Lecture Note #6: More on Task Scheduling
EECS 571
Principles of Real-Time Embedded Systems

Kang G. Shin
EECS Department
University of Michigan
Mars Pathfinder Timing Hiccups?

- **When**: landed on the Mar on July 4, 1997
- **Mission**: gathering and sending back voluminous data to Earth.
- **Problem**: resetting the entire computer system, each reset resulting in data loss ("software glitches?")

**System Used and Cause Found:**
- Wind River System's VxWorks that supports preemptive priority scheduling.
- "Information bus" (≡ shared memory) accessed via mutexes. The bus management task $T_1$ ran frequently with high priority
- Meteorological data collection task $T_3$ is an infrequent, low-priority task, and uses the info bus to publish its data. (It acquires mutex, writes data to the bus, then releases the mutex.)
- ∃ a medium-priority, long commercial task $T_2$.

- Scenario: $T_1$ blocks on the mutex held by $T_3$ and in the meantime $T_2$ preempts $T_3$; not executing $T_1$ for certain time triggers a watchdog timer resetting the system.

**How was the problem solved**: A short C program was uploaded to the spacecraft which changed parameter initialization.

*Note 6-2*
Synchronization Protocols

- Non-preemption
- Basic priority inheritance
- Highest locker’s priority
- Priority ceiling
- All of these prevent unbounded priority inversion.
Non-preemption Protocol

\( \tau_2: \{ \ldots P(S1) \ldots V(S1) \ldots \} \)

\( \tau_4: \{ \ldots P(S1) \ldots V(S1) \ldots \} \)
Advantages and Disadvantages

- **Advantages:**
  - Simplicity
  - Use with fixed-priority and dynamic-priority systems
  - No prior knowledge about resource requirement by each task
  - Good when all critical sections are short

- **Disadvantages:**
  - Every task can be blocked by every lower priority task, even when there is no resource sharing between the tasks.
  - Blocking time: $max(c_{si})$
Basic Inheritance Protocol (BIP)

\[ \tau_2: \{ \ldots P(S1) \ldots V(S1) \ldots \} \]

\[ \tau_4: \{ \ldots P(S1) \ldots V(S1) \ldots \} \]

\( \tau_1(H) \)

\( \tau_2 \)

blocked

\( \tau_3 \)

S1 locked

S1 unlocked

\( \tau_4(L) \)

inherits the priority of \( \tau_2 \) after \( \tau_2 \) is blocked

Note 6-6
Some Notations

- \( J_i \) is the \( i \)-th job of periodic task \( T \).
- \( \pi_i \) = job \( J_i \)'s assigned priority
- \( \pi_i(t) \) = current (at time \( t \)) priority of \( J_i \)
- If the decision to change the priority of \( J_i \) is made at \( t = t_1 \) then
  - \( \pi_i(t_1^-) \) = priority at and immediately before \( t_1 \),
  - \( \pi_i(t_1^+) \) = priority immediately after the priority change
- \( \Omega \) = nonexistent priority, lower than the lowest priority
Terminology and Rules

- At time $t_1$, job $J_i$ requests resource $R_k$.
- $R_k \rightarrow J_i$: Resource $R_k$ is held by job $J_i$
- $J_i \rightarrow R_k$: Job $J_i$ is blocked waiting for resource $R_k$ to be released ($J_i \rightarrow R_k \rightarrow J_i$)

**Scheduling Rules:**
- Ready jobs are scheduled on processors preemptively according to their current priorities, $\pi_i(t)$.
- Upon release of a job, its priority is equal to its assigned priority
  - if $J_i$ is released at $t = t'$, then $\pi_i(t') = \pi_i$

**Resource allocation:**
- If a resource is free, then it is allocated when it is requested
- if not free, then the request is denied and the requesting job is blocked
Priority Inheritance Rules

- **Scheduling Rule:** same as the assumptions
- **Allocation Rule:** same as the assumptions
- **Priority-Inheritance Rule:**
  - if $J_i \rightarrow R_k \rightarrow J_l$ and $\pi_l(t_1^-) = \text{priority of } J_l \text{ at } t = t_1$
  - then $\pi_l(t_1^+) = \pi_l(t_1)$
  - until $J_l$ releases $R_k$ at $t_2$ when $\pi_l(t_2^+) = \pi_l(t_1^-)$

Properties of Priority Inheritance

- For each resource (semaphore), a list of blocked tasks must be stored in a priority queue.

- A task (job) $\tau_i$ uses its assigned priority, and inherits the highest dynamic priority of all the tasks it blocks when it is in its critical section and blocks some higher priority tasks.

- Priority inheritance is transitive; that is, if task $\tau_i$ blocks $\tau_j$ and $\tau_j$ blocks $\tau_k$, then $\tau_i$ can inherit the priority of $\tau_k$.

- When $\tau_i$ releases a resource, which priority should it use?

- Chained/nested blocking if requesting multiple resources (nested mutex requests)

- Direct blocking and indirect (inheritance) blocking (when the lower priority task inherits the higher priority task’s priority).
Example of Chained/nested Blocking (BIP)

\[ \tau_1: \{ \ldots P(S1) \ldots P(S2) \ldots V(S2) \ldots V(S1) \ldots \} \]
\[ \tau_2: \{ \ldots P(S1) \ldots V(S1) \ldots \} \]
\[ \tau_3: \{ \ldots P(S2) \ldots V(S2) \ldots \} \]

Attempts to lock S1(blocked)
Attempts to lock S2(blocked)

\[ \tau_1(H) \]
S1 locked
S1 unlocked

\[ \tau_2(M) \]
S2 locked
S2 unlocked

\[ \tau_3(L) \]
Deadlock: Using BIP

\[ \tau_1 : \{ \ldots P(S1) \ldots P(S2) \ldots V(S2) \ldots V(S1) \ldots \} \]
\[ \tau_2 : \{ \ldots P(S2) \ldots P(S1) \ldots V(S1) \ldots V(S2) \ldots \} \]

Attempts to lock S2 (blocked) \Rightarrow \text{deadlock!}
### Blocking Time Under BIP

**Example**

\[ T_1 = \{\ldots P(A) \cdot 3. P(B) \cdot 2. V(B) \cdot 1. V(A) \ldots\} \]
\[ T_2 = \{\ldots P(C) \cdot 2. V(C) \ldots\} \]
\[ T_3 = \{\ldots P(A) \cdot 1. P(B) \cdot 2. V(B) \cdot 2. V(A) \ldots\} \]
\[ T_4 = \{\ldots P(A) \cdot 1. P(C) \cdot 1. P(B) \cdot 3. V(B) \cdot 1. V(C) \cdot 1. V(A) \ldots\} \]

<table>
<thead>
<tr>
<th></th>
<th>direct blocking by</th>
<th>indirect blocking by</th>
<th>blocking time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
</tr>
<tr>
<td>T1</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>T2</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Priority Ceiling Protocol (PCP)

\( \tau_2: \{ \ldots P(S1) \ldots V(S1) \ldots \} \)

\( \tau_3: \{ \ldots P(S2) \ldots V(S2) \ldots \} \)

\( \tau_4: \{ \ldots P(S1) \ldots V(S1) \ldots \} \)
Basic Priority Ceiling Rules (1)

- $\Pi(R) = \text{priority ceiling of resource } R$ – the highest priority of the tasks that request $R$
- $\Pi_S(t) = \text{system priority ceiling} – \text{the highest priority ceiling of all resources that are in use at time } t$

- **Scheduling Rule:** same as the assumptions
- **Allocation Rule:**
  - if $J_i \rightarrow R_k \rightarrow J_i$ at $t \leq t_1$ then block $J_i$ (no change)
  - if $R_k$ becomes free at $t_1$,
    - if $\pi_i(t_1) > \Pi_S(t_1)$, then $R_k \rightarrow J_i$
    - else (i.e., $\pi_i(t_1) \leq \Pi_S(t_1)$)
      - if for some $R_x \rightarrow J_i$ and $\Pi(R_x) = \Pi_S(t_1)$, then $R_k \rightarrow J_i$
        - [J_i holds resource $R_x$ whose priority ceiling is $\Pi_S(t_1)$ ]
      - else deny and block ($J_i \rightarrow R_k$)
Basic Priority Ceiling Rules (2)

- **Priority-Inheritance Rule:**
  - if $J_i \rightarrow R_k$ at $t = t_1$ and is blocked by $J_l$ (and $\pi_l(t_1^-) = \text{priority of } J_l$)
    - either $R_k \rightarrow J_l$, ($J_l$ holds the resource $R_k$)
    - or $J_l \rightarrow R_x$ and $\Pi(R_x) = \Pi_S(t_1) \geq \pi_i(t_1)$
  - then $\pi_l(t_1^+) = \pi_l(t_1)$ (**inherited priority**)
  - until $J_l$ releases all $R_x$ with $\Pi(R_x) \geq \pi_l(t_1)$, $\pi_l(t_2^+) = \pi_l(t_1^-)$ at $t = t_2$. 

Note 6-16
A task $T_H$ can be blocked by a lower-priority task $T_L$ in three ways:

- directly, i.e.,

- when $T_L$ inherits a priority higher than the priority $\pi_H$ of $T_H$.

- When $T_H$ requests a resource and the priority ceiling of all resources held by $T_L$ is equal to or higher than $\pi_H$:

Note 6-17
Blocked At Most Once (PCP)

\[ \tau_1:\{\ldots P(S1)\ldots P(S2)\ldots V(S2)\ldots V(S1)\ldots} \]
\[ \tau_2:\{\ldots P(S1)\ldots V(S1)\ldots} \]
\[ \tau_3:\{\ldots P(S2)\ldots V(S2)\ldots} \]

\( \tau_1(H) \)

\( \tau_2(M) \)

\( \tau_3(L) \)
Deadlock Avoidance: Using PCP

\[ \tau_1 : \{ \ldots P(S1) \ldots P(S2) \ldots V(S2) \ldots V(S1) \ldots \} \]

\[ \tau_2 : \{ \ldots P(S2) \ldots P(S1) \ldots V(S1) \ldots V(S2) \ldots \} \]
Stack Sharing

Sharing a stack among multiple tasks eliminates stack space fragmentation, making memory savings:

- $T_1$  
- $T_i$  
- $T_n$

However:
- Once job is preempted, 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”.

Note 6-20
Stack-Based Priority Ceiling Protocol

- **To avoid deadlocks**: Once execution begins, make sure that job is not blocked due to resource access
  - allow preemption only if the priority is higher than the ceiling of all resources in use

- **Update Current Ceiling in the usual manner**
  - If no resource allocated, $\Pi_S(t) = \Omega$

- **Scheduling Rule**:
  - $J_i$ released and blocked until $\pi_i(t) > \Pi_S(t)$
  - When not blocked, jobs are scheduled in the usual manner.

- **Allocation Rule**:
  - Allocate when requested
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, $\Pi_s(t)$ would be higher or equal to the priority of the job.
- Whenever a job is preempted, all the resources needed by the preempting job are free.
- The preempting job can complete, and then 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 context switches since once execution starts, job cannot be blocked (may be preempted)
- 4 context switches for PCP since a job may be blocked at most once
Ceiling-Priority Protocol

- Re-formulation of stack-based priority ceiling protocol for multiple stacks (w/o stack-sharing)
- Update Current Ceiling in the usual manner

**Scheduling Rule:**
- No resources held by $J_i$, $\pi_i(t) = \pi_i$
- Resource held by $J_i$, $\pi_i(t) = \max(\Pi(R_x))$ for all resources $R_x$ held by $J_i$
- FIFO scheduling among jobs with equal priority

**Allocation Rule:**
- Allocate when requested
Comparison

- Worst-case performance of stack-based and basic ceiling protocols are the same

- Stack-based version
  - supported by the Real-Time systems Annex of Ada95
  - Jobs must not self-suspend

- When jobs do not self-suspend, stack-based and ceiling-priority protocols yield the same schedules.

- Stack-based and ceiling-priority have the same worst-case blocking time.
Highest Locker’s Priority Protocol

\[ \tau_2: \{ \ldots P(S1) \ldots V(S1) \ldots \} \]

\[ \tau_4: \{ \ldots P(S1) \ldots V(S1) \ldots \} \]
## Summary of Synchronization Protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Bounded Priority Inversion</th>
<th>Blocked at Most Once</th>
<th>Deadlock Avoidance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonpreemptible critical sections</td>
<td>Yes</td>
<td>Yes$^1$</td>
<td>Yes$^1$</td>
</tr>
<tr>
<td>Highest locker's priority</td>
<td>Yes</td>
<td>Yes$^1$</td>
<td>Yes$^1$</td>
</tr>
<tr>
<td>Basic inheritance</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Priority Ceiling</td>
<td>Yes</td>
<td>Yes$^2$</td>
<td>Yes</td>
</tr>
</tbody>
</table>

$^1$ *Only if tasks do not suspend within critical sections*

$^2$ *PCP is not affected if tasks suspend within critical sections.*