Major Characteristics of RTOS (Last lecture)

- Determinism & Reliability
- Responsiveness (quoted by vendors)
  - Fast process/thread switch
  - Fast interrupt response
- Support for concurrency and real-time
  - Multi-tasking
  - Real-time
  - synchronization
- User control over OS policies
  - Mainly scheduling, many priority levels
  - Memory support (especially embedded)

Today's topic:
Real Time Programming with Ada

Real time programming

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

---

Real time programming

- Wihout OS support (without "concurrency")
  - Program your tasks in any language
  - Static schedule and Cyclic Execution
- With OS/RTOS support (e.g. LegOS assignment)
  - Program your tasks in C (or any prog. language)
  - Fix the scheduling policy in RTOS e.g. priority assignment
- With RTOS and Language support
  - Program your tasks in a RT languages e.g. RT Java, Ada
  - RTOS is "hidden", a Run-Time kernel for each program

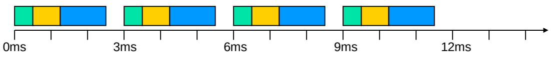
Cyclic Execution: the classic approach

the first example of real time programming without "concurrency"

Static cyclic scheduling: example

Task	Required sample rate	Processing time
t1	3ms (333Hz)	0.5ms
t2	6ms (166Hz)	0.75ms
t3	14ms (71Hz)	1.25ms

```
void main(void)
{
     do_init();
     while (1)
     {
          t1();
          t2();
          t3();
          delay_until_cycle_start();
     }
}
```



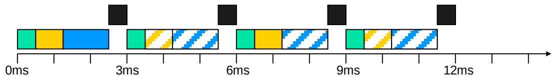
Cyclic scheduling: “overheads”

Task	Required sample rate	Processing time
t1	3ms (333Hz)	0.5ms
t2	6ms (166Hz)	0.75ms
t3	14ms (71Hz)	1.25ms

t2 requires 12.5% CPU (0.75/6), uses 25% (4*0.75/12)

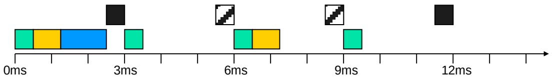
t3 requires 9% CPU (1.25/14), uses 42% (4*1.25/12)

add interrupt I with 0.5ms processing time



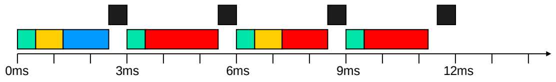
Major/minor cyclic scheduling

- 12ms major cycle containing 3ms minor cycles
  - t1 every 3ms, t2 every 6ms, t3 every 12ms
- t3 still upsampled (10.4% where 9% needed)
- time is still allocated for I every task in every cycle
  - will not always be used, but must be allowed for

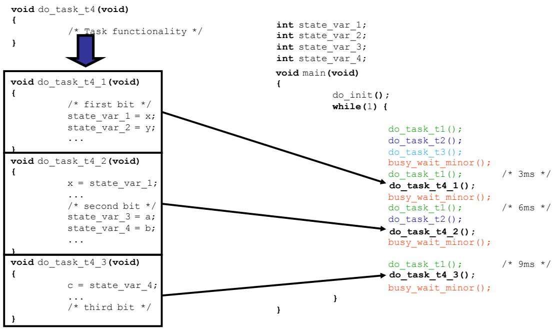


Fitting tasks to cycles

- add t4 with 14ms rate and 5ms processing time
- 12ms cycle has 5.25ms free time...
- ...but 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_minor();
                do_task_t1();          /* 3ms */
                do_task_t4_1();
                busy_wait_minor();
                do_task_t1();          /* 6ms */
                do_task_t2();
                do_task_t4_2();
                busy_wait_minor();

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

This is too ”ad hoc”, though this is often used in industry

- You just don’t want to do this for large software systems, say a few hundred of control tasks

Concurrent Programming

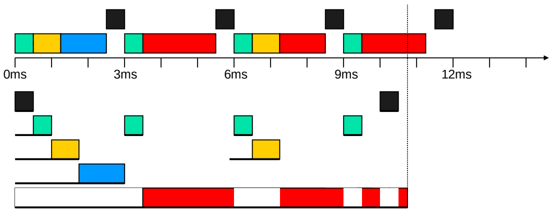
Concurrent programming:
using sequential programming languages

- Program your computation tasks, execute them concurrently with OS support e.g. in LegOS

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

Will start three concurrent tasks running foo1, foo2, foo3

Cyclic vs. Concurrent



Programming Languages for concurrent and real time programming

Let's look at Ada95

Note that there is no reason why you can't program a real time system using C. But there is no language support for concurrent tasks and real time features, so you would have to provide them yourself using e.g. exec(), sleep(20) etc, and most importantly, you would have to fix scheduling

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

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
- Abstract data type: Packages
  - Protected data type
- Tasking: concurrency
- Task communication/synchronization: rendezvous
- Real Time

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)

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;
```

If, for, case: 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;
```

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);
            TYPE            : STRING(1 .. 20);
      end record;
```

Anything new in Ada?

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(10) and Delay until next-time
  - Scheduling according to timing constraints

”Package”: abstract data type in Ada

- package definition ---- specification
- packagebody ---- implementation

Package definition

- Objects declared in specification is visible externally.

```
package MY_PACKAGE  is
    procedure myfirst_procedure;
    procedure mysecond_procedure;
end MY_PACKAGE;
```

Packagebody

- 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_procedure is
            begin
                        myfirst_procedure code here;
            end;

            function MAX (X,Y :INTEGER) return INTEGER  is
            begin
              ... ...
            end;

            procedure mysecond_procedure is
            begin
                        PUT_LINE("Hello  Im Ada Who are U");
                        GET();
            end;
end MY_PACKAGE;
```

Protected data type

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

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 nothing
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
```

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 in Ada will terminate if:
  - It completes execution of its body
  - It executes a terminate alternative of a select statement
  - It is aborted

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;

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

end foo;
```

T and user will be started concurrently

Rendezvous

```
task body A is
begin
...
B.Call;
...
end A
```

```
task body B is
begin
...
accept Call do
....
end Call
...
end A
```

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

- Each task has a queue
- A call to a task entry is inserted in the queue
- The queue is a simple FIFO without priority
- A task in an entry queue is inactive (waiting)
- The first task in the queue will be ”accepted” first (like the queue for a semaphore)

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(...)
----
```

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

A Semaphore using a Task, RV

- The specification
  - task type Semaphore is

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

A Semaphore using RV

- The body of semaphore is very simple:

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

Using the Semaphore Abstraction

- Declare an instance of a semaphore
  - Lock : Semaphore;
- Now we can use this semaphore to create a monitor, using
  - Lock.P;

    code to be protected in monitor

    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;
```