Resource Access Control in Real-Time Systems

- Resources, Resource Access, and How Things Can Go Wrong: The Mars Pathfinder Incident
- Resources, Critical Sections, Blocking
- Priority Inversion, Deadlocks
- Nonpreemptive Critical Sections
- Priority Inheritance Protocol
- Priority Ceiling Protocol
- Stack-Based Protocols

Mars Pathfinder Incident

- Landing on July 4, 1997
- “experiences software glitches”
- Pathfinder experiences repeated RESETs after starting gathering of meteorological data.
- RESETs generated by watchdog process.
- Timing overruns caused by priority inversion.
- Resources:
  research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html
Priority Inversion on Mars Pathfinder

Task bc_dist
- blocks on mutex
- becomes active
- gets preempted
- low priority

other tasks

Task ASI/MET
- starts
- locks mutex
- high priority

Task bc_sched detects overrun

Resource Access: System Model

- Processor(s)
  - $m$ types of serially reusable resources $R_1, ..., R_m$
  - An execution of a job $J_i$ requires:
    - a processor for $e_i$ units of time
    - some resources for exclusive use

- Resources
  - **Serially Reusable**: Allocated to one job at a time. Once allocated, held by the job until no longer needed.
  - Examples: semaphores, locks, servers, ...
  - Operations:
    - \texttt{lock(Ri)} \texttt{-----<critical section>------ unlock(Ri)}
  - Resources allocated non-preemptively
  - Critical sections properly nested
Preemption of Tasks in their Critical Sections

- Negative effect on schedulability and predictability.
- Traditional resource management algorithms fail (e.g. Banker’s Algorithm). They decouple resource management decisions from scheduling decisions.

Example:

\[ T_1 = (c_1=2, e_1 = 5, p_1 = 8) \quad T_2 = (4, 7, 22) \quad T_3 = (4, 6, 26) \]

Unpredictability: Scheduling Anomalies

- Example: \[ T_1 = (c_1=2, e_1 = 5, p_1 = 8) \quad T_2 = (4, 7, 22) \quad T_3 = (2.5, 6, 26) \]
Disallow Processor Preemption of Tasks in Critical Section

[A. Mok]

- Analysis identical to analysis with non-preemptable portions
- Define: $\beta$ = maximum duration of all critical sections
- Task $T_i$ is schedulable if
  $$\sum_{i=1}^{n} \frac{e_i}{p_i} + \frac{\beta}{p_i} = U_x(i)$$

- Problem: critical sections can be rather long.

---

Priority Inheritance can Control Priority Inversion

- $T_1 > T_2 > T_3$ without priority inheritance
- $T_3$ blocks $T_2$ here
- $T_3$ directly blocks $T_2$ here
- $T_3$'s priority = $\pi_1$

- $T_1$, $T_2$, $T_3$: scheduling algorithm

- $\pi_1 > \pi_2 > \pi_3$ without priority inheritance
Terminology

- A job is directly blocked when it requests a resource $R_i$, i.e. executes a `lock(R_i)`, but no resource of type $R_i$ is available.
- The scheduler grants the lock request, i.e. allocates the requested resource to the job, according to the resource allocation rules, as soon as the resources become available.
- $J'$ directly blocks $J$ if $J'$ holds some resources that $J$ has requested.
- Priority Inheritance:
  - Basic strategy for controlling priority inversion:
    Let $\pi$ be the priority of $J$
    and $\pi'$ be the priority of $J'$
    and $\pi' < \pi$
    then the priority of $J'$ is set to $\pi$ whenever $J'$ is blocked by $J$.
- New forms of blocking may be introduced by the resource management policy to control priority inversion and/or prevent deadlocks.

Basic Priority-Inheritance Protocol

- Jobs that are not blocked are scheduled according to a priority-driven algorithm preemptively on a processor.
- Priorities of tasks are fixed, except for the conditions described below:
  - A job $J$ requests a resource $R$ by executing `lock(R)`
  - If $R$ is available, it is allocated to $J$. $J$ then continues to execute and releases $R$ by executing `unlock(R)`
  - If $R$ is allocated to $J'$, $J'$ directly blocks $J$. The request for $R$ is denied.
  - However: Let $\pi = \text{priority of } J \text{ when executing } \text{lock}(R)$
    $\pi' = \text{priority of } J' \text{ at the same time}$
    - For as long as $J'$ holds $R$, its priority is $\max(\pi, \pi')$ and returns to $\pi'$ when it releases $R$.
    - That is: $J'$ inherits the priority of $J$ when $J'$ directly blocks $J$ and $J$ has a higher priority.
- Priority Inheritance is transitive.
Example: Priority Inheritance Protocol

\[ \pi_1 > \pi_2 > \pi_3 > \pi_4 > \pi_5 \]

- Task uses A
- Task uses A and B
- Task uses B

**Problem:** If \( T_3 \) tries to \( lock(B) \) while it has priority \( \pi_1 \), we have a deadlock!

Example: Priority Inheritance Protocol (2)

\[ \pi_1 > \pi_2 > \pi_3 > \pi_4 > \pi_5 \]

- Task uses A
- Task uses A and B
- Task uses B

**Problem:** If \( T_3 \) tries to \( lock(B) \) while it has priority \( \pi_1 \), we have a deadlock!
Properties of Priority Inheritance Protocol

- It does not prevent deadlock.
- Task can be blocked directly by a task with a lower priority at most once, for the duration of the (outmost) critical section.
- Consider a task whose priority is higher than $n$ other tasks:

  ![Diagram of Priority Inheritance Protocol]

- Each of the lower-priority tasks can directly block the task at most once.
- A task outside the critical section cannot directly block a higher-priority task.

Priority Ceiling Protocol

- **Assumptions:**
  - Priorities of tasks are fixed
  - Resources required by tasks are known
- **Definition (Priority Ceiling of $R$)**
  
  Priority Ceiling $\Pi_R$ of $R$ = highest priority of all tasks that will request $R$.
- Any task holding $R$ may have priority $\Pi_R$ at some point; either its own priority is $\Pi_R$, or it inherits $\Pi_R$.
- **Motivation:**
  - Suppose there are resource $A$ and $B$.
  - Both $A$ and $B$ are available. $T_1$ requests $A$.
  - $T_2$ requests $B$ after $A$ is allocated.
  
  - If $\pi_2 > \Pi_R$: $T_1$ can never preempt $T_2 \Rightarrow B$ should be allocated to $T_2$.
  - If $\pi_2 \leq \Pi_R$: $T_1$ can preempt $T_2$ (and also request $B$) at some later time. $B$ should not be allocated to $T_2$ to avoid deadlock.
Priority Ceiling Protocol (II)

- Same as the basic Priority Inheritance Protocol, except for the following:

- When a task $T$ requests for allocation of a resource $R$ by executing $\text{lock}(R)$:
  - The request is denied if
    1. $R$ is already allocated to $T'$. ($T'$ directly blocks $T$.)
    2. The priority of $T$ is not higher than all priority ceilings for resources allocated to tasks other than $T$ at the time. (These tasks block $T$.)
  - Otherwise, $R$ is allocated to $T$.

- When a task blocks other tasks, it inherits the highest of their priorities.

Priority Ceiling Protocol: Example

[Lehoczky et al., 1990]

$$\pi_1 > \pi_2 > \pi_3 \quad (\Pi_X = \pi_1, \Pi_Y = \Pi_Z = \pi_2)$$

$$L(Y) \quad L(Z) \quad L(X) \quad U(X) \quad L(Y) \quad U(Y)$$

$(*)$ lock(Z) is denied, since $\pi_2 \leq \Pi_Y$.
Priority Ceiling Protocol: Example II

(*) Fails: directly blocked by $T_3$
(**) Fails: $\pi_2 < \Pi_4$
(1) $T_4$ blocks $T_3$ (to prevent deadlock)
(2) $T_3$ blocks $T_3$ (to control priority inversion)

$\pi_1 > \pi_2 > \pi_3 > \pi_4 > \pi_5$
$\Pi_A = \pi_2, \Pi_B = \pi_1$

Schedulability Analysis: Reminders

- **Blocking**: A higher-priority task waits for a lower-priority task.

- A task $T_H$ can be blocked by a lower-priority task $T_L$ in three ways:
  - directly, i.e. $T_H$ requests a resource the priority ceiling of resources held by $T_L$ is equal to or higher than $\pi_H$:
    \[T_H \not\to \text{request for } \not\to \text{allocated to } T_L \text{ } (\pi > \pi_H)\]
  - When $T_H$ requests a resource the priority ceiling of resources held by $T_L$ is equal to or higher than $\pi_H$:
    \[T_H \not\to \text{request for } \not\to \text{allocated to } T_L \text{ } (\pi_H \leq \Pi_X)\]
Schedulability Analysis: Preliminary Observations

Consider: Task $T$ with priority $\pi$ and at release time $t$.

Define: **Current Priority Ceiling** $\Pi(t)$: Highest priority ceiling of all resources allocated at time $t$.

Preliminary Observation 1:

$T$ cannot be blocked if at time $t$, every resource allocated has a priority ceiling less than $\pi$, i.e., $\pi_T > \Pi(t)$.

Obvious:
- No task with priority lower than $\pi$ holds any resource with priority ceiling $\geq \pi$.
- $T$ will not require any of the resources allocated at time $t$ with priority ceilings $< \pi$, and will not be directly blocked waiting for them.
- No lower-priority task can inherit a priority higher than $\pi$ through resources allocated at time $t$.
- Requests for resources by $T$ will not be denied because of resource allocations made before $t$.

Schedulability Analysis: Preliminary Observations II

Preliminary Observation 2

- Suppose that
  - There is a task $T_L$ holding a resource $X$
  - $T$ (with priority $\pi$) preempts $T_L$, and then
  - $T$ is allocated a resource $Y$.
- Until $T$ completes, $T_L$ cannot inherit a priority higher or equal to $\pi$.

Reason: ($\pi_L$ = priority of $T_L$ when it is preempted.)
- $\pi_L < \pi$
- $T$ is allocated a resource
  $\Rightarrow$ $\pi$ is higher than all the priority ceilings of resources held by all lower-priority tasks when $T$ preempts $T_L$.
- $T$ cannot be blocked by $T_L$, from Preliminary Observation 1.
  $\Rightarrow$ $\pi_L$ cannot be raised to $\pi$ or higher through inheritance.
Schedulability Analysis with Resources Access

- Schedulability loss due to blocking:
- Reminder: Critical sections are properly nested
  \[ \Rightarrow \text{Duration of a critical section equals the outmost critical section.} \]

**Observation 1:**
A low-priority task \( T_L \) can block a higher-priority task \( T_H \) at most once.

- Reason: When \( T_L \) is not in critical section
  \[ \pi_L < \pi_H \]
  - \( T_L \) cannot inherit a higher priority

**Observation 2**
A task \( T \) can be blocked for at most the duration of one critical section, no matter how many tasks share resources with \( T \).

- Reason:
  - It is not possible for \( T \) to be blocked for durations of 2 critical sections of one task.
  - It is not possible for \( T \) to be blocked by \( T_1 \) and \( T_2 \) with priorities \( \pi_1 < \pi \)
    \[ \pi_2 < \pi \]

\[ \begin{align*}
T_1 & \rightarrow & \text{L(B)} & \rightarrow & \text{L(A)} \\
T_2 & \rightarrow & \text{L(A)} & \rightarrow & \text{L(B)} \\
T_3 & \rightarrow & \text{L(A)} & \rightarrow & \text{L(B)} \\
\end{align*} \]

Not possible!
\( T_1 \) is allocated \( B \) \( \Rightarrow \) \( \pi_1 \) is higher than the priority ceiling of \( A \), which is \( \pi \).

\[ \begin{align*}
T_1 & \rightarrow & \text{L(A)} & \rightarrow & \text{L(B)} \\
T_2 & \rightarrow & \text{L(B)} & \rightarrow & \text{L(A)} \\
T_3 & \rightarrow & \text{L(A)} & \rightarrow & \text{L(B)} \\
\end{align*} \]

Not possible!
\( \pi_A \geq \pi_B \) \( \Rightarrow \) \( B \) is not allocated to \( T_1 \) at \( t \), \( \pi_1 < \pi \).
Stack Sharing

- Sharing of the stack among tasks eliminates stack space fragmentation and so allows for memory savings:

  \[ T_i \rightarrow \text{stack sharing} \quad T_i \rightarrow \text{no stack sharing} \]

- However:
  - Once job is preemted, it can only resume when it returns to be on top of stack.
  - Otherwise, it may cause a deadlock.
  - Stack becomes a resource that allows for “one-way preemption”.

Stack-Sharing Priority-Ceiling Protocol

- To avoid deadlocks: Once execution begins, make sure that job is not blocked due to resource access.
- Otherwise: Low-priority, preempted, jobs may re-acquire access to CPU, but can not continue due to unavailability of stack space.

- Define: \( \Pi(t) \): highest priority ceiling of all resources currently allocated.
  If no resource allocated, \( \Pi(t) = \infty \).

Protocol:

1. Update Priority Ceiling: Whenever all resources are free, \( \Pi(t) = \infty \). The value of \( \Pi(t) \) is updated whenever resource is allocated or freed.
2. Scheduling Rule: After a job is released, it is blocked from starting execution until its assigned priority is higher then \( \Pi(t) \). At all times, jobs that are not blocked are scheduled on the processor in a priority-driven, preemptive fashion according to their assigned priorities.
3. Allocation Rule: Whenever a job requests a resource, it is allocated the resource.
Stack-Based Priority-Ceiling Protocol (cont)

- The Stack-Based Priority-Ceiling Protocol is **deadlock-free**:
  - When a job begins to execute, all the resources it will ever need are free.
  - Otherwise, $\Pi(t)$ would be higher or equal to the priority of the job.

  - Whenever a job is preempted, all the resources needed by the preemtping job are free.
  - The preempting job can complete, and the preempted job can resume.

- Worst-case blocking time of Stack-Based Protocol is the same as for Basic Priority Ceiling Protocol.

- Stack-Based Protocol smaller context-switch overhead (2 CS) than Priority Ceiling Protocol (4 CS)
  - Once execution starts, job cannot be blocked.

Ceiling-Priority Protocol

- Stack-Based Protocol does not allow for self-suspension
  - Stack is shared resource

- Re-formulation for multiple stacks (no stack-sharing) straightforward:

**Scheduling Rules:**
1. Every job executes at its assigned priority when it does not hold resources.
2. Jobs of the same priority are scheduled on FIFO basis.
3. Priority of jobs holding resources is the highest of the priority ceilings of all resources held by the job.

**Allocation Rule:**
- Whenever a job requests a resource, it is allocated the resource.
Priority-Ceiling Locking in Ada 9X
[Ada 9X; RT Annex]

• Task definitions allow for a pragma Priority as follows:
  \[
  \text{pragma Priority(expression)}
  \]

• Task priorities:
  – base priority: priority defined at task creation, or dynamically set with Dynamic_Priority.Set_Priority() method.
  – active priority: base priority or priority inherited from other sources (activation, rendez-vous, protected objects).

• Priority-Ceiling Locking:
  – Every protected object has a ceiling priority: Upper bound on active priority a task can have when it calls a protected operation on objects.
  – While task executes a protected action, it inherits the ceiling priority of the corresponding protected object.
  – When a task calls a protected operation, a check is made that its active priority is not higher than the ceiling of the corresponding protected object. a Program Error is raised if this check fails.

Priority-Ceiling Locking in Ada 9X: Implementation
[Ada 9X; RT Annex]

• Efficient implementation possible that does not rely on explicit locking.
• Mutual exclusion is enforced by priorities and priority ceiling protocol only.
• We show that Resource \( R \) can never be requested by Task \( T_2 \) while it is held by Task \( T_1 \).
• Simplified argument:
  – \( AP(T_2) \) can never be higher than \( C(R) \). Otherwise, run-time error would occur. \( \Rightarrow AP(T_2) \leq C(R) \)
  – As long as \( T_1 \) holds \( R \), it cannot be blocked.
    • Therefore, for \( T_2 \) to request \( R \) after \( T_1 \) seized it, \( T_1 \) must have been preempted (priority of \( T_1 \) does not change while \( T_1 \) is in ready queue).
    – For \( T_2 \) to request \( R \) while \( T_1 \) is in ready queue, \( T_2 \) must have higher active priority than \( T_1 \). \( \Rightarrow AP(T_2) \leq C(R) \)
  – \( T_1 \) is holding \( R \) \( \Rightarrow C(R) \leq AP(T_1) < AP(T_2) \)
  • Before \( T_2 \) requests \( R \), \( T_2 \)'s priority must drop to \( \leq C(R) \)
  Case 1: \( AP(T_2) \) drops to below \( AP(T_1) \) \( \Rightarrow T_2 \) preempted
  Case 2: \( AP(T_2) \) drops to \( AP(T_1) \) \( \Rightarrow T_2 \) must yield to \( T_1 \) (by rule)