ECE 5780 / 6780
Real-Time Systems

Ada Basics

Slides adapted from various sources

Lab 1: To Do in Ada

- Cyclic scheduler
- Cyclic scheduler with watchdog
- Communication among tasks

Real-Time Programming

- It is mostly about concurrent programming
- We also need to handle timing constraints on concurrent executions of tasks
  - And other important performance metrics such as energy consumption

Real-Time Programming

- Without OS support (bare metal)
  - Static schedule and cyclic execution
- With RTOS support
  - Program in general-purpose language like C and use scheduling policy to assign priorities
- With RTOS and language support
  - Program in RT Java or Ada
  - RTOS is “hidden”

Why Ada?

- Developed by DoD for embedded and real-time systems
  - Back when hundreds of languages were used
  - Goal was to have a standardized language for DoD’s embedded real-time systems
  - $201M
- But Ada mandate ended in 1997
  - Could not be enforced
  - But did reduce the number of languages used in DoD computer systems by 2 orders of magnitude
- Many versions
  - Ada 83, Ada 95, Ada 2005, Ada 2012

Features

- Strong typing
- Modularity mechanisms (packages)
- Run-time checking
- **Exception handling**
- Generics
- Object-oriented
- **Concurrent programming**
- **Real-time support**

Application Area

- Safety-critical systems
  - Avionics
  - Military systems
  - Space applications

First Program: Hello World!

```
-- File: hello_world.adb
with Ada.Text_IO; -- Basic I/O
use Ada.Text_IO; -- Integrate namespace

procedure hello_world is
    Message : constant String := "Hello World";
begin
    Put_Line(Message);
end hello_world;
```

First Program: Hello World!

```
-- File: hello_world.adb
with Ada.Text_IO; -- Basic I/O
use Ada.Text_IO; -- Integrate namespace

procedure hello_world is
    Message : constant String := "Hello World";
begin
    Put_Line(Message);
end hello_world;
```

First Program: Hello World!

```
-- File: hello_world.adb
with Ada.Text_IO; -- Basic I/O
use Ada.Text_IO; -- Integrate namespace

procedure hello_world is
    Message : constant String := "Hello World";
begin
    Put_Line(Message);
end hello_world;
```

First Program: Hello World!

```
-- File: hello_world.adb
with Ada.Text_IO; -- Basic I/O
use Ada.Text_IO; -- Integrate namespace

procedure hello_world is
    Message : constant String := "Hello World";
begin
    Put_Line(Message);
end hello_world;
```

First Program: Hello World!

```
-- File: hello_world.adb
with Ada.Text_IO; -- Basic I/O
use Ada.Text_IO; -- Integrate namespace

procedure hello_world is
    Message : constant String := "Hello World";
begin
    Put_Line(Message); -- Case insensitive
end hello_world;
```

Typical Program Structure

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

Declaration2 – to define operations: procedures, functions, and/or tasks (concurrent operations) to manipulate the data structures

Main program

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

Basic Structures

- Basic data types: Boolean, Integer, Float, Character, String
  - X : Integer; -- Declare X as an integer
  - X : Integer := 0; -- Declare and initialize X
- New type via subtype, array, etc…
  - subtype Die is Integer range 1 .. 6; -- One die
  - type Dice is array (0 .. 22) of Die; -- 23 dice
  - To use
    - D : Dice;
    - Value := D(index);

Control Statements (1)

```
x := y; -- Assign y to x
if x = y then
    z := 1;
else
    z := 0;
    x := 42; -- Several statements!
end if;
```

Control Statements (2)

```
loop
    z := z + 1;
    exit when z > 100;
end loop; -- Loops may be nested

for i in 1…100 loop
    Put_Line(“Inside loop”);
end loop;
```

Procedures

```
procedure F1(var: in Integer; var: out String) is
    -- Local variables
    -- Definitions of local procedures/functions/tasks/..
begin
    -- Code
end F1;
```

- In/out parameters are read/write only
- Main program: procedure without argument

Example

```
procedure example1 is
    i: Integer := 0;

    procedure printVal(a: Integer) is
    begin
        if a > 0 then
            put_line(Integer'Image(a));
        end if;
    end printVal;
begin
    loop
        printVal(a => i);
        i := i + 1;
        exit when i > 10;
    end loop;
end example1;
```

Functions

```
function fname(x1, x2: Float) return Float is
    -- local variables
    -- definitions of local procedures/functions/tasks/..
begin
    return x1 * x2;
end fname;
```

Time

- Package: Ada.Calendar
  - Type for relative time: Duration
  - Type for absolute time: Time
  - For getting present time: Clock (of type Time)
- To get current time

```
t: Time;
t := Ada.Calendar.Clock;
```

- To get time as a Duration

```
t: Duration;
t := Ada.Calendar.Seconds(Ada.Calendar.Clock);
```

Delaying a Statement

- To wait x seconds, we could have

  ```
  delay x;
  ```

- But the above would cause drifts, especially in loop
- The correct way to do this is

  ```
  delay until y;
  ```

Generating a Random Number

- A float between [0, 1]

```
G: Generator;
Reset(G)
Val := Random(G);
```

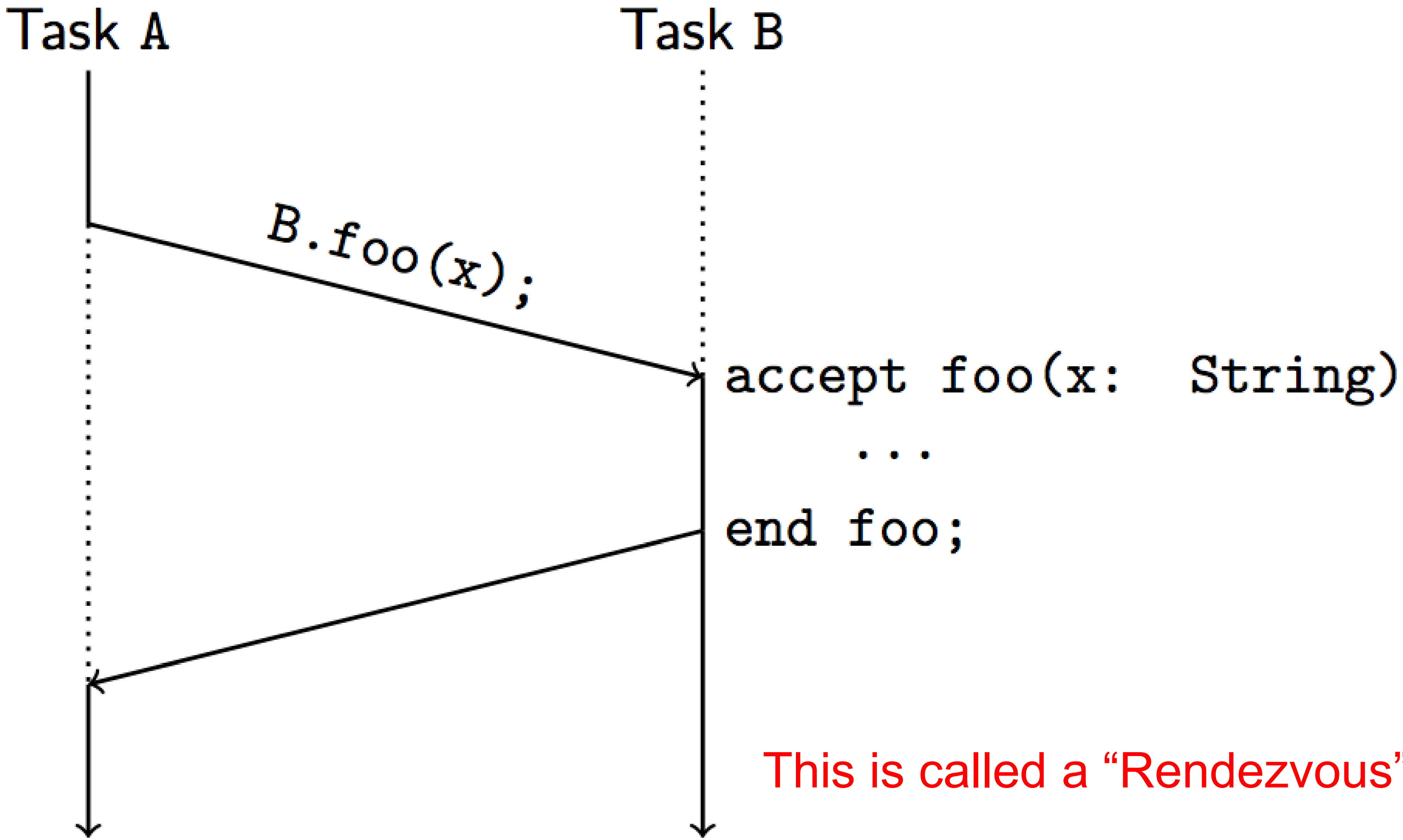
- An integer between [0, 10]

```
subtype Num_Gen is Integer range 0 .. 10;
package Random_Gen is new
    Ada.Numerics.Discrete_Random(Num_Gen);
use Random_Gen;
G: Random_Gen.Generator;
Reset(G)
Val: Num_Gen := random(G);
```

Creating a Task in Ada

```
task My_Task is
    entry Init;
    entry Get(Item: out Integer);
    entry Put(Item: in Integer);
end My_Task;
task body My_Task is
    -- Declarations needed by task code
begin
    -- Task code goes here
end My_Task;
```

Concurrent Exec. with Synch.



This is called a “Rendezvous”

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
    taskT: T;
    taskU: user;
end
end foo;
```

taskT and taskU 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 B

Select Statement

```
task body A is
    empty: Boolean := false;
begin
    loop
        select
            accept Init do
                …
            end Init;
        or
            accept Put(Item: Integer) do
                …
            end Put;
    end loop;
end A;
```

```
Procedure x is
  task type A is
    entry Init;
    entry Put(Item: Integer);
    …
  end A;

  task body A is
    …
  end A

TaskA: A;

begin
  …
  TaskA.init;
  …
  TaskA.put(y);
  ….
end x;
```

Guarded Entries: When Statement

```
select
    when not Is_Empty(Container) =>
        accept get(Item: out Integer) do
            …
        end get;
or
    when not Is_Full(Container) =>
        accept put(Item: in Integer);
            …
        end put;
end select;
```

Choice: More on 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;
        end loop;
    end server;
```

Rendezvous Implementation

- 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
  - Different queuing policies can be specified

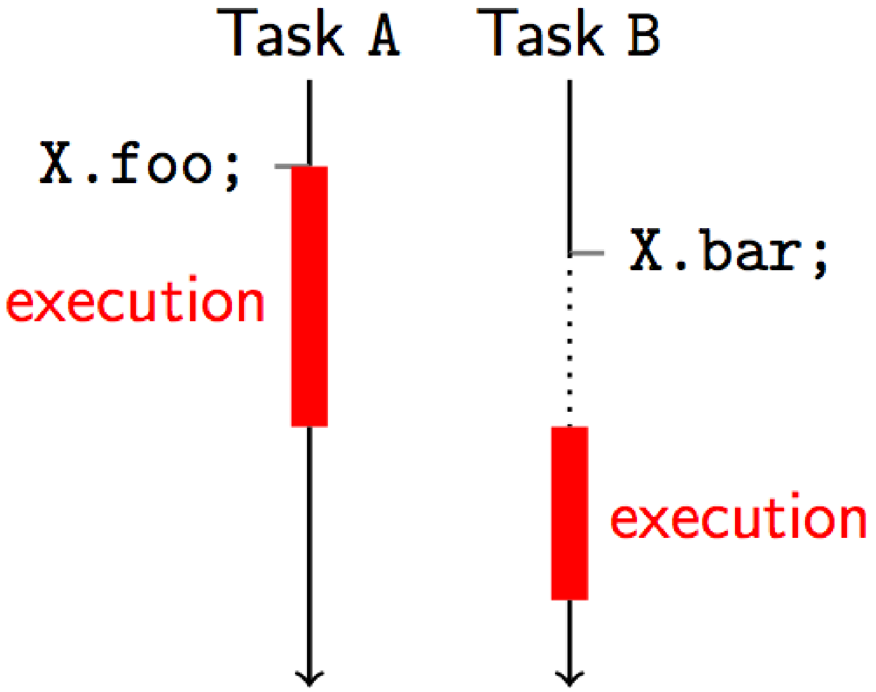
More on Task

- A task starts executing as soon as it’s created
  - For example: T: rtTask;

- Use explicit synchronization if necessary (like an “Init” entry)

Protected Types

- Provide mutual exclusion
  - For concurrent execution
  - Only one procedure/entry of instance can run at any time



Implementation

```
protected Integer_Buffer is
    entry Insert (i : in Integer);
    entry Remove (i : out Integer);
private
    buffer : Integer;
    empty : Boolean := True;
end Integer_Buffer;
protected body Integer_Buffer is
    entry Insert (i : in Integer) when empty is
    begin
        …
    end insert
end integer_buffer;
```

To use:

```
Integer_Buffer.insert(x);
Integer_Buffer.remove(y)
;
```

Options

- Procedures
  - Unconditionally changes the state of the protected object
- Entries
  - Can only be executed when the boundary condition is true
- Functions
  - Used to report the state of a protected object and do not change the state of the protected object

GNAT

- Free compiler and software development toolset for the full Ada programming language
  - Linux, Windows, and Mac compatible
- Integrated into the GCC compiler system

- To compile

```
$ gnatmake myprogram
```

- To run

```
$ ./myprogram
```