Synchronization in real-time systems

There are many concurrent entities in a real-time systems:

- Interrupt handlers
- Tasks
- Dispatchers
- Times
- …

... 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!

... and ...

Measures for synchronization are required!

Synchronization

Synchronization in real-time systems

Synchronization: the run-time overhead?

- Is the potential overhead justified for simple data-structures:
  
  ```
  int i;
  ```

  ```
  i++; {in one thread}  
  i=0; {in another thread}
  ```

- Are those operations atomic?
- Do we really need to introduce full featured synchronization methods here?

Synchronization methods

- Shared memory based synchronization
  - Semaphores
  - Conditional critical regions
  - Monitors
  - Mutual exclusive critical sections
  - Synchronized methods
  - Protected objects
  - Real-time Java
  - Ada95

- Message based synchronization
  - Asynchronous messages
  - Synchronous messages
  - Remote invocation, remote procedure call
  - Synchronization in distributed systems

There are many concurrent entities in a real-time systems:

- Interrupt handlers
- Tasks
- Dispatchers
- Times
- …

... 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!

... and ...

Measures for synchronization are required!

Synchronization

Synchronization in real-time systems

Synchronization: the run-time overhead?

- Is the potential overhead justified for simple data-structures:
  
  ```
  int i;
  ```

  ```
  i++; {in one thread}  
  i=0; {in another thread}
  ```

- Are those operations atomic?
- Do we really need to introduce full featured synchronization methods here?

Synchronization methods

- Shared memory based synchronization
  - Semaphores
  - Conditional critical regions
  - Monitors
  - Mutual exclusive critical sections
  - Synchronized methods
  - Protected objects
  - Real-time Java
  - Ada95

- Message based synchronization
  - Asynchronous messages
  - Synchronous messages
  - Remote invocation, remote procedure call
  - Synchronization in distributed systems

There are many concurrent entities in a real-time systems:

- Interrupt handlers
- Tasks
- Dispatchers
- Times
- …

... 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!

... and ...

Measures for synchronization are required!

Synchronization

Synchronization in real-time systems

Synchronization: the run-time overhead?

- Is the potential overhead justified for simple data-structures:
  
  ```
  int i;
  ```

  ```
  i++; {in one thread}  
  i=0; {in another thread}
  ```

- Are those operations atomic?
- Do we really need to introduce full featured synchronization methods here?

Synchronization methods

- Shared memory based synchronization
  - Semaphores
  - Conditional critical regions
  - Monitors
  - Mutual exclusive critical sections
  - Synchronized methods
  - Protected objects
  - Real-time Java
  - Ada95

- Message based synchronization
  - Asynchronous messages
  - Synchronous messages
  - Remote invocation, remote procedure call
  - Synchronization in distributed systems

There are many concurrent entities in a real-time systems:

- Interrupt handlers
- Tasks
- Dispatchers
- Times
- …

... 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!

... and ...

Measures for synchronization are required!
Real-Time & Embedded Systems

Synchronization

Synchronization by flags

Assuming further that there is a shared memory area between two processes:
- A set of processes agree on a (word-size) atomic variable operating as a flag to indicate synchronization conditions.
- CPU-level P indicates whether it was successful, but the operation is not blocking.

Synchronization by semaphores

(Dijkstra 1968)

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 … and ...
- an operation P on S – P stands for 'passeren' (Dutch for 'pass'):
  \[ P: \begin{cases} \text{if } S > 0 \text{ then } S := S - 1 \end{cases} \]
  also: \text{wait(}S\text{);} \text{Suspend_Until_True};
- an atomic operation V on S – V stands for 'vrijgeven' (Dutch for 'to release'):
  \[ V: \begin{cases} \text{if } S < 0 \text{ then } S := S + 1 \end{cases} \]
  \text{also: } \text{signal(}S\text{);} \text{Set_True};
- \text{the variable } S \text{ is then called a semaphore.}

OS-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.

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 \([0, 1]\); \text{Multiple \text{V}(signal) calls have the same effect than 1 call.}
  • binary semaphores are sufficient to create all other semaphore forms.
  • 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.

Program_Error will be raised with the second task trying to suspend itself.

Semaphores 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 Suspend_Until_False (S : in out Suspension_Object);

Semaphores 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 Suspend_Until_False (S : in out Suspension_Object);
Synchronization

Semaphores in POSIX

int sem_init( sem_t *sem_location, int pshared, unsigned int value);
int sem_destroy( sem_t *sem_location);
int sem_post( sem_t *sem_location);
int sem_getvalue( sem_t *sem_location, int *value);

generate semaphore for usage between processes (otherwise for threads of the same process only)

Deadlock by semaphores

with Ada.Synchronous_Task_Control; use Ada.Synchronous_Task_Control;
X, Y : Suspension.Object;
task B; task R;
task body B is task body R is
begin begin
    suspend_until_true (Y); suspend_until_true (X);
and B; and R;
end B; end R;
end

Semaphores are not bound to any resource or method or region
⇒ Adding or deleting a single semaphore operation some place might stall the whole system
Semaphores are scattered all over the code
⇒ hard to read, error-prone
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

buffer : buffer_t;
resource critical_buffer_region : buffer;

process producer;
loop
    region critical_buffer_region
        when buffer.size < N do
            -- place in buffer etc.
        end region

process consumer;
loop
    region critical_buffer_region
        when buffer.size > 0 do
            -- take from buffer etc.
        end region

Synchronization in a multiprocessor environment (each process is associated with exactly one processor).

The language Edison uses conditional critical regions for synchronization in a multiprocessor environment:

The language Edison uses conditional critical regions for synchronization in a multiprocessor environment:

Basic idea:
⇒ Collect all operations and data-structures shared in critical regions in one place, the monitor.
⇒ Formulate all operations as procedures or functions.
⇒ Prohibit access to data-structures, other than by the monitor-procedures.
⇒ Assure mutual exclusion of the monitor-procedures.
⇒ 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 or potential livelocks
⇒ As with semaphores the conditional critical regions are scattered all over the code.
⇒ on a larger scale: same problems as with semaphores.
### Monitors in Modula-1

- Monitors can be used for synchronization between processes.
- Each monitor has a condition variable that can be accessed by processes.
- Processes can enter and exit the monitor.

```pascal
BEGIN
   busy := FALSE;
END;
```

### Monitors with condition synchronization

- Condition variables are implemented by semaphores.
- `wait(s)` delays the process until the condition is satisfied.
- `signal(s)` activates the waiting processes.

```pascal
BEGIN
   IF busy THEN
      WAIT (free)
   END;
END;
```

### Monitors in 'C' / POSIX

- `pthread_mutexattr_t` and `pthread_condattr_t` are used for attribute management.
- `pthread_mutex_init()` initializes a mutex.
- `pthread_cond_init()` initializes a condition variable.

```pascal
typedef … pthread_cond_t;
typedef … pthread_mutexattr_t;
```

### Synchronization

- Semaphores are used for conditional synchronization.
- Processes can block and unblock each other.
- The `sem_post()` function is used to notify the waiting processes.
- The `sem_wait()` function blocks the current process.

```pascal
BEGIN
   busy := TRUE;
END;
```

### Example

- A simple buffer for sending and receiving data.
- Processes can block until a signal is received.

```pascal
interface MODULE resource_control;
PROCEDURE allocate;
PROCEDURE deallocate;
DEFINE allocate, deallocate;
BEGIN
   VAR busy : BOOLEAN; free : SIGNAL;
   BEGIN
      numberInBuffer := 0;
      top := 0; base := 0;
   end;

   BEGIN
      IF busy THEN
         WAIT (free)
      END;
   END;

   BEGIN
      active_in_the_monitor := FALSE;
      busy := FALSE;
      numberInBuffer := 0;
      top := 0; base := 0;
   end;
```

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

- Ensure that condition variables are manipulated correctly.
- Use semaphores instead of monitors for better performance.
- Monitor the use of condition variables to prevent deadlocks.

```pascal
BEGIN
   numberInBuffer := 0;
   top := 0; base := 0;
   BEGIN
      active_in_the_monitor := FALSE;
      busy := FALSE;
      numberInBuffer := 0;
      top := 0; base := 0;
   end;
```

### Example

- Using condition variables to control access to a shared resource.
- Processes can block and unblock each other.

```pascal
BEGIN
   numberInBuffer := 0;
   top := 0; base := 0;
   BEGIN
      active_in_the_monitor := FALSE;
      busy := FALSE;
      numberInBuffer := 0;
      top := 0; base := 0;
   end;
```

### Monitors with condition synchronization

- Condition variables are implemented by semaphores.
- Processes can be blocked and unblocked.
- `sem_post()` is used to activate waiting processes.
- `sem_wait()` blocks the current process.

```pascal
BEGIN
   busy := TRUE;
END;
```

### Conditions and mutexes

- A process can gain access to the monitor again.
- The `sem_post()` function signals the waiting processes.
- The `sem_wait()` function blocks the current process.

```pascal
BEGIN
   busy := TRUE;
END;
```

### Real-Time & Embedded Systems

- Monitors and condition variables are used for synchronization.
- Processes can be blocked and unblocked.
- The `sem_post()` function is used to notify waiting processes.
- The `sem_wait()` function blocks the current process.

```pascal
BEGIN
   busy := TRUE;
END;
```
Synchronization in Real-time Java

Monitors in 'C' / POSIX

Real-Time & Embedded Systems

Monitors in Real-time Java

Synchronization

Real-Time & Embedded Systems

Monitors in 'C' / POSIX

Real-Time & Embedded Systems

Monitors in Real-time Java

Synchronization

Real-Time & Embedded Systems

Monitors in 'C' / POSIX

Real-Time & Embedded Systems

Monitors in Real-time Java

Synchronization

Real-Time & Embedded Systems

Monitors in 'C' / POSIX

Real-Time & Embedded Systems
Monitors in Real-time Java

2. Notification methods: wait, notify, and notifyAll
   - wait suspends the thread and releases the local lock only
   - notify will call all enclosing locks.
   - 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 notifyAll is waking the threads in order
   - livelocks are prevented at this level (in opposition to Java).
   - There are no explicit conditional variables.

   - notify every notified thread needs
to wait for the lock to be released and to re-evaluate its entry condition.

   - monitor solution: wait-notifyAll method

   - public synchronized void StartWrite () throws InterruptedException
     while (writing || readers > 0)
     { /* wait pending write */
       waitingWriters++;
     }
   - private boolean writing = false;
   - public synchronized void StopWrite()
     { /* notify all readers */
       readers--; // manipulate the shared data
       if (readers == 0) notifyAll();
     }

   - public synchronized void StartRead () throws InterruptedException
     while (writing || waitingWriters > 0)
     { /* wait pending reads */
       waitingWriters--;
       if (writing) notify();
       readers++; // read the shared data only
     }
   - private int readers = 0;

   - introduction of synchronization-scopes in monitor-methods:

   - Standard monitor solution:

     - declare the monitored data-structures private to the monitor object (non-static).
     - introduce a class ConditionVariable:
       - public class ConditionVariable {
         private int readers = 0;
         private int waitingReaders = 0;
         private boolean writing = false;
       }
     - introduce synchronization-scopes in monitor-methods:
       - synchronize on the adequate conditional variables first and
         synchronize on the monitor-object second.
     - 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.

   - Standard solution:

     - use the class ConditionVariable:
     - synchronized monitor calls:
       - synchronized (OkToWrite) { /* manipulate the shared data */
         OkToWrite.wait();
         if (OkToWrite.wantToSleep) OkToWrite.wait();
       }
     - notifyAll
     - synchronized (OkToRead) { /* read the shared data only */
         OkToRead.notifyAll(); /* wake up all readers */
         if (OkToRead.wantToSleep) OkToRead.wait();
       }
     - notify
Synchronization

Monitors in Real-time Java

- public void StartRead () throws InterruptedException
  
  synchronized (OkToRead)
  
  synchronized (this)
  
  if (writing1 + writing3 > 0) {
    writingReaders++;
    OkToRead.wantToSleep = true;
  }
  else {
    OkToRead.wantToSleep = false;
  }
  if (OkToRead.wantToSleep) OkToRead.wait ()

- public void StopRead ()
  
  synchronized (OkToWrite)
  
  synchronized (this)
  
  readers--;
  if (readers == 0 & waitingWriters > 0) {
    waitingWriters--; 
    OkToWrite.notify ();
  }

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-usage idea of object-oriented programming, the re-usage 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 (Matsuoka & Yonezawa '93)

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

Synchronization

Monitors in POSIX & Real-time Java

= flexible and universal, but relies on conventions rather than compilers

POSIX offers conditional variables

Real-time Java is more supportive than POSIX in terms of data-encapsulation

Extreme care must be taken when employing object-oriented programming and monitors

Synchronization

Combining protected objects

« the encapsulation feature of monitors

« the coordinated entries of conditional critical regions

to

« Protected objects

- all controlled data and operations are encapsulated
- all operations are mutual exclusive
- entry guards are attached to operations
- the protected interface allows for operations on data
- no protected data is accessible (other than by defined operations)
- tasks are queued (according to their priorities)

Synchronization

Monitors in Real-time Java

- public void StartRead ()
  
  synchronized (OkToRead)

- public void StopRead ()
  
  synchronized (OkToWrite)

Synchronization

Nested monitor calls

Assuming a thread in a monitor is calling an operation in another monitor and is suspended at a conditional variable there:

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

« the calling monitor is possibly not aware of the suspension and keeps its lock!

« the unjustified locked calling monitor reduces the system performance and leads to potential deadlocks.

Suggestions to solve this situation:

- Maintain the lock Anyway: e.g. POSIX, Real-time Java
- Prohibit nested procedure calls: e.g. Modula-1
- Provide constructs which specify the release of a monitor lock for remote calls, e.g. Ada95

Synchronization

Criticism 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

(simultaneous read-access)

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

protected type SharedData (Initial : Data_Item) is

  function Read (Data : Data_Item) return Data, Item;

procedure Write (New_Value : in Data_Item);

private

  The_Data : Data_Item := Initial;

and Shared_Data_Item;

- protected functions can have 'r' parameters only and are not allowed to alter the private data (enforced by the compiler).

« protected functions allow simultaneous access (but mutual exclusive with other operations).

« there is no defined priority between functions and other protected operations in Ada95.

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

Buffer_Size : constant Integer := 16;

type Index is mod Buffer_Size;

subtype Count is Natural range 0 .. Buffer_Size;

type Buffer is array (Index) of Data_Item;

protected type Bounded_Buffer is

  entry Get (Item : out Data_Item);

end Bounded_Buffer;
Synchronization

Synchronization by protected objects in Ada95 (barriers)

protected body Bounded_Buffer is
        entry Get (Item : out Data_Item) when Num > 0 is
        begin       New_Message := M;       Arrived := Receive'count > 0;
                    Item := Buffer (First);       First := First + 1;
                    Num := Num - 1;       end Get;
        entry Put (Item : in Data_Item) when Num < Buffer_Size is
        begin       Last := Last + 1;
                    Buffer (Last) := Item;
                    Num := Num + 1;       end Put;
        end Bounded_Buffer;

Synchronization

Synchronization by protected objects in Ada95 (operations on entry queues)

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

protected type Node_T is
        (Takeoff, Recent, Cruising, Descent, Landing);
protected Node_Data is
        procedure SetNode (Node: in Node_T) is
        begin       Current_Note := Node;
        end SetNode;
        entry WaitForNode (Node_T) is
        begin       Wait := True;
        end WaitForNode;
private
        Current_Note : Node_T := Takeoff;
        Node_Data;
        end Nodes;

protected Blocker is
        procedure Trigger;
private
        Release : Boolean := False;
        end Blocker;

protected body Blocker is
        entry Proceed when Proceed'count = 5 or Release is
        begin       Release := Proceed'count > 0;
                    and Proceed;
                    and Blocker;
        end Proceed;

package Nodes is
        type Node_T is
                (Takeoff, Recent, Cruising, Descent, Landing);
        protected Node_Data is
                procedure SetNode (Node: in Node_T) is
                begin       Current_Note := Node;
                end SetNode;
                entry WaitForNode (Node_T) is
                begin       Wait := True;
                end WaitForNode;
        private
                Current_Note : Node_T := Takeoff;
                Node_Data;
        end Nodes;

package body Blocker is
        entry Proceed when Proceed'count = 5 or Release is
        begin       Release := Proceed'count > 0;
                    and Proceed;
                    and Blocker;
        end Proceed;

package body Nodes is
        type Node_T is
                (Takeoff, Recent, Cruising, Descent, Landing);
        protected Node_Data is
                procedure SetNode (Node: in Node_T) is
                begin       Current_Note := Node;
                end SetNode;
                entry WaitForNode (Node_T) is
                begin       Wait := True;
                end WaitForNode;
        private
                Current_Note : Node_T := Takeoff;
                Node_Data;
        end Nodes;

protected body Bounded_Buffer is
        entry Get (Item : out Data_Item) when Num > 0 is
        begin       New_Message := M;       Arrived := Receive'count > 0;
                    Item := Buffer (First);
                    First := First + 1;
                    Num := Num - 1;
        end Get;
        entry Put (Item : in Data_Item) when Num < Buffer_Size is
        begin       Last := Last + 1;
                    Buffer (Last) := Item;
                    Num := Num + 1;
        end Put;
        end Bounded_Buffer;

Synchronization by protected objects in Ada95 (entry families, response & private entries)

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.
- Request facility
  protected operations can use 'request' to redirect tasks to other internal, external, or private entries. The current protected operation is finished and the lock on the object is released 'before the work is done'.
- Private entries
  protected entries which are not accessible from outside the protected object, but can be employed as destinations for request operations.

Synchronization

Synchronization by protected objects in Ada95 (entry families, response & private entries)

protected body Node_T is
        entry WaitForNode (Node_T) is
        begin       Wait := True;
        end WaitForNode;
private
        Current_Note : Node_T := Takeoff;
        Node_Data;
        end Nodes;

protected body Blocker is
        entry Proceed when Proceed'count = 5 or Release is
        begin       Release := Proceed'count > 0;
                    and Proceed;
                    and Blocker;
        end Proceed;

package Nodes is
        type Node_T is
                (Takeoff, Recent, Cruising, Descent, Landing);
        protected Node_Data is
                procedure SetNode (Node: in Node_T) is
                begin       Current_Note := Node;
                end SetNode;
                entry WaitForNode (Node_T) is
                begin       Wait := True;
                end WaitForNode;
        private
                Current_Note : Node_T := Takeoff;
                Node_Data;
        end Nodes;

package body Nodes is
        type Node_T is
                (Takeoff, Recent, Cruising, Descent, Landing);
        protected Node_Data is
                procedure SetNode (Node: in Node_T) is
                begin       Current_Note := Node;
                end SetNode;
                entry WaitForNode (Node_T) is
                begin       Wait := True;
                end WaitForNode;
        private
                Current_Note : Node_T := Takeoff;
                Node_Data;
        end Nodes;

package body Blocker is
        entry Proceed when Proceed'count = 5 or Release is
        begin       Release := Proceed'count > 0;
                    and Proceed;
                    and Blocker;
        end Proceed;

package Nodes is
        type Node_T is
                (Takeoff, Recent, Cruising, Descent, Landing);
        protected Node_Data is
                procedure SetNode (Node: in Node_T) is
                begin       Current_Note := Node;
                end SetNode;
                entry WaitForNode (Node_T) is
                begin       Wait := True;
                end WaitForNode;
        private
                Current_Note : Node_T := Takeoff;
                Node_Data;
        end Nodes;

package body Blocker is
        entry Proceed when Proceed'count = 5 or Release is
        begin       Release := Proceed'count > 0;
                    and Proceed;
                    and Blocker;
        end Proceed;

package Bounded_Buffer is
        entry Get (Item : out Data_Item) when Num > 0 is
        begin       Last := Last + 1;
                    Buffer (Last) := Item;
                    Num := Num + 1;
        end Get;
        entry Put (Item : in Data_Item) when Num < Buffer_Size is
        begin       Last := Last + 1;
                    Buffer (Last) := Item;
                    Num := Num + 1;
        end Put;
        end Bounded_Buffer;

Barrier evaluations and task activations:
- on entering a protected entry, the associated barrier is evaluated (only those parts of the barrier which might have changed since the last evaluation).
- on leaving a protected procedure or entry, related barriers with tasks queued are evaluated (only those parts of the barriers which might have been altered by this procedure / entry or which might have changed since the last evaluation).
Barriers are not evaluated while inside a protected object or on leaving a protected function.

Synchronization

Synchronization by protected objects in Ada95 (entry families, response & private entries)

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

protected body Single_Release is
        entry Wait;
        procedure Trigger;
        private
        Front_Door, Main_Door: Boolean := False;
        entry Queue;
        end Single_Release;

protected body Single_Release is
        entry Wait;
        procedure Trigger;
        private
        Front_Door, Main_Door: Boolean := False;
        entry Queue;
        end Single_Release;

protected body Single_Release is
        entry Wait;
        procedure Trigger;
        private
        Front_Door, Main_Door: Boolean := False;
        entry Queue;
        end Single_Release;

protected body Single_Release is
        entry Wait;
        procedure Trigger;
        private
        Front_Door, Main_Door: Boolean := False;
        entry Queue;
        end Single_Release;
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 `faqueue` facility allows for a potentially blocking operation, but releases the current lock!

Real-Time & Embedded Systems

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

Modula-1, CHILL
- full monitor implementation (Dijkstra-Hoare monitor concept).
  - no more, no less, ...
  - all features of and criticism about monitors apply.

Modula-2
- modular and object oriented.
- complete synchronization support
- low-level semaphores for very special cases.
- predictable timing (os scheduler).
- no full monitor oriented synchronization conditions are realzed by the compiler or the run-time environment directly rather than the programmer.

Ada95
- all features of and criticism about monitors apply.
  - no more, no less, ...
  - all features of and criticism about monitors apply.

Real-Time & Embedded Systems

Message-based synchronization

Asynchronous messages

If there is a listener:
☞ send the message directly

If the receiver becomes available at a later stage:
☞ the message need to be buffered

Real-Time & Embedded Systems

Message-based synchronization

Asynchronous messages

If there is a listener:
☞ send the message directly

If the receiver becomes available at a later stage:
☞ the message need to be buffered

Real-Time & Embedded Systems

Message-based synchronization

Asynchronous messages

If there is a listener:
☞ send the message directly

If the receiver becomes available at a later stage:
☞ the message need to be buffered
**Real-Time & Embedded Systems**

### 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 got the message
- a receiver executed an addressed routine

**Asynchronous remote invocation**

Delay the sender, until:
- a receiver becomes available
- a receiver got the message
- a receiver executed an addressed routine
Real-Time & Embedded Systems

Synchronization

Message-based synchronization

Asynchronous remote invocation

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

Real-Time & Embedded Systems

Synchronization

Synchronous vs. asynchronous communications

Purpose ‘synchronous’: => synchronous messages / remote invocations
Purpose ‘in-time delivery’: => 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 allowing for broadcasts).

Asynchronous remote invocation

Practical message-passing systems:

Real-Time & Embedded Systems

Synchronization

Message structure

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

Practical message-passing systems:

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- or bus-system</td>
</tr>
</tbody>
</table>

Real-Time & Embedded Systems

Synchronization

Message structure

Communication medium:

package Ada.Streams is
  pragma Pure (Streams);
  type Root_Stream_Type is abstract tagged limited private;
  type Stream_Element is record
    ... handling streams (packets) of a basic element type only ...
    ... manually (POSIX)
    ... semi-automatic (Real-time CORBA)
    ... automatic and are typed-persistent (Ada95)
  end record;
  procedure Read (…) is abstract;
  procedure Write (…) is abstract;
  procedure S'Output (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’Class);
  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);
  procedure S'Class'Read (Stream : access Ada.Streams.Root_Stream_Type’Class; Item : out T);
end Ada.Streams;

Real-Time & Embedded Systems

Synchronization

Message-based synchronization

Real-Time & Embedded Systems

Synchronization

Message-based synchronization

Real-Time & Embedded Systems

Synchronization

Message-based synchronization
Message-based synchronization in Ada95

Ada95 supports remote invocations (extended rendezvous) in form of:

- entry points in tasks
- full set of parameter profiles supported
- parameters incl. bounds and discriminants are 'tunnelled' through byte-stream-formats.

If the local and the remote task are on different architectures, or if an intermediate communication system is employed:

- strong support for concurrency, synchronization, and communication
- monitors, buffered message passing, synchronous channels
- no communication via messages available

Remote invocation

Remote invocation (Extended 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 CHILL

CHILL is the 'CCITT High Level Language', where CCITT is the Comité Consultatif International Télégraphique et Téléphonique. The CHILL language development was started in 1973 and standardized in 1979.

- strong support for concurrency, synchronization, and communication
- monitors, buffered message passing, synchronous channels
- no communication via messages available

Remote invocation

Remote invocation (Rendezvous)

Delay the sender, until:
- a receiver becomes available
- a receiver got the message
- a receiver started an addressed routine
### Selective waiting

**Dijkstra's guarded commands:**

```
if x < y then x := x
else y := y
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:**

```
switch x { case 1: r := 3; case 2: r := 1; case 3: r := 2; break; }
```

- The sequence of alternatives has a crucial role.

### Message-based selective synchronization in Ada95

**Extended rendezvous:**

```
<entry_name>[(index)] parameters do synchronized accept <entry_name>[(index)] parameters do... -- waiting for synchronization... --... --... --... --... --... --... --... --... --... -- blocked return results
```

**Message-based synchronization in Ada95**

- Guards are referring to boolean expressions and/or channel input operations.
- The boolean expressions are local operations, i.e., if none of them evaluates to true at the time of the evaluation of the ALT-statement, then the process is stopped.
- If all triggered channel input operations evaluate to false, the process is suspended until another entry of the same name is addressed by another task.
- The ALT-statement is non-deterministic (there is also a deterministic version: PRI ALT).

```
ALT Guard1 Process1
ALT Guard2 Process2
```

- Guards refer to boolean expressions and/or channel input operations.
- The boolean expressions are local operations, i.e., if none of them evaluates to true at the time of the evaluation of the ALT-statement, then the process is stopped.
- If all triggered channel input operations evaluate to false, the process is suspended until another entry of the same name is addressed by another task.
- The ALT-statement is non-deterministic (there is also a deterministic version: PRI ALT).

### Message-based selective synchronization in Ada95

**select-statement:**

```
select accept in its full syntactical form in Ada95:
```

**select**

- `select` accepts an entry (possibly protected).
- `accept` accepts an entry (possibly protected).
- `select` implements...
  - **... wait for more than a single rendezvous at any one time**
  - **... time-out if no rendezvous is forthcoming within a specified time**
  - **... withdraw its offer to communicate if no rendezvous is available immediately**
  - **... terminate if no clients can possibly call its entries**

**select**

- `select` accepts an entry (possibly protected).
- `accept` accepts an entry (possibly protected).
- `select` implements...
  - **... wait for more than a single rendezvous at any one time**
  - **... time-out if no rendezvous is forthcoming within a specified time**
  - **... withdraw its offer to communicate if no rendezvous is available immediately**
  - **... terminate if no clients can possibly call its entries**

**accept**

- `accept` accepts an entry (possibly protected).
- `accept` accepts an entry (possibly protected).
- `accept` implements...
  - **... wait for more than a single rendezvous at any one time**
  - **... time-out if no rendezvous is forthcoming within a specified time**
  - **... withdraw its offer to communicate if no rendezvous is available immediately**
  - **... terminate if no clients can possibly call its entries**
The real-world behavior of a system is often best described in terms of its events and the actions that follow them. In this section, we will explore the concepts of delay and selective synchronization in the context of real-time and embedded systems. These concepts are crucial for understanding how systems manage events and ensure timely responses.

### Delay

Delay is a mechanism used to pause the execution of a task for a specified amount of time. It is often used in scenarios where a task needs to wait for a certain amount of time before proceeding, such as in timer-based protocols. Delays can be explicit or implicit, and they are used to model situations where the system has to wait for a specific duration before proceeding.

### Selective Synchronization

Selective synchronization is a mechanism used to choose among multiple concurrent events or actions. It is a fundamental concept in real-time and embedded systems, as it allows the system to decide which event to respond to, based on the conditions or priorities of the events.

#### Basic Forms of Selective Synchronization

There are several basic forms of selective synchronization, including:

- **Guarded Select**
- **Delay**
- **Terminate**
- **非-guarded Select-or-else**

Each of these forms has its own characteristics and uses. For example, a guarded select allows the system to choose between multiple events based on certain conditions, while a delay simply pauses the system for a fixed amount of time.

#### Conditional & Timed Entry-calls

Conditional entry-calls and timed entry-calls are two important types of entry-calls that are used in the context of selective synchronization. Conditional entry-calls allow for the execution of an entry based on certain conditions, while timed entry-calls provide a way to schedule the execution of an entry for a specific time in the future.

### Non-determinism in Selective Synchronization

Non-determinism is a common issue in selective synchronizations. It arises when there are multiple possible actions that can be taken, and the system must choose one of them. This can lead to unpredictable behavior, especially in real-time systems where timing is critical.

To mitigate non-determinism, systems often employ techniques such as preemption or scheduling algorithms that ensure a consistent order of events. By carefully managing the selection of events, systems can provide predictable and reliable behavior.

### Conclusion

In summary, delay and selective synchronization are essential concepts in real-time and embedded systems. They enable systems to manage events effectively, ensuring timely responses and predictable behavior. Understanding these concepts is crucial for developing reliable and efficient systems.
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, selectee calls
  • Indeterminism in message based synchronization