Real Time Programming with Ada (1)

Real time programming

- It is mostly about "Concurrent programming"
- We also need to handle Timing Constraints on concurrent executions of tasks

However, remember:

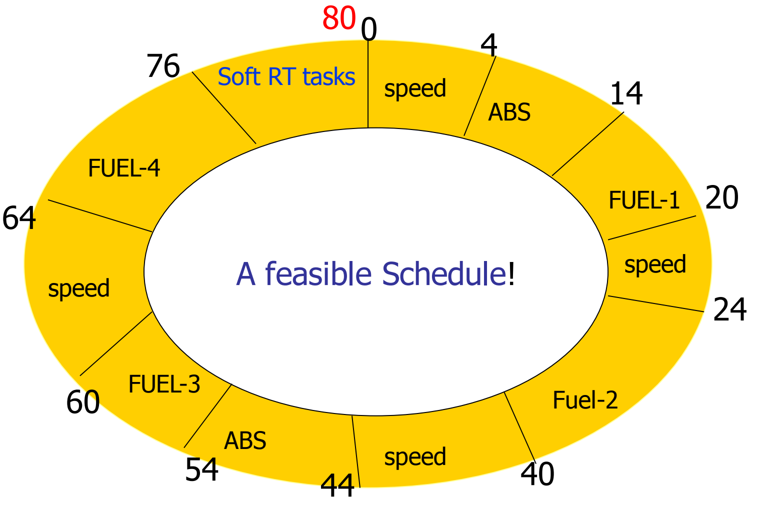
- "concurrency" is a way of structuring computer programs
  e.g. three "concurrent modules": task 1, task 2 task 3
- "concurrency" is often implemented by "fast sequential computation" using a scheduler

Programming the car controller

Process Speed: Loop next := get-time + 0.02 read sensor,compute,display... sleep until next End loop	Process ABS Loop next:=get-time + 0.04 Read sensor, compute, react sleep until next End loop
Process Fuel Loop next:=get-time + 0.08 read data, compute, inject ... sleep until next End loop	Soft RT Processes Loop read temperature elevator, stereo .... End loop

Qustion: do we need 4 CPUs to run these concurrently**?**

Programming the car controller (3)



This is the classic approach: cyclic execution

- Program your tasks in any sequential language

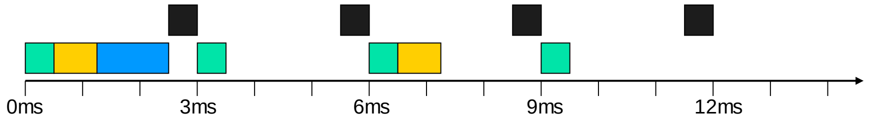
```
loop
  do task 1
  do task 2
  do task 3
end loop
```

Efficient code, deterministic, predictable,
But (1) difficult to make it right, (2) difficult to reuse existing design
(3) extremely difficult for constructing large systems

Cyclic Execution

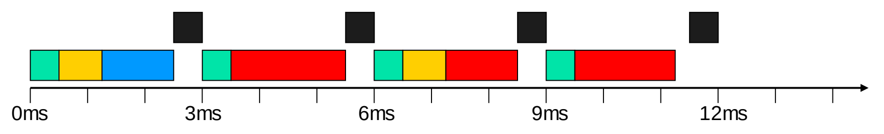
Task	Required sample rate	Processing time
t1	3ms (333Hz)	0.5ms
t2	6ms (166Hz)	0.75ms
t3	12ms (83Hz)	1.25ms

In addition, there is an interrupt handling task: I
with 0.5ms processing time every 3ms



Adding new functionality … …

- add task t4 with 12ms rate and 5ms processing time
- 12ms cycle has 5.25ms free time...
- ... t4 has to be artificially partitioned



Effect of new task at code level

```
void do_task_t4(void)
{
        /* Task functionality */
}
```

```
void do_task_t4_1(void)
{
        /* first bit */
        state_var_1 = x;
        state_var_2 = y;
        ...
}
void do_task_t4_2(void)
{
        x = state_var_1;
        ...
        /* second bit */
        state_var_3 = a;
        state_var_4 = b;
        ...
}
void do_task_t4_3(void)
{
        c = state_var_4;
        ...
        /* third bit */
}
```

```
int state_var_1;
int state_var_2;
int state_var_3;
int state_var_4;
void main(void)
{
        do_init();
        while(1) {

                do_task_t1();
                do_task_t2();
                do_task_t3();
                busy_wait();
                do_task_t1();        /* 3ms */
                do_task_t4_1();
                busy_wait();
                do_task_t1();        /* 6ms */
                do_task_t2();
                do_task_t4_2();
                busy_wait();

                do_task_t1();        /* 9ms */
                do_task_t4_3();
                busy_wait();
        }
}
```

This is "ad hoc", but it is often used in industry

- You just don't want to do this for large software systems, say a few hundreds of control tasks
- This was why "Multitasking" came into the picture

9

Concurrent programming with multitasking:

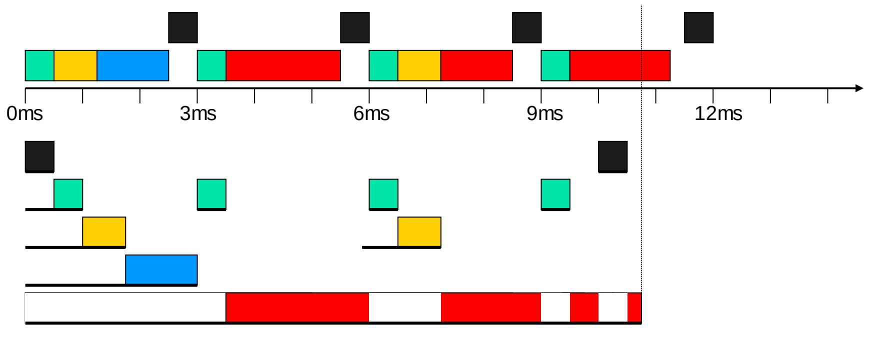
- Program your computation tasks, execute them concurrently with OS support e.g. in LegOS (or in Ada in slightly different syntax)

```
execi(foo1, ..., priority1, ...);
execi(foo2, ..., priority2, ...);
execi(foo3, ..., priority3, ...);
```

Will start three concurrent tasks running foo1, foo2, foo3

10

Cyclic Execution vs. Multitasking



Ada95

- It is strongly typed OO language, looks like Pascal
- Originally designed by the US DoD as a language for large safety critical systems i.e. Military systems
  - Ada83
  - Ada95 + RT annex + Distributed Systems Annex
  - Ada 2005 (allows scheduling policies e.g. RR, EDF, dynamic priorities for protected types …)

The basic structures in Ada

- A large part in common with other languages
  - Procedures
  - Functions
  - Basic types: integers, characters, ...
  - Control statements: if, for, ..., case analysis
- Any thing new?
  - Abstract data type: Packages (objects)
    - Protected data type
  - Tasking: concurrency
  - Task communication/synchronization: rendezvous
  - Real Time

13

Typical structure of programs

Program Foo(...)

Declaration 1 ←----- to introduce identities/variables
and define data structures

Declaration 2 ←----- to define "operations" : procedures, functions
and/or tasks (concurrent operations)
to manopulate the data structures

Main program
(Program body) ←------ a sequence of statements or "operations" to
compute the result (output)

14

Declarations and statements

- Before each block, you have to declare (define) the variables used, just like any sequential program

```
procedure PM (A : in  INTEGER;
               B:  in out  INTEGER;
                 C : out     INTEGER) is
begin
    B := B+A;
    C := B + A;
end PM;
```

15

If, case, for: contrl-statements

```
if TEMP < 15 then
    some smart code;
else
 do something else..;
end if;

case TAL is
    when <2 =>
          PUT_LINE(”one or two”);
    when >4 =>
          PUT_LINE(”greater than 4);
end case;

for I in 1..12 loop
    PUT(”in the loop”);
end loop;
```

16

Types (like in Pascal or any other fancy languages)

```
type LINE_NUMBER is range 1 .. 72
type WEEKDAY is (Monday, Tuesday, Wednesday);
type serie is array (1..10) of FLOAT;

type CAR is
   record
        REG_NUMBER   : STRING(1 .. 6);
        MODEL        : STRING(1 .. 20);
   end record;
```

17

Concurrent and Real-Time Programming with Ada

- Abstract data types
  - package
  - protected data type
- Concurrency
  - Task creation
  - Task execution
- Communication/synchronization
  - Rendezvous
- Real time:
  - Delay ”time period” and Delay until ”next-time point”
  - Real-time scheduling/”Fixed-priority scheduling”

18

Package --- Class/Object

”Package”: abstract data type in Ada

- package definition ---- specification
- Package body ---- implementation

Package definition -- Specificaiton

- Objects declared in specification is visible externally.

```
package MY_PACKAGE is
-- declare/define data structures
    Type myobject is record
        Name: string
        Personalnr: integer
    End myobject
-- declare/define all public operations
    procedure myfirst_operation;
    procedure mysecond_operation;
    function mythird_operation (name: string) return myobject;
end MY_PACKAGE;
```


Package body -- Implementation

- Implements package specification

```
(you probably want to use some other packages here e.g.. )
with TEXT_IO;
use TEXT_IO;

package body MY_PACKAGE is
        procedure myfirst_operation is
        begin
                myfirst_operation code here;
        end;
        function MAX (X,Y :INTEGER) return INTEGER is
        begin
          … …
        end;
        procedure mysecond_operation is
        begin
                PUT_LINE("Hello Im Ada Who are U");
                GET();
        end;
        function mythird_operation (name: string) return myobject is
                                begin
                              … …
                                end;

end MY_PACKAGE;
```

Protected data type

```
protected Buffer is
    procedure read(x: out integer)
    procedure write(x: in integer)
    private
      v: integer := 0  /* initial value */

protected body Buffer is
    procedure read(x: out integer) is
      begin x:=v end
    procedure write(x: in integer) is
      begin  v:= x end
```

(note that you can solve similar problems with semaphores)

Tasking

Ada tasking: concurrent programming

- Ada provides at the language level light-weight tasks. These often refered to as threads in some other languages. The basic form is:

```
task T is                    <--------- specification
--- operations/entry (or simply: task T)
end T;

task body T is               <---------- implementation/body
begin
---- processing----
end T;
```


Example: the sequential case

```
procedure shopping is
begin
buy-meat;
buy-salad;
buy-wine;
end
```

Assume pre-defined procedures:
buy-meat
buy-salad
buy-wine

The concurrent version

```
procedure shopping is

task get-salad;
task body get-salad is
begin
buy-salad;
end  get-salad;

task get-wine;
task body get-wine is
begin
buy-wine;
end  get-wine;
begin
buy-meat;
end
```

buy-salad and buy-wine
will be activated concurrently
here

And then run in parallel with
buy-meat

Creating Tasks

- Tasks may be declared at any program level
- Created implicitly upon entry to the scope of their declaration.
- Possible to declare task types to start several task instances of the same task type

example

```
procedure Example1 is
    task type A_Type;
    task B;
    A,C : A_Type;

    task body A_Type is
            --local declarations for task A and C
    begin
            --sequence of statements for task A and C
    end A_Type;

    task body B is
            --local declarations for task B
    begin
            --sequence of statements for task B
    end B;

begin
    --task A,C and B start their executions before the first statement of this procedure.
end Example1;
```


Task scheduling

- Allow priorities to be assigned to tasks in task definition
- Allow task dispatching policy to be set (Default: highest priority first)

```
task Controller is
    pragma Priority(10)
end Controller
```

Task termination: a task will terminate if:

- It completes execution of its body
- It executes a terminate alternative of a select statement
- It is aborted:
  - abort_statement ::= abort task_name {, task_name};

Communication/Synchronization

Task communication/synchronization

- Message passing using ”rendezvous”
  - entry and accept
- Shared variables
  - protected objects/variables

Rendezvous

```
procedure foo

     task T is
      entry E(...in/out parameter...);
     end;

     task body T is
          begin
               -------
                accept E(... ...) do
               ------- sequence of statements
          end E;
      end T;

      task user;
      task body user is
      begin
      ---
      T.E(... ...)
      ---
      end
begin
...
end
```

T and user will be started concurrently

Rendezvous

```
task body A is              task body B is
begin                       begin
...                         ...
B.Call;   <------------>    accept Call do
...                         ....
end A                       end Call
                            ...
                            end A
```


This is implemented with Entry queues (the compiler takes care of this!)

- Each entry has a queue for tasks waiting to be accepted
  - a call to the entry is inserted in the queue
  - the first task in the queue will be ”accepted” first (like the queue for a semaphore)
- By default, the queuing policy is FIFO
  - it can be different queuing policies

An Example: Buffer

```
task buffer is
entry put(X: in integer)
entry get(x: out integer)
end;

task body buffer is
  v: integer;
 begin
 loop accept put(x: in integer) do v:= x end put;
        accpet get(x: out integer) do x:= v end get;
 end loop;
 end buffer;
```

```
---
buffer.put(...)      ←-------------------------------------------- other tasks (users)!!
Buffer.get(...)
----
```

37

Potential deadlocks

- Task A: …. B.b; accept a …
- Task B: …. A.a; accept b …

38

An Example, the Semaphore

- The Idea of a (binary) semaphore
- Two operations, p and v
    - p grabs semaphore or waits if not available
    - v releases the semaphore

Program Semaphore using Task & RV. Synch.

The specification

- ```
  task type Semaphore is
          entry p;
          entry v;
  end Semaphore;
  ```

The implementation

- ```
  task body Semaphore is
  begin
          loop
                  accept p;
                  accept v;
          end loop;
  end Semaphore;
  ```

Program Semaphore using Task & RV

- The implementation:
  - task body Semaphore is

```
task body Semaphore is
begin
        loop
                accept p;
                accept v;
        end loop;
end Semaphore;
```

Using the Semaphore

- Declare an instance of a semaphore
  - Lock : Semaphore;
  - Now we can use Lock to protect critical sections

```
Lock.P;
        Code for Critical Section
Lock.V;
```

Choice: Select statement

```
task Server is
      entry S1(...);
      entry S2(...);
end Server;

task body Server is
      ...
begin
      loop
                --prepare for service
                select
                   when <boolean expression> =>
                        accept S1(...) do
                                --code for this service
                        end S1;
                or
                        accept S2(...) do
                                --code for this service
                        end S2;
                or
                       terminate;
                end select;
      --do any house keeping
      end loop;
end Server;
```

43

Real-Time Facilities (next lecture)

44