Synchronization
Basic functions of RTOS

- Time management
- Task management
- Interrupt handling
- Memory management
- Exception handling
- Task scheduling
- Task synchronization
Synchronization primitives

- **Semaphore**: counting semaphore and binary semaphore
  - A semaphore is created with initial_count, which is the number of allowed holders of the semaphore lock. (initial_count=1: binary sem)
  - Sem_wait will decrease the count; while sem_signal will increase it.
  - A task can get the semaphore when the count > 0; otherwise, block on it.
- **Mutex**: similar to a binary semaphore, but mutex has an owner.
  - A semaphore can be “waited for” and “signaled” by any task,
  - while only the task that has taken a mutex is allowed to release it.
- **Spinlock**: lock mechanism for multi-processor systems,
  - A task wanting to get spinlock has to get a lock shared by all processors.
- **Barrier**: to synchronize a lot of tasks,
  - they should wait until all of them have reached a certain “barrier.”
Potential problems in synchronization

- **Critical section** (data, service, code) protected by lock mechanism e.g. Semaphore etc. In a RTOS, the maximum time a task can be delayed because of locks held by other tasks should be less than its timing constraints.

- **Deadlock, livelock, starvation** Some deadlock avoidance/prevention algorithms are too complicate and indeterministic for real-time execution. Simplicity preferred, e.g.
  - all tasks always take locks in the same order.

- **Priority inversion** using priority-based task scheduling and locking primitives should know the “priority inversion” danger: a medium-priority job runs while a high-priority task is ready to proceed.
Solutions:
Resource Sharing and Priority Ceiling Protocols
A classic paper on real-time systems

The simplest form of priority inversion

Task 1
- -
P(S)
Using R
V(S)
- -

Task 9
- -
P(S)
Using R
V(S)
- -

Shared Resource R

Task 1

Task 9

computing
using R
blocked
Priority inversion problem

- Assume 3 tasks: A, B, C with priorities Ap<Bp<Cp
- Assume semaphore: S shared by A and C
- The following may happen:
  - A gets S by P(S)
  - C wants S by P(S) and blocked
  - B is released and preempts A
  - Now B can run for a long long period ..... 
  - A is blocked by B, and C is blocked by A
  - So C is blocked by B
- The above senario is called ‘priority inversion’
- It can be much worse if there are more tasks with priorities in between Bp and Cp, that may block C as B does!
Un-bounded priority inversion

Task 1
- 
- P(S) Using R
- V(S)
- 

Task 9
- 
- P(S) Using R 
- V(S)
- 

Shared Resource R

Task 2
- 
- 
- 
- 

Task 1

Task 2

Task 9

P(S) ⋯ V(S)

P(S) ⋯ V(S)

computing
using R
blocked
Solutions

- Tasks are ‘forced’ to follow **pre-defined rules** when requesting and releasing resources (locking and unlocking semaphores)
- The rules are called ‘Resource access protocols’
Semaphore, Dijkstra 60s

- A semaphore is a simple data structure with
  - a counter
    - the number of “copies of a resource”
    - binary semaphore
  - a queue
    - Tasks waiting

and two operations:

- P(S): get or wait for semaphore
- V(S): release semaphore

Shared resources may be protected using semaphores
## Implementation of Semaphores: SCB

- **SCB**: Semaphores Control Block

<table>
<thead>
<tr>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue of TCBs (tasks waiting)</td>
</tr>
<tr>
<td>Pointer to next SCB</td>
</tr>
</tbody>
</table>

The queue should be sorted by priorities (Why not FIFO?)
Implementation of semaphores: P-operation

- P(scb):
  Disable-interrupt;
  If scb.counter>0 then
    scb.counter -= 1;
  end then
  else
    save-context();
    current-tcb.state := blocked;
    insert(current-tcb, scb.queue);
    dispatch();
    load-context();
  end else
  Enable-interrupt
Implementation of Semaphores: V-operation

- V(scb):
  
  Disable-interrupt;
  
  If not-empty(scb.queue) then
    tcb := get-first(scb.queue);
    tcb.state := ready;
    insert(tcb, ready-queue);
    save-context();
    schedule();  /* dispatch invoked*/
    load-context();
  end then
  else scb.counter ++1;
  end else
  Enable-interrupt
Resource Access Protocols

- Highest Priority Inheritance
  - Non preemption protocol (NPP)

- Basic Priority Inheritance Protocol (BIP)
  - POSIX (RT OS standard) mutexes

- Priority Ceiling Protocols (PCP)

- Immediate Priority Inheritance
  - Highest Locker’s priority Protocol (HLP)
    - Ada95 (protected object) and POSIX mutexes
Non Preemption Protocol (NPP)

- Modify $P(S)$ so that the “caller” is assigned the highest priority if it succeeds in locking $S$
  - Highest priority=non preemption!
- Modify $V(S)$ so that the “caller” is assigned its own priority back when it releases $S$

This is the simplest method to avoid Priority Inversion!
NPP: + and −

- Simple and easy to implement (+), how?
- Deadlock free (++)
- Number of blockings = 1 (+)
- Allow low-priority tasks to block high-priority tasks including those that have no sharing resources (-)

Missinig all deadlines!
Basic Priority Inheritance Protocol (BIP)

- supported in RT POSIX

- **Idea:**
  - A gets semaphore S
  - B with higher priority tries to lock S, and blocked by S
  - B transfers its priority to A (so A is resumed and run with B’s priority)

- **Run time behaviour:** whenever a lower-priority task blocks a higher priority task, it inherits the priority of the blocked task
Un-bounded priority inversion

Task 1
- 
-  
P(S)  
Using R  
V(S)  
-  
-  

Task 9
- 
-  
P(S)  
Using R  
V(S)  
-  
-  

Shared Resource R

Task 2
- 
- 
- 
- 

Task 1

P(S)  

V(S)  

Task 2

...  

Task 9

P(S)  

V(S)  

...  

computing

using R

blocked
BIP protocol: Example

Task 1
- -
P(S) Using R
V(S)
- -

Task 9
- -
P(S) Using R
V(S)
- -

Shared Resource R

Run with the priority of Task 1
Implementation of Ceiling Protocols

- Main ideas:
  - Priority-based scheduling
  - Implement P/V operations on Semaphores to assign task priorities dynamically
Semaphore Control Block for BIP

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>counter</td>
<td>queue</td>
<td>Pointer to next SCB</td>
<td>Holder</td>
<td></td>
</tr>
</tbody>
</table>
Standard P-operation (without BIP)

- P(scb):
  
  Disable-interrupt;
  If scb.counter > 0 then {scb.counter -= 1;
  else
  {save-context( );
   current-task.state := blocked;
   insert(current-task, scb.queue);
   dispatch();
   load-context() }
  Enable-interrupt
P-operation with BIP

- **P(scb):**
  
  ```
  Disable-interrupt;
  If scb.counter>0 then {scb.counter -= 1;
   scb.holder:= current-task
   add(current-task.sem-list,scb)}
  ```

  ```
  else
  {save-context( );
   current-task.state := blocked;
   insert(current-task, scb.queue);
   save(scb.holder.priority);
   scb.holder.priority := current-task.priority;
   dispatch();
   load-context() }
  ```

  **Enable-interrupt**
Standard V-operation (without BIP)

- V(scb):
  
  Disable-interrupt;

  If not-empty(scb.queue) then
  
  {  
  next-to-run := get-first(scb.queue);
  next-to-run.state := ready;
  insert(next-to-run, ready-queue);
  save-context();
  schedule(); /* dispatch invoked*/
  load-context() }

  else scb.counter ++1;

  Enable-interrupt
V-operation with BIP

- **V(scb):**
  
  Disable-interrupt;
  
  current-task.priority := ”original/previous priority”
  /* restore the previous priority of the ” caller” */
  
  If not-empty(scb.queue) then
  {
    next-to-run := get-first(scb.queue);
    next-to-run.state := ready;
    scb.holder := next-to-run;
    add(next-to-run.sem-list, scb);
    insert(next-to-run, ready-queue);
    save-context();
    schedule(); /* dispatch invoked*/
    load-context()
  }
  else scb.counter ++1;
  
  Enable-interrupt
Problem 1: potential deadlock

Task 2: ... P(S2) ... P(S1)...
Task 1: ... P(S1) ... P(S2)...

Deadlock!
Problem 2: chained blocking – many preemptions

Task H needs M resources may be blocked M times:
→ many preemptions/run-time overheads
→ maximal blocking= at least, the sum of all CS sections for lower-priority tasks

- Priority inversion
- Using S1
- Using S2

Running with priority H
Properties of BIP: + and -

- Bounded Priority inversion (+)
- Reasonable Run-time performance (+)
- Potential deadlocks (-)
- Chain-blocking – many preemptions (-)
Immediate Priority Inheritance:
=Highest Locker’s Priority Protocol (HLP)

- Adopted in Ada95 (protected object), POSIX mutexes
- **Idea:** define the ceiling \( C(S) \) of a semaphore \( S \) to be the highest priority of all tasks that use \( S \) during execution. Note that \( C(S) \) can be calculated statically (off-line).
Run-time behaviour of HLP

- Whenever a task succeeds in holding a semaphor $S$, its priority is changed dynamically to the maximum of its current priority and $C(S)$.
- When it finishes with $S$, it sets its priority back to what it was before.
## Priority Ceiling of Semaphore: Examples

<table>
<thead>
<tr>
<th>Task</th>
<th>Priority</th>
<th>Share Semaphors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>H</td>
<td>S3</td>
</tr>
<tr>
<td>Task 2</td>
<td>M</td>
<td>S1, S</td>
</tr>
<tr>
<td>Task 3</td>
<td>L</td>
<td>S1, S2</td>
</tr>
<tr>
<td>Task 4</td>
<td>Lower</td>
<td>S2, S</td>
</tr>
</tbody>
</table>

**Ceiling of semaphors**

- \( C(S_1) = \text{M} \)
- \( C(S_2) = \text{L} \)
- \( C(S_3) = \text{H} \)
- \( C(S) = \text{M} \)
Un-bounded priority inversion

Task 1
- 
- 
P(S) Using R
V(S) 
-

Shared Resource R

Task 9
- 
- 
P(S) Using R
V(S) 
-

Task 2
- 
- 
-

Task 1
Task 2
Task 9

- P(S) ••• - V(S)
- ••• -
- P(S) - V(S)

- computing
- using R
- blocked
HLP protocol: Example

Task 1
- 
- 
P(S) Using R
V(S)
- 

Task 9
- 
- 
P(S) Using R
V(S)
- 

Shared Resource R

Task 2
- 
- 
- 

C(S)=H

Task 1 (H)

Task 2 (M)

Task 9 (L)

Run with the priority of C(S): H
HLP Protocol: Example

M and Lower share S

- Computing
- Blocked
- Using resource

New release of Task 2

Task 1

Task 2

Task 3
Property 1: Deadlock free (HLP)

Once task 2 gets S1, it runs with pri H, task 1 will be blocked (no chance to get S2 before task 2)
BIP: Chained blocking – many preemptions

Task H needs M resources may be blocked M times:
→ many preemptions/run-time overheads
→ maximal blocking= at least, the sum of all CS sections for lower-priority tasks
Property 2: at most one blocking (HLP)

Task 1

Task 2

Task 3

Running with priority C(S2): H

C(S1)=H
C(S2)=H

Priority inversion
Using S1
Using S2
HLP: Blocking time calculation

\[ B(i) = \max\{CS(k, i, S) | \text{pri}(k) < \text{pri}(i) \leq C(S)\} \]

where
- \( B(i) \) is the maximal time that task \( i \) may be blocked due to HLP
- \( CS(k, i, S) \) is the length of a critical section that may be locked by task \( k \) and \( i \) with \( S \).
Implementation of HLP

- Calculate the ceiling for all semaphores
- Modify SCB
- Modify P and V-operations
Semaphore Control Block for HLP

- counter
- queue
- Pointer to next SCB
- Ceiling
P-operation with HLP

- P(scb):
  
  Disable-interrupt;
  
  If scb.counter>0 then
  
  { scb.counter - 1;
    save(current-task.priority);
    current-task.priority := Ceiling(scb) }

  else
  
  {save-context();
    current-task.state := blocked
    insert(current-task, scb.queue);
    dispatch();
    load-context() }

Enable-interrupt
V-operation with HLP

- V(scb):
  Disable-interrupt;
  "restore previous priority"
  If not-empty(scb.queue) then
    next-to-run := get-first(scb.queue);
    next-to-run.state := ready;
    next-to-run.priority := Ceiling(scb);  /*this is not necessary! */
    insert(next-to-run, ready-queue);
    save-context();
    schedule(); /* dispatch invoked*/
    load-context();
  end then
  else scb.counter ++1;
end else
Enable-interrupt
Properties of HLP: + and -

- Bounded priority inversion
- Deadlock free (+), Why?
- Number of blocking = 1 (+), Why?
- The extreme case of HLP=Non-preemption (-)
  - E.g when the highest priority task uses all semaphors, the lower priority tasks will inherit the highest priority
# Summary

<table>
<thead>
<tr>
<th></th>
<th>NPP</th>
<th>BIP</th>
<th>HLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded Priority Inversion</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Avoid deadlock</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Avoid Un-necessary blocking</td>
<td>no</td>
<td>yes</td>
<td>yes/no</td>
</tr>
<tr>
<td>Blocking time calculation</td>
<td>Easy</td>
<td>hard</td>
<td>easy</td>
</tr>
</tbody>
</table>
Un-bounded priority inversion

Task 1
- -
- P(S)
Using R
V(S)
- -

Task 2
- -
- -
- -

Task 9
- -
P(S)
Using R
V(S)
- -

Shared Resource R

Task 1
- -
P(S) \cdots V(S)

Task 2
- -
- -

Task 9
- -
P(S) \cdots V(S)

\[\text{computing}\]
\[\text{using R}\]
\[\text{blocked}\]
Priority Ceiling Protocol (combining HLP and BIP)

- Each semaphore $S$ has a Ceiling $C(S)$
- Run-time behaviour:
  - Assume that $S$ is the semaphore with highest ceiling of all semaphores locked by other tasks currently:
    - $C(S)$ is “the current system ceiling”
  - If $A$ wants to lock any semaphore, it must have a strictly higher priority than $C(S)$ i.e. $\text{Pri}(A) > C(S)$. Otherwise $A$ is blocked, and it transmits its priority($+\varepsilon$) to the task currently holding $S$
Example: HLP

A: \ldots P(S_1) \ldots V(S_1) \ldots
B: \ldots P(S_2) \ldots P(S_3) \ldots V(S_3) \ldots V(S_2) \ldots
C: \ldots P(S_3) \ldots P(S_2) \ldots V(S_2) \ldots V(S_3)

\text{Prio}(A) = H \\
\text{Prio}(B) = M \\
\text{Prio}(C) = L

C(S_1) = H \\
C(S_2) = C(S_3) = M

Task A

Task B

Task C

Run with priority “M”

Run with its own priority
Example: PCP

A: \( \ldots P(S_1) \ldots V(S_1) \ldots \)
B: \( \ldots P(S_2) \ldots P(S_3) \ldots V(S_3) \ldots V(S_2) \ldots \)
C: \( \ldots P(S_3) \ldots P(S_2) \ldots V(S_2) \ldots V(S_3) \ldots \)

\[ \text{Prio}(A) = H \]
\[ \text{Prio}(B) = M \]
\[ \text{Prio}(C) = L \]

\[ C(S_1) = H \]
\[ C(S_2) = C(S_3) = M \]

Task A

Task B

Task C

Run with priority “M+ε”

Run with its own priority
Why adding $\varepsilon$?

Example: assume

- tasks $H$ and $L$ ($H$ has higher priority than $L$)
- both share semaphores $S_1, S_2$
- $H$ uses $S$ (not shared)
- Thus, $\text{ceiling}(S_i) = H$ and
- $\text{SystemCeiling} = H$ if any semaphore is locked

If not $H+\varepsilon$, $L$ will be blocked
PCP: Blocking time calculation
(the same as HLP)

\[ B(i) = \max\{CS(k, i, S) | \text{pri}(k) < \text{pri}(i) \leq C(S)\} \]

where

- \( B(i) \) is the maximal time that task \( i \) may be blocked due to HLP
- \( CS(k, i, S) \) is the length of a critical section that may be locked by task \( k \) and \( i \) with \( S \).
Exercise: implementation of PCP

- Implement P,V-operations that follow PCP
- (this is not so easy)
Properties of PCP: + and -

- Bounded priority inversion (+)
- Deadlock free (+)
- Number of blocking = 1 (+)
- Better response times for high priority tasks (+)
  - Avoid un-necessary blocking
- Not easy to implement (-)
## Summary

<table>
<thead>
<tr>
<th>Feature</th>
<th>NPP</th>
<th>BIP</th>
<th>HLP</th>
<th>PCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded Priority Inversion</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Deadlock free</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Un-necessary blocking</td>
<td>yes</td>
<td>no</td>
<td>yes/no</td>
<td>no</td>
</tr>
<tr>
<td>Blocking time calculation</td>
<td>easy</td>
<td>hard</td>
<td>easy</td>
<td>easy</td>
</tr>
<tr>
<td>Number of blocking</td>
<td>1</td>
<td>&gt;1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Implementation</td>
<td>easy</td>
<td>easy</td>
<td>easy</td>
<td>hard</td>
</tr>
</tbody>
</table>