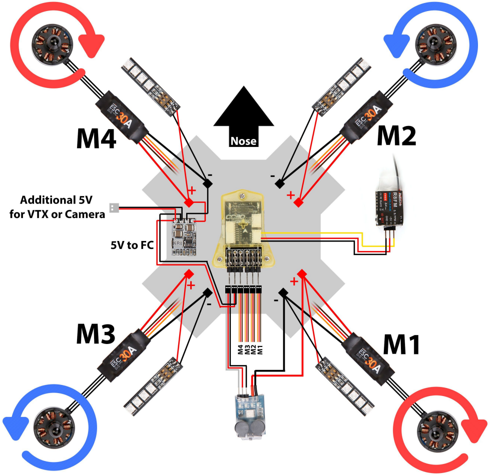
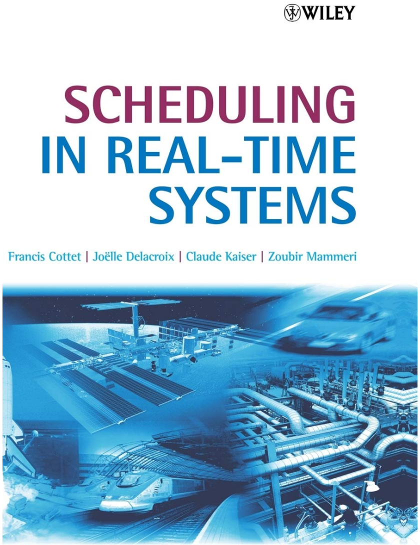


Scheduling of Dependent Tasks

Kizito NKURIKIYEYEZU, Ph.D.

Readings

- Read Chapter 3 of Cottet et al. (2002). Scheduling in Real-Time Systems.
- Topics
  - Task precedence relationships
  - Sharing critical resources
  - Mutual exclusion
  - Priority inversion
  - Deadlock



[1] Readings are based on Cottet, F., Delacroix, J., Mammeri, Z., & Kaiser, C. (2002). Scheduling in Real-Time Systems. Wiley.

Introduction

- The previous lecture assumed tasks were independent, i.e., there was no relationship between them
- This is too simplistic and does not reflect reality
- In most real-world application, inter-task cooperation and inter-task dependencies are a must
  - some tasks must respect the processing order

Introduction

- The previous lecture assumed tasks were independent, i.e., there was no relationship between them
- This is too simplistic and does not reflect reality
- In most real-world application, inter-task cooperation and inter-task dependencies are a must
  - some tasks must respect the processing order
  - mutual exclusion to protect shared resources

Introduction

- The previous lecture assumed tasks were independent, i.e., there was no relationship between them
- This is too simplistic and does not reflect reality
- In most real-world application, inter-task cooperation and inter-task dependencies are a must
  - some tasks must respect the processing order
  - mutual exclusion to protect shared resources
  - precedence constraints that correspond to synchronization or communication among tasks

Tasks with Precedence Relationships

- precedence constraint between two tasks τ_i and τ_j is denoted as $\tau_i \rightarrow \tau_j$ if the execution of task τ_i precedes that of task τ_j.
- In this case, task τ_j must await the completion of task τ_i before it can execute

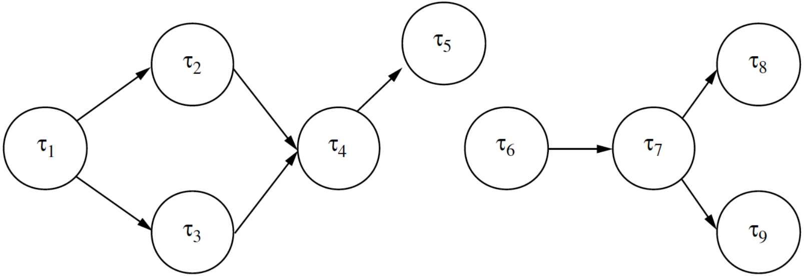


FIG 1. Example of two precedence graphs related to a set of nine tasks
The relationships is described through a graph where the nodes represent tasks and the arrows express the precedence constraint between two nodes.

Tasks with Precedence Relationships

- The previous precedence acyclic graph, however, represents a partial order on the task set.

Tasks with Precedence Relationships

- The previous precedence acyclic graph, however, represents a partial order on the task set.
- In general, we consider cases where *n* successive instance of a task can precede one instance of another task or vice versa.

Tasks with Precedence Relationships

- The previous precedence acyclic graph, however, represents a partial order on the task set.
- In general, we consider cases where *n* successive instance of a task can precede one instance of another task or vice versa.
- Fig. 2 shows an example of a generalized precedence relationship where the rate of communicating task are not equal.

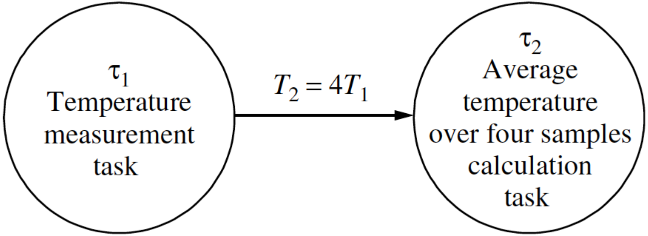


FIG 2. Example of a generalized precedence relationship between two tasks with different perio

Tasks with Precedence Relationships

Let's consider an example of in which τ_i has to communicate its results to task τ_j

- τ_i and τ_j have to be scheduled in a way that the execution of the k^{th} instance of task τ_i precedes the the execution of the k^{th} instance of the task τ_j. Thus, these task have the same rate, i.e., $T_i = T_j$

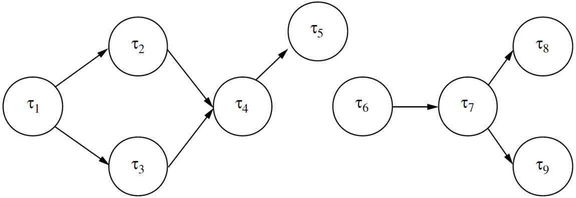


FIG 3. Example of two precedence graphs related to a set of nine tasks.
Note that tasks τ_1 to τ_5 have the same period and tasks τ_6 to τ_9 also have the same period.

[1] Unless there is a mitigating mechanisms such as cyclical asynchronous message buffers that are beyond the scope of this course

Tasks with Precedence Relationships

Let's consider an example of in which τ_i has to communicate its results to task τ_j

- τ_i and τ_j have to be scheduled in a way that the execution of the k^{th} instance of task τ_i precedes the the execution of the k^{th} instance of the task τ_j. Thus, these task have the same rate, i.e., $T_i = T_j$
- $T_i \neq T_j$, then tasks will run at the lowest rate sooner or later; consequently, the task with the shortest period will miss its deadline[1].

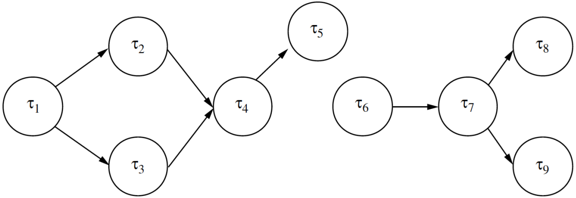


FIG 3. Example of two precedence graphs related to a set of nine tasks.
Note that tasks τ_1 to τ_5 have the same period and tasks τ_6 to τ_9 also have the same period.

[1] Unless there is a mitigating mechanisms such as cyclical asynchronous message buffers that are beyond the scope of this course

Tasks with Precedence Relationships

if $\tau_i \rightarrow \tau_j$, then the task parameters must be in accordance with the following rules[2]:

- release times: $r_j \geq r_i$

[2] Błazewicz, J. (1979). Deadline scheduling of tasks with ready times and resource constraints. Information Processing Letters, 8(2), 60–63. https://doi.org/10.1016/0020-0190(79)90143-1

Tasks with Precedence Relationships

if $\tau_i \rightarrow \tau_j$, then the task parameters must be in accordance with the following rules[2]:

- release times: $r_j \geq r_i$
- priorities: $priority_i \geq priority_j$, in accordance with the scheduling algorithm

[2] Błazewicz, J. (1979). Deadline scheduling of tasks with ready times and resource constraints. Information Processing Letters, 8(2), 60–63. https://doi.org/10.1016/0020-0190(79)90143-1

Precedence constraints and fixed-priority with rate monotonic algorithm

- We consider the rate monotonic (RM) and deadline monotonic (DM) algorithms

Precedence constraints and fixed-priority with rate monotonic algorithm

- We consider the rate monotonic (RM) and deadline monotonic (DM) algorithms
- In RM, tasks with shorter period get higher priorities.

Precedence constraints and fixed-priority with rate monotonic algorithm

- We consider the rate monotonic (RM) and deadline monotonic (DM) algorithms
- In RM, tasks with shorter period get higher priorities.
- We want to respect this rule and figure out how to modify the task parameters in order to take account of precedence constraints, i.e. to obtain an independent task set with modified parameters with the following rules:

Precedence constraints and fixed-priority with rate monotonic algorithm

- We consider the rate monotonic (RM) and deadline monotonic (DM) algorithms
- In RM, tasks with shorter period get higher priorities.
- We want to respect this rule and figure out how to modify the task parameters in order to take account of precedence constraints, i.e. to obtain an independent task set with modified parameters with the following rules:
  - A task cannot start before its predecessors

Precedence constraints and fixed-priority with rate monotonic algorithm

- We consider the rate monotonic (RM) and deadline monotonic (DM) algorithms
- In RM, tasks with shorter period get higher priorities.
- We want to respect this rule and figure out how to modify the task parameters in order to take account of precedence constraints, i.e. to obtain an independent task set with modified parameters with the following rules:
  - A task cannot start before its predecessors
  - A task cannot preempt its successors.

Precedence constraints and fixed-priority with rate monotonic algorithm

- We consider the rate monotonic (RM) and deadline monotonic (DM) algorithms
- In RM, tasks with shorter period get higher priorities.
- We want to respect this rule and figure out how to modify the task parameters in order to take account of precedence constraints, i.e. to obtain an independent task set with modified parameters with the following rules:
  - A task cannot start before its predecessors
  - A task cannot preempt its successors.
- If $\tau_i \rightarrow \tau_j$, then the release time and the priority of task parameters must be modified as follows:

Precedence constraints and fixed-priority with rate monotonic algorithm

- We consider the rate monotonic (RM) and deadline monotonic (DM) algorithms
- In RM, tasks with shorter period get higher priorities.
- We want to respect this rule and figure out how to modify the task parameters in order to take account of precedence constraints, i.e. to obtain an independent task set with modified parameters with the following rules:
  - A task cannot start before its predecessors
  - A task cannot preempt its successors.
- If $\tau_i \rightarrow \tau_j$, then the release time and the priority of task parameters must be modified as follows:
  - $r_j^* \geq max(r_j, r_i^*)$, where r_i^* is the modified release time of task τ_i

Precedence constraints and fixed-priority with rate monotonic algorithm

- We consider the rate monotonic (RM) and deadline monotonic (DM) algorithms
- In RM, tasks with shorter period get higher priorities.
- We want to respect this rule and figure out how to modify the task parameters in order to take account of precedence constraints, i.e. to obtain an independent task set with modified parameters with the following rules:
  - A task cannot start before its predecessors
  - A task cannot preempt its successors.
- If $\tau_i \rightarrow \tau_j$, then the release time and the priority of task parameters must be modified as follows:
  - $r_j^* \geq max(r_j, r_i^*)$, where r_i^* is the modified release time of task τ_i
  - $priority_i \geq pirority_j$ in according with the RM algorithm

Example

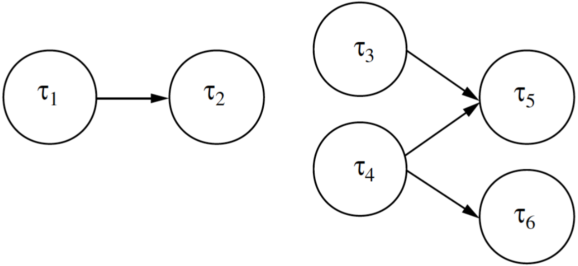


FIG 4. Precedence graphs of a set of six tasks

Example

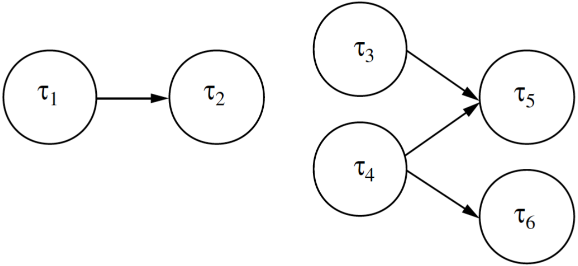


FIG 4. Precedence graphs of a set of six tasks

TAB 1. Example of priority mapping taking care of precedence constraints and using the RM scheduling algorithm

Task	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
Priority	6	5	4	3	2	1

Precedence constraints and fixed-priority with deadline monotonic algorithm

- With the deadline monotonic scheduling algorithm, tasks with shorter relative deadline get higher priorities
- The modifications of task parameters are close to those applied for RM scheduling except that the relative deadline is also changed in order to respect the priority assignment.
- If $\tau_i \rightarrow \tau_j$, then the release time, the relative deadline and the priority of the task parameters must be modified as follows:
  - $r_j^* \geq max(r_j, r_i^*)$, when r_i^* is the modified release time of task τ_i

Precedence constraints and fixed-priority with deadline monotonic algorithm

- With the deadline monotonic scheduling algorithm, tasks with shorter relative deadline get higher priorities
- The modifications of task parameters are close to those applied for RM scheduling except that the relative deadline is also changed in order to respect the priority assignment.
- If $\tau_i \to \tau_j$, then the release time, the relative deadline and the priority of the task parameters must be modified as follows:
  - $r_j^* \geq max(r_j, r_i^*)$, when r_i^* is the modified release time of task τ_i
  - $D_j^* \geq max(D_j, D_i^*)$, when D_i^* is the modified relative deadline of task τ_i

Precedence constraints and fixed-priority with deadline monotonic algorithm

- With the deadline monotonic scheduling algorithm, tasks with shorter relative deadline get higher priorities
- The modifications of task parameters are close to those applied for RM scheduling except that the relative deadline is also changed in order to respect the priority assignment.
- If $\tau_i \rightarrow \tau_j$, then the release time, the relative deadline and the priority of the task parameters must be modified as follows:
  - $r_j^* \geq max(r_j, r_i^*)$, when r_i^* is the modified release time of task τ_i
  - $D_j^* \geq max(D_j, D_i^*)$, when D_i^* is the modified relative deadline of task τ_i
  - $priority_i \geq priority_j$ in accordance with the DM scheduling algorithm

Precedence constraints and the EDF algorithm

review—the earliest deadline first (EDF) algorithm assigns priority to tasks according to their absolute deadline: the task with the earliest deadline will be executed as the highest priority.

- with the EDF algorithm, the modification of task parameters relies on the deadline d.
- Rules for modifying release times and deadlines of tasks are based on the following observations[3], [4]:
  1. To get $\tau_i \rightarrow \tau_j$, the release time r_j^* of task τ_j must be greater than or equal to its initial value or to the new release times τ_i^* of its immediate predecessors τ_i increased by their execution times C_i

$$r_i^* \geq max((r_i^* + C_i), r_j) \quad (1)$$

[3] Blazewicz J. (1997), Scheduling dependent tasks with different arrival times to meet deadlines, in Beilner H. and Gelenbe E. (eds) Modeling and Performance Evaluation of Computer Systems, North Holland, Amsterdam, pp. 57–65

[4] Chetto H., Silly M. and Bouchentouf T.(1990). Dynamic scheduling of real-time tasks under

Constraints and the EDF algorithm

2. If we have to get $\tau_i \rightarrow \tau_j$, the deadline d_i^* of task τ_i has to be replaced by the minimum between its initial value d_i by the new dealine d_j^* of the immediate successors τ_j decreased by their execution times C_j :

$$d_i^* \geq min((d_j^* - C_j), d_i) \quad (2)$$

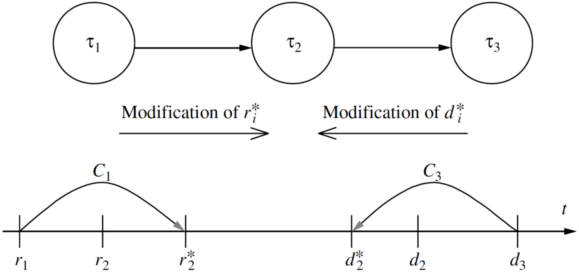


FIG 5. Modifications of task parameters in the case of EDF scheduling
The modifications begin with the tasks that have no predecessors for modifying their release times and with those with no successors for changing their deadlines. **Please see example on page 54.**

Tasks Sharing Critical Resources

Resource Sharing

- example of shared resource—data structures (e.g., queue), variables, main memory area, file, set of registers, I/O unit, etc.

[5] must be kept very short, otherwise they will adversely affect interrupt response times.

Resource Sharing

- example of shared resource—data structures (e.g., queue), variables, main memory area, file, set of registers, I/O unit, etc.
- Many shared resources do not allow simultaneous accesses but require mutual exclusion . These resources are called exclusive resources.

[5] must be kept very short, otherwise they will adversely affect interrupt response times.

Resource Sharing

- example of shared resource—data structures (e.g., queue), variables, main memory area, file, set of registers, I/O unit, etc.
- Many shared resources do not allow simultaneous accesses but require mutual exclusion . These resources are called exclusive resources.
- No two tasks are allowed to operate on the resource at the same time.

[5] must be kept very short, otherwise they will adversely affect interrupt response times.

Resource Sharing

- example of shared resource—data structures (e.g., queue), variables, main memory area, file, set of registers, I/O unit, etc.
- Many shared resources do not allow simultaneous accesses but require mutual exclusion . These resources are called exclusive resources.
- No two tasks are allowed to operate on the resource at the same time.
- Protection methods: interrupt disabling[5] and using semaphore or mutex

[5] must be kept very short, otherwise they will adversely affect interrupt response times.

Resource Sharing

- example of shared resource—data structures (e.g., queue), variables, main memory area, file, set of registers, I/O unit, etc.
- Many shared resources do not allow simultaneous accesses but require mutual exclusion . These resources are called exclusive resources.
- No two tasks are allowed to operate on the resource at the same time.
- Protection methods: interrupt disabling[5] and using semaphore or mutex
- In FreeRTOS, The taskENTER_CRITICAL() and taskEXIT_CRITICAL() provide a basic critical section implementation that works by simply disabling interrupts, either globally, or up to a specific interrupt priority level.

```

taskENTER_CRITICAL();
/* access to some exclusive resource*/
taskEXIT_CRITICAL();
```

LISTING 5: Mutual exclusion by disabling interrupts in FreeRTOS

[5] must be kept very short, otherwise they will adversely affect interrupt response times.

Resource Sharing

- Task J_2 has higher priority than task J_1

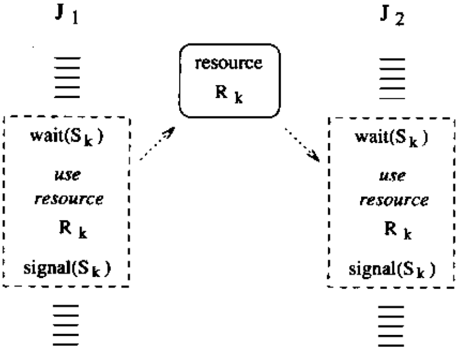


FIG 6. Two tasks sharing one resource

Resource Sharing

- Task J_2 has higher priority than task J_1
- Task J_1 is activated first and use the resource R (i.e, enters the critical section)

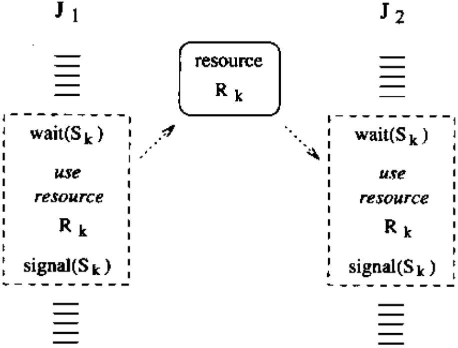


FIG 6. Two tasks sharing one resource

Resource Sharing

- Task J_2 has higher priority than task J_1
- Task J_1 is activated first and use the resource R (i.e, enters the critical section)
- If task J_2 (with higher priority) tries access the processor, it will preempt task J_1. However, if it tries to access the shared resources, it is blocked due to the mutual exclusion guaranteed by the semaphore.

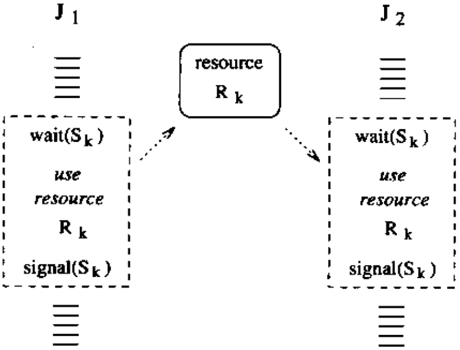


FIG 6. Two tasks sharing one resource

Resource Sharing

- Task J_2 has higher priority than task J_1
- Task J_1 is activated first and use the resource R (i.e, enters the critical section)
- If task J_2 (with higher priority) tries access the processor, it will preempt task J_1. However, if it tries to access the shared resources, it is blocked due to the mutual exclusion guaranteed by the semaphore.
- When blocked, the task J_1 can resume its execution and complete using the resource R

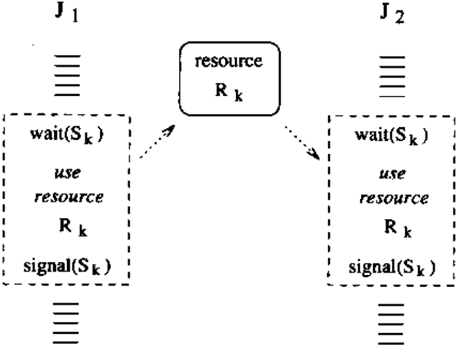


FIG 6. Two tasks sharing one resource

Resource Sharing

- Task J_2 has higher priority than task J_1
- Task J_1 is activated first and use the resource R (i.e, enters the critical section)
- If task J_2 (with higher priority) tries access the processor, it will preempt task J_1. However, if it tries to access the shared resources, it is blocked due to the mutual exclusion guaranteed by the semaphore.
- When blocked, the task J_1 can resume its execution and complete using the resource R
- This may lead to an uncontrolled blocking time for task J_2—which should normally run first since it has higher priority

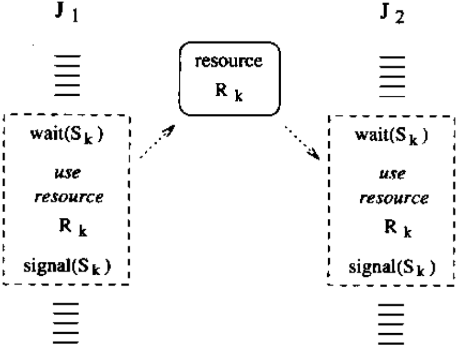


FIG 6. Two tasks sharing one resource

Resource Sharing

- Task J_2 has higher priority than task J_1
- Task J_1 is activated first and use the resource R (i.e, enters the critical section)
- If task J_2 (with higher priority) tries access the processor, it will preempt task J_1. However, if it tries to access the shared resources, it is blocked due to the mutual exclusion guaranteed by the semaphore.
- When blocked, the task J_1 can resume its execution and complete using the resource R
- This may lead to an uncontrolled blocking time for task J_2—which should normally run first since it has higher priority
- How do we solve this?

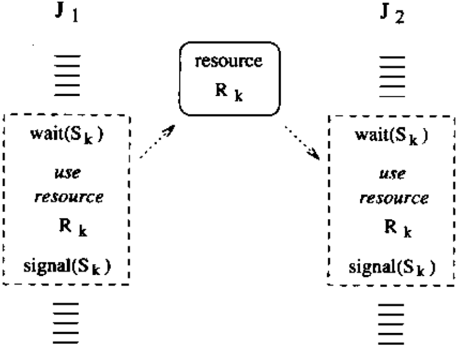


FIG 6. Two tasks sharing one resource

Mutual exclusion

In FreeRTOS , a mutex is a special type of semaphore that is used to control access to a resource that is shared between two or more tasks.

- When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared.

Mutual exclusion

In FreeRTOS , a mutex is a special type of semaphore that is used to control access to a resource that is shared between two or more tasks.

- When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared.
- For a task to access the resource legitimately, it must first successfully take the token. When the token holder has finished with the resource, it must give the token back.

Mutual exclusion

In FreeRTOS , a mutex is a special type of semaphore that is used to control access to a resource that is shared between two or more tasks.

- When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared.
- For a task to access the resource legitimately, it must first successfully take the token. When the token holder has finished with the resource, it must give the token back.
- Only when the token has been returned can another task successfully take the token, and then safely access the same shared resource.

Mutual exclusion

In FreeRTOS , a mutex is a special type of semaphore that is used to control access to a resource that is shared between two or more tasks.

- When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared.
- For a task to access the resource legitimately, it must first successfully take the token. When the token holder has finished with the resource, it must give the token back.
- Only when the token has been returned can another task successfully take the token, and then safely access the same shared resource.

Mutual exclusion

In FreeRTOS , a mutex is a special type of semaphore that is used to control access to a resource that is shared between two or more tasks.

- When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared.
- For a task to access the resource legitimately, it must first successfully take the token. When the token holder has finished with the resource, it must give the token back.
- Only when the token has been returned can another task successfully take the token, and then safely access the same shared resource.

```
SemaphoreHandle_t xMutex
int main( void){
  xMutex = xSemaphoreCreateMutex()
  if(xMutex != NULL ){
    // Create tasks that use the mutex
  }
```

```c
void vTask1( void *pvParameters ){
  while(true){
     ...
     xSemaphoreTake(xMutex,portMAX_DELAY);
     /* access to exclusive resource */
     xSemaphoreGive(xMutex)
     ...
  }
}
void vTask2( void *pvParameters ){
  while(true){
     ...
     xSemaphoreTake(xMutex,portMAX_DELAY);
     /* access to exclusive resource */
     xSemaphoreGive(xMutex)
     ...
  }
}
```

Priority inversion

- Priority inversion may occur in preemptive scheduling that is driven by fixed priority and where critical resources are protected by a mutual exclusion mechanism.

Priority inversion

- Priority inversion may occur in preemptive scheduling that is driven by fixed priority and where critical resources are protected by a mutual exclusion mechanism.
- Priority inversion —a case where a medium priority task is executed prior to a high priority task; this occurs because the latter is blocked —for an unbounded amount of time —by a low priority task. It is a consequence of shared resource access.

Priority inversion

- Priority inversion may occur in preemptive scheduling that is driven by fixed priority and where critical resources are protected by a mutual exclusion mechanism.
- Priority inversion —a case where a medium priority task is executed prior to a high priority task; this occurs because the latter is blocked —for an unbounded amount of time —by a low priority task. It is a consequence of shared resource access.
- Priority inversion, contravenes the scheduling specification and can induce deadline missing

Priority inversion

Consider a task set composed of four tasks τ_1, τ_2,τ_3,τ_4 having decreasing priorities (i.e., τ_1 has the highest priority and τ_4 the lowest) and where Tasks τ_2 and τ_4 share a critical resource R_1, the access of which is mutually exclusive

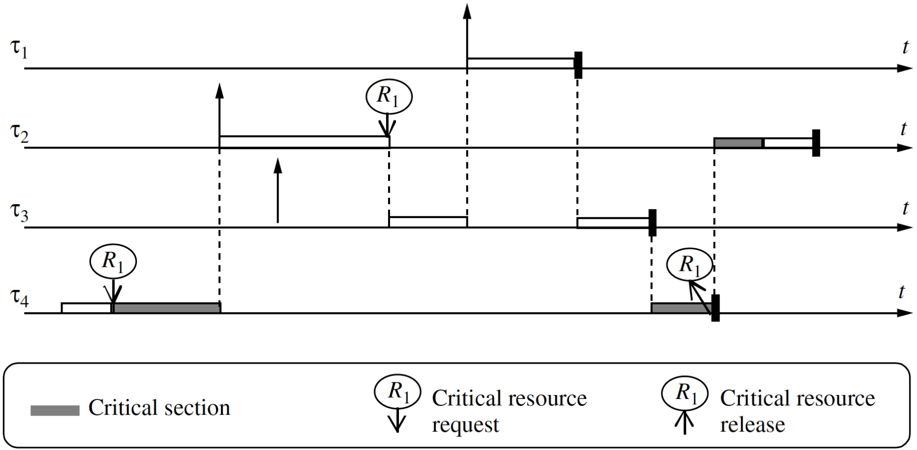


FIG 7. Example of priority inversion phenomenon

Priority inversion

- The lowest priority task τ_4 starts its execution first and after some time it enters a critical section using resource R_1.

Priority inversion

- The lowest priority task τ_4 starts its execution first and after some time it enters a critical section using resource R_1.
- When task τ_4 is in its critical section, the higher priority task τ_2 is released and preempts task τ_4

Priority inversion

- The lowest priority task τ_4 starts its execution first and after some time it enters a critical section using resource R_1.
- When task τ_4 is in its critical section, the higher priority task τ_2 is released and preempts task τ_4
- During the execution of task τ_2 , task τ_3 is released.

Priority inversion

- The lowest priority task τ_4 starts its execution first and after some time it enters a critical section using resource R_1.
- When task τ_4 is in its critical section, the higher priority task τ_2 is released and preempts task τ_4
- During the execution of task τ_2 , task τ_3 is released.
- Nevertheless, task τ_3, having a lower priority than task τ_2, must wait

Priority inversion

- The lowest priority task τ_4 starts its execution first and after some time it enters a critical section using resource R_1.
- When task τ_4 is in its critical section, the higher priority task τ_2 is released and preempts task τ_4
- During the execution of task τ_2 , task τ_3 is released.
- Nevertheless, task τ_3, having a lower priority than task τ_2, must wait
- When task τ_2 needs to enter its critical section, associated with the critical resource R_1 shared with task τ_4, it finds that the corresponding resource R_1 is held by task τ_4.—Thus it is blocked

Priority inversion

- The lowest priority task τ_4 starts its execution first and after some time it enters a critical section using resource R_1.
- When task τ_4 is in its critical section, the higher priority task τ_2 is released and preempts task τ_4
- During the execution of task τ_2 , task τ_3 is released.
- Nevertheless, task τ_3, having a lower priority than task τ_2, must wait
- When task τ_2 needs to enter its critical section, associated with the critical resource R_1 shared with task τ_4, it finds that the corresponding resource R_1 is held by task τ_4.—Thus it is blocked
- The highest priority task able to execute is task τ_3, So task τ_3, gets the processor and runs.

Priority inversion

- The lowest priority task τ_4 starts its execution first and after some time it enters a critical section using resource R_1.
- When task τ_4 is in its critical section, the higher priority task τ_2 is released and preempts task τ_4
- During the execution of task τ_2 , task τ_3 is released.
- Nevertheless, task τ_3, having a lower priority than task τ_2, must wait
- When task τ_2 needs to enter its critical section, associated with the critical resource R_1 shared with task τ_4, it finds that the corresponding resource R_1 is held by task τ_4.—Thus it is blocked
- The highest priority task able to execute is task τ_3, So task τ_3, gets the processor and runs.
- During this execution, the highest priority task τ_1 awakes. As a consequence task τ_3 is suspended and the processor is allocated to task τ_1.

- At the end of execution of task τ_1, task τ_3 can resume its execution until it reaches the end of its code.

- At the end of execution of task τ_1, task τ_3 can resume its execution until it reaches the end of its code.
- Now, only the lowest priority task τ_4, preempted in its critical section, can execute again. It resumes its execution until it releases critical resource R_1 required by the higher priority task τ_2

- At the end of execution of task τ_1, task τ_3 can resume its execution until it reaches the end of its code.
- Now, only the lowest priority task τ_4, preempted in its critical section, can execute again. It resumes its execution until it releases critical resource R_1 required by the higher priority task τ_2
- Then, task τ_2 can resume its execution by holding critical resource R1 necessary for its activity

- At the end of execution of task τ_1, task τ_3 can resume its execution until it reaches the end of its code.
- Now, only the lowest priority task τ_4, preempted in its critical section, can execute again. It resumes its execution until it releases critical resource R_1 required by the higher priority task τ_2
- Then, task τ_2 can resume its execution by holding critical resource R1 necessary for its activity
- Remarks:

- At the end of execution of task τ_1, task τ_3 can resume its execution until it reaches the end of its code.
- Now, only the lowest priority task τ_4, preempted in its critical section, can execute again. It resumes its execution until it releases critical resource R_1 required by the higher priority task τ_2
- Then, task τ_2 can resume its execution by holding critical resource R1 necessary for its activity
- Remarks:
  - Task τ_2's maximum blocking time varies and depends on the duration of the critical section of the lower priority tasks sharing the resource with it (e.g., τ_2)

- At the end of execution of task τ_1, task τ_3 can resume its execution until it reaches the end of its code.
- Now, only the lowest priority task τ_4, preempted in its critical section, can execute again. It resumes its execution until it releases critical resource R_1 required by the higher priority task τ_2
- Then, task τ_2 can resume its execution by holding critical resource R1 necessary for its activity
- Remarks:
  - Task τ_2's maximum blocking time varies and depends on the duration of the critical section of the lower priority tasks sharing the resource with it (e.g., τ_2)
  - The blocking time also depends on the execution time of the higher priority task τ_1

- At the end of execution of task τ_1, task τ_3 can resume its execution until it reaches the end of its code.
- Now, only the lowest priority task τ_4, preempted in its critical section, can execute again. It resumes its execution until it releases critical resource R_1 required by the higher priority task τ_2
- Then, task τ_2 can resume its execution by holding critical resource R1 necessary for its activity
- Remarks:
  - Task τ_2's maximum blocking time varies and depends on the duration of the critical section of the lower priority tasks sharing the resource with it (e.g., τ_2)
  - The blocking time also depends on the execution time of the higher priority task τ_1
  - A lower priority task, τ_3, increased the blocking time of a higher priority task τ_2, even if τ_3 does not share any critical resource with τ_2

- At the end of execution of task τ_1, task τ_3 can resume its execution until it reaches the end of its code.
- Now, only the lowest priority task τ_4, preempted in its critical section, can execute again. It resumes its execution until it releases critical resource R_1 required by the higher priority task τ_2
- Then, task τ_2 can resume its execution by holding critical resource R1 necessary for its activity
- Remarks:
  - Task τ_2's maximum blocking time varies and depends on the duration of the critical section of the lower priority tasks sharing the resource with it (e.g., τ_2)
  - The blocking time also depends on the execution time of the higher priority task τ_1
  - A lower priority task, τ_3, increased the blocking time of a higher priority task τ_2, even if τ_3 does not share any critical resource with τ_2
  - When there is priority inversion, the blocking time of each task cannot be bounded—this can lead to uncontrolled response time of each task.

Why this course?

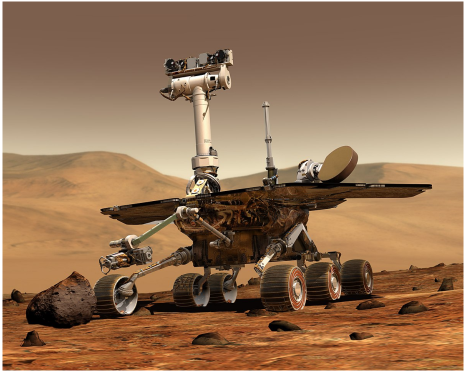


FIG 8. Artist's conception of NASA's Mars Exploration Rover on Mars. It's mission almost failed due priority inversion.

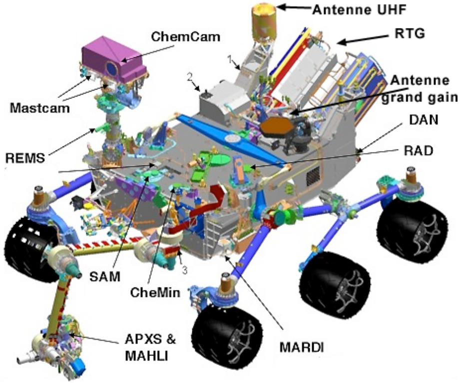


FIG 9. Instrumentation of the Mars Rover

[2] http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html

Mars rover and priority inversion

- A few days into the mission, the rover began experiencing total system resets, each resulting in losses of data[2].

[2] http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html

[6] VxWorks is a deterministic, priority-based preemptive RTOS with low latency and minimal jitter and is used for mission critical embedded systems. https://en.wikipedia.org/wiki/VxWorks

Mars rover and priority inversion

- A few days into the mission, the rover began experiencing total system resets, each resulting in losses of data[2].
- Priority inversion was the root cause because VxWorks[6]'s preemptive priority scheduling

[2] http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html

[6] VxWorks is a deterministic, priority-based preemptive RTOS with low latency and minimal jitter and is used for mission critical embedded systems. https://en.wikipedia.org/wiki/VxWorks

Mars rover and priority inversion

- A few days into the mission, the rover began experiencing total system resets, each resulting in losses of data[2].
- Priority inversion was the root cause because VxWorks[6]'s preemptive priority scheduling
  - Its bus management task ran frequently with high priority and access to the bus was synchronized with mutual exclusion locks

[2] http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html

[6] VxWorks is a deterministic, priority-based preemptive RTOS with low latency and minimal jitter and is used for mission critical embedded systems. https://en.wikipedia.org/wiki/VxWorks

Mars rover and priority inversion

- A few days into the mission, the rover began experiencing total system resets, each resulting in losses of data[2].
- Priority inversion was the root cause because VxWorks[6]'s preemptive priority scheduling
  - Its bus management task ran frequently with high priority and access to the bus was synchronized with mutual exclusion locks
  - The meteorological data gathering task ran a low priority thread and acquire a mutex when publishing its data, writes to the bus, and release the mutex

[2] http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html

[6] VxWorks is a deterministic, priority-based preemptive RTOS with low latency and minimal jitter and is used for mission critical embedded systems. https://en.wikipedia.org/wiki/VxWorks

Mars rover and priority inversion

- A few days into the mission, the rover began experiencing total system resets, each resulting in losses of data[2].
- Priority inversion was the root cause because VxWorks[6]'s preemptive priority scheduling
  - Its bus management task ran frequently with high priority and access to the bus was synchronized with mutual exclusion locks
  - The meteorological data gathering task ran a low priority thread and acquire a mutex when publishing its data, writes to the bus, and release the mutex
  - A communications task that ran with medium priority.

[2] http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html

[6] VxWorks is a deterministic, priority-based preemptive RTOS with low latency and minimal jitter and is used for mission critical embedded systems. https://en.wikipedia.org/wiki/VxWorks

Mars rover and priority inversion

- A few days into the mission, the rover began experiencing total system resets, each resulting in losses of data[2].
- Priority inversion was the root cause because VxWorks[6]'s preemptive priority scheduling
  - Its bus management task ran frequently with high priority and access to the bus was synchronized with mutual exclusion locks
  - The meteorological data gathering task ran a low priority thread and acquire a mutex when publishing its data, writes to the bus, and release the mutex
  - A communications task that ran with medium priority.
- It was possible for an interrupt to occur that caused the the medium priority communications task to be scheduled during the short interval while the high priority information bus thread was blocked waiting for the low priority meteorological data thread, consequently preventing the blocked information bus task from running.

[2] http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html

[6] VxWorks is a deterministic, priority-based preemptive RTOS with low latency and minimal jitter and is used for mission critical embedded systems. https://en.wikipedia.org/wiki/VxWorks

Solutions to Priority Inversion

- Disallow preemption during the execution of all critical sections.

[7] https://www.embedded.com/introduction-to-priority-inversion/
[8] https://www.embedded.com/how-to-use-priority-inheritance/
[9] https://en.wikipedia.org/wiki/Priority_ceiling_protocol
[10] https://en.wikipedia.org/wiki/Stack_Resource_Policy

Solutions to Priority Inversion

- Disallow preemption during the execution of all critical sections.
  - simple approach

[7] https://www.embedded.com/introduction-to-priority-inversion/
[8] https://www.embedded.com/how-to-use-priority-inheritance/
[9] https://en.wikipedia.org/wiki/Priority_ceiling_protocol
[10] https://en.wikipedia.org/wiki/Stack_Resource_Policy

Solutions to Priority Inversion

- Disallow preemption during the execution of all critical sections.
  - simple approach
  - but it creates unnecessary blocking as unrelated tasks may be blocked.

[7] https://www.embedded.com/introduction-to-priority-inversion/
[8] https://www.embedded.com/how-to-use-priority-inheritance/
[9] https://en.wikipedia.org/wiki/Priority_ceiling_protocol
[10] https://en.wikipedia.org/wiki/Stack_Resource_Policy

Solutions to Priority Inversion

- Disallow preemption during the execution of all critical sections.
  - simple approach
  - but it creates unnecessary blocking as unrelated tasks may be blocked.
- Resource access protocols—modify the priority of those tasks that cause blocking. When a task τ_i blocks one or more higher priority tasks, it temporarily assumes a higher priority. Several approaches exist:

[7] https://www.embedded.com/introduction-to-priority-inversion/

[8] https://www.embedded.com/how-to-use-priority-inheritance/

[9] https://en.wikipedia.org/wiki/Priority_ceiling_protocol

[10] https://en.wikipedia.org/wiki/Stack_Resource_Policy

Solutions to Priority Inversion

- Disallow preemption during the execution of all critical sections.
  - simple approach
  - but it creates unnecessary blocking as unrelated tasks may be blocked.
- Resource access protocols—modify the priority of those tasks that cause blocking. When a task τ_i blocks one or more higher priority tasks, it temporarily assumes a higher priority. Several approaches exist:
  - Priority Inheritance Protocol (PIP), for static priorities[7], [8]

[7] https://www.embedded.com/introduction-to-priority-inversion/

[8] https://www.embedded.com/how-to-use-priority-inheritance/

[9] https://en.wikipedia.org/wiki/Priority_ceiling_protocol

[10] https://en.wikipedia.org/wiki/Stack_Resource_Policy

Solutions to Priority Inversion

- Disallow preemption during the execution of all critical sections.
  - simple approach
  - but it creates unnecessary blocking as unrelated tasks may be blocked.
- Resource access protocols—modify the priority of those tasks that cause blocking. When a task τ_i blocks one or more higher priority tasks, it temporarily assumes a higher priority. Several approaches exist:
  - Priority Inheritance Protocol (PIP), for static priorities[7], [8]
  - Priority Ceiling Protocol (PCP), for static priorities[9]

[7] https://www.embedded.com/introduction-to-priority-inversion/

[8] https://www.embedded.com/how-to-use-priority-inheritance/

[9] https://en.wikipedia.org/wiki/Priority_ceiling_protocol

[10] https://en.wikipedia.org/wiki/Stack_Resource_Policy

Solutions to Priority Inversion

- Disallow preemption during the execution of all critical sections.
  - simple approach
  - but it creates unnecessary blocking as unrelated tasks may be blocked.
- Resource access protocols—modify the priority of those tasks that cause blocking. When a task τ_i blocks one or more higher priority tasks, it temporarily assumes a higher priority. Several approaches exist:
  - Priority Inheritance Protocol (PIP), for static priorities[7], [8]
  - Priority Ceiling Protocol (PCP), for static priorities[9]
  - Stack Resource Policy (SRP), for static and dynamic priorities[10]

[7] https://www.embedded.com/introduction-to-priority-inversion/

[8] https://www.embedded.com/how-to-use-priority-inheritance/

[9] https://en.wikipedia.org/wiki/Priority_ceiling_protocol

[10] https://en.wikipedia.org/wiki/Stack_Resource_Policy

Priority Inheritance Protocol

- summary—When a task τ_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks. It allows this task to use the critical resource as early as possible without going through the preemption. It avoids the unbounded priority inversion[11].

[11] https://www.geeksforgeeks.org/priority-inheritance-protocol-pip-in-synchronization/
[12] https://en.wikipedia.org/wiki/Deadlock
[13] https://www.informit.com/articles/article.aspx?p=30188&seqNum=3

Priority Inheritance Protocol

- summary—When a task τ_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks. It allows this task to use the critical resource as early as possible without going through the preemption. It avoids the unbounded priority inversion[11].
- assumptions

[11] https://www.geeksforgeeks.org/priority-inheritance-protocol-pip-in-synchronization/
[12] https://en.wikipedia.org/wiki/Deadlock
[13] https://www.informit.com/articles/article.aspx?p=30188&seqNum=3

Priority Inheritance Protocol

- summary—When a task τ_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks. It allows this task to use the critical resource as early as possible without going through the preemption. It avoids the unbounded priority inversion[11].
- assumptions
  - n tasks which cooperate through m shared resources

[11] https://www.geeksforgeeks.org/priority-inheritance-protocol-pip-in-synchronization/
[12] https://en.wikipedia.org/wiki/Deadlock
[13] https://www.informit.com/articles/article.aspx?p=30188&seqNum=3

Priority Inheritance Protocol

- summary—When a task τ_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks. It allows this task to use the critical resource as early as possible without going through the preemption. It avoids the unbounded priority inversion[11].
- assumptions
  - n tasks which cooperate through m shared resources
  - fixed priorities

[11] https://www.geeksforgeeks.org/priority-inheritance-protocol-pip-in-synchronization/
[12] https://en.wikipedia.org/wiki/Deadlock
[13] https://www.informit.com/articles/article.aspx?p=30188&seqNum=3

Priority Inheritance Protocol

- summary—When a task τ_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks. It allows this task to use the critical resource as early as possible without going through the preemption. It avoids the unbounded priority inversion[11].
- assumptions
  - n tasks which cooperate through m shared resources
  - fixed priorities
  - all critical sections on a resource begin with a take() and end with a give operation

[11] https://www.geeksforgeeks.org/priority-inheritance-protocol-pip-in-synchronization/
[12] https://en.wikipedia.org/wiki/Deadlock
[13] https://www.informit.com/articles/article.aspx?p=30188&seqNum=3

Priority Inheritance Protocol

- summary—When a task τ_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks. It allows this task to use the critical resource as early as possible without going through the preemption. It avoids the unbounded priority inversion[11].
- assumptions
  - *n* tasks which cooperate through *m* shared resources
  - fixed priorities
  - all critical sections on a resource begin with a take() and end with a give operation
- advantages

[11] https://www.geeksforgeeks.org/priority-inheritance-protocol-pip-in-synchronization/
[12] https://en.wikipedia.org/wiki/Deadlock
[13] https://www.informit.com/articles/article.aspx?p=30188&seqNum=3

Priority Inheritance Protocol

- summary—When a task τ_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks. It allows this task to use the critical resource as early as possible without going through the preemption. It avoids the unbounded priority inversion[11].
- assumptions
  - *n* tasks which cooperate through *m* shared resources
  - fixed priorities
  - all critical sections on a resource begin with a take() and end with a give operation
- advantages
  - It allows the different priority tasks to share the critical resources.

[11] https://www.geeksforgeeks.org/priority-inheritance-protocol-pip-in-synchronization/
[12] https://en.wikipedia.org/wiki/Deadlock
[13] https://www.informit.com/articles/article.aspx?p=30188&seqNum=3

Priority Inheritance Protocol

- summary—When a task τ_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks. It allows this task to use the critical resource as early as possible without going through the preemption. It avoids the unbounded priority inversion[11].
- assumptions
  - n tasks which cooperate through m shared resources
  - fixed priorities
  - all critical sections on a resource begin with a take() and end with a give operation
- advantages
  - It allows the different priority tasks to share the critical resources.
  - it avoids the unbounded priority inversion.

[11] https://www.geeksforgeeks.org/priority-inheritance-protocol-pip-in-synchronization/
[12] https://en.wikipedia.org/wiki/Deadlock
[13] https://www.informit.com/articles/article.aspx?p=30188&seqNum=3

Priority Inheritance Protocol

- summary—When a task τ_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks. It allows this task to use the critical resource as early as possible without going through the preemption. It avoids the unbounded priority inversion[11].
- assumptions
  - n tasks which cooperate through m shared resources
  - fixed priorities
  - all critical sections on a resource begin with a take() and end with a give operation
- advantages
  - It allows the different priority tasks to share the critical resources.
  - it avoids the unbounded priority inversion.
- disadvantages

[11] https://www.geeksforgeeks.org/priority-inheritance-protocol-pip-in-synchronization/
[12] https://en.wikipedia.org/wiki/Deadlock
[13] https://www.informit.com/articles/article.aspx?p=30188&seqNum=3

Priority Inheritance Protocol

- summary—When a task τ_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks. It allows this task to use the critical resource as early as possible without going through the preemption. It avoids the unbounded priority inversion[11].
- assumptions
  - n tasks which cooperate through m shared resources
  - fixed priorities
  - all critical sections on a resource begin with a take() and end with a give operation
- advantages
  - It allows the different priority tasks to share the critical resources.
  - it avoids the unbounded priority inversion.
- disadvantages
  - can lead to deadlock[12]

[11] https://www.geeksforgeeks.org/priority-inheritance-protocol-pip-in-synchronization/
[12] https://en.wikipedia.org/wiki/Deadlock
[13] https://www.informit.com/articles/article.aspx?p=30188&seqNum=3

Priority Inheritance Protocol

- summary—When a task τ_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks. It allows this task to use the critical resource as early as possible without going through the preemption. It avoids the unbounded priority inversion[11].
- assumptions
  - n tasks which cooperate through m shared resources
  - fixed priorities
  - all critical sections on a resource begin with a take() and end with a give operation
- advantages
  - It allows the different priority tasks to share the critical resources.
  - it avoids the unbounded priority inversion.
- disadvantages
  - can lead to deadlock[12]
  - can lead to chain blocking[13]

[11] https://www.geeksforgeeks.org/priority-inheritance-protocol-pip-in-synchronization/
[12] https://en.wikipedia.org/wiki/Deadlock
[13] https://www.informit.com/articles/article.aspx?p=30188&seqNum=3

Deadlock phenomenon

summary—a situation in which two or more tasks are blocked indefinitely because each task is waiting for a resource acquired by another blocked task (Fig. 10).

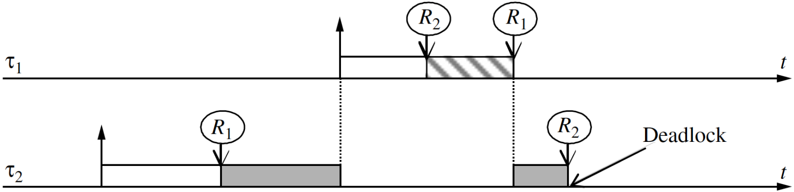


FIG 10. Example of the deadlock phenomenon

- Two tasks τ_1 and τ_1 use two critical resources R_1 and R_2.
- τ_1 and τ_2 access R_1 and R_2 in reverse order. Moreover, the priority of task τ_1 is greater than that of task τ_2.
- Now, suppose that task τ_2 executes first and locks resource R_1.

Deadlock phenomenon

- During the critical section of task τ_2 using resource R_1, task τ_1 awakes and preempts task τ_2 before it can lock the second resource R_2.

Deadlock phenomenon

- During the critical section of task τ_2 using resource R_1, task τ_1 awakes and preempts task τ_2 before it can lock the second resource R_2.
- Task τ_1 needs resource R_2 first, which is free, and it locks it.

Deadlock phenomenon

- During the critical section of task τ_2 using resource R_1, task τ_1 awakes and preempts task τ_2 before it can lock the second resource R_2.
- Task τ_1 needs resource R_2 first, which is free, and it locks it.
- Then task τ_1 needs resource R_1, which is held by task τ_2. So task τ_2 resumes and asks for resource R_2, which is not free.

Deadlock phenomenon

- During the critical section of task τ_2 using resource R_1, task τ_1 awakes and preempts task τ_2 before it can lock the second resource R_2.
- Task τ_1 needs resource R_2 first, which is free, and it locks it.
- Then task τ_1 needs resource R_1, which is held by task τ_2. So task τ_2 resumes and asks for resource R_2, which is not free.
- The final result is that task τ_2 is in possession of resource R_1 but is waiting for resource R_2 and task τ_1 is in possession of resource R_2 but is waiting for resource R_1.

Deadlock phenomenon

- During the critical section of task τ_2 using resource R_1, task τ_1 awakes and preempts task τ_2 before it can lock the second resource R_2.
- Task τ_1 needs resource R_2 first, which is free, and it locks it.
- Then task τ_1 needs resource R_1, which is held by task τ_2. So task τ_2 resumes and asks for resource R_2, which is not free.
- The final result is that task τ_2 is in possession of resource R_1 but is waiting for resource R_2 and task τ_1 is in possession of resource R_2 but is waiting for resource R_1.
- Neither task τ_1 nor task τ_2 will release the resource until its pending request is satisfied.

The end