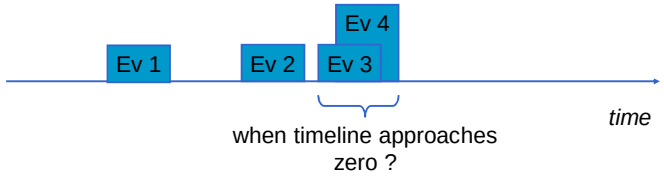
Low-level synchronisation

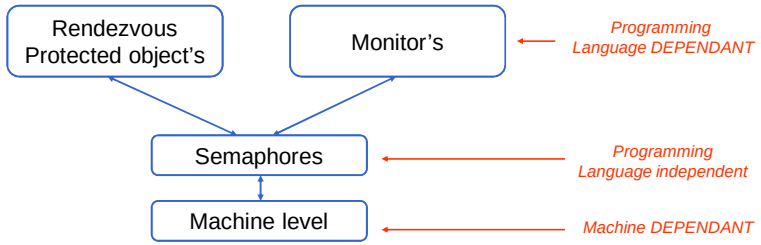
- What is synchronisation and how can we do this in a sequential execution environment?
- Monitors, Semaphores,..
- Demonstration of assignments 11,12 and 15
- Requeue Statement in Ada +
  - Train Lab Related Issues

What is synchronisation ?

- ”Synchronisation” – from Greece ”SYNCHRON”, i.e. ”simultaneously, at the same time”, ”agreed upon time”, ”agreed upon the sequential ordering of tasks”...
- An implicit agreement upon that events will be sequentially ordered (in absolute time) and this order is appreciated equivalent between all ”stakeholders”.



Synchronisation levels of abstraction



- Synchronisation (in sequentially computing systems) at high level can only be implemented if synchronisation at lower level is available...

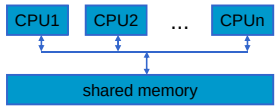
Synchronisation at machine level

```
procedure Do_Critical_Thing ( ... ) is
...
begin
        SetHighHardwareInterruptLevel;
        ...
        -- Do critical manipulations
        RestorePreviousHardwareInterruptLevel;
end Do_Critical_Thing;
```

- By raising CPU priority level to maximum, all other interrupts (including real-time clock) will be blocked out ...

Synchronisation at machine level, multiprocessor systems

- Raising HW interrupt priority will not work in a multiprocessor environment ...
- In this case we need:
  - A shared memory
  - The ”test and set” instruction




```
TESTANDSET semantics:
function TAS( lock_variable:  in out boolean) return boolean is
         previous: boolean;
         begin
            -- a single instruction ( uninterruptible )
            previous := lock_variable;
            lock_variable:= true;
                  return previous;
          end TAS;
```

See Jan’s Lecture 7

Monitors

- Monitors has several similarities with Ada95 ”protected objects”. Monitors are not supported by Ada95 however they are of general interest in parallel programming.
- A monitor guarantees mutual exclusion. A monitor can export procedures and functions but not entries. Entries can be simulated using so called “condition variables”.
- There are three possible operations on condition variables.
  - **`Wait(Condition_variable)`** – calling task is queued (FIFO) on “Condition_variable”.
  - **`Signal(Condition_variable)`** – first task in corresponding queue becomes eligible for execution (is ”waked up”). A signal operation on an empty queue has no effect.
  - **`Non_Empty(Condition_variable)`** – this function returns Boolean TRUE if there is at least one task in the corresponding queue, returns FALSE otherwise.

Assignment 11

Assume that the “monitor” construct is available in Ada95. Implement a notice board using a monitor `Board`.

There is only room for a single message at the same time on this board. The monitor exports two procedures, `Putmsg` to place a message on the board and `Getmsg` to read the current message.

When a message is submitted to the board through `Putmsg`, any previously message is destroyed.

Client tasks can read messages from a non-empty board (i.e. the same message may be read multiple times). If no message is available, then it cannot be read.

`Putmsg` and `Getmsg` should be accepted in arbitrary order but only one of them, at the same time.

Monitors, example, assignment 11

```
monitor Board is
   procedure Putmsg(Msg : in Msgtype);
   procedure Getmsg(Msg : out Msgtype);
   Contents : Msgtype;
   Empty : Boolean := True;
   cond_contents : Condition_Variable;
   waiting: integer:=0;
end Board;
```

- Procedures in the monitor are mutual exclusive (just as with ”protected objects”).
- Variables are ”private”, i.e. only visible from procedures.
- Condition_Variable is actually a queue.

Monitors, example, assignment 11

```
procedure Putmsg(Msg : in Msgtype) is
begin
   Contents := Msg;
   Empty := False;
   if waiting > 0 then
     waiting := waiting - 1;
     Signal(Cond_contents);
   end if;
end Putmsg;
```

```
procedure Getmsg(Msg : out Msgtype) is
begin
   if not Empty then
      Msg := Contents;
   else
      waiting := waiting + 1;
      Wait(Cond_contents);
      Msg := Contents;
      if waiting > 0 then
        waiting := waiting - 1;
        Signal(Cond_contents);
      end if;
   end if;
end Getmsg;
```

- Monitor's are easy to use, but its language dependent...

Semaphores

- A semaphore **s is an integer variable with value domain ≥ 0**
- Atomic operations on semaphores:

Init(s,n): assign s an initial value n

Wait(s): if s > 0 then
s := s - 1;
else
"block calling task";

Signal(s): if "any task that has called Wait(s) is blocked" then
"allow one such task to execute";
else
s := s + 1;

- If semaphore "S" already is in use, then the calling task is moved to a queue ("waiting on S") otherwise the task will continue execution.
- The "signal" primitive, releases semaphore "S", and thus moves any task in waiting queue(S) to the "Ready" state...

```
...
wait( S );        -- request single access
-- Do critical region
signal( S );      -- release semaphore
...
```


Semaphores, implementation "busy-wait"

```
procedure wait(semaphore_id : integer ) is
begin
  loop
    SetHighHardwareInterruptLevel;      -- lock out interrupts
    if semaphore_id.count > 0
      semaphore_id.count := semaphore_id.count - 1;
      RestorePreviousHardwareInterruptLevel; -- allow...return
    else
      RestorePreviousHardwareInterruptLevel; -- continue loop...
    end if;
  end loop;
end;

procedure signal( semaphore_id : integer ) is
begin
  SetHighHardwareInterruptLevel;        -- lock out interrupts
  semaphore_id.count := semaphore_id.count + 1;
  RestorePreviousHardwareInterruptLevel; -- allow...
end;
```

- Serious drawback: causes a single task to use all of it's time slice even if this use means "busy waiting"
- Remedy: Allow task to abandon cpu when hitting a blocked semaphore

Assignment 12

As with monitors, semaphores are of general interest in parallel programming. Semaphores, unlike monitors can be implemented regardless of programming language. Assume we have an Ada implementation of semaphores:

```
package semaphores is
   type Semaphore is ...—- useful type...
   procedure wait(Semaphore: in s);
   procedure signal(Semaphore: in s);
end semaphores;
```

Implement a package `Board` using semaphores.
There is only room for a single message at the same time on this board. The package exports two procedures, `Putmsg` to place a message on the board and `Getmsg` to read the current message.
When a message is submitted to the board through `Putmsg`, any previously message is destroyed.
Client tasks can read messages from a non-empty board. I.e. the same message may be read multiple times. If no message is available, then it cannot be read.
`Putmsg` and `Getmsg` should be accepted in arbitrary order but only one of them, at the same time.

Solution

```
package body Board is
   Contents : Msgtype;
   Empty : Boolean := True;
   sem_board, Sem_contents: Semaphore;
   waiting: integer:=0;
begin

   procedure Putmsg(Msg : in Msgtype) is
   begin
    -- code ...
   end Putmsg;

   procedure Getmsg(Msg : out Msgtype) is
   begin
    --code
   end Getmsg;

end Board;
```


Solution

```
procedure Putmsg(Msg : in Msgtype) is
begin
   wait(sem_board);
   Contents := Msg;
   Empty := False;
   while waiting > 0 loop
      signal(Sem_contents);
      waiting := waiting - 1;
   end loop;
   signal(sem_board);
end Putmsg;
```

```
procedure Getmsg(Msg : out Msgtype is
begin
   wait(sem_board);
   if not Empty then
      Msg := Contents;
   else
      waiting := waiting + 1;
      signal(sem_board);
      wait(Sem_contents);
      wait(sem_board);
      Msg := Contents;
   end if;
   signal(sem_board);
end Getmsg;
```

- Conclusions; Semaphores are easy to implement but difficult to use.

Assignment 15: Consider a real-time system with seven tasks (P1-P7) and two semaphores (S1 and S2). The tasks make use of semaphores due to the following .

1	2	3	4	5	6	7
..	..	..	..	..	..	..
wait(S1)	signal(S2)	signal(S1)	wait(S1)	signal(S2)	signal(S1)	signal(S2)
..	..	..	..	..	..	..
	wait(S1)	wait(S2)		signal(S1)	wait(S2)	wait(S1)
	..	..		..		
		wait(S1)		wait(S1)		

At a certain moment, just before the above scenario, the tasks are in the following states

P1	P2	P3	P4	P5	P6	P7
RUNNING	RUNNABLE	RUNNABLE	RUNNABLE	WAITING ON S1	WAITING ON S1	WAITING ON S2

Show how the task's states are changed (RUNNING, RUNNABLE and WAITING ON Sx) from that moment and on

Task 1 (Running) executes 'wait(S1)'

1	2	3	4	5	6	7
..	..	..	..	..	..	..
wait(S1)	signal(S2)	signal(S1)	wait(S1)	signal(S2)	signal(S1)	signal(S2)
..	..	..	..	..	..	..
	wait(S1)	wait(S2)		signal(S1)	wait(S2)	wait(S1)
	..	..		..		
		wait(S1)		wait(S1)		

```
Running:      1
Ready:        2,3,4

S1:           5,6
S1.Val:       '0'

S2:           7
S2.Val:       '0'
```

```
Running:      -
Ready:  2,3,4

S1:           5,6,1
S1.Val: 0

S2:           7
S2.Val: 0
```

```
Running:      2
Ready:        3,4

S1:           5,6,1
S1.Val:       '0'

S2:           7
S2.Val:       '0'
```

1	2

Task 2 (Running) executes 'signal(S2)'

1	2	3	4	5	6	7
..	..	..	..	..	..	..
wait(S1)	**signal(S2)**	signal(S1)	wait(S1)	signal(S2)	signal(S1)	signal(S2)
..	..	..	..	..	..	..
	wait(S1)	wait(S2)		signal(S1)	wait(S2)	wait(S1)
	..	..		..		
		wait(S1)		wait(S1)		

Running:	1
Ready:	2,3,4
S1:	5,6
S1.Val:	'0'
S2:	7
S2.Val:	'0'

Running:	2
Ready:	3,4,**7**
S1:	5,6,1
S1.Val:	'0'
S2:	{}
S2.Val:	'0'

1	2

Task 2 (Running) executes 'wait(S1)'

1	2	3	4	5	6	7
..	..	..	..	..	..	..
wait(S1)	signal(S2)	signal(S1)	wait(S1)	signal(S2)	signal(S1)	signal(S2)
..	..	..	..	..	..	..
	wait(S1)	wait(S2)		signal(S1)	wait(S2)	wait(S1)
	..	..		..		
		wait(S1)		wait(S1)		

Running:	2
Ready:	3,4,7
S1:	5,6,1
S1.Val:	'0'
S2:	{}
S2.Val:	'0'

Running:	-
Ready:	3,4,7
S1:	5,6,1,**2**
S1.Val:	'0'
S2:	{}
S2.Val:	'0'

Running:	**3**
Ready:	4,7
S1:	5,6,1,2
S1.Val:	'0'
S2:	{}
S2.Val:	'0'

1	2	3

Task 3 (Running) executes 'signal(S1)'

1	2	3	4	5	6	7
..	..	..	..	..	..	..
wait(S1)	signal(S2)	**signal(S1)**	wait(S1)	signal(S2)	signal(S1)	signal(S2)
..	..	..	..	..	..	..
	wait(S1)	wait(S2)		signal(S1)	wait(S2)	wait(S1)
	..	..		..		
		wait(S1)		wait(S1)		

Running:	3
Ready:	4,7
S1:	5,6,1,2
S1.Val:	'0'
S2:	{}
S2.Val:	'0'

Running:	3
Ready:	4,7,**5**
S1:	6,1,2
S1.Val:	'0'
S2:	{}
S2.Val:	'0'

1	2	3

Task 3 (Running) executes ' wait(S2)'

1	2	3	4	5	6	7
..	..	..	..	..	..	..
wait(S1)	signal(S2)	signal(S1)	wait(S1)	signal(S2)	signal(S1)	signal(S2)
..	..	..	..	..	..	..
	wait(S1)	**wait(S2)**		signal(S1)	wait(S2)	wait(S1)
	..	..		..		
		wait(S1)		wait(S1)		

Running:	3
Ready:	4,7,5
S1:	6,1,2
S1.Val:	'0'
S2:	{}
S2.Val:	'0'

Running:	-
Ready:	4,7,5
S1:	6,1,2
S1.Val:	'0'
S2:	**3**
S2.Val:	'0'

Running:	**4**
Ready:	7,5
S1:	6,1,2
S1.Val:	'0'
S2:	3
S2.Val:	'0'

1	2	3	4

Task 4 (Running) executes ' wait(S1)'

1	2	3	4	5	6	7
..	..	..	..	..	..	..
wait(S1)	signal(S2)	signal(S1)	**wait(S1)**	signal(S2)	signal(S1)	signal(S2)
..	..	..	..	..	..	..
	wait(S1)	wait(S2)		signal(S1)	wait(S2)	wait(S1)
	..	..		..		
		wait(S1)		wait(S1)		

```
Running:    4          Running:    -          Running:    7
Ready:      7,5        Ready:      7,5        Ready:      5

S1:         6,1,2      S1:         6,1,2,4    S1:         6,1,2,4
S1.Val:     '0'        S1.Val:     '0'        S1.Val:     '0'

S2:         3          S2:         3          S2:         3
S2.Val:     '0'        S2.Val:     '0'        S2.Val:     '0'
```

1	2	3	4	7

Task 7 (Running) executes ' signal(S2)'

1	2	3	4	5	6	7
..	..	..	..	..	..	..
wait(S1)	signal(S2)	signal(S1)	wait(S1)	signal(S2)	signal(S1)	**signal(S2)**
..	..	..	..	..	..	..
	wait(S1)	wait(S2)		signal(S1)	wait(S2)	wait(S1)
	..	..		..		
		wait(S1)		wait(S1)		

```
Running:    7          Running:    7
Ready:      5          Ready:      5,3

S1:         6,1,2,4    S1:         6,1,2,4
S1.Val:     '0'        S1.Val:     '0'

S2:         3          S2:         {}
S2.Val:     '0'        S2.Val:     '0'
```

1	2	3	4	7

Task 7 (Running) executes ' wait(S1)'

1	2	3	4	5	6	7
..	..	..	..	..	..	..
wait(S1)	signal(S2)	signal(S1)	wait(S1)	signal(S2)	signal(S1)	signal(S2)
..	..	..	..	..	..	**wait(S1)**
	wait(S1)	wait(S2)		signal(S1)	wait(S2)	
	..	..		..		
		wait(S1)		wait(S1)		

```
Running:    7          Running:    -          Running:    5
Ready:      5,3        Ready:      5,3        Ready:      3

S1:         6,1,2,4    S1:         6,1,2,4,7  S1:         6,1,2,4,7
S1.Val:     '0'        S1.Val:     '0'        S1.Val:     '0'

S2:         {}         S2:         {}         S2:         {}
S2.Val:     '0'        S2.Val:     '0'        S2.Val:     '0'
```

1	2	3	4	7	5

Task 5 (Running) executes ' signal(S2)'

1	2	3	4	5	6	7
..	..	..	..	..	..	..
wait(S1)	signal(S2)	signal(S1)	wait(S1)	**signal(S2)**	signal(S1)	signal(S2)
..	..	..	..	..	..	..
	wait(S1)	wait(S2)		signal(S1)	wait(S2)	wait(S1)
	..	..		..		
		wait(S1)		wait(S1)		

```
Running:    5          Running:    5
Ready:      3          Ready:      3

S1:         6,1,2,4,7  S1:         6,1,2,4,7
S1.Val:     '0'        S1.Val:     '0'

S2:         {}         S2:         {}
S2.Val:     '0'        S2.Val:     '1'
```

1	2	3	4	7	5

Task 5 (Running) executes ' signal(S1)'

1	2	3	4	5	6	7
..	..	..	..	..	..	..
wait(S1)	signal(S2)	signal(S1)	wait(S1)	signal(S2)	signal(S1)	signal(S2)
..	..	..	..	..	..	..
	wait(S1)	wait(S2)		signal(S1)	wait(S2)	wait(S1)
	..	..		..		
		wait(S1)		wait(S1)		

Running:	5
Ready:	3
S1:	6,1,2,4,7
S1.Val:	'0'
S2:	{}
S2.Val:	'1'

Running:	5
Ready:	3,**6**
S1:	1,2,4,7
S1.Val:	'0'
S2:	{}
S2.Val:	'1'

1	2	3	4	7	5

Task 5 (Running) executes ' wait(S1)'

1	2	3	4	5	6	7
..	..	..	..	..	..	..
wait(S1)	signal(S2)	signal(S1)	wait(S1)	signal(S2)	signal(S1)	signal(S2)
..	..	..	..	..	..	..
	wait(S1)	wait(S2)		signal(S1)	wait(S2)	wait(S1)
	..	..		..		
		wait(S1)		wait(S1)		

Running:	5
Ready:	3,6
S1:	1,2,4,7
S1.Val:	'0'
S2:	{}
S2.Val:	'1'

Running:	-
Ready:	3,6
S1:	1,2,4,7,**5**
S1.Val:	'0'
S2:	{}
S2.Val:	'1'

Running:	**3**
Ready:	6
S1:	1,2,4,7,5
S1.Val:	'0'
S2:	{}
S2.Val:	'1'

1	2	3	4	7	5	3

Task 3 (Running) executes ' wait(S1)'

1	2	3	4	5	6	7
..	..	..	..	..	..	..
wait(S1)	signal(S2)	signal(S1)	wait(S1)	signal(S2)	signal(S1)	signal(S2)
..	..	..	..	..	..	..
	wait(S1)	wait(S2)		signal(S1)	wait(S2)	wait(S1)
	..	..		..		
		wait(S1)		wait(S1)		

Running:	3
Ready:	6
S1:	1,2,4,7,5
S1.Val:	'0'
S2:	{}
S2.Val:	'1'

Running:	**{}**
Ready:	6
S1:	1,2,4,7,5,**3**
S1.Val:	'0'
S2:	{}
S2.Val:	'1'

Running:	**6**
Ready:	{}
S1:	1,2,4,7,5,3
S1.Val:	'0'
S2:	{}
S2.Val:	'1'

1	2	3	4	7	5	3	6

Task 6 (Running) executes ' signal(S1)'

1	2	3	4	5	6	7
..	..	..	..	..	..	..
wait(S1)	signal(S2)	signal(S1)	wait(S1)	signal(S2)	**signal(S1)**	signal(S2)
..	..	..	..	..	..	..
	wait(S1)	wait(S2)		signal(S1)	wait(S2)	wait(S1)
	..	..		..		
		wait(S1)		wait(S1)		

Running:	6
Ready:	{}
S1:	1,2,4,7,5,3
S1.Val:	'0'
S2:	{}
S2.Val:	'1'

Running:	6
Ready:	**1**
S1:	2,4,7,5,3
S1.Val:	'0'
S2:	{}
S2.Val:	'1'

1	2	3	4	7	5	3	6

Task 6 (Running) executes ' wait(S2)' ..and then to termination…

1	2	3	4	5	6	7
.. wait(S1) ..	.. signal(S2) .. wait(S1) ..	.. signal(S1) .. wait(S2) .. wait(S1)	.. wait(S1) ..	.. signal(S2) .. signal(S1) .. wait(S1)	.. signal(S1) .. wait(S2)	.. signal(S2) .. wait(S1)

```
Running:        6
Ready:          1

S1:     2,4,7,5,3
S1.Val:         '0'

S2:             {}
S2.Val:         '1'
```

```
Running:        6
Ready:          1

S1:     2,4,7,5,3
S1.Val:         '0'

S2:             {}
S2.Val:         '0'
```

```
Running:        {}
Ready:          1

S1:     2,4,7,5,3
S1.Val:         '0'

S2:             {}
S2.Val:         '0'
```

1	2	3	4	7	5	3	6

Task 1, only runnable task.. executes to termination…

1	2	3	4	5	6	7
.. wait(S1) ..	.. signal(S2) .. wait(S1) ..	.. signal(S1) .. wait(S2) .. wait(S1)	.. wait(S1) ..	.. signal(S2) .. signal(S1) .. wait(S1)	.. signal(S1) .. wait(S2)	.. signal(S2) .. wait(S1)

```
Running:        {}
Ready:          1

S1:     2,4,7,5,3
S1.Val:         '0'

S2:             {}
S2.Val:         '0'
```

```
Running:        1
Ready:          {}

S1:     2,4,7,5,3
S1.Val:         '0'

S2:             {}
S2.Val:         '0'
```

No runnable task, but the semaphore queue is nonempty...

1	2	3	4	7	5	3	6	1

Task 1, only runnable task.. executes to termination…

1	2	3	4	5	6	7
.. wait(S1) ..	.. signal(S2) .. wait(S1) ..	.. signal(S1) .. wait(S2) .. wait(S1)	.. wait(S1) ..	.. signal(S2) .. signal(S1) .. wait(S1)	.. signal(S1) .. wait(S2)	.. signal(S2) .. wait(S1)

```
Running:        {}
Ready:          1

S1:     2,4,7,5,3
S1.Val:         '0'

S2:             {}
S2.Val:         '0'
```

```
Running:        1
Ready:          {}

S1:     2,4,7,5,3
S1.Val:         '0'

S2:             {}
S2.Val:         '0'
```

No runnable task, but the semaphore queue is nonempty...

System is locked…. (ANSWER)

1	2	3	4	7	5	3	6	1

Requeue

The requeue statement is primarily designed to handle two situations:

- After an accept statement or entry body begins execution, it may be determined that the request cannot he satisfied immediately, instead it is necessary to requeue the caller until such a time arises that it can be handled.
- Part of a request may be handled immediately, but there are additional steps in the process that need to be performed at a later point.

These scenarios arise e.g. when a task is queued for a particular event (Think about *waitsens(sensor_nr_trype, yes_no_type)* procedure in "command.adb").

They also arise in resource allocation problems, e.g. when a client must gather a number of resources before it is able to proceed.

Requeue

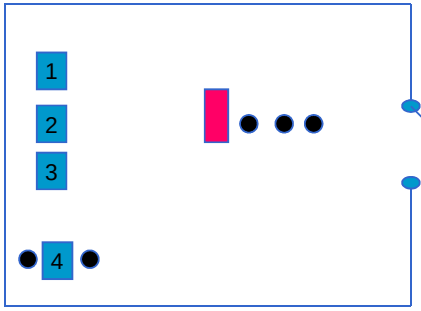
The syntax of requeue is defined as

```
requeue_statement ::= requeue entry_name
```

The entry name (called the target entry) may be:

- another entry within the same protected object, in which case the entry name normally consists only of the identifier naming that entry.
- an entry name within another protected object, possibly of a different protected type.
- an entry within a task object.
- Think about the parameter list for an entry when calling…

Example: Waiting for a dinner table




```
procedure tabletest is
begin
  tables.GET_TABLE(2);
  --   use TABLE
  tables.RELEASE_TABLE(2);
exception
  when table_error ⇒
     put("main: table error");
  when others ⇒
     put("main: exception");
end tabletest;
```


Example: Waiting for a dinner table

```
procedure GET_TABLE(nr : integer) is
  begin
    if (nr ≥ 1) and (nr ≤ N)  then
      table_handler.req(nr);
    else
      put_line("Table_handler:” & “requested nonexistent table");
      raise table_error;
    end if;
  end GET_TABLE;

procedure RELEASE_TABLE(nr : integer) is
  begin
    if (nr ≥ 1) and (nr ≤ N)  then
      table_handler.rel(nr);
    else
      put_line ("Table_handler:” &  “released nonexistent table");
      raise table_error;
    end if;
  end RELEASE_TABLE;
```


Example: Waiting for a dinner table

```
package tables is
        table_error : exception;
        procedure GET_TABLE(nr : integer);
        procedure RELEASE_TABLE(nr : integer);
end tables;

package body  tables is
N : constant INTEGER := 2;
type INDEX is range 1..N;
type free_array is array (INDEX) of boolean;

protected table_handler is
        entry req(nr : integer);
        procedure rel(nr : integer);
private
        free : free_array := (others ⇒ true);
        waiters : integer := 0;
        entry again(nr : integer);
end table_handler;
```

Example: Waiting for a dinner table

```
entry req(nr : integer) when true is
  begin
    if not free(nr) then
      requeue again;
    end if;
    free(nr) := FALSE;
  end;

entry again(nr : integer) when waiters > 0 is
  begin
    waiters := waiters - 1;
    if not free(nr) then
      requeue again;
    end if;
    free(nr) := FALSE;
  end again;
```


Example: Waiting for a dinner table

```
procedure rel(nr : integer) is

begin
   if free(nr) then
        put_line("Releasing free table");
        raise table_error;
   else
        free(nr) := TRUE;
        waiters := again'count;
   end if;
end rel;
```


Tips Trains Lab

- You have a procedure waitsens(nr:sensor_number_type, yn: yes_no_type) in command.adb.
  - WAITS FOR A SENSOR TO BE SET AS YES/NO
  - Note it is not a blocking procedure. But you need blocking effect. Why?
  - What you have to do?
- Implement the my_waitsens(nr,yn) and again(nr,yn) entry similar to the example (dinner table) in an protected object.
  - Do we need a new protected object in *command.adb* file?
- Call *interrurpt.my_waitsens(nr,yn)* where *interruprt* is the protected object name.

Summary

Synchronisation constructs: Monitors, semaphores
Requeue Statement
Assignment 11, 12 and 15