IN4343 Real-Time Systems, Lecture 9

Handling Shared Resources

Contact

m.nasri@tudelft.nl
(appointment by email)
Sources

Giorgio Buttazzo’s book: chapter 7

Some slides have been taken from Giorgio’s website: http://retis.sssup.it/~giorgio/rts-MECS.html
Handling shared resources in real-time system
Critical sections

If the system supports concurrent execution (e.g., preemptive scheduling), then the access to the shared resources must be protected, e.g., by Semaphores.

\[ a = x + 1; \]
\[ b = y + 2; \]
\[ c = x + y; \]

\( \tau_1 \)

\( \tau_2 \)

wait(s)

write

read

global variables (global memory buffer)

int x;
int y;

signal(s)

signal(s)

critical section

critical section
Why do we need to protect shared resources?

Imagine that \((x, y)\) is the position of a robot in the room.

\[
\begin{align*}
x &= 3; \\
y &= 5;
\end{align*}
\]

\[
\begin{align*}
a &= x + 1; \\
b &= y + 2; \\
c &= x + y;
\end{align*}
\]

\[\tau_1 \text{ high priority}\]

\[\tau_2 \text{ low priority}\]

Data inconsistency!

Task \(\tau_2\) has used data from two different locations.
Semaphores

• Each shared resource is protected by a different semaphore.
  • $s = 1 \Rightarrow$ free resource
  • $s = 0 \Rightarrow$ busy (locked) resource

wait($s$):
  if $s == 0$, then
    The task must be blocked on a queue of the semaphore. The queue management policy depends on the OS (usually it is FIFO or priority-based).
  else
    set $s = 0$.

signal($s$):
  if there are blocked tasks, then the first in the queue is awaken ($s$ remains 0),
  else set $s = 1$. 

\[
\tau_1
\]

\[
\begin{align*}
\text{wait}(S) \\
x &= 3; \\
y &= 5; \\
\text{signal}(S)
\end{align*}
\]

\[
\begin{align*}
\text{wait}(B) \\
a &= t + 3; \\
b &= 5; \\
\text{signal}(B)
\end{align*}
\]
Using semaphores to protect shared resources

Semaphores guarantee that at any time, only one of these tasks can enter the critical section.

Preemption

No data inconsistency
Guidelines for real-time systems engineers
Hints: shorten critical sections

Make critical sections as short as possible.

```c
int x, y; // these are global shared variables
mutex s;  // this is the semaphore to protect them
```

```c
task reader() {
    int i;          // these are local variables
    float d, v[DIM];
    ...
    wait(s);
    d = sqrt(x*x + y*y);
    for (i=0; i++<DIM) {
        v[i] = i*(x + y);
        if (v[i] < x*y) v[i] = x + y;
    }
    signal(s);
    ...
}
```
Hints: shorten critical sections

A possibility is to **copy global variables** into local variables:

```plaintext
task reader() {
    int i;
    float d, v[DIM];
    float a, b;
    // these are local variables
    // two new local variables
    ...
    wait(s);
    a = x; b = y;
    // to local vars
    signal(s);
    d = sqrt(a*a + b*b);
    // make computation
    for (i=0; i++; i<DIM) {
        v[i] = i*(a + b);
        if (v[i] < a*b) v[i\] = a + b;
    }
    wait(s);
    // copy local vars
    x = a; y = b;
    // to global vars
    signal(s);
    ...
}
```
Hints: avoid critical sections across loops or conditions

```
... wait(s);
results = x + y;
while (result > 0) {
    v[i] = j*(x + y);
    if (v[i] < x*y)
        results = results - y;
    else
        signal(s);
}
...```

This code is very **UNSAFE** since "signal" could never be executed, and $\tau_1$ could be blocked forever!

Anime: “One piece”
Hints: avoid nested critical sections

Because to reach the inner critical section
the task must acquire 2 locks: $S_A$ and $S_B$.

While the task holds the first lock $S_A$ and
waits for the second one $S_B$, no other task
can access the first lock $S_A$!
Hints: avoid cross-cutting critical sections

- Make critical sections as short as possible.
- Avoid making critical sections across loops or conditional statements.
- Try to avoid nested critical sections.
- If nested critical sections are unavoidable, at least avoid cross-cutting critical sections.
  - Because it makes the analysis very hard
Do these guidelines solve the “blocking” problem?
Impact on schedulability

Schedule with no conflicts

priority

$\tau_1$

$\tau_2$

$\tau_3$

Critical section
Impact on schedulability

Conflict on a critical section

In this case, it is a direct blocking from the low-priority task to the high-priority task
Impact on schedulability

Conflict on a critical section

This situation is called “priority inversion”: A high-priority task is blocked by a lower-priority task.

Solution
Introduce a concurrency control protocol for accessing critical sections.
Key aspects in designing an access protocol

**Access Rule:**
Determines when to block or whether to block or not.

Example: if a task is in a critical section, then block the task that has just arrived

**Progress Rule:**
Determines how to execute inside a critical section.

Example: inside critical section, execute non-preemptively

**Release Rule:**
Determines how to order the pending requests of the blocked tasks.

Example: At exit, enable preemption
Resource access protocols

- Classical semaphores (No protocol)
- Non-Preemptive Protocol (NPP)
- Highest-Locker Priority (HLP)
- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)
- Stack Resource Policy (SRP)  
  (will not be covered in the exam)

There will certainly be some exam questions from these protocols
Assumption

Critical sections are correctly accessed by tasks:
Non-preemptive protocol (NPP)

High-level idea
Whenever a task accesses a resource, it enters a non-preemptive mode until it releases the resource.

- **Access Rule:** A task never blocks at the entrance of a critical section, but at its activation time.
- **Progress Rule:** Disable preemption when executing inside a critical section.
- **Release Rule:** At exit, enable preemption so that the resource is assigned to the pending task with the highest priority.

How many tasks can be in their critical sections at the same time?
Only one task!
NPP: implementation notes

A possible method to implement NPP protocol:

• Each task $\tau_i$ must have two priorities:
  • a nominal priority $P_i$ (fixed) assigned by the application developer;
  • a dynamic priority $p_i$ (initialized to $P_i$) used to schedule the task and affected by the protocol.

• Then, the protocol can be implemented by changing the behavior of the wait and signal primitives:

  \[
  \begin{align*}
  \text{wait(s):} & \quad p_i = \min\{P_1, \ldots, P_n\} \\
  \text{signal(s):} & \quad p_i = P_i
  \end{align*}
  \]
NPP: pro & cons

**ADVANTAGES:** simplicity and efficiency.

- Semaphore queues are not needed, because tasks never block on a `wait(s)`.  
- Each task can block at most on a single critical section.  
- It prevents deadlocks and allows stack sharing.  
- It is transparent to the programmer.

**PROBLEMS:**

1. Tasks may be blocked **even if they do not use any shared resource.**  
2. Long critical sections delay all high-priority tasks  
3. A task could be blocked even if it “may” not access a critical section
NPP: problems

- Tasks may be blocked even if they do not use any shared resource.
- Long critical sections delay all high priority tasks

Can we deal with this situation using a non-work-conserving scheduling algorithm?

Have any idea? :D Let’s collaborate on a paper
NPP: problems

A task could be blocked even if it “may” not access a critical section
High-level idea

When a task accesses a resource (e.g., `wait(S)`), its priority **upgrades** to the priority of the highest-priority task that **may use** the resource `S`.

- **Access Rule**: A task never blocks at the entrance of a critical section, but at its activation time.
- **Progress Rule**: Inside the critical section for resource `R`, the task executes at the **highest priority of the tasks that use** `R`.
- **Release Rule**: At exit, the dynamic priority of the task is reset to its nominal priority `P_i`.

Diagram:
- Priority axis with `τ_1`, `τ_2`, and `τ_3`.
- `wait(S)` and `signal(S)` actions.
- `B_2` block time for `τ_2`.
- `τ_2` is blocked, but `τ_1` can preempt.
Highest-locker priority (HLP) protocol

- **Access Rule:** A task never blocks at the entrance of a critical section, but at its activation time.
- **Progress Rule:** Inside the critical section for resource \( R \), the task executes at the highest priority of the tasks that use \( R \).
- **Release Rule:** At exit, the dynamic priority of the task is reset to its nominal priority \( P_i \).

Priority assigned to \( \tau_i \) when it uses semaphore \( S \):

\[
p_i(S) = \min\{P_j | \forall \tau_j, \tau_j \text{ uses } S\}
\]

What is \( p_3(S) \)?

It is 2 because \( \tau_2 \) is the highest-priority task that uses \( S \).
HLP: implementation notes

• Each task $\tau_i$ is assigned a nominal priority $P_i$ and a dynamic priority $p_i$.
• Each semaphore $S$ is assigned a resource ceiling $C(S)$:

$$C(S) = \min\{P_j | \forall \tau_j, \tau_j \text{ uses } S\}$$

Change the wait and signal primitives as follows:

**Wait (S):**  
$p_i = C(S)$

**Signal (S):**  
$p_i = P_i$

**Note:** HLP is also known as Immediate-Priority Ceiling (IPC).
Consider HLP protocol:

What is $p_1(A)$? 1

What is $p_2(A)$? 1

What is $p_2(B)$? 2

What is $p_3(C)$? 3

What is $p_4(C)$? 3

What is $p_4(B)$? 2

Reminder: $P_1 < P_2 < P_3 < \ldots < P_n$
HLP: pro & cons

ADVANTAGES: simplicity and efficiency.

• Semaphores queues are not needed, because tasks never block on a wait(s).
• Each task can block at most on a single critical section.
• It prevents deadlocks and allows stack sharing.

PROBLEMS:

1. A task could be blocked even if it “may” not access a critical section (similar to NPP).
2. It is not transparent to programmers (due to ceilings).
Priority-inheritance protocol (PIP)

**High-level idea**
Whenever a task accesses a resource $S$ that is locked by another task, the priority of the locking task upgrades to the priority of the highest-priority task that is currently blocked on resource $S$.

- **Access Rule**: A task blocks at the entrance of a critical section if the resource is locked.
- **Progress Rule**: Inside resource $R$, a task executes with the highest priority of the tasks blocked on $R$.
- **Release Rule**: At exit, the dynamic priority of the task is reset to its nominal priority $P_i$. 
PIP: types of blocking

• **Direct blocking**
  • A task blocks on a locked semaphore

• **Indirect blocking (push-through blocking)**
  • A task is blocked because a lower-priority task inherited a higher priority.

**Blocking:**

a delay caused by lower-priority tasks
PIP: implementation notes

- Inside a resource $S$ the dynamic priority $p_i$ is set to
  
  $$p_i(S) = \min\{P_j \mid \tau_j \text{ is blocked on } S\}$$

```c
wait (S) if (S == 0) {
    // suspend the calling task $\tau_c$ in the semaphore queue
    <suspend the calling task $\tau_c$ in the semaphore queue>
    <find the task $\tau_k$ that is locking the semaphore $S$>
    $p_k = \min\{P_c, p_k\}$  // $\tau_k$ inherits the priority of $\tau_c$ if $p_k > P_c$
    <call the scheduler>
}
else S = 0;
```

```c
signal (S) if (there are blocked tasks) {
    // awake the highest-priority task in the semaphore queue
    <awake the highest-priority task in the semaphore queue>
    $p_i = P_i$
    <call the scheduler>
}
else S = 1;
```
Identifying blocking resources

Under PIP, a task $\tau_i$ can be **blocked** on a semaphore $S_k$ only if:

1. $S_k$ is directly shared between $\tau_i$ and lower-priority tasks (direct blocking), or
2. $S_k$ is shared between tasks with priority lower than $\tau_i$ and tasks having priority higher than $\tau_i$ (push-through blocking).
Which tasks can block $\tau_1$? $\tau_2$ (on A2 or C2) and $\tau_3$ (on B3 or D3)

How many times any low-priority task can block a high-priority task?

**ONLY once!** Because right after the end of the critical section of that low-priority task, the high-priority task starts its execution and then no other low-priority task can preempt the high-priority one.

Which tasks can block $\tau_3$? $\tau_3$ cannot be blocked

Which tasks can block $\tau_2$? $\tau_3$ (on B3 or D3)

Notation guide:
$B_2 =$ the access of task 2 to resource B
Given the following resource accesses protected by semaphores A, B, C, and D,

Part 1. Is it possible that under the PIP protocol, \( \tau_3 \) directly blocks \( \tau_2 \) on any resource?

Part 2. Generate an execution scenario in which \( \tau_2 \) is indirectly blocked by \( \tau_3 \) on the access of \( \tau_3 \) to resource B.

\( \tau_2 \) is indirectly blocked by \( \tau_3 \) because as soon as \( \tau_1 \) wants to enter its critical section on resource B, \( \tau_2 \)'s priority is upgraded to 1.
Under PIP, generate an execution scenario in which $\tau_2$ is indirectly blocked by $\tau_3$ on the access of $\tau_3$ to resource $D$. 
Identifying blocking resources

**Lemma 1:** A task $\tau_i$ can be blocked at most once by a lower priority task.

If there are $n_i$ tasks with priority lower than $\tau_i$, then $\tau_i$ can be blocked at most at most $n_i$ times, independently of the number of critical sections that can block $\tau_i$. 
Identifying blocking resources

**Lemma 2:** A task $\tau_i$ can be blocked at most once on a semaphore $S_k$.

If there are $m_i$ distinct semaphores that can block a task $\tau_i$, then $\tau_i$ can be blocked at most $m_i$ times, independently of the number of critical sections that can block $\tau_i$. 
Bounding blocking times

Theorem:

\( \tau_i \) can be blocked at most for \( \alpha_i = \min(n_i, m_i) \) critical sections.

\[ n_i = \text{number of tasks with priority less than } \tau_i \]
\[ m_i = \text{number of semaphores that can block } \tau_i \]
(either directly or indirectly).
PIP: pro & cons

ADVANTAGES:

• It removes the pessimisms of NPP and HLP (a task is blocked only when really needed).
• It is transparent to the programmer.

PROBLEMS:

1. More complex to implement (especially to support nested critical sections).
2. It is prone to chained blocking.
3. It does not avoid deadlocks.
NOTE: $\tau_1$ can be blocked at most once for each lower priority task.
Typical deadlock

• It can only occur with nested critical sections:

\[ P_1 < P_2 \]
PIP deadline example

\[ P_1 < P_2 \]

\[ \tau_1 \]

\[ \tau_2 \]

Wait(A)

Wait(B)

signal(A)

signal(B)

Wait(B)

Wait(A)

\[ \tau_1 \text{ is blocked on A} \]

Wait(B)

Wait(A)

\[ \tau_2 \text{ is blocked on B} \]

Deadlock!
Agenda

- Classical semaphores (No protocol)
- Non-Preemptive Protocol (NPP)
- Highest-Locker Priority (HLP)
- Priority Inheritance Protocol (PIP)
- Priority Ceiling Protocol (PCP)
- Stack Resource Policy (SRP)

(will not be covered in the exam)

There will certainly be some exam questions from these protocols
Priority Ceiling Protocol (PCP)

High-level idea
A task can access a resource only if it passes the PCP access test. If the test is passed, the rest is like PIP protocol.

PCP can be viewed as PIP + access test

• **Access Rule**: A task can access a resource only if it passes the PCP access test.
• **Progress Rule**: Inside resource R, a task executes with the highest priority of the tasks blocked on R.
• **Release Rule**: At exit, the dynamic priority of the task is reset to its nominal priority $P_i$. 
PCP implementation

To keep track of resource usage by high-priority tasks, each resource is assigned a resource ceiling:

\[ C(S_k) = \min \{ P_j \mid \forall \tau_j, \tau_j \text{ uses } S_k \} \]

A task \( \tau_i \) can enter a critical section only if its priority is higher than the maximum ceiling of the locked semaphores:

**PCP access test:**

\[ P_i < \min \{ C(S_k) \mid S_k \text{ locked by tasks } \neq \tau_i \} \]
PCP: deadlock avoidance

$P_1 < P_2$

$C(S_A) = P_1$

$C(S_B) = P_1$

blocked by PCP
Avoiding chained blocking

How can we avoid chained blocking?

To avoid multiple blocking of $\tau_1$, we must prevent $\tau_3$ and $\tau_2$ to enter their critical sections (even if they are free), because a low priority task ($\tau_4$) is holding a resource used by $\tau_1$. 
PCP: example

\[ C(A) = \min\{P_j | \forall \tau_j, \tau_j \text{ uses } A\} \]

\[ C(B) = \min\{P_j | \forall \tau_j, \tau_j \text{ uses } B\} \]

\[ P_3 < \min\{C(S_k) | S_k \in \{A, B\} \text{ and } S_k \text{ locked by tasks } \neq \tau_3\} \]

The test passes because no resource has been locked so far
PCP: example

\( \tau_2 \) does not get the permission to enter its critical section even though it was accessing a resource that was not locked.

\[ P_2 < \min \{ C(S_k) \mid S_k \in \{ A, B \} \text{ and } S_k \text{ locked by tasks } \neq \tau_2 \} \]

\[ C(A) = \min \{ P_j \mid \forall \tau_j, \tau_j \text{ uses } A \} \quad 1 \]

\[ C(B) = \min \{ P_j \mid \forall \tau_j, \tau_j \text{ uses } B \} \quad 1 \]
PCP: example

\( \tau_1 \)

\( \tau_2 \)

\( \tau_3 \)

Ceiling blocking
PCP: another example

At $t_1$: $\tau_2$ is blocked by the PCP, since $P_2 > C(S_A)$
PCP: properties

**Theorem 1**
Under PCP, a task can be blocked at most on a single critical section.

**Theorem 2**
PCP prevents chained blocking.

**Theorem 3**
PCP prevents deadlocks.
PCP: pro & cons

ADVANTAGES:

• It limits blocking to the length of a single critical section.
• It avoids deadlocks when using nested critical sections.

PROBLEMS:

1. More complex to implement (like PIP).
2. It can create unnecessary blocking (it is pessimistic like HLP).
3. It is not transparent to the programmer: resource ceilings must be specified in the source code.
# Summary

<table>
<thead>
<tr>
<th>protocol</th>
<th>Compatible with scheduling algorithm</th>
<th>pessimism</th>
<th>Blocking at</th>
<th>Transparent to user</th>
<th>Deadlock free</th>
<th>implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPP</td>
<td>any</td>
<td>high</td>
<td>Arrival</td>
<td>yes</td>
<td>yes</td>
<td>easy</td>
</tr>
<tr>
<td>HLP</td>
<td>FP</td>
<td>medium</td>
<td>Arrival</td>
<td>no</td>
<td>yes</td>
<td>easy</td>
</tr>
<tr>
<td>PIP</td>
<td>FP</td>
<td>low</td>
<td>Resource access</td>
<td>yes</td>
<td>no</td>
<td>hard</td>
</tr>
<tr>
<td>PCP</td>
<td>FP</td>
<td>medium</td>
<td>Resource access</td>
<td>no</td>
<td>yes</td>
<td>harder</td>
</tr>
</tbody>
</table>
Extra slides about semaphores
Multi-unit resources

- If a resource has $n$ parallel units that can be accessed by $n$ tasks simultaneously, it can be protected by a semaphore initialized to $n$.

- `wait(s):`
  - if $s == 0$, the task is blocked on the semaphore queue;
  - else $s$ is decremented.

- `signal(s):`
  - If there are blocked tasks, the first in the queue is awakened ($s$ remains $0$), else $s$ is incremented.
Implementation notes

\[ s = \text{create\_sem}(n) \]
creates the semaphore structure, including a counter \((s.\text{count})\)
initialized to \(n\), and a queue of tasks \((s.\text{queue})\).

\[
\text{wait}(s) \{ \\
\text{if } (s.\text{count} == 0) \\
\quad <\text{block the calling task on } s.\text{queue}> \\
\text{else } s.\text{count}--; \\
\}
\]

\[
\text{signal}(s) \{ \\
\text{if } (!\text{empty}(s.\text{queue})) \\
\quad <\text{unblock the first task in } s.\text{queue}> \\
\text{else } s.\text{count}++; \\
\}
\]
Problem with semaphores

Semaphores (when properly used) guarantee the consistency of shared global data, but introduce extra **blocking delays in high priority tasks**.