Real-Time Operating Systems (0_KRI)
Process Interactions & Blocking

Ivan Cibrario Bertolotti

IEIIT-CNR / Politecnico di Torino

Academic Year 2006-2007

Outline

Further Extensions of the Process Model

- The basic process model provides a simple framework for schedulability analysis, but embodies several “unrealistic” assumptions.
- The assumption $D_i = T_i$ has already been replaced by $D_i \leq T_i$ to support, for example, sporadic processes as well as periodic processes.

Process interaction is useful

- Another simplistic assumption is the need for processes to be independent, because process interaction is needed in most real-world applications.
- The interaction leads to the possibility of a process being blocked until some future event occurs.

Further Extensions of the Process Model

- The basic process model provides a simple framework for schedulability analysis, but embodies several "unrealistic" assumptions.
- The assumption $D_i = T_i$ has already been replaced by $D_i \leq T_i$ to support, for example, sporadic processes as well as periodic processes.

Process interaction is useful

- Another simplistic assumption is the need for processes to be independent, because process interaction is needed in most real-world applications.
- The interaction leads to the possibility of a process being blocked until some future event occurs.

Further Extensions of the Process Model

- The basic process model provides a simple framework for schedulability analysis, but embodies several "unrealistic" assumptions.
- The assumption $D_i = T_i$ has already been replaced by $D_i \leq T_i$ to support, for example, sporadic processes as well as periodic processes.

Process interaction is useful

- Another simplistic assumption is the need for processes to be independent, because process interaction is needed in most real-world applications.
- The interaction leads to the possibility of a process being blocked until some future event occurs.

Process Interaction

- In a real-time system, process interaction must be designed with great care, above all when the tasks being synchronized have different priorities.
- In fact, when a process is waiting for a lower-priority process to complete some required computation, the process priority scheme is, in some sense, being undermined.
- This happens, for example, when processes access shared resources by means of a critical region protected by a mutual exclusion semaphore: if a lower-priority process is inside its critical region, any higher-priority process wanting to enter the critical region must wait until it exits.

Priority inversion

This phenomenon is called priority inversion and can have adverse effects on schedulability.

Process Interaction

- In a real-time system, process interaction must be designed with great care, above all when the tasks being synchronized have different priorities.
- In fact, when a process is waiting for a lower-priority process to complete some required computation, the process priority scheme is, in some sense, being undermined.
- This happens, for example, when processes access shared resources by means of a critical region protected by a mutual exclusion semaphore: if a lower-priority process is inside its critical region, any higher-priority process wanting to enter the critical region must wait until it exits.

Priority inversion

This phenomenon is called priority inversion and can have adverse effects on schedulability.

Process Interaction

- In a real-time system, process interaction must be designed with great care, above all when the tasks being synchronized have different priorities.
- In fact, when a process is waiting for a lower-priority process to complete some required computation, the process priority scheme is, in some sense, being undermined.
- This happens, for example, when processes access shared resources by means of a critical region protected by a mutual exclusion semaphore: if a lower-priority process is inside its critical region, any higher-priority process wanting to enter the critical region must wait until it exits.

Priority inversion

This phenomenon is called priority inversion and can have adverse effects on schedulability.

Process Interaction

- In a real-time system, process interaction must be designed with great care, above all when the tasks being synchronized have different priorities.
- In fact, when a process is waiting for a lower-priority process to complete some required computation, the process priority scheme is, in some sense, being undermined.
- This happens, for example, when processes access shared resources by means of a critical region protected by a mutual exclusion semaphore: if a lower-priority process is inside its critical region, any higher-priority process wanting to enter the critical region must wait until it exits.

Priority inversion

This phenomenon is called priority inversion and can have adverse effects on schedulability.

Dealing with Priority Inversion

In an ideal world, priority inversion should not exist but it cannot, in general, be totally eliminated. Nevertheless, its adverse effects can be minimized in several ways:

- Avoid useless interactions, when the problem can be solved in another way.
- Bound the blocking time by means of an appropriate technique, and compute the maximum blocking time in order to test for schedulability.
- Some blocking is inevitable, but it should be made as small as possible.

Dealing with Priority Inversion

In an ideal world, priority inversion should not exist but it cannot, in general, be totally eliminated. Nevertheless, its adverse effects can be minimized in several ways:

- Avoid useless interactions, when the problem can be solved in another way.
- Bound the blocking time by means of an appropriate technique, and compute the maximum blocking time in order to test for schedulability.
- Some blocking is inevitable, but it should be made as small as possible.

Dealing with Priority Inversion

In an ideal world, priority inversion should not exist but it cannot, in general, be totally eliminated. Nevertheless, its adverse effects can be minimized in several ways:

- Avoid useless interactions, when the problem can be solved in another way.
- Bound the blocking time by means of an appropriate technique, and compute the maximum blocking time in order to test for schedulability.
- Some blocking is inevitable, but it should be made as small as possible.

Dealing with Priority Inversion

In an ideal world, priority inversion should not exist but it cannot, in general, be totally eliminated. Nevertheless, its adverse effects can be minimized in several ways:

- Avoid useless interactions, when the problem can be solved in another way.
- Bound the blocking time by means of an appropriate technique, and compute the maximum blocking time in order to test for schedulability.
- Some blocking is inevitable, but it should be made as small as possible.

Unbounded Priority Inversion

- Consider the execution of 3 processes *H* (high priority), *M* (middle priority), and *L* (low priority), scheduled by a fixed priority scheduler.
- Let us also assume that *H* ed *L* share information by means of a critical region. The following sequence of events may happen:
  - *L* enters the critical region while neither *H* nor *M* are ready to run.
  - Then, both *H* and *M* are released and the scheduler makes *H* run.
  - *H* tries to enter the critical region, but waits on the mutex, because *L* is already inside its critical region.
  - On a single-processor system, *L* cannot run while *M* is running and, therefore, it cannot leave the critical region.

A low-priority process (*L*) prevents a higher-priority process (*H*) from entering the critical region. Even if the programmers which wrote *H* e *L* may even be unaware that *M* exists, the blocking time depends on the behavior of *M* and may last forever.

Unbounded Priority Inversion

- Consider the execution of 3 processes *H* (high priority), *M* (middle priority), and *L* (low priority), scheduled by a fixed priority scheduler.
- Let us also assume that *H* ed *L* share information by means of a critical region. The following sequence of events may happen:
  - *L* enters the critical region while neither *H* nor *M* are ready to run.
  - Then, both *H* and *M* are released and the scheduler makes *H* run.
  - *H* tries to enter the critical region, but waits on the mutex, because *L* is already inside its critical region.
  - On a single-processor system, *L* cannot run while *M* is running and, therefore, it cannot leave the critical region.

A low-priority process (*L*) prevents a higher-priority process (*H*) from entering the critical region. Even if the programmers which wrote *H* e *L* may even be unaware that *M* exists, the blocking time depends on the behavior of *M* and may last forever.

Unbounded Priority Inversion

- Consider the execution of 3 processes *H* (high priority), *M* (middle priority), and *L* (low priority), scheduled by a fixed priority scheduler.
- Let us also assume that *H* ed *L* share information by means of a critical region. The following sequence of events may happen:
  - *L* enters the critical region while neither *H* nor *M* are ready to run.
  - Then, both *H* and *M* are released and the scheduler makes *H* run.
  - *H* tries to enter the critical region, but waits on the mutex, because *L* is already inside its critical region.
  - On a single-processor system, *L* cannot run while *M* is running and, therefore, it cannot leave the critical region.

A low-priority process (*L*) prevents a higher-priority process (*H*) from entering the critical region. Even if the programmers which wrote *H* e *L* may even be unaware that *M* exists, the blocking time depends on the behavior of *M* and may last forever.

Unbounded Priority Inversion

- Consider the execution of 3 processes *H* (high priority), *M* (middle priority), and *L* (low priority), scheduled by a fixed priority scheduler.
- Let us also assume that *H* ed *L* share information by means of a critical region. The following sequence of events may happen:
  - *L* enters the critical region while neither *H* nor *M* are ready to run.
  - Then, both *H* and *M* are released and the scheduler makes *H* run.
  - *H* tries to enter the critical region, but waits on the mutex, because *L* is already inside its critical region.
  - On a single-processor system, *L* cannot run while *M* is running and, therefore, it cannot leave the critical region.

A low-priority process (*L*) prevents a higher-priority process (*H*) from entering the critical region. Even if the programmers which wrote *H* e *L* may even be unaware that *M* exists, the blocking time depends on the behavior of *M* and may last forever.

Unbounded Priority Inversion

- Consider the execution of 3 processes *H* (high priority), *M* (middle priority), and *L* (low priority), scheduled by a fixed priority scheduler.
- Let us also assume that *H* ed *L* share information by means of a critical region. The following sequence of events may happen:
  - *L* enters the critical region while neither *H* nor *M* are ready to run.
  - Then, both *H* and *M* are released and the scheduler makes *H* run.
  - *H* tries to enter the critical region, but waits on the mutex, because *L* is already inside its critical region.
  - On a single-processor system, *L* cannot run while *M* is running and, therefore, it cannot leave the critical region.

A low-priority process (*L*) prevents a higher-priority process (*H*) from entering the critical region. Even if the programmers which wrote *H* e *L* may even be unaware that *M* exists, the blocking time depends on the behavior of *M* and may last forever.

Unbounded Priority Inversion

- Consider the execution of 3 processes *H* (high priority), *M* (middle priority), and *L* (low priority), scheduled by a fixed priority scheduler.
- Let us also assume that *H* ed *L* share information by means of a critical region. The following sequence of events may happen:
  - *L* enters the critical region while neither *H* nor *M* are ready to run.
  - Then, both *H* and *M* are released and the scheduler makes *H* run.
  - *H* tries to enter the critical region, but waits on the mutex, because *L* is already inside its critical region.
  - On a single-processor system, *L* cannot run while *M* is running and, therefore, it cannot leave the critical region.

A low-priority process (*L*) prevents a higher-priority process (*H*) from entering the critical region. Even if the programmers which wrote *H* e *L* may even be unaware that *M* exists, the blocking time depends on the behavior of *M* and may last forever.

Unbounded Priority Inversion

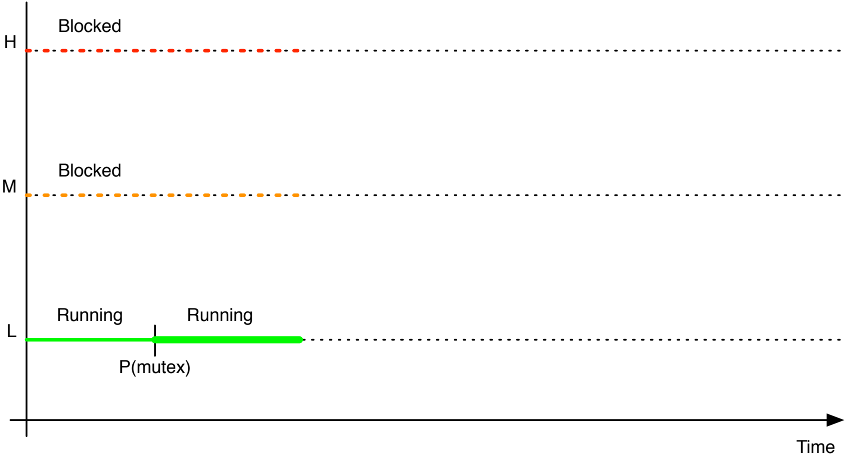
- Consider the execution of 3 processes *H* (high priority), *M* (middle priority), and *L* (low priority), scheduled by a fixed priority scheduler.
- Let us also assume that *H* ed *L* share information by means of a critical region. The following sequence of events may happen:
  - *L* enters the critical region while neither *H* nor *M* are ready to run.
  - Then, both *H* and *M* are released and the scheduler makes *H* run.
  - *H* tries to enter the critical region, but waits on the mutex, because *L* is already inside its critical region.
  - On a single-processor system, *L* cannot run while *M* is running and, therefore, it cannot leave the critical region.

A low-priority process (*L*) prevents a higher-priority process (*H*) from entering the critical region. Even if the programmers which wrote *H* e *L* may even be unaware that *M* exists, the blocking time depends on the behavior of *M* and may last forever.

Example

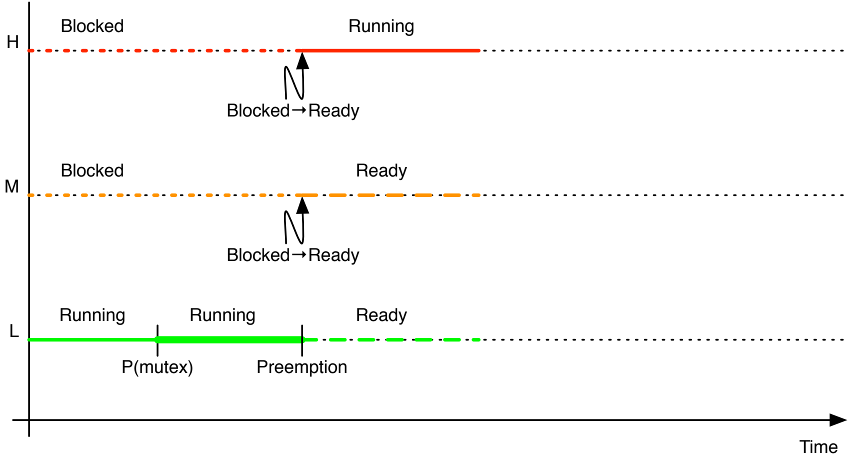
H and *M* are not ready, *L* is ready $\longrightarrow$ *L* runs.

Example



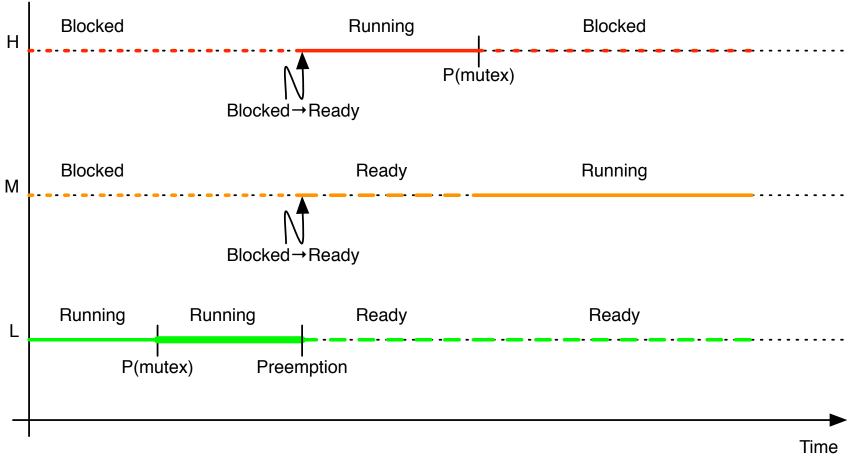
L performs *P*(mutex) without waiting, because the critical region is free $\longrightarrow$ *L* keeps running.

Example



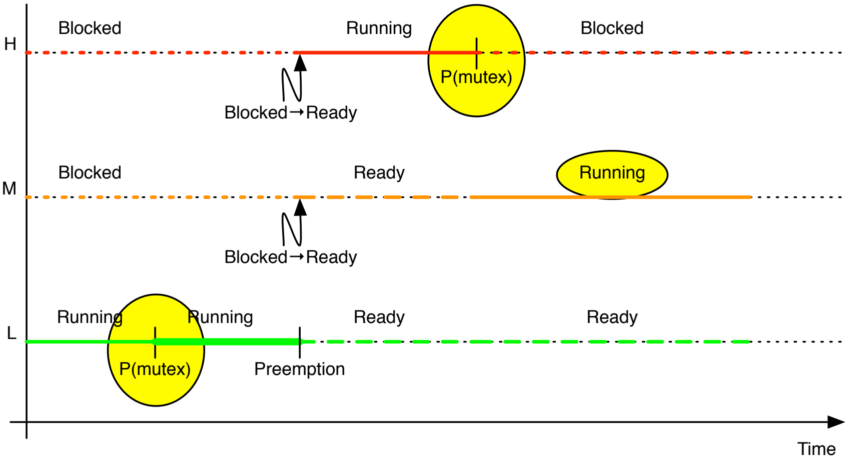
M and *H* are released → *L* is preempted and *H* starts running on its place.

Example



H tries to enter the critical region by means of *P*(mutex) and waits because *L* is already inside → *M* starts running.

Example



H may wait indefinitely, depending on the behavior of *M* → unbounded priority inversion.

Summary of the Example

1. In the example just considered, a certain amount of blocking cannot be eliminated:
   - To enter the critical region, H must be prepared to wait up to the maximum time needed by L to execute its critical region.
   - This is a direct consequence of the mutual exclusion necessary to access the shared resources in a safe way.
2. On the other hand, the blocking time of H cannot be bounded by the worst-case execution time, or duration, of the critical region of L in the general case:
   - The blocking time also depends on the worst-case execution time of M, a process which has "nothing to do" with H and L.
   - Moreover, any other intermediate-priority process which can preempt L will also have an effect on the blocking time of H.

Summary of the Example

1. In the example just considered, a certain amount of blocking cannot be eliminated:
    - To enter the critical region, *H* must be prepared to wait up to the maximum time needed by *L* to execute its critical region.
    - This is a direct consequence of the mutual exclusion necessary to access the shared resources in a safe way.
2. On the other hand, the blocking time of *H* cannot be bounded by the worst-case execution time, or duration, of the critical region of *L* in the general case:
    - The blocking time also depends on the worst-case execution time of *M*, a process which has "nothing to do" with *H* and *L*.
    - Moreover, any other intermediate-priority process which can preempt *L* will also have an effect on the blocking time of *H*.

Summary of the Example

1. In the example just considered, a certain amount of blocking cannot be eliminated:
   - To enter the critical region, *H* must be prepared to wait up to the maximum time needed by *L* to execute its critical region.
   - This is a direct consequence of the mutual exclusion necessary to access the shared resources in a safe way.
2. On the other hand, the blocking time of *H* cannot be bounded by the worst-case execution time, or duration, of the critical region of *L* in the general case:
   - The blocking time also depends on the worst-case execution time of *M*, a process which has "nothing to do" with *H* and *L*.
   - Moreover, any other intermediate-priority process which can preempt *L* will also have an effect on the blocking time of *H*.

Summary of the Example

1. In the example just considered, a certain amount of blocking cannot be eliminated:
   - To enter the critical region, H must be prepared to wait up to the maximum time needed by L to execute its critical region.
   - This is a direct consequence of the mutual exclusion necessary to access the shared resources in a safe way.
2. On the other hand, the blocking time of H cannot be bounded by the worst-case execution time, or duration, of the critical region of L in the general case:
   - The blocking time also depends on the worst-case execution time of M, a process which has “nothing to do” with H and L.
   - Moreover, any other intermediate-priority process which can preempt L will also have an effect on the blocking time of H.

Summary of the Example

1. In the example just considered, a certain amount of blocking cannot be eliminated:
   - To enter the critical region, *H* must be prepared to wait up to the maximum time needed by *L* to execute its critical region.
   - This is a direct consequence of the mutual exclusion necessary to access the shared resources in a safe way.
2. On the other hand, the blocking time of *H* cannot be bounded by the worst-case execution time, or duration, of the critical region of *L* in the general case:
   - The blocking time also depends on the worst-case execution time of *M*, a process which has “nothing to do” with *H* and *L*.
   - Moreover, any other intermediate-priority process which can preempt *L* will also have an effect on the blocking time of *H*.

Summary of the Example

1. In the example just considered, a certain amount of blocking cannot be eliminated:
   - To enter the critical region, *H* must be prepared to wait up to the maximum time needed by *L* to execute its critical region.
   - This is a direct consequence of the mutual exclusion necessary to access the shared resources in a safe way.
2. On the other hand, the blocking time of *H* cannot be bounded by the worst-case execution time, or duration, of the critical region of *L* in the general case:
   - The blocking time also depends on the worst-case execution time of *M*, a process which has “nothing to do” with *H* and *L*.
   - Moreover, any other intermediate-priority process which can preempt *L* will also have an effect on the blocking time of *H*.

A Simple Solution

- A very simple solution to the unbounded priority inversion problem is to completely disallow preemption during the execution of all critical regions.
- This can be obtained by either disabling interrupts or locking the scheduler within critical regions.
- In other words, it is “as if” a process inside a critical region implicitly assumed the highest possible priority in the system.

Limited applicability

However, this is only appropriate for very short critical regions, because it causes unnecessary blocking. A more sophisticated approach is needed in the general case.

A Simple Solution

- A very simple solution to the unbounded priority inversion problem is to completely disallow preemption during the execution of all critical regions.
- This can be obtained by either disabling interrupts or locking the scheduler within critical regions.
- In other words, it is "as if" a process inside a critical region implicitly assumed the highest possible priority in the system.

Limited applicability

However, this is only appropriate for very short critical regions, because it causes unnecessary blocking. A more sophisticated approach is needed in the general case.

A Simple Solution

- A very simple solution to the unbounded priority inversion problem is to completely disallow preemption during the execution of all critical regions.
- This can be obtained by either disabling interrupts or locking the scheduler within critical regions.
- In other words, it is "as if" a process inside a critical region implicitly assumed the highest possible priority in the system.

Limited applicability

However, this is only appropriate for very short critical regions, because it causes unnecessary blocking. A more sophisticated approach is needed in the general case.

A Simple Solution

- A very simple solution to the unbounded priority inversion problem is to completely disallow preemption during the execution of all critical regions.
- This can be obtained by either disabling interrupts or locking the scheduler within critical regions.
- In other words, it is "as if" a process inside a critical region implicitly assumed the highest possible priority in the system.

Limited applicability

However, this is only appropriate for very short critical regions, because it causes unnecessary blocking. A more sophisticated approach is needed in the general case.

The Priority Inheritance Protocol

The priority inheritance protocol

The priority inheritance protocol (Sha, Rajkumar and Lehoczky, 1990) offers a straightforward solution to the problem of unbounded priority inversion.

- The basic idea behind it is to dynamically increase the priority of the processes that cause blocking.
- In particular, when a process L is blocking one or more higher-priority processes $H_1, \ldots, H_n$, it temporarily inherits the highest priority of the blocked processes.
- This prevents any intermediate-priority process M from preempting L and thus prolonging the blocking experienced by the higher-priority processes.

The Priority Inheritance Protocol

The priority inheritance protocol

The priority inheritance protocol (Sha, Rajkumar and Lehoczky, 1990) offers a straightforward solution to the problem of unbounded priority inversion.

- The basic idea behind it is to dynamically increase the priority of the processes that cause blocking.
- In particular, when a process L is blocking one or more higher-priority processes $H_1, \ldots, H_n$, it temporarily inherits the highest priority of the blocked processes.
- This prevents any intermediate-priority process M from preempting L and thus prolonging the blocking experienced by the higher-priority processes.

The Priority Inheritance Protocol

The priority inheritance protocol

The priority inheritance protocol (Sha, Rajkumar and Lehoczky, 1990) offers a straightforward solution to the problem of unbounded priority inversion.

- The basic idea behind it is to dynamically increase the priority of the processes that cause blocking.
- In particular, when a process L is blocking one or more higher-priority processes $H_1, \ldots, H_n$, it temporarily inherits the highest priority of the blocked processes.
- This prevents any intermediate-priority process M from preempting L and thus prolonging the blocking experienced by the higher-priority processes.

The Priority Inheritance Protocol

The priority inheritance protocol

The priority inheritance protocol (Sha, Rajkumar and Lehoczky, 1990) offers a straightforward solution to the problem of unbounded priority inversion.

- The basic idea behind it is to dynamically increase the priority of the processes that cause blocking.
- In particular, when a process L is blocking one or more higher-priority processes $H_1, \ldots, H_n$, it temporarily inherits the highest priority of the blocked processes.
- This prevents any intermediate-priority process M from preempting L and thus prolonging the blocking experienced by the higher-priority processes.

Underlying Hypotheses

- Each process has a fixed initial (or baseline) priority and an active priority which is set dynamically and is initially set to the initial priority.
- Processes are scheduled by a fixed-priority scheduler on a single-processor system, that is, if several processes are ready to run, the highest-priority job will be run.
- Processes with the same priority are executed according to a First Come, First Served (FCFS) discipline.
- Semaphore wait queues are ordered by active priority.
- There is not any other source of blocking, for example I/O operations, in the system.

Underlying Hypotheses

- Each process has a fixed initial (or baseline) priority and an active priority which is set dynamically and is initially set to the initial priority.
- Processes are scheduled by a fixed-priority scheduler on a single-processor system, that is, if several processes are ready to run, the highest-priority job will be run.
- Processes with the same priority are executed according to a First Come, First Served (FCFS) discipline.
- Semaphore wait queues are ordered by active priority.
- There is not any other source of blocking, for example I/O operations, in the system.

Underlying Hypotheses

- Each process has a fixed initial (or baseline) priority and an active priority which is set dynamically and is initially set to the initial priority.
- Processes are scheduled by a fixed-priority scheduler on a single-processor system, that is, if several processes are ready to run, the highest-priority job will be run.
- Processes with the same priority are executed according to a First Come, First Served (FCFS) discipline.
- Semaphore wait queues are ordered by active priority.
- There is not any other source of blocking, for example I/O operations, in the system.

Underlying Hypotheses

- Each process has a fixed initial (or baseline) priority and an active priority which is set dynamically and is initially set to the initial priority.
- Processes are scheduled by a fixed-priority scheduler on a single-processor system, that is, if several processes are ready to run, the highest-priority job will be run.
- Processes with the same priority are executed according to a First Come, First Served (FCFS) discipline.
- Semaphore wait queues are ordered by active priority.
- There is not any other source of blocking, for example I/O operations, in the system.

Underlying Hypotheses

- Each process has a fixed initial (or baseline) priority and an active priority which is set dynamically and is initially set to the initial priority.
- Processes are scheduled by a fixed-priority scheduler on a single-processor system, that is, if several processes are ready to run, the highest-priority job will be run.
- Processes with the same priority are executed according to a First Come, First Served (FCFS) discipline.
- Semaphore wait queues are ordered by active priority.
- There is not any other source of blocking, for example I/O operations, in the system.

Protocol Definition

- When a process *H* tries to enter a critical region which is "busy" it blocks, but it also transmits its active priority to the process *L* which is blocking it, if the active priority of *L* is lower than *H*'s.
- Hence, *L* will execute the rest of its critical region with a priority equal to the inherited priority. In general, a process inherits the highest priority of the processes it blocks.
- When a process *L* exits a critical region, its active priority returns back to the nominal priority if it does not block any other process.
- Otherwise (critical regions can be nested) its active priority is set to the highest priority of the processes it still blocks.

Protocol Definition

- When a process H tries to enter a critical region which is "busy" it blocks, but it also transmits its active priority to the process L which is blocking it, if the active priority of L is lower than H's.
- Hence, L will execute the rest of its critical region with a priority equal to the inherited priority. In general, a process inherits the highest priority of the processes it blocks.
- When a process L exits a critical region, its active priority returns back to the nominal priority if it does not block any other process.
- Otherwise (critical regions can be nested) its active priority is set to the highest priority of the processes it still blocks.

Protocol Definition

- When a process H tries to enter a critical region which is "busy" it blocks, but it also transmits its active priority to the process L which is blocking it, if the active priority of L is lower than H's.
- Hence, L will execute the rest of its critical region with a priority equal to the inherited priority. In general, a process inherits the highest priority of the processes it blocks.
- When a process L exits a critical region, its active priority returns back to the nominal priority if it does not block any other process.
- Otherwise (critical regions can be nested) its active priority is set to the highest priority of the processes it still blocks.

Protocol Definition

- When a process *H* tries to enter a critical region which is "busy" it blocks, but it also transmits its active priority to the process *L* which is blocking it, if the active priority of *L* is lower than *H*'s.
- Hence, *L* will execute the rest of its critical region with a priority equal to the inherited priority. In general, a process inherits the highest priority of the processes it blocks.
- When a process *L* exits a critical region, its active priority returns back to the nominal priority if it does not block any other process.
- Otherwise (critical regions can be nested) its active priority is set to the highest priority of the processes it still blocks.

Example of Application

If the priority inheritance protocol is applied to the previous example, the following sequence of events occurs:

- When *L* tries to enter the critical region it does not block, and its priority does not change, as before.
- When *H* tries to enter the critical region it blocks as before, but it also transmits its priority to *L*, the process which is blocking it.
- Hence, *L* has an higher priority than *M* and runs on its place, because now only a process with a priority higher than *H*'s can preempt *L* (but it would have preempted *H*, too, according to the priority model in use).
- The priority of *L* goes back to its nominal value when it exits the critical region, because it does not block any other process.
- At this point, the execution of *H* is resumed and it enters the critical region.

Example of Application

If the priority inheritance protocol is applied to the previous example, the following sequence of events occurs:

- When *L* tries to enter the critical region it does not block, and its priority does not change, as before.
- When *H* tries to enter the critical region it blocks as before, but it also transmits its priority to *L*, the process which is blocking it.
- Hence, *L* has an higher priority than *M* and runs on its place, because now only a process with a priority higher than *H*'s can preempt *L* (but it would have preempted *H*, too, according to the priority model in use).
- The priority of *L* goes back to its nominal value when it exits the critical region, because it does not block any other process.
- At this point, the execution of *H* is resumed and it enters the critical region.

Example of Application

If the priority inheritance protocol is applied to the previous example, the following sequence of events occurs:

- When *L* tries to enter the critical region it does not block, and its priority does not change, as before.
- When *H* tries to enter the critical region it blocks as before, but it also transmits its priority to *L*, the process which is blocking it.
- Hence, *L* has an higher priority than *M* and runs on its place, because now only a process with a priority higher than *H*'s can preempt *L* (but it would have preempted *H*, too, according to the priority model in use).
- The priority of *L* goes back to its nominal value when it exits the critical region, because it does not block any other process.
- At this point, the execution of *H* is resumed and it enters the critical region.

Example of Application

If the priority inheritance protocol is applied to the previous example, the following sequence of events occurs:

- When *L* tries to enter the critical region it does not block, and its priority does not change, as before.
- When *H* tries to enter the critical region it blocks as before, but it also transmits its priority to *L*, the process which is blocking it.
- Hence, *L* has an higher priority than *M* and runs on its place, because now only a process with a priority higher than *H*'s can preempt *L* (but it would have preempted *H*, too, according to the priority model in use).
- The priority of *L* goes back to its nominal value when it exits the critical region, because it does not block any other process.
- At this point, the execution of *H* is resumed and it enters the critical region.

Example of Application

If the priority inheritance protocol is applied to the previous example, the following sequence of events occurs:

- When *L* tries to enter the critical region it does not block, and its priority does not change, as before.
- When *H* tries to enter the critical region it blocks as before, but it also transmits its priority to *L*, the process which is blocking it.
- Hence, *L* has an higher priority than *M* and runs on its place, because now only a process with a priority higher than *H*'s can preempt *L* (but it would have preempted *H*, too, according to the priority model in use).
- The priority of *L* goes back to its nominal value when it exits the critical region, because it does not block any other process.
- At this point, the execution of *H* is resumed and it enters the critical region.

Example of Application

If the priority inheritance protocol is applied to the previous example, the following sequence of events occurs:

- When *L* tries to enter the critical region it does not block, and its priority does not change, as before.
- When *H* tries to enter the critical region it blocks as before, but it also transmits its priority to *L*, the process which is blocking it.
- Hence, *L* has an higher priority than *M* and runs on its place, because now only a process with a priority higher than *H*'s can preempt *L* (but it would have preempted *H*, too, according to the priority model in use).
- The priority of *L* goes back to its nominal value when it exits the critical region, because it does not block any other process.
- At this point, the execution of *H* is resumed and it enters the critical region.

Pictorial Representation

TBD

Kinds of Blocking

The introduction of the priority inheritance protocol makes the concept of blocking more complex. There are now two distinct kinds of blocking:

Direct blocking. It occurs when a high-priority process tries to acquire a resource held by a lower-priority process. Direct blocking was already present and is necessary to ensure the consistency of the shared resources.

Push-through blocking. This kind of blocking is a consequence of the priority inheritance protocol, occurs when an intermediate-priority process (M in the previous example) cannot run because a lower-priority job (L) has temporarily inherited a higher priority, and can affect even processes which does not use any shared resource. Nevertheless, it is necessary to avoid unbounded priority inversion.

Kinds of Blocking

The introduction of the priority inheritance protocol makes the concept of blocking more complex. There are now two distinct kinds of blocking:

Direct blocking. It occurs when a high-priority process tries to acquire a resource held by a lower-priority process. Direct blocking was already present and is necessary to ensure the consistency of the shared resources.

Push-through blocking. This kind of blocking is a consequence of the priority inheritance protocol, occurs when an intermediate-priority process (M in the previous example) cannot run because a lower-priority job (L) has temporarily inherited a higher priority, and can affect even processes which does not use any shared resource. Nevertheless, it is necessary to avoid unbounded priority inversion.

Kinds of Blocking

The introduction of the priority inheritance protocol makes the concept of blocking more complex. There are now two distinct kinds of blocking:

Direct blocking. It occurs when a high-priority process tries to acquire a resource held by a lower-priority process. Direct blocking was already present and is necessary to ensure the consistency of the shared resources.

Push-through blocking. This kind of blocking is a consequence of the priority inheritance protocol, occurs when an intermediate-priority process (M in the previous example) cannot run because a lower-priority job (L) has temporarily inherited a higher priority, and can affect even processes which does not use any shared resource. Nevertheless, it is necessary to avoid unbounded priority inversion.

Further Steps

- In the following, the main properties of the priority inheritance protocol are presented.
- These properties are then used to show that the maximum blocking time that each process may experience is bounded.
- The same properties will also be useful to define several algorithms to compute the maximum blocking time for each process, in order to analyze the schedulability of a periodic task set.
- As usual, these algorithms entail a trade-off between the tightness of the bound they compute and their complexity.

Further Steps

- In the following, the main properties of the priority inheritance protocol are presented.
- These properties are then used to show that the maximum blocking time that each process may experience is bounded.
- The same properties will also be useful to define several algorithms to compute the maximum blocking time for each process, in order to analyze the schedulability of a periodic task set.
- As usual, these algorithms entail a trade-off between the tightness of the bound they compute and their complexity.

Further Steps

- In the following, the main properties of the priority inheritance protocol are presented.
- These properties are then used to show that the maximum blocking time that each process may experience is bounded.
- The same properties will also be useful to define several algorithms to compute the maximum blocking time for each process, in order to analyze the schedulability of a periodic task set.
- As usual, these algorithms entail a trade-off between the tightness of the bound they compute and their complexity.

Further Steps

- In the following, the main properties of the priority inheritance protocol are presented.
- These properties are then used to show that the maximum blocking time that each process may experience is bounded.
- The same properties will also be useful to define several algorithms to compute the maximum blocking time for each process, in order to analyze the schedulability of a periodic task set.
- As usual, these algorithms entail a trade-off between the tightness of the bound they compute and their complexity.

Nesting of Critical Regions

For simplicity, in the following discussion the fact that critical regions can be nested into each other will be neglected.

- Under the assumption that critical regions are properly nested, the set of critical regions belonging to the same process is partially ordered by region inclusion.
- For each process, it is possible to restrict the attention to the set of maximal critical regions, that is, the regions which are not included within any other.
- It can be shown that most results discussed in the following are still valid even if only maximal critical regions are taken into account, unless otherwise specified.

Nesting of Critical Regions

For simplicity, in the following discussion the fact that critical regions can be nested into each other will be neglected.

- Under the assumption that critical regions are properly nested, the set of critical regions belonging to the same process is partially ordered by region inclusion.
- For each process, it is possible to restrict the attention to the set of maximal critical regions, that is, the regions which are not included within any other.
- It can be shown that most results discussed in the following are still valid even if only maximal critical regions are taken into account, unless otherwise specified.

Nesting of Critical Regions

For simplicity, in the following discussion the fact that critical regions can be nested into each other will be neglected.

- Under the assumption that critical regions are properly nested, the set of critical regions belonging to the same process is partially ordered by region inclusion.
- For each process, it is possible to restrict the attention to the set of maximal critical regions, that is, the regions which are not included within any other.
- It can be shown that most results discussed in the following are still valid even if only maximal critical regions are taken into account, unless otherwise specified.

Nesting of Critical Regions

For simplicity, in the following discussion the fact that critical regions can be nested into each other will be neglected.

- Under the assumption that critical regions are properly nested, the set of critical regions belonging to the same process is partially ordered by region inclusion.
- For each process, it is possible to restrict the attention to the set of maximal critical regions, that is, the regions which are not included within any other.
- It can be shown that most results discussed in the following are still valid even if only maximal critical regions are taken into account, unless otherwise specified.

Conditions for Blocking

Lemma (1)

Under the priority inheritance protocol a process H can be blocked by a lower-priority process L only if L is executing within a critical region Z which satisfies either one of the following two conditions when H is initiated:

1. The critical region is guarded by the same semaphore as a critical region of *H*. In this case, *L* can block *H* directly, when *H* tries to enter its critical region.
2. The critical region can lead *L* to inherit a priority higher than or equal to the priority of *H*. In this case, *L* can block *H* by means of push-through blocking.

Otherwise, *L* can be preempted by *H* and can never block it.

Conditions for Blocking

Lemma (1)

Under the priority inheritance protocol a process H can be blocked by a lower-priority process L only if L is executing within a critical region Z which satisfies either one of the following two conditions when H is initiated:

1. The critical region is guarded by the same semaphore as a critical region of *H*. In this case, *L* can block *H* directly, when *H* tries to enter its critical region.
2. The critical region can lead *L* to inherit a priority higher than or equal to the priority of *H*. In this case, *L* can block *H* by means of push-through blocking.

□

Otherwise, *L* can be preempted by *H* and can never block it.

Conditions for Blocking

Lemma (1)

Under the priority inheritance protocol a process H can be blocked by a lower-priority process L only if L is executing within a critical region Z which satisfies either one of the following two conditions when H is initiated:

1. The critical region is guarded by the same semaphore as a critical region of *H*. In this case, *L* can block *H* directly, when *H* tries to enter its critical region.
2. The critical region can lead *L* to inherit a priority higher than or equal to the priority of *H*. In this case, *L* can block *H* by means of push-through blocking.

□

Otherwise, *L* can be preempted by *H* and can never block it.

Conditions for Blocking

Lemma (1)

Under the priority inheritance protocol a process H can be blocked by a lower-priority process L only if L is executing within a critical region Z which satisfies either one of the following two conditions when H is initiated:

1. The critical region is guarded by the same semaphore as a critical region of *H*. In this case, *L* can block *H* directly, when *H* tries to enter its critical region.
2. The critical region can lead *L* to inherit a priority higher than or equal to the priority of *H*. In this case, *L* can block *H* by means of push-through blocking.

□

Otherwise, *L* can be preempted by *H* and can never block it.

Additional Nomenclature

When the hypotheses of Lemma 1 are satisfied, then process L can block H.

- The same concept can also be expressed by saying that the critical region Z, being executed by L, can block H.
- Since the access to Z is protected by a certain mutual exclusion semaphore S, it can also be said that semaphore S can block H.

Additional Nomenclature

When the hypotheses of Lemma 1 are satisfied, then process L can block H.

- The same concept can also be expressed by saying that the critical region Z, being executed by L, can block H.
- Since the access to Z is protected by a certain mutual exclusion semaphore S, it can also be said that semaphore S can block H.

Additional Nomenclature

When the hypotheses of Lemma 1 are satisfied, then process L can block H.

- The same concept can also be expressed by saying that the critical region Z, being executed by L, can block H.
- Since the access to Z is protected by a certain mutual exclusion semaphore S, it can also be said that semaphore S can block H.

Each Lower-Priority Process Blocks *H* at Most Once

Lemma (2)

Under the priority inheritance protocol a process H can be blocked by a lower-priority process L for at most the duration of one critical region of L which can block H, regardless of the number of semaphores H and L share.

- By Lemma 1, for *L* to block *H*, *L* must be executing a critical region which can block *H* by either direct or push-through blocking.
- When *L* exits that critical region its active priority will certainly go back to a value less than the priority of *H*.
- From this point on, *H* can preempt *L* and cannot be blocked by *L* again.

Each Lower-Priority Process Blocks H at Most Once

Lemma (2)

Under the priority inheritance protocol a process H can be blocked by a lower-priority process L for at most the duration of one critical region of L which can block H, regardless of the number of semaphores H and L share.

- By Lemma 1, for L to block H, L must be executing a critical region which can block H by either direct or push-through blocking.
- When L exits that critical region its active priority will certainly go back to a value less than the priority of H.
- From this point on, H can preempt L and cannot be blocked by L again.

□

Each Lower-Priority Process Blocks *H* at Most Once

Lemma (2)

Under the priority inheritance protocol a process H can be blocked by a lower-priority process L for at most the duration of one critical region of L which can block H, regardless of the number of semaphores H and L share.

- By Lemma 1, for *L* to block *H*, *L* must be executing a critical region which can block *H* by either direct or push-through blocking.
- When *L* exits that critical region its active priority will certainly go back to a value less than the priority of *H*.
- From this point on, *H* can preempt *L* and cannot be blocked by *L* again.

Each Lower-Priority Process Blocks *H* at Most Once

Lemma (2)

Under the priority inheritance protocol a process H can be blocked by a lower-priority process L for at most the duration of one critical region of L which can block H, regardless of the number of semaphores H and L share.

- By Lemma 1, for *L* to block *H*, *L* must be executing a critical region which can block *H* by either direct or push-through blocking.
- When *L* exits that critical region its active priority will certainly go back to a value less than the priority of *H*.
- From this point on, *H* can preempt *L* and cannot be blocked by *L* again.

□

The Maximum Blocking Time is Bounded

Lemma (3)

Under the priority inheritance protocol, a process H for which there are n lower-priority processes $L_1, \ldots, L_n$ can be blocked for at most the duration of one critical region which can block H for each L_i, regardless of the number of semaphores used by H.

- Lemma 2 states that each process L_i can block H for at most the duration of one of its critical regions which can block H.
- In the worst case, the same situation may happen for each of the n lower-priority processes, hence H can be blocked for at most n times. This proves the lemma.

□

If all process only spend a finite amount of time within their critical regions, this lemma shows that the maximum blocking time is bounded.

The Maximum Blocking Time is Bounded

Lemma (3)

Under the priority inheritance protocol, a process H for which there are n lower-priority processes $L_1, \ldots, L_n$ can be blocked for at most the duration of one critical region which can block H for each L_i, regardless of the number of semaphores used by H.

- Lemma 2 states that each process L_i can block H for at most the duration of one of its critical regions which can block H.
- In the worst case, the same situation may happen for each of the n lower-priority processes, hence H can be blocked for at most n times. This proves the lemma.

□

If all process only spend a finite amount of time within their critical regions, this lemma shows that the maximum blocking time is bounded.

The Maximum Blocking Time is Bounded

Lemma (3)

Under the priority inheritance protocol, a process H for which there are n lower-priority processes $L_1, \ldots, L_n$ can be blocked for at most the duration of one critical region which can block H for each L_i, regardless of the number of semaphores used by H.

- Lemma 2 states that each process L_i can block H for at most the duration of one of its critical regions which can block H.
- In the worst case, the same situation may happen for each of the n lower-priority processes, hence H can be blocked for at most n times. This proves the lemma.

□

If all process only spend a finite amount of time within their critical regions, this lemma shows that the maximum blocking time is bounded.

The Maximum Blocking Time is Bounded

Lemma (3)

Under the priority inheritance protocol, a process H for which there are n lower-priority processes $L_1, \ldots, L_n$ can be blocked for at most the duration of one critical region which can block H for each L_i, regardless of the number of semaphores used by H.

- Lemma 2 states that each process L_i can block H for at most the duration of one of its critical regions which can block H.
- In the worst case, the same situation may happen for each of the n lower-priority processes, hence H can be blocked for at most n times. This proves the lemma.

$\square$

If all process only spend a finite amount of time within their critical regions, this lemma shows that the maximum blocking time is bounded.

More Information on Push-Through Blocking

Lemma (4)

A semaphore S can cause push-through blocking to process H only if it is accessed both by a process which has a priority lower than the priority of H, and by a process which has or can inherit a priority higher than the priority of H.

- Suppose that S is accessed by a process L with a priority lower than H.
- If S is not accessed by any process which has or can inherit a priority higher than H, then the priority inheritance mechanism will never give to L an active priority higher than H.
- In this case, H can always preempt L and the lemma follows.

□

More Information on Push-Through Blocking

Lemma (4)

A semaphore S can cause push-through blocking to process H only if it is accessed both by a process which has a priority lower than the priority of H, and by a process which has or can inherit a priority higher than the priority of H.

- Suppose that S is accessed by a process L with a priority lower than H.
- If S is not accessed by any process which has or can inherit a priority higher than H, then the priority inheritance mechanism will never give to L an active priority higher than H.
- In this case, H can always preempt L and the lemma follows.

□

More Information on Push-Through Blocking

Lemma (4)

A semaphore S can cause push-through blocking to process H only if it is accessed both by a process which has a priority lower than the priority of H, and by a process which has or can inherit a priority higher than the priority of H.

- Suppose that S is accessed by a process L with a priority lower than H.
- If S is not accessed by any process which has or can inherit a priority higher than H, then the priority inheritance mechanism will never give to L an active priority higher than H.
- In this case, H can always preempt L and the lemma follows.

□

More Information on Push-Through Blocking

Lemma (4)

A semaphore S can cause push-through blocking to process H only if it is accessed both by a process which has a priority lower than the priority of H, and by a process which has or can inherit a priority higher than the priority of H.

- Suppose that S is accessed by a process L with a priority lower than H.
- If S is not accessed by any process which has or can inherit a priority higher than H, then the priority inheritance mechanism will never give to L an active priority higher than H.
- In this case, H can always preempt L and the lemma follows.

□

The Maximum Blocking Time is Bounded (Again)

Lemma (5)

If process H can suffer blocking from m distinct semaphores $S_1, \ldots, S_m$, then H can be blocked at most for the duration of m critical regions, one for each of the m semaphores.

- Since resources are accessed with mutual exclusion, only one of the lower-priority processes L_i can be within a blocking critical region protected by any given semaphore S_i.
- When L_i leaves the blocking critical region, it can be preempted by *H* and can no longer block it, by Lemma 1.
- Repeating the argument for the m resources proves the lemma.

The Maximum Blocking Time is Bounded (Again)

Lemma (5)

If process H can suffer blocking from m distinct semaphores $S_1, \dots, S_m$, then H can be blocked at most for the duration of m critical regions, one for each of the m semaphores.

- Since resources are accessed with mutual exclusion, only one of the lower-priority processes L_i can be within a blocking critical region protected by any given semaphore S_i.
- When L_i leaves the blocking critical region, it can be preempted by H and can no longer block it, by Lemma 1.
- Repeating the argument for the m resources proves the lemma.

□

The Maximum Blocking Time is Bounded (Again)

Lemma (5)

If process H can suffer blocking from m distinct semaphores $S_1, \ldots, S_m$, then H can be blocked at most for the duration of m critical regions, one for each of the m semaphores.

- Since resources are accessed with mutual exclusion, only one of the lower-priority processes L_i can be within a blocking critical region protected by any given semaphore S_i.
- When L_i leaves the blocking critical region, it can be preempted by H and can no longer block it, by Lemma 1.
- Repeating the argument for the m resources proves the lemma.

□

The Maximum Blocking Time is Bounded (Again)

Lemma (5)

If process H can suffer blocking from m distinct semaphores $S_1, \ldots, S_m$, then H can be blocked at most for the duration of m critical regions, one for each of the m semaphores.

- Since resources are accessed with mutual exclusion, only one of the lower-priority processes L_i can be within a blocking critical region protected by any given semaphore S_i.
- When L_i leaves the blocking critical region, it can be preempted by H and can no longer block it, by Lemma 1.
- Repeating the argument for the m resources proves the lemma.

□

A Coarse Bound

Theorem (1 – Sha, Rajkumar and Lehoczky)

Under the priority inheritance protocol, a process H can be blocked for at most the worst-case execution time, or duration, of $\min(n, m)$ critical regions in the system, where:

- *n is the number of lower-priority processes that can block H, and*
- *m is the number of distinct semaphores that can block H.*

- The proof immediately follows from Lemmas 3 and 5.

□

Remark

It should be noted that a critical region or a semaphore can block H even if the critical region does not belong to H, or H does not use the semaphore.

A Coarse Bound

Theorem (1 – Sha, Rajkumar and Lehoczky)

Under the priority inheritance protocol, a process H can be blocked for at most the worst-case execution time, or duration, of $\min(n, m)$ critical regions in the system, where:

- *n is the number of lower-priority processes that can block H, and*
- *m is the number of distinct semaphores that can block H.*

- The proof immediately follows from Lemmas 3 and 5.

□

Remark

It should be noted that a critical region or a semaphore can block *H* even if the critical region does not belong to *H*, or *H* does not use the semaphore.

A Coarse Bound

Theorem (1 – Sha, Rajkumar and Lehoczky)

Under the priority inheritance protocol, a process H can be blocked for at most the worst-case execution time, or duration, of $\min(n, m)$ critical regions in the system, where:

- *n is the number of lower-priority processes that can block H, and*
- *m is the number of distinct semaphores that can block H.*

- The proof immediately follows from Lemmas 3 and 5.

□

Remark

It should be noted that a critical region or a semaphore can block H even if the critical region does not belong to H, or H does not use the semaphore.

Transitive Priority Inheritance

Definition

A transitive priority inheritance occurs when a high-priority process *H* is blocked by an intermediate-priority process *M*, which in turn is blocked by a low-priority process *L*.

- In this case, the priority of *H* must be transitively transmitted not only to *M*, but also to *L*.
- Otherwise, the presence of any other intermediate-priority process could still give rise to an unbounded priority inversion, by preempting *L*.
- A critical region of a task can block a higher-priority task via transitive priority inheritance, too.

Transitive Priority Inheritance

Definition

A transitive priority inheritance occurs when a high-priority process H is blocked by an intermediate-priority process M, which in turn is blocked by a low-priority process L.

- In this case, the priority of H must be transitively transmitted not only to M, but also to L.
- Otherwise, the presence of any other intermediate-priority process could still give rise to an unbounded priority inversion, by preempting L.
- A critical region of a task can block a higher-priority task via transitive priority inheritance, too.

Transitive Priority Inheritance

Definition

A transitive priority inheritance occurs when a high-priority process *H* is blocked by an intermediate-priority process *M*, which in turn is blocked by a low-priority process *L*.

- In this case, the priority of *H* must be transitively transmitted not only to *M*, but also to *L*.
- Otherwise, the presence of any other intermediate-priority process could still give rise to an unbounded priority inversion, by preempting *L*.
- A critical region of a task can block a higher-priority task via transitive priority inheritance, too.

Transitive Priority Inheritance

Definition

A transitive priority inheritance occurs when a high-priority process *H* is blocked by an intermediate-priority process *M*, which in turn is blocked by a low-priority process *L*.

- In this case, the priority of *H* must be transitively transmitted not only to *M*, but also to *L*.
- Otherwise, the presence of any other intermediate-priority process could still give rise to an unbounded priority inversion, by preempting *L*.
- A critical region of a task can block a higher-priority task via transitive priority inheritance, too.

When Can Transitive Priority Inheritance Occur?

Lemma (6)

Transitive priority inheritance can occur only in presence of nested critical regions.

- Since H is blocked by M, then M must hold a semaphore, say S_M.
- But, by hypothesis, M is also blocked by L on a different semaphore held by L, say S_L.
- As a consequence, M performed a blocking `P`$(\cdot)$ on S_L inside the critical region protected by S_M.
- This corresponds to the definition of nested critical regions.

When Can Transitive Priority Inheritance Occur?

Lemma (6)

Transitive priority inheritance can occur only in presence of nested critical regions.

- Since H is blocked by M, then M must hold a semaphore, say S_M.
- But, by hypothesis, M is also blocked by L on a different semaphore held by L, say S_L.
- As a consequence, M performed a blocking `P(·)` on S_L inside the critical region protected by S_M.
- This corresponds to the definition of nested critical regions.

□

When Can Transitive Priority Inheritance Occur?

Lemma (6)

Transitive priority inheritance can occur only in presence of nested critical regions.

- Since H is blocked by M, then M must hold a semaphore, say S_M.
- But, by hypothesis, M is also blocked by L on a different semaphore held by L, say S_L.
- As a consequence, M performed a blocking $\mathtt{P}(\cdot)$ on S_L inside the critical region protected by S_M.
- This corresponds to the definition of nested critical regions.

□

When Can Transitive Priority Inheritance Occur?

Lemma (6)

Transitive priority inheritance can occur only in presence of nested critical regions.

- Since H is blocked by M, then M must hold a semaphore, say S_M.
- But, by hypothesis, M is also blocked by L on a different semaphore held by L, say S_L.
- As a consequence, M performed a blocking $\mathtt{P}(\cdot)$ on S_L inside the critical region protected by S_M.
- This corresponds to the definition of nested critical regions.

□

When Can Transitive Priority Inheritance Occur?

Lemma (6)

Transitive priority inheritance can occur only in presence of nested critical regions.

- Since H is blocked by M, then M must hold a semaphore, say S_M.
- But, by hypothesis, M is also blocked by L on a different semaphore held by L, say S_L.
- As a consequence, M performed a blocking $\mathtt{P}(\cdot)$ on S_L inside the critical region protected by S_M.
- This corresponds to the definition of nested critical regions.

□

An Additional Hypothesis

Lemma (7)

In the absence of nested critical regions, a semaphore (or resource) can block a process H only if it is used both by (at least) one process with a priority less than H, and (at least) one process with a priority higher than or equal to H.

- Similar to Lemma 4, but the possibility for higher-priority processes to have acquired that priority by inheritance is ruled out.
- This reasoning is valid because Lemma 6 rules out transitive inheritance if critical regions are not nested.
- Transitive inheritance is the only way for a process to acquire a priority higher than the highest-priority process with which it shares a resource.

An Additional Hypothesis

Lemma (7)

In the absence of nested critical regions, a semaphore (or resource) can block a process H only if it is used both by (at least) one process with a priority less than H, and (at least) one process with a priority higher than or equal to H.

- Similar to Lemma 4, but the possibility for higher-priority processes to have acquired that priority by inheritance is ruled out.
- This reasoning is valid because Lemma 6 rules out transitive inheritance if critical regions are not nested.
- Transitive inheritance is the only way for a process to acquire a priority higher than the highest-priority process with which it shares a resource.

An Additional Hypothesis

Lemma (7)

In the absence of nested critical regions, a semaphore (or resource) can block a process H only if it is used both by (at least) one process with a priority less than H, and (at least) one process with a priority higher than or equal to H.

- Similar to Lemma 4, but the possibility for higher-priority processes to have acquired that priority by inheritance is ruled out.
- This reasoning is valid because Lemma 6 rules out transitive inheritance if critical regions are not nested.
- Transitive inheritance is the only way for a process to acquire a priority higher than the highest-priority process with which it shares a resource.

An Additional Hypothesis

Lemma (7)

In the absence of nested critical regions, a semaphore (or resource) can block a process H only if it is used both by (at least) one process with a priority less than H, and (at least) one process with a priority higher than or equal to H.

- Similar to Lemma 4, but the possibility for higher-priority processes to have acquired that priority by inheritance is ruled out.
- This reasoning is valid because Lemma 6 rules out transitive inheritance if critical regions are not nested.
- Transitive inheritance is the only way for a process to acquire a priority higher than the highest-priority process with which it shares a resource.

A Tighter Bound

Theorem (2)

Let K be the total number of semaphores (resources) in the system. If critical regions cannot be nested, the worst-case blocking time experienced by each activation of task τ_i under the priority inheritance protocol is bounded by B_i:

$$B_i = \sum_{k=1}^{K} \text{usage}(k, i) C(k)$$

- $\text{usage}(k, i)$ is a function which returns 1 if resource k is used by (at least) one process with a priority less than τ_i and (at least) one process with a priority higher than or equal to τ_i, and 0 otherwise.
- $C(k)$ is the worst-case execution time among all critical regions corresponding to resource k.

A Tighter Bound

Theorem (2)

Let K be the total number of semaphores (resources) in the system. If critical regions cannot be nested, the worst-case blocking time experienced by each activation of task τ_i under the priority inheritance protocol is bounded by B_i:

$$B_i = \sum_{k=1}^{K} \text{usage}(k, i) C(k)$$

- $\text{usage}(k, i)$ is a function which returns 1 if resource k is used by (at least) one process with a priority less than τ_i and (at least) one process with a priority higher than or equal to τ_i, and 0 otherwise.
- $C(k)$ is the worst-case execution time among all critical regions corresponding to resource k.

A Tighter Bound

Theorem (2)

Let K be the total number of semaphores (resources) in the system. If critical regions cannot be nested, the worst-case blocking time experienced by each activation of task τ_i under the priority inheritance protocol is bounded by B_i:

$$B_i = \sum_{k=1}^{K} \text{usage}(k, i) C(k)$$

- $\text{usage}(k, i)$ is a function which returns 1 if resource k is used by (at least) one process with a priority less than τ_i and (at least) one process with a priority higher than or equal to τ_i, and 0 otherwise.
- $C(k)$ is the worst-case execution time among all critical regions corresponding to resource k.

Proof and Comments

The proof of this theorem descends from the straightforward application of Lemma 7.

- This algorithm is not optimal for the priority inheritance protocol, but it is an acceptable compromise between the tightness of the bound it calculates and its computational complexity.
- Better algorithms exist, and are able to provide a tighter bound of the worst-case blocking time, but they have an exponential complexity.

Proof and Comments

The proof of this theorem descends from the straightforward application of Lemma 7.

□

- This algorithm is not optimal for the priority inheritance protocol, but it is an acceptable compromise between the tightness of the bound it calculates and its computational complexity.
- Better algorithms exist, and are able to provide a tighter bound of the worst-case blocking time, but they have an exponential complexity.

Proof and Comments

The proof of this theorem descends from the straightforward application of Lemma 7.

□

- This algorithm is not optimal for the priority inheritance protocol, but it is an acceptable compromise between the tightness of the bound it calculates and its computational complexity.
- Better algorithms exist, and are able to provide a tighter bound of the worst-case blocking time, but they have an exponential complexity.

Shortcomings of the Priority Inheritance Protocol

While the priority inheritance protocol just described gives an upper bound on the number and the duration of blocks a high-priority process can encounter, it has several shortcomings:

- In the worst case, if *H* accesses *n* semaphores which have been locked by *n* lower-priority processes, *H* will be blocked for the duration of *n* critical regions (chained blocking).
- The priority inheritance protocol does not prevent deadlock from occurring. Deadlock must be avoided by some other means, for example by imposing a total order on the semaphore accesses.

Shortcomings of the Priority Inheritance Protocol

While the priority inheritance protocol just described gives an upper bound on the number and the duration of blocks a high-priority process can encounter, it has several shortcomings:

- In the worst case, if H accesses n semaphores which have been locked by n lower-priority processes, H will be blocked for the duration of n critical regions (chained blocking).
- The priority inheritance protocol does not prevent deadlock from occurring. Deadlock must be avoided by some other means, for example by imposing a total order on the semaphore accesses.

Shortcomings of the Priority Inheritance Protocol

While the priority inheritance protocol just described gives an upper bound on the number and the duration of blocks a high-priority process can encounter, it has several shortcomings:

- In the worst case, if H accesses n semaphores which have been locked by n lower-priority processes, H will be blocked for the duration of n critical regions (chained blocking).
- The priority inheritance protocol does not prevent deadlock from occurring. Deadlock must be avoided by some other means, for example by imposing a total order on the semaphore accesses.

Introduction

The priority ceiling protocols

All of these issues are addressed by the priority ceiling protocols, also proposed by Sha, Rajkumar and Lehoczky (1990); in particular, we will discuss the original priority ceiling protocol and its immediate variant. Both have the following properties:

1. A high-priority process can be blocked at most once during its execution by lower-priority processes.
2. They prevent transitive blocking.
3. They prevent deadlock.
4. Mutual exclusive access to resources is ensured by the protocols themselves.

Introduction

The priority ceiling protocols

All of these issues are addressed by the priority ceiling protocols, also proposed by Sha, Rajkumar and Lehoczky (1990); in particular, we will discuss the original priority ceiling protocol and its immediate variant. Both have the following properties:

1. A high-priority process can be blocked at most once during its execution by lower-priority processes.
2. They prevent transitive blocking.
3. They prevent deadlock.
4. Mutual exclusive access to resources is ensured by the protocols themselves.

Introduction

The priority ceiling protocols

All of these issues are addressed by the priority ceiling protocols, also proposed by Sha, Rajkumar and Lehoczky (1990); in particular, we will discuss the original priority ceiling protocol and its immediate variant. Both have the following properties:

1. A high-priority process can be blocked at most once during its execution by lower-priority processes.
2. They prevent transitive blocking.
3. They prevent deadlock.
4. Mutual exclusive access to resources is ensured by the protocols themselves.

Introduction

The priority ceiling protocols

All of these issues are addressed by the priority ceiling protocols, also proposed by Sha, Rajkumar and Lehoczky (1990); in particular, we will discuss the original priority ceiling protocol and its immediate variant. Both have the following properties:

1. A high-priority process can be blocked at most once during its execution by lower-priority processes.
2. They prevent transitive blocking.
3. They prevent deadlock.
4. Mutual exclusive access to resources is ensured by the protocols themselves.

Introduction

The priority ceiling protocols

All of these issues are addressed by the priority ceiling protocols, also proposed by Sha, Rajkumar and Lehoczky (1990); in particular, we will discuss the original priority ceiling protocol and its immediate variant. Both have the following properties:

1. A high-priority process can be blocked at most once during its execution by lower-priority processes.
2. They prevent transitive blocking.
3. They prevent deadlock.
4. Mutual exclusive access to resources is ensured by the protocols themselves.

Basic Idea

- The basic idea of this method is to extend the priority inheritance protocol with an additional rule for granting lock requests to a free semaphore.
- The protocol ensures that if process L holds a resource, and it could lead to the blocking of an higher-priority process H, then no other resource that could also block H is allowed to be acquired by any process other than L.
- Hence, a process can be delayed not only by attempting to lock a busy semaphore, but also when granting a lock to a free semaphore could lead to multiple blocking on higher-priority processes.
- This form of blocking, that the priority ceiling protocol introduces in addition to direct and push-through blocking, is called ceiling blocking.

Basic Idea

- The basic idea of this method is to extend the priority inheritance protocol with an additional rule for granting lock requests to a free semaphore.
- The protocol ensures that if process *L* holds a resource, and it could lead to the blocking of an higher-priority process *H*, then no other resource that could also block *H* is allowed to be acquired by any process other than *L*.
- Hence, a process can be delayed not only by attempting to lock a busy semaphore, but also when granting a lock to a free semaphore could lead to multiple blocking on higher-priority processes.
- This form of blocking, that the priority ceiling protocol introduces in addition to direct and push-through blocking, is called ceiling blocking.

Basic Idea

- The basic idea of this method is to extend the priority inheritance protocol with an additional rule for granting lock requests to a free semaphore.
- The protocol ensures that if process L holds a resource, and it could lead to the blocking of an higher-priority process H, then no other resource that could also block H is allowed to be acquired by any process other than L.
- Hence, a process can be delayed not only by attempting to lock a busy semaphore, but also when granting a lock to a free semaphore could lead to multiple blocking on higher-priority processes.
- This form of blocking, that the priority ceiling protocol introduces in addition to direct and push-through blocking, is called ceiling blocking.

Basic Idea

- The basic idea of this method is to extend the priority inheritance protocol with an additional rule for granting lock requests to a free semaphore.
- The protocol ensures that if process L holds a resource, and it could lead to the blocking of an higher-priority process H, then no other resource that could also block H is allowed to be acquired by any process other than L.
- Hence, a process can be delayed not only by attempting to lock a busy semaphore, but also when granting a lock to a free semaphore could lead to multiple blocking on higher-priority processes.
- This form of blocking, that the priority ceiling protocol introduces in addition to direct and push-through blocking, is called ceiling blocking.

Original Priority Ceiling Protocol

- The underlying hypotheses are the same as those of the priority inheritance protocol.
- Each resource (semaphore) has a static ceiling value defined, the maximum priority of all processes that use it.
- As in the priority inheritance protocol, each process has a dynamic active priority that is the maximum of its initial priority and any it inherits due to it blocking higher-priority processes.
- A process can only lock a resource (semaphore) if its active priority is higher than the ceiling of any currently locked resource, excluding any resource that the process has already locked itself.
- It should be noted that the last rule can block the access to busy as well as free resources.

Original Priority Ceiling Protocol

- The underlying hypotheses are the same as those of the priority inheritance protocol.
- Each resource (semaphore) has a static ceiling value defined, the maximum priority of all processes that use it.
- As in the priority inheritance protocol, each process has a dynamic active priority that is the maximum of its initial priority and any it inherits due to it blocking higher-priority processes.
- A process can only lock a resource (semaphore) if its active priority is higher than the ceiling of any currently locked resource, excluding any resource that the process has already locked itself.
- It should be noted that the last rule can block the access to busy as well as free resources.

Original Priority Ceiling Protocol

- The underlying hypotheses are the same as those of the priority inheritance protocol.
- Each resource (semaphore) has a static ceiling value defined, the maximum priority of all processes that use it.
- As in the priority inheritance protocol, each process has a dynamic active priority that is the maximum of its initial priority and any it inherits due to it blocking higher-priority processes.
- A process can only lock a resource (semaphore) if its active priority is higher than the ceiling of any currently locked resource, excluding any resource that the process has already locked itself.
- It should be noted that the last rule can block the access to busy as well as free resources.

Original Priority Ceiling Protocol

- The underlying hypotheses are the same as those of the priority inheritance protocol.
- Each resource (semaphore) has a static ceiling value defined, the maximum priority of all processes that use it.
- As in the priority inheritance protocol, each process has a dynamic active priority that is the maximum of its initial priority and any it inherits due to it blocking higher-priority processes.
- A process can only lock a resource (semaphore) if its active priority is higher than the ceiling of any currently locked resource, excluding any resource that the process has already locked itself.
- It should be noted that the last rule can block the access to busy as well as free resources.

Original Priority Ceiling Protocol

- The underlying hypotheses are the same as those of the priority inheritance protocol.
- Each resource (semaphore) has a static ceiling value defined, the maximum priority of all processes that use it.
- As in the priority inheritance protocol, each process has a dynamic active priority that is the maximum of its initial priority and any it inherits due to it blocking higher-priority processes.
- A process can only lock a resource (semaphore) if its active priority is higher than the ceiling of any currently locked resource, excluding any resource that the process has already locked itself.
- It should be noted that the last rule can block the access to busy as well as free resources.

Properties of the Priority Ceiling Protocol

Lemma (8)

If a process L is preempted within a critical region Z_L by another process M that enters a critical region Z_M then, under the priority ceiling protocol, L cannot inherit a priority higher than or equal to the priority of M until M completes.

- If L inherits a priority higher than or equal to M, then it must block a process H (with a priority higher than or equal to M). Hence, it must be $P_H \geq P_M$.
- On the other hand, since M was allowed to enter Z_M, its priority must be strictly higher than the maximum ceiling C^* of the semaphores currently locked. Hence, it must be $P_M > C^*$.
- But, since $P_H > C^*$, we can conclude that H cannot be blocked by L. This is a contradiction and the lemma follows.

□

Properties of the Priority Ceiling Protocol

Lemma (8)

If a process L is preempted within a critical region Z_L by another process M that enters a critical region Z_M then, under the priority ceiling protocol, L cannot inherit a priority higher than or equal to the priority of M until M completes.

- If L inherits a priority higher than or equal to M, then it must block a process H (with a priority higher than or equal to M). Hence, it must be $P_H \geq P_M$.
- On the other hand, since M was allowed to enter Z_M, its priority must be strictly higher than the maximum ceiling C^* of the semaphores currently locked. Hence, it must be $P_M > C^*$.
- But, since $P_H > C^*$, we can conclude that H cannot be blocked by L. This is a contradiction and the lemma follows.

□

Properties of the Priority Ceiling Protocol

Lemma (8)

If a process L is preempted within a critical region Z_L by another process M that enters a critical region Z_M then, under the priority ceiling protocol, L cannot inherit a priority higher than or equal to the priority of M until M completes.

- If L inherits a priority higher than or equal to M, then it must block a process H (with a priority higher than or equal to M). Hence, it must be $P_H \geq P_M$.
- On the other hand, since M was allowed to enter Z_M, its priority must be strictly higher than the maximum ceiling C^* of the semaphores currently locked. Hence, it must be $P_M > C^*$.
- But, since $P_H > C^*$, we can conclude that H cannot be blocked by L. This is a contradiction and the lemma follows.

□

Properties of the Priority Ceiling Protocol

Lemma (8)

If a process L is preempted within a critical region Z_L by another process M that enters a critical region Z_M then, under the priority ceiling protocol, L cannot inherit a priority higher than or equal to the priority of M until M completes.

- If *L* inherits a priority higher than or equal to *M*, then it must block a process *H* (with a priority higher than or equal to *M*). Hence, it must be $P_H \geq P_M$.
- On the other hand, since *M* was allowed to enter Z_M, its priority must be strictly higher than the maximum ceiling C^* of the semaphores currently locked. Hence, it must be $P_M > C^*$.
- But, since $P_H > C^*$, we can conclude that *H* cannot be blocked by *L*. This is a contradiction and the lemma follows.

□

No Transitive Blocking

Lemma (9)

The priority ceiling protocol prevents transitive blocking.

- Suppose that a transitive blocking occurs, that is, there exist three processes H, M and L, with decreasing priorities, such that L blocks M and M blocks H.
- Then, by definition, L must inherit the priority of H by transitivity.
- But this contradicts Lemma 8, which states that L cannot inherit a priority higher than or equal to M.

□

No Transitive Blocking

Lemma (9)

The priority ceiling protocol prevents transitive blocking.

- Suppose that a transitive blocking occurs, that is, there exist three processes H, M and L, with decreasing priorities, such that L blocks M and M blocks H.
- Then, by definition, L must inherit the priority of H by transitivity.
- But this contradicts Lemma 8, which states that L cannot inherit a priority higher than or equal to M.

□

No Transitive Blocking

Lemma (9)

The priority ceiling protocol prevents transitive blocking.

- Suppose that a transitive blocking occurs, that is, there exist three processes H, M and L, with decreasing priorities, such that L blocks M and M blocks H.
- Then, by definition, L must inherit the priority of H by transitivity.
- But this contradicts Lemma 8, which states that L cannot inherit a priority higher than or equal to M.

□

No Transitive Blocking

Lemma (9)

The priority ceiling protocol prevents transitive blocking.

- Suppose that a transitive blocking occurs, that is, there exist three processes H, M and L, with decreasing priorities, such that L blocks M and M blocks H.
- Then, by definition, L must inherit the priority of H by transitivity.
- But this contradicts Lemma 8, which states that L cannot inherit a priority higher than or equal to M.

□

No Deadlock (I)

Theorem (3)

The priority ceiling protocol prevents deadlock.

- Assuming that a process cannot deadlock "by itself", a deadlock can only be formed by a cycle of n processes $\{\tau_1, \ldots, \tau_n\}$ waiting for each other (circular wait condition).
- Each of these processes must be within one of its critical regions, otherwise (hold & wait condition) deadlock cannot occur.
- By Lemma 9 it must be $n = 2$, otherwise transitive blocking would occur, hence we consider only $\{\tau_1, \tau_2\}$.
- Without loss of generality, suppose that τ_2 was preempted by τ_1 while it was within a critical region. Then, τ_1 enters its own critical region.

No Deadlock (I)

Theorem (3)

The priority ceiling protocol prevents deadlock.

- Assuming that a process cannot deadlock "by itself", a deadlock can only be formed by a cycle of n processes $\{\tau_1, \ldots, \tau_n\}$ waiting for each other (circular wait condition).
- Each of these processes must be within one of its critical regions, otherwise (hold & wait condition) deadlock cannot occur.
- By Lemma 9 it must be $n = 2$, otherwise transitive blocking would occur, hence we consider only $\{\tau_1, \tau_2\}$.
- Without loss of generality, suppose that τ_2 was preempted by τ_1 while it was within a critical region. Then, τ_1 enters its own critical region.

No Deadlock (I)

Theorem (3)

The priority ceiling protocol prevents deadlock.

- Assuming that a process cannot deadlock "by itself", a deadlock can only be formed by a cycle of n processes $\{\tau_1, \ldots, \tau_n\}$ waiting for each other (circular wait condition).
- Each of these processes must be within one of its critical regions, otherwise (hold & wait condition) deadlock cannot occur.
- By Lemma 9 it must be $n = 2$, otherwise transitive blocking would occur, hence we consider only $\{\tau_1, \tau_2\}$.
- Without loss of generality, suppose that τ_2 was preempted by τ_1 while it was within a critical region. Then, τ_1 enters its own critical region.

No Deadlock (I)

Theorem (3)

The priority ceiling protocol prevents deadlock.

- Assuming that a process cannot deadlock "by itself", a deadlock can only be formed by a cycle of n processes $\{\tau_1, \ldots, \tau_n\}$ waiting for each other (circular wait condition).
- Each of these processes must be within one of its critical regions, otherwise (hold & wait condition) deadlock cannot occur.
- By Lemma 9 it must be $n = 2$, otherwise transitive blocking would occur, hence we consider only $\{\tau_1, \tau_2\}$.
- Without loss of generality, suppose that τ_2 was preempted by τ_1 while it was within a critical region. Then, τ_1 enters its own critical region.

No Deadlock (I)

Theorem (3)

The priority ceiling protocol prevents deadlock.

- Assuming that a process cannot deadlock “by itself”, a deadlock can only be formed by a cycle of n processes $\{\tau_1, \ldots, \tau_n\}$ waiting for each other (circular wait condition).
- Each of these processes must be within one of its critical regions, otherwise (hold & wait condition) deadlock cannot occur.
- By Lemma 9 it must be $n = 2$, otherwise transitive blocking would occur, hence we consider only $\{\tau_1, \tau_2\}$.
- Without loss of generality, suppose that τ_2 was preempted by τ_1 while it was within a critical region. Then, τ_1 enters its own critical region.

No Deadlock (II)

Theorem (3, continued)

The priority ceiling protocol prevents deadlock.

- By Lemma 8, τ_2 cannot inherit a priority higher than or equal to the priority of τ_1.
- On the other hand, if τ_1 is blocked by τ_2, then τ_2 will inherit the priority of τ_1.
- This is a contradiction and hence the theorem follows.

$\square$

This is useful. . .

Under the priority ceiling protocol, programmers can write arbitrary sequences of (properly nested) semaphore accesses. As long as each job does not deadlock with itself, there will be no deadlock in the system.

No Deadlock (II)

Theorem (3, continued)

The priority ceiling protocol prevents deadlock.

- By Lemma 8, τ_2 cannot inherit a priority higher than or equal to the priority of τ_1.
- On the other hand, if τ_1 is blocked by τ_2, then τ_2 will inherit the priority of τ_1.
- This is a contradiction and hence the theorem follows.

□

This is useful. . .

Under the priority ceiling protocol, programmers can write arbitrary sequences of (properly nested) semaphore accesses. As long as each job does not deadlock with itself, there will be no deadlock in the system.

No Deadlock (II)

Theorem (3, continued)

The priority ceiling protocol prevents deadlock.

- By Lemma 8, τ_2 cannot inherit a priority higher than or equal to the priority of τ_1.
- On the other hand, if τ_1 is blocked by τ_2, then τ_2 will inherit the priority of τ_1.
- This is a contradiction and hence the theorem follows.

□

This is useful. . .

Under the priority ceiling protocol, programmers can write arbitrary sequences of (properly nested) semaphore accesses. As long as each job does not deadlock with itself, there will be no deadlock in the system.

No Deadlock (II)

Theorem (3, continued)

The priority ceiling protocol prevents deadlock.

- By Lemma 8, τ_2 cannot inherit a priority higher than or equal to the priority of τ_1.
- On the other hand, if τ_1 is blocked by τ_2, then τ_2 will inherit the priority of τ_1.
- This is a contradiction and hence the theorem follows.

□

This is useful...

Under the priority ceiling protocol, programmers can write arbitrary sequences of (properly nested) semaphore accesses. As long as each job does not deadlock with itself, there will be no deadlock in the system.

No Deadlock (II)

Theorem (3, continued)

The priority ceiling protocol prevents deadlock.

- By Lemma 8, τ_2 cannot inherit a priority higher than or equal to the priority of τ_1.
- On the other hand, if τ_1 is blocked by τ_2, then τ_2 will inherit the priority of τ_1.
- This is a contradiction and hence the theorem follows.

□

This is useful. . .

Under the priority ceiling protocol, programmers can write arbitrary sequences of (properly nested) semaphore accesses. As long as each job does not deadlock with itself, there will be no deadlock in the system.

Processes are Blocked at Most Once

Theorem (4 – Sha, Rajkumar and Lehoczky)

Under the priority ceiling protocol, a process H can be blocked for at most the duration of one critical region.

- Suppose that H is blocked by two lower-priority processes L and M, where $P_L \leq P_H$ and $P_M \leq P_H$.
- Let L enter its critical region first, and let C_L^* be the highest priority ceiling among all semaphores locked by L at this point.
- In this situation, if M enters its critical region it must be $P_M > C_L^*$, otherwise M would be blocked.
- Moreover, since we assumed that H can be blocked by L, it must be $P_H \leq C_L^*$.
- But this means that $P_M > P_H$, thus contradicting the hypothesis.

Processes are Blocked at Most Once

Theorem (4 – Sha, Rajkumar and Lehoczky)

Under the priority ceiling protocol, a process H can be blocked for at most the duration of one critical region.

- Suppose that H is blocked by two lower-priority processes L and M, where $P_L \leq P_H$ and $P_M \leq P_H$.
- Let L enter its critical region first, and let C_L^* be the highest priority ceiling among all semaphores locked by L at this point.
- In this situation, if M enters its critical region it must be $P_M > C_L^*$, otherwise M would be blocked.
- Moreover, since we assumed that H can be blocked by L, it must be $P_H \leq C_L^*$.
- But this means that $P_M > P_H$, thus contradicting the hypothesis.

□

Processes are Blocked at Most Once

Theorem (4 – Sha, Rajkumar and Lehoczky)

Under the priority ceiling protocol, a process H can be blocked for at most the duration of one critical region.

- Suppose that H is blocked by two lower-priority processes L and M, where $P_L \leq P_H$ and $P_M \leq P_H$.
- Let L enter its critical region first, and let C_L^* be the highest priority ceiling among all semaphores locked by L at this point.
- In this situation, if M enters its critical region it must be $P_M > C_L^*$, otherwise M would be blocked.
- Moreover, since we assumed that H can be blocked by L, it must be $P_H \leq C_L^*$.
- But this means that $P_M > P_H$, thus contradicting the hypothesis.

□

Processes are Blocked at Most Once

Theorem (4 – Sha, Rajkumar and Lehoczky)

Under the priority ceiling protocol, a process H can be blocked for at most the duration of one critical region.

- Suppose that H is blocked by two lower-priority processes L and M, where $P_L \leq P_H$ and $P_M \leq P_H$.
- Let L enter its critical region first, and let C_L^* be the highest priority ceiling among all semaphores locked by L at this point.
- In this situation, if M enters its critical region it must be $P_M > C_L^*$, otherwise M would be blocked.
- Moreover, since we assumed that H can be blocked by L, it must be $P_H \leq C_L^*$.
- But this means that $P_M > P_H$, thus contradicting the hypothesis.

Processes are Blocked at Most Once

Theorem (4 – Sha, Rajkumar and Lehoczky)

Under the priority ceiling protocol, a process H can be blocked for at most the duration of one critical region.

- Suppose that H is blocked by two lower-priority processes L and M, where $P_L \leq P_H$ and $P_M \leq P_H$.
- Let L enter its critical region first, and let C_L^* be the highest priority ceiling among all semaphores locked by L at this point.
- In this situation, if M enters its critical region it must be $P_M > C_L^*$, otherwise M would be blocked.
- Moreover, since we assumed that H can be blocked by L, it must be $P_H \leq C_L^*$.
- But this means that $P_M > P_H$, thus contradicting the hypothesis.

□

Processes are Blocked at Most Once

Theorem (4 – Sha, Rajkumar and Lehoczky)

Under the priority ceiling protocol, a process H can be blocked for at most the duration of one critical region.

- Suppose that H is blocked by two lower-priority processes L and M, where $P_L \leq P_H$ and $P_M \leq P_H$.
- Let L enter its critical region first, and let C_L^* be the highest priority ceiling among all semaphores locked by L at this point.
- In this situation, if M enters its critical region it must be $P_M > C_L^*$, otherwise M would be blocked.
- Moreover, since we assumed that H can be blocked by L, it must be $P_H \leq C_L^*$.
- But this means that $P_M > P_H$, thus contradicting the hypothesis.

□

Identifying the Regions of Interest

Lemma (11)

Under the priority ceiling protocol, a critical region Z, belonging to process L and guarded by semaphore S, can block another process H only if $P_L < P_H$, and the priority ceiling of S, C^*_S, is greater than or equal to P_H.

- If $P_L \geq P_H$, then H cannot preempt L and hence cannot be blocked by S.
- Now, assume that the $P_L < P_H$, $C^*_S < P_H$ and suppose that H is blocked by S.
- If H is blocked, then its priority must be less than or equal to the maximum ceiling C^* among all semaphores locked by jobs other than itself: $P_H \leq C^*$.
- But it is $P_H > C^*_S$ by hypothesis; hence, $C^*_S < C^*$ and another semaphore must be the source of the blocking.

Identifying the Regions of Interest

Lemma (11)

Under the priority ceiling protocol, a critical region Z, belonging to process L and guarded by semaphore S, can block another process H only if $P_L < P_H$, and the priority ceiling of S, C^*_S, is greater than or equal to P_H.

- If $P_L \geq P_H$, then H cannot preempt L and hence cannot be blocked by S.
- Now, assume that the $P_L < P_H$, $C^*_S < P_H$ and suppose that H is blocked by S.
- If H is blocked, then its priority must be less than or equal to the maximum ceiling C^* among all semaphores locked by jobs other than itself: $P_H \leq C^*$.
- But it is $P_H > C^*_S$ by hypothesis; hence, $C^*_S < C^*$ and another semaphore must be the source of the blocking.

Identifying the Regions of Interest

Lemma (11)

Under the priority ceiling protocol, a critical region Z, belonging to process L and guarded by semaphore S, can block another process H only if $P_L < P_H$, and the priority ceiling of S, C_S^*, is greater than or equal to P_H.

- If $P_L \geq P_H$, then H cannot preempt L and hence cannot be blocked by S.
- Now, assume that the $P_L < P_H$, $C_S^* < P_H$ and suppose that H is blocked by S.
- If H is blocked, then its priority must be less than or equal to the maximum ceiling C^* among all semaphores locked by jobs other than itself: $P_H \leq C^*$.
- But it is $P_H > C_S^*$ by hypothesis; hence, $C_S^* < C^*$ and another semaphore must be the source of the blocking.

Identifying the Regions of Interest

Lemma (11)

Under the priority ceiling protocol, a critical region Z, belonging to process L and guarded by semaphore S, can block another process H only if $P_L < P_H$, and the priority ceiling of S, C_S^*, is greater than or equal to P_H.

- If $P_L \geq P_H$, then H cannot preempt L and hence cannot be blocked by S.
- Now, assume that the $P_L < P_H$, $C_S^* < P_H$ and suppose that H is blocked by S.
- If H is blocked, then its priority must be less than or equal to the maximum ceiling C^* among all semaphores locked by jobs other than itself: $P_H \leq C^*$.
- But it is $P_H > C_S^*$ by hypothesis; hence, $C_S^* < C^*$ and another semaphore must be the source of the blocking.

Identifying the Regions of Interest

Lemma (11)

Under the priority ceiling protocol, a critical region Z, belonging to process L and guarded by semaphore S, can block another process H only if $P_L < P_H$, and the priority ceiling of S, C_S^*, is greater than or equal to P_H.

- If $P_L \geq P_H$, then H cannot preempt L and hence cannot be blocked by S.
- Now, assume that the $P_L < P_H$, $C_S^* < P_H$ and suppose that H is blocked by S.
- If H is blocked, then its priority must be less than or equal to the maximum ceiling C^* among all semaphores locked by jobs other than itself: $P_H \leq C^*$.
- But it is $P_H > C_S^*$ by hypothesis; hence, $C_S^* < C^*$ and another semaphore must be the source of the blocking.

Worst-Case Blocking Time

Theorem (5)

Let K be the total number of semaphores (resources) in the system. The worst-case blocking time experienced by each activation of task τ_i under the priority ceiling protocol is bounded by B_i:

$$B_i = \max_{k=1}^{K} \{\text{usage}(k, i) C(k)\}$$

- $\text{usage}(k, i)$ is a function which returns 1 if resource k is used by (at least) one process with a priority less than τ_i and (at least) one process with a priority higher than or equal to τ_i, and 0 otherwise.
- $C(k)$ is the worst-case execution time among all critical regions corresponding to resource k.

Worst-Case Blocking Time

Theorem (5)

Let K be the total number of semaphores (resources) in the system. The worst-case blocking time experienced by each activation of task τ_i under the priority ceiling protocol is bounded by B_i:

$$B_i = \max_{k=1}^{K} \{\text{usage}(k, i) C(k)\}$$

- usage(k, i) is a function which returns 1 if resource k is used by (at least) one process with a priority less than τ_i and (at least) one process with a priority higher than or equal to τ_i, and 0 otherwise.
- $C(k)$ is the worst-case execution time among all critical regions corresponding to resource k.

Worst-Case Blocking Time

Theorem (5)

Let K be the total number of semaphores (resources) in the system. The worst-case blocking time experienced by each activation of task τ_i under the priority ceiling protocol is bounded by B_i:

$$B_i = \max_{k=1}^{K} \{\text{usage}(k, i) C(k)\}$$

- usage(k, i) is a function which returns 1 if resource k is used by (at least) one process with a priority less than τ_i and (at least) one process with a priority higher than or equal to τ_i, and 0 otherwise.
- $C(k)$ is the worst-case execution time among all critical regions corresponding to resource k.

Proof and Comments

The proof of this theorem descends from the straightforward application of:

- Theorem 4, which limits the blocking time to the duration of one critical region, the longest critical region among those that can block τ_i.
- Lemma 11, that identifies which critical regions must be considered for the analysis.

$\square$

Remark

The benefit of the priority ceiling protocol is that a high-priority process can only be blocked once per activation by any lower-priority process. The price to be paid is that more processes will experience this block.

Proof and Comments

The proof of this theorem descends from the straightforward application of:

- Theorem 4, which limits the blocking time to the duration of one critical region, the longest critical region among those that can block τ_i.
- Lemma 11, that identifies which critical regions must be considered for the analysis.

□

Remark

The benefit of the priority ceiling protocol is that a high-priority process can only be blocked once per activation by any lower-priority process. The price to be paid is that more processes will experience this block.

Proof and Comments

The proof of this theorem descends from the straightforward application of:

- Theorem 4, which limits the blocking time to the duration of one critical region, the longest critical region among those that can block τ_i.
- Lemma 11, that identifies which critical regions must be considered for the analysis.

□

Remark

The benefit of the priority ceiling protocol is that a high-priority process can only be blocked once per activation by any lower-priority process. The price to be paid is that more processes will experience this block.

Proof and Comments

The proof of this theorem descends from the straightforward application of:

- Theorem 4, which limits the blocking time to the duration of one critical region, the longest critical region among those that can block τ_i.
- Lemma 11, that identifies which critical regions must be considered for the analysis.

$\square$

Remark

The benefit of the priority ceiling protocol is that a high-priority process can only be blocked once per activation by any lower-priority process. The price to be paid is that more processes will experience this block.

Immediate Priority Ceiling Protocol

The immediate priority ceiling protocol takes a more straightforward approach and raises the priority of a process to the priority ceiling associated with a resource as soon as the process acquires it, rather than only when the process is blocking a higher-priority process. Hence, it is defined as follows:

1. Each process has a static, initial priority assigned.
2. Each resource (semaphore) has a static ceiling defined, this is the maximum priority of all processes that use it.
3. At each instant a process has a dynamic, active priority that is the maximum of its static, initial priority and the ceiling values of any resource (semaphore) it has acquired (locked).

The last rule implies that a process will only suffer a block at the very beginning of its execution.

Immediate Priority Ceiling Protocol

The immediate priority ceiling protocol takes a more straightforward approach and raises the priority of a process to the priority ceiling associated with a resource as soon as the process acquires it, rather than only when the process is blocking a higher-priority process. Hence, it is defined as follows:

1. Each process has a static, initial priority assigned.
2. Each resource (semaphore) has a static ceiling defined, this is the maximum priority of all processes that use it.
3. At each instant a process has a dynamic, active priority that is the maximum of its static, initial priority and the ceiling values of any resource (semaphore) it has acquired (locked).

The last rule implies that a process will only suffer a block at the very beginning of its execution.

Immediate Priority Ceiling Protocol

The immediate priority ceiling protocol takes a more straightforward approach and raises the priority of a process to the priority ceiling associated with a resource as soon as the process acquires it, rather than only when the process is blocking a higher-priority process. Hence, it is defined as follows:

1. Each process has a static, initial priority assigned.
2. Each resource (semaphore) has a static ceiling defined, this is the maximum priority of all processes that use it.
3. At each instant a process has a dynamic, active priority that is the maximum of its static, initial priority and the ceiling values of any resource (semaphore) it has acquired (locked).

The last rule implies that a process will only suffer a block at the very beginning of its execution.

Immediate Priority Ceiling Protocol

The immediate priority ceiling protocol takes a more straightforward approach and raises the priority of a process to the priority ceiling associated with a resource as soon as the process acquires it, rather than only when the process is blocking a higher-priority process. Hence, it is defined as follows:

1. Each process has a static, initial priority assigned.
2. Each resource (semaphore) has a static ceiling defined, this is the maximum priority of all processes that use it.
3. At each instant a process has a dynamic, active priority that is the maximum of its static, initial priority and the ceiling values of any resource (semaphore) it has acquired (locked).

The last rule implies that a process will only suffer a block at the very beginning of its execution.

Immediate Priority Ceiling Protocol

The immediate priority ceiling protocol takes a more straightforward approach and raises the priority of a process to the priority ceiling associated with a resource as soon as the process acquires it, rather than only when the process is blocking a higher-priority process. Hence, it is defined as follows:

1. Each process has a static, initial priority assigned.
2. Each resource (semaphore) has a static ceiling defined, this is the maximum priority of all processes that use it.
3. At each instant a process has a dynamic, active priority that is the maximum of its static, initial priority and the ceiling values of any resource (semaphore) it has acquired (locked).

The last rule implies that a process will only suffer a block at the very beginning of its execution.

Immediate Vs. Original Priority Ceiling Protocol

The worst-case behavior of the two protocols is identical, but. . .

+ The immediate priority ceiling is easier to implement, as blocking relationships must not be monitored.
+ It leads to less context switches as blocking is prior to the first execution.
– It requires more priority movements as this happens with all resource usages rather than only if an actual block has occurred.

Immediate Vs. Original Priority Ceiling Protocol

The worst-case behavior of the two protocols is identical, but...

- \+ The immediate priority ceiling is easier to implement, as blocking relationships must not be monitored.
- \+ It leads to less context switches as blocking is prior to the first execution.
- – It requires more priority movements as this happens with all resource usages rather than only if an actual block has occurred.

Immediate Vs. Original Priority Ceiling Protocol

The worst-case behavior of the two protocols is identical, but. . .

+ The immediate priority ceiling is easier to implement, as blocking relationships must not be monitored.
+ It leads to less context switches as blocking is prior to the first execution.
– It requires more priority movements as this happens with all resource usages rather than only if an actual block has occurred.

Immediate Vs. Original Priority Ceiling Protocol

The worst-case behavior of the two protocols is identical, but…

+ The immediate priority ceiling is easier to implement, as blocking relationships must not be monitored.
+ It leads to less context switches as blocking is prior to the first execution.
− It requires more priority movements as this happens with all resource usages rather than only if an actual block has occurred.

Extended Definition of Response Time

Extended definition

Given that a value for the worst-case blocking time B_i that a task τ_i can suffer has been obtained, its worst-case response time R_i can be redefined to take B_i into account as:

$$R_i = C_i + B_i + I_i$$

- The corresponding recurrence relationship becomes:

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

- This formulation is now pessimistic (no longer necessary and sufficient) because whether a process actually suffers its worst-case blocking time depends upon the relative phases of the processes.

Extended Definition of Response Time

Extended definition

Given that a value for the worst-case blocking time B_i that a task τ_i can suffer has been obtained, its worst-case response time R_i can be redefined to take B_i into account as:

$$R_i = C_i + B_i + I_i$$

- The corresponding recurrence relationship becomes:

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

- This formulation is now pessimistic (no longer necessary and sufficient) because whether a process actually suffers its worst-case blocking time depends upon the relative phases of the processes.

Extended Definition of Response Time

Extended definition

Given that a value for the worst-case blocking time B_i that a task τ_i can suffer has been obtained, its worst-case response time R_i can be redefined to take B_i into account as:

$$R_i = C_i + B_i + I_i$$

- The corresponding recurrence relationship becomes:

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

- This formulation is now pessimistic (no longer necessary and sufficient) because whether a process actually suffers its worst-case blocking time depends upon the relative phases of the processes.

Pictorial Representation

TBD

POSIX and Priority Inversion

The IEEE Std 1003.1 (the POSIX standard) allows either the priority inheritance or the immediate priority ceiling protocols to be specified for each mutex by setting its `protocol` attribute as follows:

`PTHREAD_PRIO_NONE`: acquiring a mutex does not influence thread scheduling (default).

`PTHREAD_PRIO_INHERIT`: priority inheritance is requested on the mutex.

`PTHREAD_PRIO_PROTECT`: immediate priority ceiling (also known as priority ceiling emulation) is requested on the mutex.

POSIX and Priority Inversion

The IEEE Std 1003.1 (the POSIX standard) allows either the priority inheritance or the immediate priority ceiling protocols to be specified for each mutex by setting its `protocol` attribute as follows:

`PTHREAD_PRIO_NONE`: acquiring a mutex does not influence thread scheduling (default).

`PTHREAD_PRIO_INHERIT`: priority inheritance is requested on the mutex.

`PTHREAD_PRIO_PROTECT`: immediate priority ceiling (also known as priority ceiling emulation) is requested on the mutex.

POSIX and Priority Inversion

The IEEE Std 1003.1 (the POSIX standard) allows either the priority inheritance or the immediate priority ceiling protocols to be specified for each mutex by setting its `protocol` attribute as follows:

`PTHREAD_PRIO_NONE`: acquiring a mutex does not influence thread scheduling (default).

`PTHREAD_PRIO_INHERIT`: priority inheritance is requested on the mutex.

`PTHREAD_PRIO_PROTECT`: immediate priority ceiling (also known as priority ceiling emulation) is requested on the mutex.

POSIX and Priority Inversion

The IEEE Std 1003.1 (the POSIX standard) allows either the priority inheritance or the immediate priority ceiling protocols to be specified for each mutex by setting its `protocol` attribute as follows:

`PTHREAD_PRIO_NONE`: acquiring a mutex does not influence thread scheduling (default).

`PTHREAD_PRIO_INHERIT`: priority inheritance is requested on the mutex.

`PTHREAD_PRIO_PROTECT`: immediate priority ceiling (also known as priority ceiling emulation) is requested on the mutex.

Setting the Ceiling

- In POSIX the set of threads that compete to lock a mutex is not known in advance. Hence, to use the priority ceiling emulation protocol the `ceiling` attribute of the mutex must be explicitly set.
- Other operating systems, for example OSEK/VDK, have more information on the threads and can compute the right value of ceiling autonomously, at system generation time.
- When a mutex is released the choice of which thread (among those waiting for it) must be chosen to enter the critical region is under the control of the scheduling policies.

Setting the Ceiling

- In POSIX the set of threads that compete to lock a mutex is not known in advance. Hence, to use the priority ceiling emulation protocol the `ceiling` attribute of the mutex must be explicitly set.
- Other operating systems, for example OSEK/VDK, have more information on the threads and can compute the right value of ceiling autonomously, at system generation time.
- When a mutex is released the choice of which thread (among those waiting for it) must be chosen to enter the critical region is under the control of the scheduling policies.

Setting the Ceiling

- In POSIX the set of threads that compete to lock a mutex is not known in advance. Hence, to use the priority ceiling emulation protocol the `ceiling` attribute of the mutex must be explicitly set.
- Other operating systems, for example OSEK/VDK, have more information on the threads and can compute the right value of ceiling autonomously, at system generation time.
- When a mutex is released the choice of which thread (among those waiting for it) must be chosen to enter the critical region is under the control of the scheduling policies.

Is It a Good Idea?

- The general-purpose algorithms to cope with priority inversion have a non-negligible overhead.
- For a specific case, their effect on the average and worst-case response time of the processes involved can be less than optimal.

Hence...

- Some Authors suggest to avoid attacking the priority inversion problem at the operating system level, and work at the application level instead, by means of:
  - Accurate design.
  - Adoption of lock and wait-free data structures.
  - Temporary and explicit modification of process priorities.

Is It a Good Idea?

- The general-purpose algorithms to cope with priority inversion have a non-negligible overhead.
- For a specific case, their effect on the average and worst-case response time of the processes involved can be less than optimal.

Hence…

- Some Authors suggest to avoid attacking the priority inversion problem at the operating system level, and work at the application level instead, by means of:
  - Accurate design.
  - Adoption of lock and wait-free data structures.
  - Temporary and explicit modification of process priorities.

Is It a Good Idea?

- The general-purpose algorithms to cope with priority inversion have a non-negligible overhead.
- For a specific case, their effect on the average and worst-case response time of the processes involved can be less than optimal.

Hence…

- Some Authors suggest to avoid attacking the priority inversion problem at the operating system level, and work at the application level instead, by means of:
    - Accurate design.
    - Adoption of lock and wait-free data structures.
    - Temporary and explicit modification of process priorities.