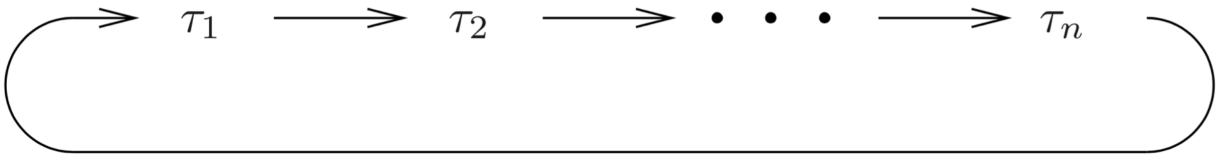
- By the transitivity of the protocol, τ_3 will inherit the priority of τ_1.

- This contradicts the Lemma, which shows that τ_3 cannot inherit a priority higher than or equal to P_2.

- Thus, PCP prevents transitive blocking.

Lemma and Proof

The Priority Ceiling Protocol prevents deadlocks



- Assume that a task cannot deadlock by itself, a deadlock can only be formed by a cycle of tasks waiting for each other
- By the transitivity of the protocol, task τ_n would inherit the priority of τ_1, which is assumed to be higher than P_n.
- This contradicts prior Lemma.
- Hence PCP prevents deadlock.

Blocking Time Computation

A task τ_i can only be blocked by critical sections belonging to lower priority tasks with a resource ceiling higher than or equal to P_i.

$$\gamma_i = \{Z_{j,k} \mid (P_j < P_i) \text{ and } C(R_k) \geq P_i\}.$$

Since τ_i can be blocked at most once, the maximum blocking time τ_i can suffer is given by the duration of the longest critical section among those that can block τ_i

$$B_i = \max_{j,k}\{\delta_{j,k} - 1 \mid Z_{j,k} \in \gamma_i\}$$