Measures for synchronization are required!

In anything higher than assembler level on small, predictable µcontrollers:

the set of operations, which atomicity is guaranteed by the underlying system (e.g. hardware). CPU-operations and interrupt structures is nevertheless possible and done frequently.

Atomic operations:

• On assembler level: synchronization by employing knowledge about the atomicity of — and no other value is ever observable.

x = 0 either will result in

either

x := 5;

Task 2:

| x := 0; |

i++; {in one thread}  …

Assuming that any access to a word in the system is an atomic operation:

synchronize a task with an event given by another task.

Critical sections:

Assuming that all ‘perhapses’ are applying: how to expand this code?

☞ … but perhaps the processor supplies atomic operations for the actual case.

or

local (and protected by language-, os-, or hardware-mechanisms)

… and perhaps it is a register.

Any manipulations on the main memory … will not be atomic

… but perhaps it is an 8-bit integer.

Handling a 64-bit integer on a 8- or 16-bit controller

☞ … and … real-time systems are often complex and is possibly expanded at a later stage …

Thus all data is declared …

☞ … either local (and protected by language-, os-, or hardware-mechanisms)

☞ … or it is ‘out in the open’ and all access need to be synchronized!

Synchronization:

Synchronization in real-time systems

Some synchronization terms:

• Condition synchronization: synchronize a task with an event given by another task.

• Critical sections: code fragments which contain access to shared resources and need to be executed without interference with other critical sections, sharing the same resources.

• Mutual exclusion: protection against asynchronous access to critical sections.

• Atomic operations: the set of operations, which atomicity is guaranteed by the underlying system (e.g. hardware).

☞ there must be a set of atomic operations to start with!

Word-access atomicity:

Assuming that any access to a word in the system is an atomic operation:

e.g. assigning two values (not wider than the size of word) to a memory cell simultaneously:

Task 1: \[ x = 0; \]

Task 2: \[ x = 5; \]

will result in either \[ x = 0 \text{ or } x = 5 \] — and no other value is ever observable.

References for this chapter


[Hoare95] Tony Hoare Communicating sequential processes Prentice Hall, 1995


Hardin, Peter Dibble, James Gosling, Mark Turnbull & Rudy Belliardi

© 2009 Uwe R. Zimmer, The Australian National University

Page 431 of 769 (chapter 6: to 549)
Semaphores

Types of semaphores:
- General semaphores (counting semaphores): non-negative number; (range limited by the system) P and V increment and decrement the semaphore by one.
- Binary semaphores: restricted to 0,1; Multiple U ($\uparrow$signal) calls have the same effect than 1 call.
- Atomic 'test-and-set' operations at hardware level are usually binary semaphores.
- Quantity semaphores: The increment (and decrement) value for the semaphore is specified as a parameter with P and V.

Synchronization by semaphores

Assuming further that there is a shared memory area between two processes:
- A set of processes agree on a variable $S$ operating as a flag to indicate synchronization conditions.
- An atomic operation P on $S \geq 0$ stands for 'pass' (Dutch for 'pass'):
  - P: if $S \geq 0$ then $S := S - 1$ also: Wait(Synchron Until_True)
- An atomic operation P on $S = 0$ stands for 'vrygeven' (Dutch for 'to release'):
  - U: if $S = 1$ also: Signal(|Set,True)

The variable $S$ is then called a semaphore.

Ok-level P is usually also suspending the current task until $S > 0$.
CPU-level P indicates whether it was successful, but the operation is not blocking.

Synchronization by flags

Assuming further that there is a shared memory between two processes:
- A set of processes agree on a variable $S$ operating as a flag to indicate synchronization conditions.
- A set of processes agree on a variable $S$ operating as a flag to indicate synchronization conditions.

Sequence of operations: $\{A | X \}=\{B | Y\}$

Synchronization in Ada95

package Ada.Synchronous_Task_Control is
  type Suspension_Object is limited private;
  procedure Set,True (S : in out Suspension_Object);
  procedure Set,False (S : in out Suspension_Object);
  function Current_State (S : Suspension_Object) return Boolean;
  procedure Suspend,Until_True (S : in out Suspension_Object; procedure P1; statement X; ... end P1;)
  procedure Suspend_Until_True (S : in out Suspension_Object; procedure P2; statement A; ... end P2;)

  only one task can be blocked at Suspend,Until_True ('strict version of a binary semaphore')
  Program_Error will be raised with the second task trying to suspend itself
  no queues $\Rightarrow$ minimal run-time overhead
Deadlocks can be generated by all kinds of synchronization methods. Deadlocks in a multiprocessor environment each process is associated with exactly one processor.}

Semaphores are considered not adequate for the real-time domain. (all concurrent and real-time languages offer more abstract and safer synchronization methods).

Conditional critical regions

All guards need to be re-evaluated, when any conditional critical region is left:

- All involved processes are activated to test their guards
- There is no order in the re-evaluation phase
- Potential deadlocks

With semaphores the conditional critical regions are scattered all over the code. On a larger scale: same problems as with semaphores.

The language Edison uses conditional critical regions for synchronization in a multiprocessor environment (each process is associated with exactly one processor).
Monitors with condition synchronization

Monitors in Modula-1

Monitors in ‘C’ / POSIX

Monitors in Hoare notation

Attributes include:

- semantics for trying to lock a mutex which is locked already by the same thread
- sharing of mutexes and condition variables between processes
- priority ceiling
- clock used for timeouts

Suggestions to overcome the multiple-tasks-in-monitor-problem:

- A signal operation has the side-effect of executing a return statement.
- More efficient evaluation of the guards:
  - the task leaving the monitor can evaluate all guards and the right tasks can be activated.
  - Blocked tasks may be ordered and livelocks prevented.

Suggestions to overcome the multiple-tasks-in-monitor-problem:

- A signal operation which unblocks another process does not block the caller, but the unblocked process must gain access to the monitor again.

The signalling and the waiting processes are both active in the monitor!
Monitors in C' / POSIX

(types and creation)

• Static data is shared between all objects of a class.

• Synchronized methods and code blocks

Considerations:

1. Synchronized methods and code blocks:
   - In order to implement a monitor all methods in an object need to be synchronized.
     » any other standard method can break the monitor and enter at any time.
   - Methods outside the monitor-object can synchronize at this object.
   - it is impossible to analyse a monitor locally, since lock accesses can exist all over the system.
   - Static data is shared between all objects of a class.
     » access to static data need to be synchronized over the whole class.

Java provides two mechanisms to construct monitors:

- Synchronized methods and code blocks
  - all methods and code blocks which are using the synchronized tag are mutually exclusive with respect to the addressed class.
  - Notification methods are mutually exclusive with respect to the addressed class.

- Monitors in Real-time Java

Considerations:

- in static methods:
  - access to static data need to be synchronized over the whole class.
  - or in static methods public synchronized static method();
Monitors in Real-time Java

Considerations:

2. Notification methods: wait, notify, and notifyAll
   - wait suspends the thread and releases the local lock only
   - notify calls will keep all enclosing locks.
   - notify() and notifyAll() does not release the lock.
   - methods, which are activated via notification need to wait for lock-access.
   - wait-suspended threads are held in a queue, thus notify() is waking the threads in order
   - livelocks are prevented at this level (in opposition to Java).
   - There are no explicit conditional variables.
   - every notified thread needs to wait for the lock to be released and to re-evaluate its entry condition

Monitors in Real-time Java

(multiple-readers-one-writer-example: wait-notifyAll method)

```java
public synchronized void StartWrite () throws InterruptedException {
    if (writing | readers > 0) {
        synchronized (OkToRead) {
            wait();
            readers++;    // wakeup one writer
        }
    }
    writing = true;
}

public synchronized void StopWrite () {
    writing = false;
    while (readers > 0 || writing)
        wait();
}
```

Standard monitor solution:

- declare the monitored data-structures private to the monitor object (non-static).
- introduce a class ConditionVariable:
  ```java
  public class ConditionVariable {
      public boolean wantToSleep = false;
  }
  ```
- introduce synchronization-scopes in monitor-methods:
  ```java
  public boolean wantToSleep {
      synchronized (this) {
          if (writing | readers > 0) {
              // synchronized block
              writing = false;
              notify();
          } else {
              wait();
              // synchronized block
              writing = true;
              notifyAll();
          }
      }
      return wantToSleep;
  }
  ```
- make sure that all methods in the monitor are implementing the correct synchronizations.
- make sure that no other method in the whole system is synchronizing on this monitor-object.

Monitors in Real-time Java

(multiple-readers-one-writer-example: usage of external conditional variables)

```java
public class ReadersWriters {
    private int     readers        = 0;
    private int     waitingReaders = 0;
    private boolean writing        = false;
    private int     waitingWriters = 0;
    private boolean wantToSleep = false;

    public synchronized void StartWrite () throws InterruptedException {
        if (waitingWriters > 0) {
            waitingWriters--;
            if (writing | readers > 0) {
                synchronized (OkToWrite) {
                    OkToWrite.wait();
                    OkToWrite.wantToSleep = true;
                }
            } else {
                synchronized (OkToRead) {
                    OkToRead.wait();
                    OkToRead.wantToSleep = false;
                }
            }
        } else {                  writing = false;
            synchronized (OkToRead) {
                OkToRead.notifyAll();    // wakeup all readers
            }
        }
        readers++;   // notify one writer
    }

    public synchronized void StopWrite () {
        writing = true;
        while (readers > 0 || writing)
            wait();
    }

    public synchronized void StartRead () throws InterruptedException {
        if (waitingReaders > 0) {
            waitingReaders--;
            if (readers == 0)
                readers = 0;
            else {
                synchronized (OkToWrite) {
                    OkToWrite.wait();
                    OkToWrite.wantToSleep = false;
                }
            }
        } else {                  writing = false;
            synchronized (OkToWrite) {
                OkToWrite.notifyAll();    // wakeup all readers
            }
            waitingReaders = 0;
        }
    }

    public synchronized void StopRead () {
        readers = 0;
        while (readers > 0 || writing)
            wait();
    }
}
```
Monitors in Real-time Java

- public void StartRead () throws InterruptedException
  
  synchronized (OkToRead)
  
  synchronized (this)
  
  if (writing | waitingReaders > 0)
    writingReaders++;
    OkToRead.waitToSleep = true;
  else
    
    OkToRead.waitToSleep = false;
  
  if (OkToRead.waitToSleep) OkToRead.wait ();

The parent class might need to be adapted in order to suit the global synchronization scheme.

The unjustified locked calling monitor

the called monitor is aware of the suspension and allows other threads to enter.

Synchronization by protected objects

Synchronization by protected objects in Ada95

protected type Shared_Data (Initial : Data_Item) is

  function Read return Data_Item;

procedure Write (New_Value : in Data_Item);

private

  The_Data : Data_Item := Initial;

end Shared_Data;

Synchronization by protected objects in Ada95

Some read-only operations do not need to be mutual exclusive:

protected type Shared_Data (Initial : Data_Item) is

  function Read return Data_Item;

procedure Write (New_Value : in Data_Item);

private

  The_Data : Data_Item := Initial;

end Shared_Data;

protected type Bounded_Buffer is

  subtype Count is Natural range 0 .. Buffer_Size;

type Buffer_T is array (Index) of Data_Item;

The_Data : Data_Item := Initial;

end Bounded_Buffer;

Object-orientation and synchronization

Since mutual exclusion, notification, and condition synchronization schemes need to be designed and analysed considering the implementation of all involved methods and guards:

- new methods cannot be added without re-evaluating the whole class!

In opposition to the general re-use idea of object-oriented programming, the re-use of synchronized classes (e.g. monitors) need to be considered carefully.

- The parent class might need to be adapted in order to suit the global synchronization scheme.

- Inheritance anomaly (Matsuoaka & Yonezawa ’93)

Methods to design and analyse expandable synchronized systems exist, but are fairly complex and are not provided in any current object-oriented language.

Criticalism of monitors

- Mutual exclusion is solved elegantly and safely.

- Conditional synchronization is on the level of semaphores still

- All criticism on semaphores apply

- mixture of low-level and high-level synchronization constructs.

Synchronization by protected objects

Synchronization by protected objects in Ada95

Condition synchronization is realized in the form of protected procedures combined with boolean conditional variables (barriers). entries in Ada95:

Buffer_Size : constant Integer := 10;

type Index is mod Buffer_Size;

subtype Count is Natural range 0 .. Buffer_Size;

type Buffer_T is array (Index) of Data_Item;

protected type Bounded_Buffer is

  entry Set (Item : out Data_Item);

  entry Put (Item : in Data_Item);

private

  First : Index is Index’First;

  Last : Index is Index’Last;

  Num : Count := 0;

  Buffer : Buffer_T;

  and Bounded_Buffer;
Synchronization by protected objects in Ada95

The count attribute indicates the number of tasks waiting at a specific queue:

protected body BoundedBuffer is
entry Get (item : out Data_Item) when Full is
begin
item := Buffer (First);
First := First + 1;
Num := Num - 1;
end Get;
entry Put (item : in Data_Item) when Empty is
begin
Last := Last + 1;
Buffer (Last) := item;
Num := Num + 1;
end Put;
end BoundedBuffer;

Synchronization by protected objects in Ada95

The count attribute indicates the number of tasks waiting at a specific queue:

protected type Broadcast is
entry Receive (m : out Message);
entry Send (m : in Message);
protected type Blocker is
entry Proceed;
private
Release : Boolean := False;
end Blocker;

package body Blocker is
begin
select
Buffer.Put (Some_Data);
when Arrived
Buffer.Put (Some_Data);
end select;
else
delay 10.0;
then abort;
end if;
end Blocker;

package body Broadcast is
entry Receive (m : out Message) when Empty is
begin
m := New_Message;
end Receive;
end Broadcast;

package body BoundedBuffer is
entry Get (item : out Data_Item) when Full is
begin
item := Buffer (First);
First := First + 1;
Num := Num - 1;
end Get;
end BoundedBuffer;

The count attribute indicates the number of tasks waiting at a specific queue:

protected type Broadcast is
entry Receive (m : out Message);
entry Send (m : in Message);
protected type Blocker is
entry Proceed;
private
Release : Boolean := False;
end Blocker;

package body Blocker is
begin
select
Buffer.Put (Some_Data);
when Arrived
Buffer.Put (Some_Data);
end select;
else
delay 10.0;
then abort;
end if;
end Blocker;

package body Broadcast is
entry Receive (m : out Message) when Empty is
begin
m := New_Message;
end Receive;
end Broadcast;

How to implement a queue, at which every task can be released only once per triggering event? E.g., by employing two entries:

package SingleRelease is
entry Wait;
entry Display;
private
MainDoor, FrontDoor : Boolean := False;
end SingleRelease;

Synchronization by protected objects in Ada95

How to implement a queue, at which every task can be released only once per triggering event? E.g., by employing two entries:

package SingleRelease is
entry Wait;
entry Display;
private
MainDoor, FrontDoor : Boolean := False;
end SingleRelease;

Barriers are not evaluated while inside a protected object or on leaving a protected function.

Further refinements on task control by:

- **Entry families**: a protected entry declaration can contain a discrete subtype selector, which can be evaluated by the barrier (other parameters cannot be evaluated by barriers) and implements an array of protected entries.
- **Requeue facility**: protected operations can use `requeue` to redirect tasks to other internal, external, or private entries. The current protected operation is finished and the lock on the object is released.
- **Internal progress first-rule**: internally requeued tasks are placed at the head of the waiting queue!
- **Private entries**: protected entries which are not accessible from outside the protected object, but can be employed as destinations for requeue operations.
Synchronization

Synchronization by protected objects in Ada95 (restrictions applying to protected operations)

Code inside a protected procedure, function or entry is bound to non-blocking operations (which would keep the whole protected object locked): Thus the following operations are prohibited:

- entry call statements
- delay statements
- task creations or activations
- calls to sub-programs which contains a potentially blocking operation
- select statements
- accept statements

☞ The requeue facility allows for a potentially blocking operation, but releases the current lock!

Real-Time Java

- mutual exclusion (synchronized methods) as the only support.
- general notification feature (no conditional variables)
- non-restricted object oriented extension introduces hard to predict timing behaviours.

Modula-1, CHILL

- full monitor implementation (Dijkstra-Hoare monitor concept).
- no more, no less, ...
- all features of and criticism about monitors apply.

 POSIX

- all low level constructs available.
- no connection with the actual data-structures.
- error-probe.
- non-determinism introduced by 'release some' semantics of conditional variables (cond_signal).

Ada95

- complete synchronization support
- low-level semaphores for very special cases.
- predictable timing (v= scheduler).
- most memory oriented synchronization conditions are realized by the compiler or the run-time environment directly rather then the programmer.

(Ada95 is currently without any mainstream)

Summary

Shared memory based synchronization

General

Criteria:

- level of abstraction
  - centralized vs. distributed concepts
  - support for consistency and correctness validations
  - error sensitivity
  - predictability
  - efficiency

POSIX

- Synchronous
  - Remote invocation

Ada95

- Synchronization model
  - Asynchronous
  - Synchronous
  - Remote invocation

Addressing (name space)

- direct communication
- mail-box communication

Message structure

- arbitrary
- restricted to 'basic' types
- restricted to un-typed communications

Summary

Shared memory based synchronization

Modula-1, CHILL

- full monitor implementation (Dijkstra-Hoare monitor concept).

 POSIX

- Synchronous
  - Remote invocation

Ada95

- Synchronization model
  - Asynchronous messages
  - send the message directly

If there is a listener:

If the receiver becomes available at a later stage:

☞ the message need to be buffered
Synchronization

Message-based synchronization

Synchronous messages

Delay the sender:
- until the receiver got the message

Remote invocation

Delay the sender, until:
- a receiver becomes available
- a receiver executed an addressed routine
- a receiver got the message

Asynchronous remote invocation

Delay the sender, until:
- a receiver becomes available
- a receiver got the message
Async. receive

Real-Time & Embedded Systems

Synchronization

Message-based synchronization

Asynchronous remote invocation

Delay the sender, until:
- a receiver becomes available
- a receiver got the message

P1
async. send
async. receive
async. send
P2

Real-Time & Embedded Systems

Synchronization

Synchronous vs. asynchronous communications

Purpose 'synchronous':
- synchronous messages / remote invocations

Purpose 'asynchronous':
- asynchronous messages / asynchronous remote invocations

- 'Real' synchronous message passing in distributed systems requires hardware support.
- Asynchronous message passing requires the usage of (infinite?) buffers.

- Synchronous communications are emulated by a combination of asynchronous messages in some systems.
- Asynchronous communications can be emulated in synchronous message passing systems by introducing 'buffer-tasks' (de-coupling sender and receiver as well as allowing for broadcasts).

Real-Time & Embedded Systems

Synchronization

Addressing (name space)

Communication medium:

<table>
<thead>
<tr>
<th>Connections</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-to-one</td>
<td>buffer, queue, synchronization</td>
</tr>
<tr>
<td>one-to-many</td>
<td>multicast</td>
</tr>
<tr>
<td>one-to-all</td>
<td>broadcast</td>
</tr>
<tr>
<td>many-to-one</td>
<td>local server, synchronization</td>
</tr>
<tr>
<td>all-to-one</td>
<td>general server, synchronization</td>
</tr>
<tr>
<td>many-to-many</td>
<td>general network, bus-system</td>
</tr>
</tbody>
</table>

Reading and writing values of any type to a stream:

procedure S'Write (Stream: access Ada.Streams.Root_Stream_Type'Class; Item : in T); procedure S'Class Write (Stream: access Ada.Streams.Root_Stream_Type'Class; Item : in T'Class);
procedure S'Read (Stream: access Ada.Streams.Root_Stream_Type'Class; Item : out T);
procedure S'Class Read (Stream: access Ada.Streams.Root_Stream_Type'Class; Item : out T'Class);

package Ada.Streams is
  procedure Write (…) is abstract;
  procedure Read (…) is abstract;
private
  … not specified by the language end Ada.Streams;
end Ada.Streams;

Real-Time & Embedded Systems

Synchronization

Message structure (Ada95)

• Machine dependent representations need to be taken care of in a distributed environment.
• Communication system is often outside the typed language environment.

Most communication systems are handling streams (packets) of a basic element type only.

Conversion routines for data-structures other then the basic element type are supplied:
... manually (POSIX)
... semi-automatic (real-time CORBA)
... automatic and are typed-persistent (Ada95)

Asynchronous remote invocation

Real-Time & Embedded Systems

Synchronization

Message-based synchronization

Practical message-passing systems:

POSIX:
- ordered indirect (asymmetrical | symmetrical) asynchronous byte-level many-to-many message passing

CHILL:
- ordered indirect [asymmetrical | symmetrical] [synchronous | asynchronous] typed [many-to-many | many-to-one] message passing

Occam2:
- "channels":
  - indirect synchronous fully-typed one-to-one message passing

Ada95:
- ordered direct [asymmetrical | synchronous | asynchronous] fully-typed many-to-one remote invocation

Real-time Java:
- no communication via messages available
Message-based synchronization

Communication is ensured by means of a 'channel', which:
- can be used by one writer and one reader process only
- and is synchronous:

```
CHILL OF INT SensorChannel:
PPR:
  | reading:     SEQ i = 0 FOR 1000     SEQ -- generate reading
  |     SensorChannel i reading     INT data:     SEQ i = 0 FOR 1000     SEQ -- display data
  |                               SensorChannel data
| send SensorBuffer (reading)    receive case (SensorBuffer in data): _
| receive case (SensorBuffer in data): _
signal SensorChannel = (int) to consumer-type;
| send SensorChannel (reading)    receive case to consumer
  | (SensorChannel in data): _
  | (SensorChannel in data): _
```

tasks are synchronized at these points

Remote invocation

Delay the sender, until:
- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine
- a receiver passed the results

Remote invocation (Rendezvous)

Delay the sender, until:
- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine
- a receiver passed the results

Remote invocation (Rendezvous)

Delay the sender, until:
- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine
- a receiver passed the results

Remote invocation (Rendezvous)

Delay the sender, until:
- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine
- a receiver passed the results

Remote invocation (Rendezvous)

Delay the sender, until:
- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine
- a receiver passed the results
Message-based synchronization in Ada95

(Extended rendezvous)

```
select statement ::= selective accept | conditional accept | timed accept
```

\[\text{selective accept} \] implements:
- \(\ldots\) wait for more than a single rendezvous at any one time
- \(\ldots\) time-out if no rendezvous is forthcoming within a specified time
- \(\ldots\) withdraw its offer to communicate if no rendezvous is available immediately
- \(\ldots\) terminate if no clients can possibly call its entries

```
select alt ::= select | guarded alternative | timed alternative | asynchronous select
```

```
select alt::=
| guarded alternative ::= accept alternative | delayed alternative |
| timed alternative ::= accept alternative | delayed alternative |
| asynchronous select ::= guarded alternative |
```

Message-based synchronization in Ada95

(Extended rendezvous)

```
select statement ::= selective accept | conditional accept | timed accept
```

```
select alt ::= select | guarded alternative | timed alternative | asynchronous select
```

```
select alt::=
| guarded alternative ::= accept alternative | delayed alternative |
| timed alternative ::= accept alternative | delayed alternative |
| asynchronous select ::= guarded alternative |
```

Selective waiting in Occam2

```
ALT
Guard1  Process1
Guard2  Process2
```

```
Guard1 ::= if x <= y -> m := x; 
        | x >= y -> m := y; 
        fi
```

Dijkstra's guarded commands:
- \(\text{if } x \leq y \rightarrow m := x \) \(\text{and } x \geq y \rightarrow m := y \) \(\text{fi}\)
- the programmer needs to design the alternatives as 'parallel' options:
- all cases need to be covered and overlapping conditions need to lead to the same result

- Extremely different philosophy: C-switch:
  - \(\text{switch } (x) \{\)
    - case 1: \(r := 3\); break;
    - case 2: \(r := 2\); break;
    - case 3: \(r := 1\);
  - \(\text{fi}\)
- the sequence of alternatives has a crucial role.

Synchronization

Some things to consider for task-entries:
- In contrast to protected-object-entries, task-entries can call other blocking operations.
- Accept statements can be nested (but need to be different).
- \(\Rightarrow\) helpful e.g. to synchronize more than two tasks.
- Accept statements can have a dedicated exception handler (like any other code-block).
- Exceptions, which are not handled during the rendezvous phase are propagated to all involved tasks.
- Parameters cannot be direct access parameters, but can be access-types.
- \(\Rightarrow\) count on task-entries is defined, but is only accessible from inside the tasks owning the entry.
- Entry families (arrays of entries) are supported.
- Private entries (accessible for internal tasks) are supported.
The idea in both cases is to withdraw a synchronization request and not to implement polling or busy-waiting.
Summary

Synchronization

• Shared memory based synchronization
  • Flags, condition variables, semaphores,
    ... conditional critical regions, monitors, protected objects.
  • Guard evaluation times, nested monitor calls, deadlocks, ...
    ... simultaneous reading, queue management.
  • Synchronization and object orientation, blocking operations and re-queuing.

• Message based synchronization
  • Synchronization models, addressing modes, message structures
  • Selective accepts, selective calls
  • Indeterminism in message based synchronization