http://cs.LTH.se/EDA040

Real-Time and Concurrent Programming

Lecture 7 (F7):

Scheduling Analysis

Klas Nilsson

2016-10-11

Content

Scheduling Test – Schedulability

- **Schedulability**
  - A system is schedulable if every thread always meets its deadline.
- **Schedulability Test**
  - Given a set of threads and knowledge of their properties, a schedulability test answers the question *Is this system schedulable?*.
  - Output: Yes or No

Scheduling Test – Schedulability

- **Schedulability**
  - A system is schedulable if every thread always meets its deadline.
- **Schedulability Test**
  - Given a set of threads and knowledge of their properties, a schedulability test answers the question *Is this system schedulable?*.
  - Output: Yes or No
- Scheduling analysis
  - The efforts to determine schedulability, carried out during engineering of the system.

Scheduling Test – Schedulability

- **Schedulability**
  - A system is schedulable if every thread always meets its deadline.
- **Schedulability Test**
  - Given a set of threads and knowledge of their properties, a schedulability test answers the question *Is this system schedulable?*.
  - Output: Yes or No
- Scheduling analysis
  - The efforts to determine schedulability, carried out during engineering of the system.
- Scheduling
  - Special case: Static scheduling done at engineering time.
  - Normal case: Dynamic scheduling by run-time system.
  - Schedulability implies: scheduling (should be) possible.
  - Priorities and deadlines are for real-time correctness; concurrency correctness should not depend on priorities!

Simplifications in scheduling analysis

We start with the following simplifications:

- Periodic threads
- No blocking
- Deadline = period
- Fixed-priority scheduling, e.g. RMS

Legend

- T = period
- C = worst case execution time
- U = C/T = CPU-utilization (in the worst case)
- R = response time
- D = deadline

RMS - Upper Bound Analysis (Liu & Layland)

Generally it is possible to guarantee schedulability if (N=number of threads):

$$\sum \left(\frac{C_i}{T_i}\right) < n\left(2^{\frac{1}{n}} - 1\right)$$

n	U
1	1
2	0.83
3	0.78
...	...
∞	0.69

HOWEVER: A system might be schedulable even if the CPU utilization is higher than the above utilization bound. Exact analysis is required in such cases.

RMS - Exact Analysis (Joseph & Pandya)

- In RMS upper bound analysis we can only tell that all threads will finish before their respective deadline. How much earlier? Look at the worst-case response times!

RMS - Exact Analysis (Joseph & Pandya)

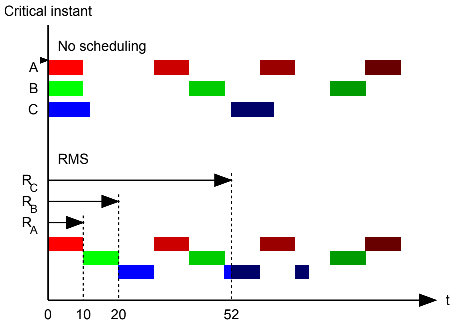
- In RMS upper bound analysis we can only tell that all threads will finish before their respective deadline. How much earlier? Look at the worst-case response times!
- Study what happens at the *critical instant* - when all threads are released simultaneously.

RMS - Exact Analysis (Joseph & Pandya)

- In RMS upper bound analysis we can only tell that all threads will finish before their respective deadline. How much earlier? Look at the worst-case response times!
- Study what happens at the *critical instant* - when all threads are released simultaneously.
- Theorem: If all threads will meet their first deadline after a critical instant, they will also meet all subsequent ones since all other scheduling situations are "easier".

RMS - Exact Analysis - Scheduling Diagram

Consider the example below, all threads are released at t=0:



$$U_A = \frac{C_A}{T_A} = \frac{10}{30} = 0.33$$
$$U_B = \frac{C_B}{T_B} = \frac{10}{40} = 0.25$$
$$U_C = \frac{C_C}{T_C} = \frac{12}{52} = 0.23$$

$$U = \sum U_i = 0.81 > 0.78$$

Worst-case response times:

$R_a = C_a = 10 \quad R_b = C_b + C_a = 20 \quad R_c = C_c + C_a + C_b + C_a + C_b = 52$

RMS - Exact Response-time Analysis

The worst case response time is the shortest time R_i that satisfies the following equation:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

$hp(i)$	set of activities with higher priority than i
$\left\lceil \frac{R_i}{T_j} \right\rceil$	number of times activity i is preempted by the higher-priority activity j
$\lceil\rceil$	ceil, i.e. rounding upwards, e.g. $\lceil 1.6 \rceil = 2$

RMS - Exact Analysis - Iterative Calculation

Calculate response times using iteration (iteration index as superscript):

$$R_i^{k+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j$$

$R_i^0 = 0$ used as starting value

R_i^k iteratively calculated until stable

RMS - Exact Analysis - Iterative Calculation Example

Example as before:

$$R_i^{k+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j$$

Thread	C	T
A	10	30
B	10	40
C	12	52

Iterative calculation for $i \in$ A, B, C:

$R_a^0 = 0$
$R_a^1 = 10$ (*)
$R_a^2 = 10$ (stable)

$R_b^0 = 0$
$R_b^1 = 10 + 0 \cdot 10 = 10$
$R_b^2 = 10 + 1 \cdot 10 = 20$
$R_b^3 = 10 + 1 \cdot 10 = 20$ (stable)

$R_c^0 = 0$
$R_c^1 = 12 + 0 \cdot 10 + 0 \cdot 10 = 12$
$R_c^2 = 12 + 1 \cdot 10 + 1 \cdot 10 = 32$
$R_c^3 = 12 + 2 \cdot 10 + 1 \cdot 10 = 42$
$R_c^4 = 12 + 2 \cdot 10 + 2 \cdot 10 = 52$
$R_c^5 = 12 + 2 \cdot 10 + 2 \cdot 10 = 52$ (stable)

* no higher priority threads

Generalized Rate Monotonic Analysis

The assumptions made in RMS are rarely the case in practice:

- Periodic threads
- No blocking
- Deadline (D) = Period (T)
- Instantaneous context switch

Therefore, *Generalized Rate Monotonic Analysis* extends the RMS analysis (by Sha, Rajkumar & Lehoczky, 1994):

1. Shorter deadlines ($D < T$)
2. Blocking (on shared resources, bounded by priority inheritance)

Generalized Rate Monotonic Analysis

The assumptions made in RMS are rarely the case in practice:

- Periodic threads
- No blocking
- Deadline (D) = Period (T)
- Instantaneous context switch

Therefore, *Generalized Rate Monotonic Analysis* extends the RMS analysis (by Sha, Rajkumar & Lehoczky, 1994):

1. Shorter deadlines ($D < T$)
2. Blocking (on shared resources, bounded by priority inheritance)
3. Non-periodic threads (to avoid overly pessimistic results)
4. Non-instantaneous release jitter / context switches / clock interrupts

Generalized Rate Monotonic Analysis

The assumptions made in RMS are rarely the case in practice:

- Periodic threads
- No blocking
- Deadline (D) = Period (T)
- Instantaneous context switch

Therefore, *Generalized Rate Monotonic Analysis* extends the RMS analysis (by Sha, Rajkumar & Lehoczky, 1994):

1. Shorter deadlines $(D < T)$
2. Blocking (on shared resources, bounded by priority inheritance)
3. Non-periodic threads (to avoid overly pessimistic results)
4. Non-instantaneous release jitter / context switches / clock interrupts

Items 1 and 2 you should know and be able to apply/use.
Items 3 and 4 you should know about.

Deadline Monotonic Scheduling - Shorter deadlines

- The situation *Deadline* = *Period* ($D = T$) is unusual - often $D < T$
  - Seldom occurring events but which must be handled quickly (time critical)
- Scheduling test
  - Answers *yes* or *no* to the question *Is this system of threads schedulable?*
  - Method
    1. Calculate the worst-case response time, R_i, for each thread, τ_i
    2. The system is schedulable if and only if: $R_i <= D_i$ (the deadline) for each thread τ_i
- $D < T$ - Deadline Monotonic Scheduling (variant of RMS)
  - Assign priority according to D (as opposed to T)
  - Calculate maximum response time (R) as before
  - Check that $R < D$
  - Scheduling analysis according to Joseph & Pandya still holds

The effect of blocking

Worst-case response-time in presence of blocking is:

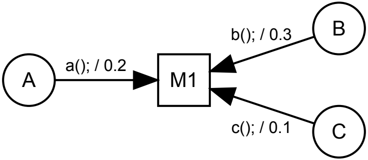
$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

B_i = the *blocking factor*, i.e. the maximum time *thread i* can be blocked by lower-priority threads. The blocking factor is the sum of:

- *Normal blocking* - The thread is blocked waiting for another thread to release a resource the thread has tried to lock.
- *Push-through blocking* - The thread is blocked by a lower priority thread which inherits a higher priority because it is blocking a higher-priority thread (only when priority inheritance is used).
- *Ceiling blocking* - The thread is blocked because the ceilings of other locked resources is too high. Only with the priority ceiling protocol.

Blocking - example 1

Can we successfully schedule the following system? The system uses RMS and basic inheritance protocol.



Thread	C	D	T
A	1	2	10
B	2	3	15
C	4	10	20

$R_A^0 = 0$

$R_A^1 = 1 + max(0.3, 0.1) + 0 = 1.3$

$R_A^2 = 1 + max(0.3, 0.1) + 0 = 1.3$

$R_B^0 = 0$

$R_B^1 = 2 + 0.1 + 0 = 2.1$

$R_B^2 = 2 + 0.1 + 1 \cdot 1 = 3.1$

$R_B^3 = 2 + 0.1 + 1 \cdot 1 = 3.1$ (*)

$R_C^0 = 0$

$R_C^1 = 4 + 0 + 0 = 4$

$R_C^2 = 4 + 0 + (1 \cdot 1 + 1 \cdot 2) = 7$

$R_C^3 = 4 + 0 + (1 \cdot 1 + 1 \cdot 2) = 7$

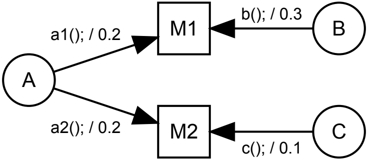
* $R_B > D_B$, *not* ok

Conclusion:

- Thread B does not meet its deadline due to blocking by thread C. Not a schedulable system!

Blocking - example 2

Can we remove the blocking problem by splitting the monitor into two?
The system uses RMS and basic inheritance protocol.



Thread	C	D	T
A	1	2	10
B	2	3	15
C	4	10	20

$R_A^0 = 0$
$R_A^1 = 1 + (0.3 + 0.1) + 0 = 1.4$
$R_A^2 = 1 + (0.3 + 0.1) + 0 = 1.4$

$R_B^0 = 0$
$R_B^1 = 2 + 0.1^{(**)} + 0 = 2.1$
$R_B^2 = 2 + 0.1 + 1 \cdot 1 = 3.1$
$R_B^3 = 2 + 0.1 + 1 \cdot 1 = 3.1$ (*)

$R_C^0 = 0$
$R_C^1 = 4 + 0 + 0 = 4$
$R_C^2 = 4 + 0 + (1 \cdot 1 + 1 \cdot 2) = 7$
$R_C^3 = 4 + 0 + (1 \cdot 1 + 1 \cdot 2) = 7$

* $R_B > D_B$, *not* ok

** push-through blocking

Conclusion:

- No, not possible due to push-through blocking!

The Ceiling Blocking Time

- Avoid multiple blockings for the highest priority thread.
- At the expense of avarage blocking times and runtime overhead.
- Should know properties and principles (details of the protocol is presently not part of the course).

The following plain-type slides (and page numbers) are from the Real-Time Systems course (L4 of FRTN01)

The Priority Ceiling Protocol

L. Sha, R. Rajkumar, J. Lehoczky, *Priority Inheritance Protocols: An Approach to Real-Time Synchronization,* IEEE Transactions on Computers, Vol. 39, No. 9, 1990

Restrictions on how we can lock (Wait, EnterMonitor) and unlock (Signal, LeaveMonitor) resources:

- a task must release all resources between invocations
- the computation time that a task i needs while holding semaphore s is bounded. $cs_{i,s}$ = the time length of the critical section for task i holding semaphore s
- a task may only lock semaphores from a fixed set of semaphores known a priory. $uses(i) =$ the set of semaphores that may be used by task i

The protocol:

- the *ceiling* of a semaphore, $ceil(s)$, is the priority of the highest priority task that uses the semaphore
- notation: $pri(i)$ is the priority of task i
- At run-time:
  - if a task i wants to lock a semaphore s, it can only do so if $pri(i)$ is **strictly higher** than the ceilings of all semaphores currently locked by **other** tasks
  - if not, task i will be blocked (task i is said to be blocked on the semaphore, S^*, with the highest priority ceiling of all semaphores currently locked by other jobs and task i is said to be blocked by the task that holds S^*)
  - when task i is blocked on S^*, the task currently holding S^* inherits the priority of task i

Properties:

- deadlock free
- a given task i is delayed at most once by a lower priority task
- the delay is a function of the time taken to execute the critical section

Deadlock free

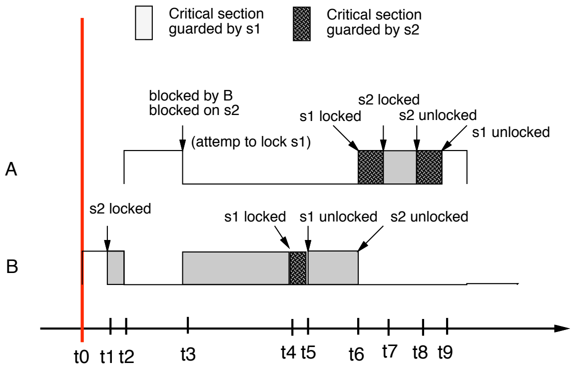
Example:

Task name	T	Priority
A	50	10
B	500	9

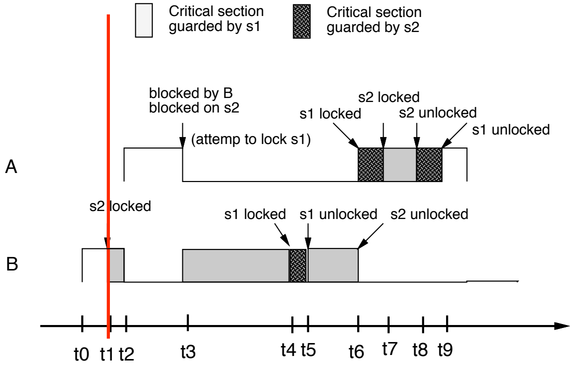
```
Task A        Task B

lock(s1)      lock(s2)
lock(s2)      lock(s1)
...           ...
unlock(s1)    unlock(s1)
unlock(s2)    unlock(s1)
```

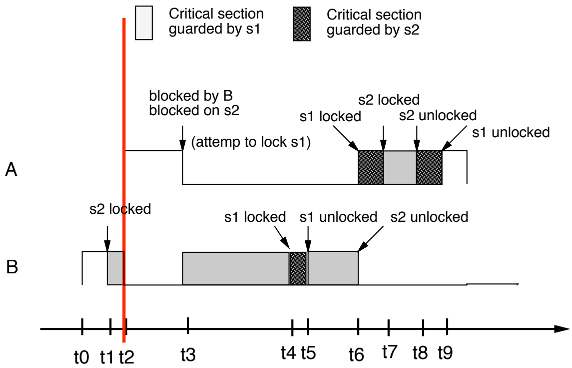
$$ceil(s_1) = 10, ceil(s_2) = 10$$



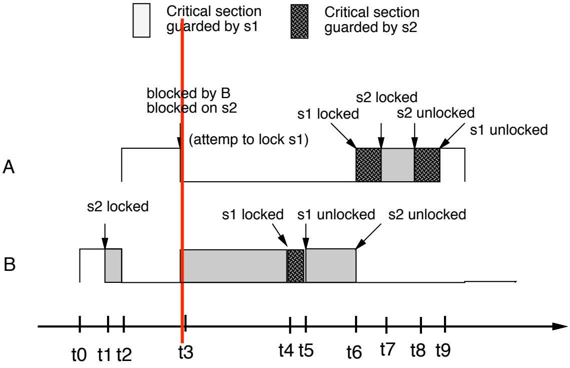
- t_0: B starts executing



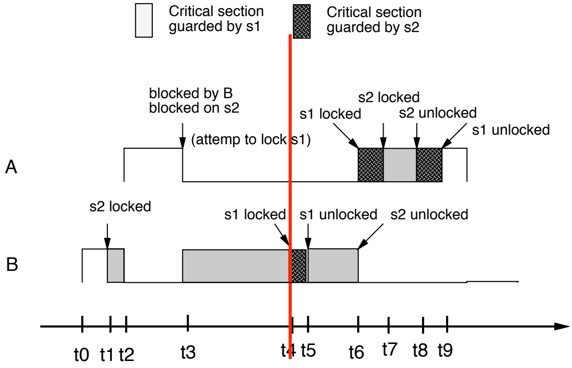
- t_1: B attempts to lock s_2. It succeeds since no lock is held by another task.



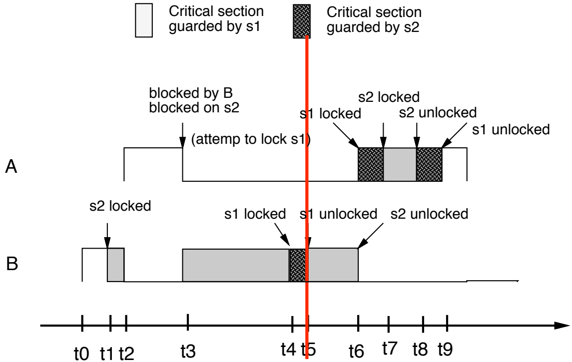
- t_2: A preempts B



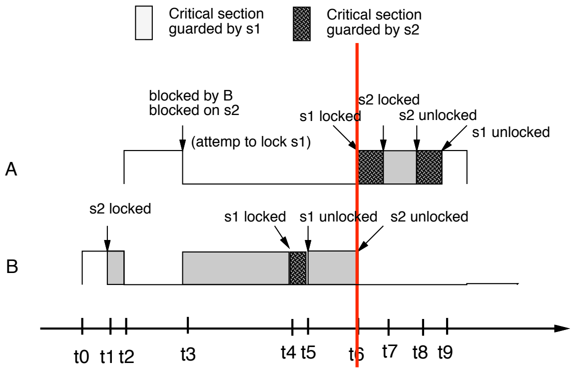
- t_3: A tries to lock s_1. A fails since A's priority (10) is not strictly higher than the ceiling of s_2 (10) that is held by B
- A is blocked by B
- A is blocked on s_2
- The priority of B is raised to 10.



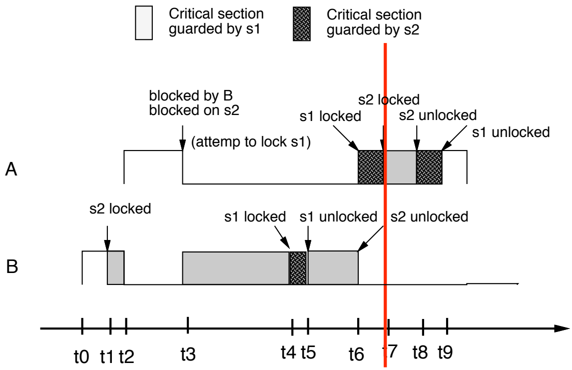
- t_4: B attempts to lock s_1. B succeeds since there are no locks held by any other tasks.



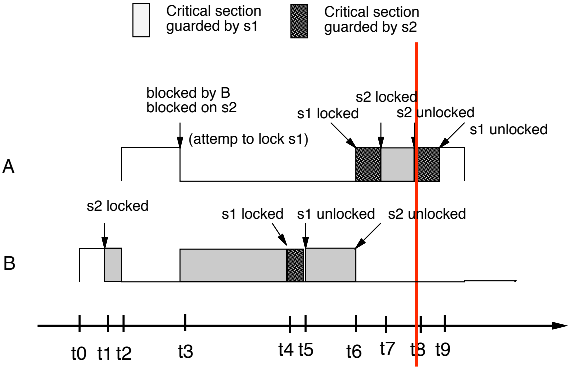
- t_5: B unlocks s_1



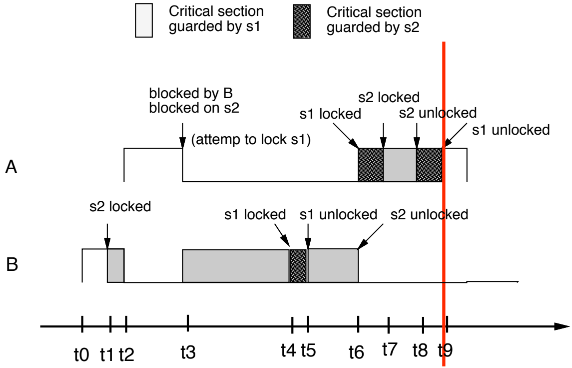
- t_6: B unlocks s_2
- The priority of B is lowered to its assigned priority (9)
- A preempts B, attempts to lock s_1 and succeeds



- t_7: A attempts to lock s_2. Succeeds



- t_8: A unlocks s_2



- t_9: A unlocks s_1

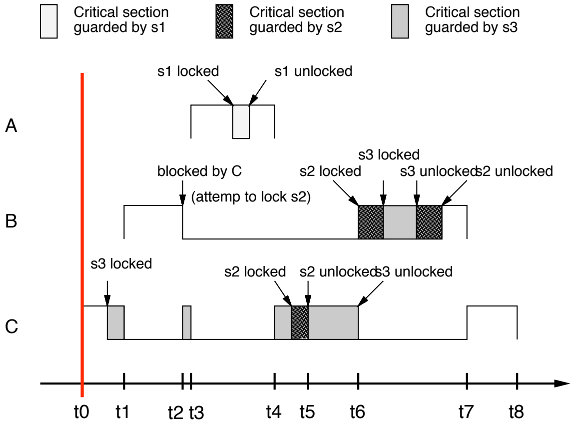
Example:

Task name	T	Priority
A	50	10
B	500	9
C	3000	8

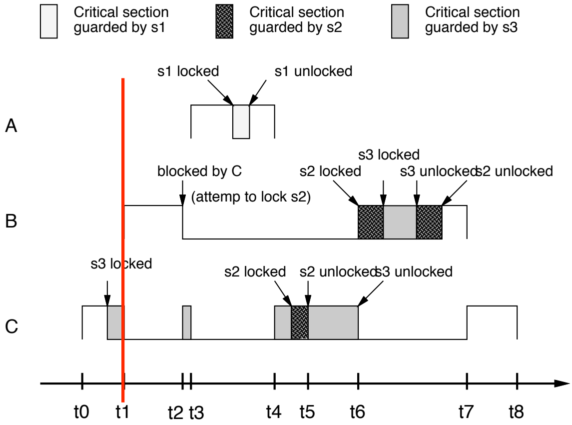
```
Task A        Task B        Task C

lock(s1)      lock(s2)      lock(s3)
..            ..            ..
unlock(s1)    lock(s3)      lock(s2)
..            ..            ..
              unlock(s3)    unlock(s2)
              ..            ..
              unlock(s2)    unlock(s3)
```

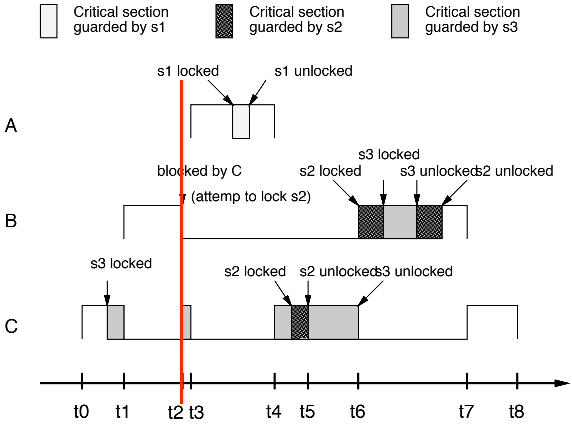
$$ceil(s_1) = 10, ceil(s_2) = ceil(s_3) = 9$$



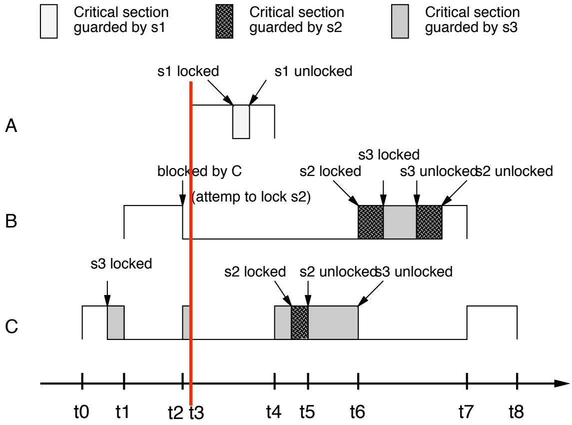
- t_0: C starts execution and then locks s_3



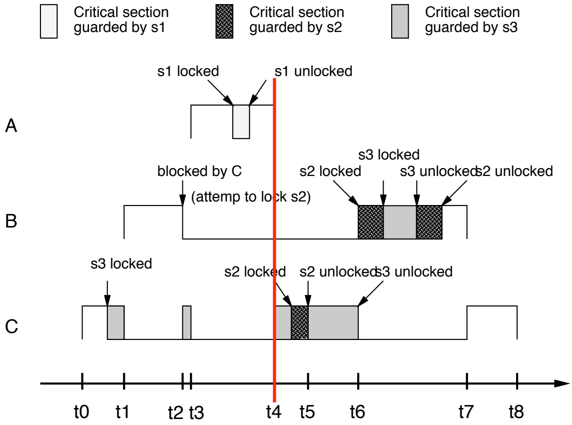
- t_1: B preempts C



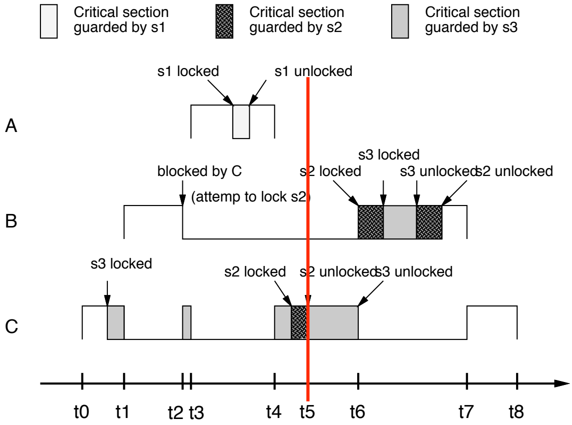
- t_2: B tries to lock s_2. B fails (the priority of B is not strictly higher than the ceiling of s_3 that is held by C) and blocks on s_3 (B is blocked by C). C inherits the priority of B.



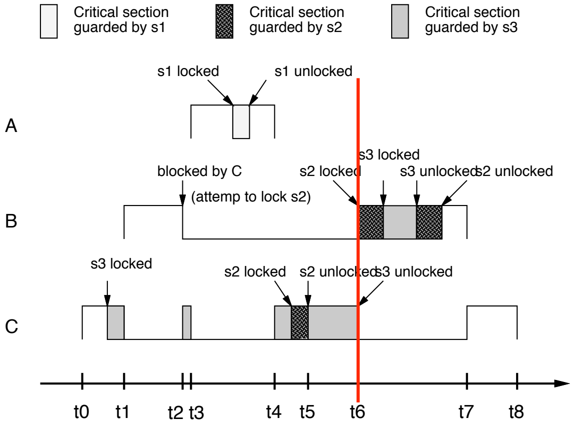
- t_3: A preempts C. Later is tries to lock s_1 and succeeds (the priority of A is higher than the ceiling of s_3).



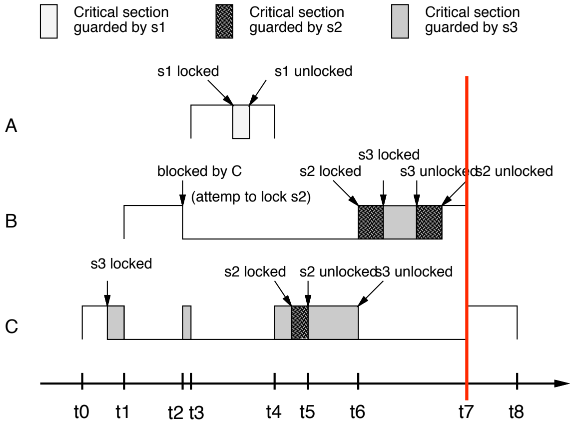
- t_4: A completes. C resumes and later tries to lock s_2 and succeeds (it is C itself that holds s_3).



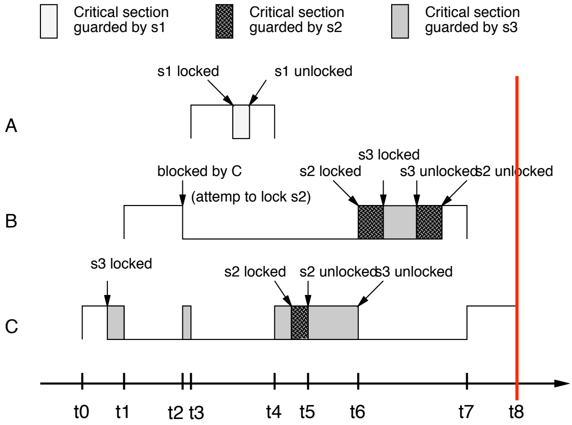
- t_5: C unlocks s_2



- t_6: C unlocks s_3, and gets back its basic priority. B preempts C, tries to lock s_2 and succeeds. Then B locks s_3, unlocks s_3 and unlocks s_2



- t_7: B completes and C is resumed.



- t_8: C completes

- A is never blocked
- B is blocked by C during the intervals $[t_2, t_3]$ and $[t_4, t_6]$. However, B is blocked for no more than the duration of one time critical section of the lower priority task C even though the actual blocking occurs over disjoint time intervals

General properties:

- with ordinary priority inheritance, a task i can be blocked for at most the duration of $\min(n, m)$ critical sections, where n is the number of lower priority tasks that could block i and m is the number of semaphores that can be used to block i
- with the priority ceiling inheritance, a task i can be blocked for at most the duration of one longest critical section
- sometimes priority ceiling introduces unnecessary blocking but the worst-case blocking delay is much less than for ordinary priority inheritance

The Immediate Inheritance Protocol

- when a task obtains a lock the priority of the task is immediately raised to the ceiling of the lock
- the same worst-case timing behavior as the priority ceiling protocol (also known as the Priority Ceiling Emulation Protocol and as the Priority Protect Protocol)
- easy to implement
- on a single-processor system it is not necessary to have any queues of blocked tasks for the locks (semaphores, monitors) – tasks waiting to acquire the locks will have lower priority than the task holding the lock and can, therefore be queued in `ReadyQueue`.

Priority Inheritance

Priority inheritance is a common, but not mandatory, feature of most Java implementations.

The Real-Time Java Specification requires that the priority inheritance protocol is implemented by default. The priority ceiling protocol is optional.

Sporadic (non-periodic) threads

A sporadic thread is triggered at unpredictable points in time.

Pessimistic analysis (our aproach so far)

- Model as a periodic thread with period equal to the minimum time between triggering events.
- Such a minimum time practically always exists due to physical limitations in the environment.
- Apply standard schedulability test.

Sporadic (non-periodic) threads

A sporadic thread is triggered at unpredictable points in time.

Pessimistic analysis (our aproach so far)

Less pessimistic: Analysis using a *sporadic server*

- Idea: Reserve a certain percentage of the CPU bandwidth for handling of sporadic events.
- Construct a periodic thread, the *sporadic server*, which handles all sporadic jobs. Let the thread run at most $C_{sporadic}$ time units every period, $T_{sporadic}$.
- Apply standard schedulability test on the sporadic server thread.
- Does not assume a minimum time between two external events.
- Cannot guarantee that deadlines are met for all sporadic events.

Including the run-time overhead

Release jitter

- Variations in the time it takes to actually release a thread once an external event triggering the thread has arrived (J_i).

Context switch

- It takes time to switch to another thread (C_{sw}).

Clock interrupts

- Periodic clock interrupts drives preemption and context switches (C_{tick}, T_{tick}, C_{queue}).

$$\omega_i = C_i + 2C_{sw} + B_i + \sum_{j \in hp(i)} \left\lceil \frac{\omega_i + J_i + T_{tick}}{T_j} \right\rceil (C_j + 2C_{sw})$$

$$+ \sum_{j \in alltasks} \left\lceil \frac{\omega_i + J_i + T_{tick}}{T_j} \right\rceil C_{queue} + \left\lceil \frac{\omega_i}{T_{tick}} \right\rceil C_{tick}$$

$$R_i = J_i + T_{tick} + \omega_i$$

Determining execution times

How do we determine C, B, etc. used in the scheduling analysis?

- Actual measurements
  - Run the application many times with many different inputs.
  - Measure the execution times, e.g. using a logic analyzer, and remember the longest.
  - Use the longest encountered execution times as C / B. (Optionally add 10-50%)
  - It can never be guaranteed that we have encountered the worst case!
  - Modern CPU features complicates matters: Cacheing? Pipelining?
- Formal code analysis
  - Analyze the program code statement by statement. Accumulate worst-case execution times for the statements.
  - How do you analyse loops? For-statements, while-statements, etc.
  - Pessimistic! Cacheing? Pipelining?
  - Automatic tools needed! Few available.

Difficult problem! Still an open and important area for future research!