Part IV

Other Systems: II

Ada Tasks: A Brief Review

My duty as a teacher is to train, educate future programmers

Niklaus Wirth
A DoD study in the early and middle 1970s indicated that enormous saving in software costs (about $24 billion between 1983 and 1999) might be achieved if the DoD used one common programming language for all its applications instead of 450 programming languages and incompatible dialects used by its programmers.

An international competition was held to design a language based on DoD’s requirements.

Seventeen proposals were submitted and four were selected as semifinalists.
The Development of Ada: 2/2

- All semifinalists chose to base their languages on Pascal.
- The final winner was the team lead by Jean Ichibiah of CII Honeywell Bull.
- With some minor modifications, this language referred to as Ada was adopted as an ANSI standard in February 1983 (i.e., Ada 83).
- Ada was overhauled in 1995 (i.e., Ada 95) and then in 2005 with less changes (i.e., Ada 2005) and more changes in Ada 2012.
Ada Major Features

- Ada was originally designed for embedded and real-time systems.

- Major features of Ada include:
  - Strong typing, runtime checking, parallel processing (tasks, synchronous message passing), exception handling, generic, OOP, polymorphism, etc.

- We will only focus on Ada’s task and synchronization capabilities.

- A language is said to be strongly typed if it has stricter typing rules at compile time.
A task requires two components: a task **type** (definition) and a task **body** (implementation).

```ada
task type My_Task is
  declarations of exported identifies
end;

task body My_Task is
  local declarations and statements
end;
```

If there is nothing to be exported, the **task type** section can be simplified as follows:

```ada
task My_Task;
```
A task requires two components: a task **type** (definition) and a task **body** (implementation).

```ada
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
  task my_Task;
  task body my_Task is
    begin
      Put_line("Hello world!");
    end my_Task;
  begin
    Put_Line("Hello from the Main");
  end Main;
```

Hello World!
Hello from the Main

Output of this program, but the order may be different.

All tasks will run when the **Main** starts. There is no need to start a task.

The **Main** terminates only if all its tasks terminate. No join needed.
procedure To_Do is

  task Study_for_Exam;
  task Call_Mom;
  task Go_Shopping;

  task body Study_for_Exam is
    -- statements
  end Study_for_Exam;

  task body Call_Mom is
    -- statements
  end Call_Mom;

  task body Go_Shopping is
    -- statements
  end Go_Shopping;

-- these tasks are automatically created and run

begin
  -- To_Do
  null;
  -- procedure To_Do waits for all tasks to terminate
end To_Do;
Static tasks can be declared as follows:

```ada
agent : myTask;
philosophers : array (1..5) of myTask;
```

Tasks can also be dynamically allocated:

```ada
type access_to_myTask is access myTask;
to_myTask : access_to_myTask;
-- other statements
to_myTask := new myTask;
```
entry-accept: 1/4

- A task can only export its entry points to which other tasks can call.
- The **accept** block, the *rendezvous* section, contains the statements to handle this call.

```haskell
task type myTask is
  entry put(data : integer);
  entry get(result: integer);
end myTask;

task body myTask is
  myData : integer;
  begin
    -- other statement
    accept put(x : integer) do
      -- the rendezvous section
      end put;
    -- other statements
  end;
```

these entries are used to access the task
entry-accept: 2/4

- Tasks run independently until
  - an **accept** statement
    ✓ waits for someone to call this entry, then proceeds to the rendezvous section. After this, both tasks execute their ways.
  - an **entry** call
    ✓ waits for the corresponding task reaching its **accept** statement, then proceeds to the rendezvous section. After this, both tasks execute their ways.
- This is a *synchronous communication*. 
Multiple `accept` s may be used in a task body.

Communication between tasks takes place, when they rendezvous, through the actual parameters of the `entry` call and the formal parameters in the corresponding `accept` statement.

The task that accepts the entry call causes suspension of the calling task, retrieves information from parameters, processes them, and passes the results back through parameters.

The caller resumes its execution once the `accept` completes.
Thus, the **entry-accept** pair is a synchronous channel communication.

The task executes the **entry** call is the sender and the task executes the corresponding **accept** statement is the receiver.

If the task executing the **accept** statement only saves the information in the parameters and ends the rendezvous, this is a simple one-direction message passing.

The task executing the **accept** statement may return some data via the parameters.
Terminate and Delay

- The **terminate** statement terminates the task that executes this **terminate** statement.
- The **delay** statement has the following syntax:
  
  ```
  delay exp;
  ```
  
  - The **delay** statement suspends the task for at least \( exp \) seconds.
  - If \( exp \) is zero or negative, the **delay** statement has no effect.
A Simple Example: 1/2

task PRODUCER;
-- if nothing is exported,
-- a task declaration is simple

begin
  C : character;
  while not END_OF_FILE(STANDARD_INPUT) loop
    GET(C);  -- read a character from
    CONSUMER.REC(C);  -- send it to CONSUMER
  end loop;
end PRODUCER;

REC() is an entry in task CONSUMER
A Simple Example: 2/2

task type CONSUMER is
  entry REC(C: in character);
end CONSUMER;

task body CONSUMER is
  X : character;
  begin
    loop
      accept REC(C: in character) do
        X := C;
        -- retrieve the input character
      end REC;
      PUT(UPPER(X));
      -- convert to upper case & print
    end loop;
  end CONSUMER;

rendezvous section
A Simple Mutex Lock

**task type** Mutex is

**entry** Lock;

**entry** Unlock;

**end** Mutex;

**task body** Mutex is

**begin**

**loop**

**accept** Lock;

**accept** Unlock;

**end loop**;

**end** Mutex;

MyLock : Mutex;

MyLock.Lock;

-- critical section

MyLock.Unlock;

This implementation is incomplete, because there is no built-in ownership.
The Select Statement: 1/2

- The `select` statement is used to provide for the selection of alternative choices involving a rendezvous between two tasks.

1. When `select` is used in a **called task**, it allows multi-way choices known as `selective-accepts`;
2. When `select` is used in a **calling task**, it allows two-way choices known as `conditional entry calls` and `timed entry calls`. 
The Select Statement: 2/2

select
  select_alternative
or
  select_alternative
or
  select_alternative
  -- other or select_alternatives
else
  -- sequence_of_statements
end select;

Each select_alternative may be an accept, a delay followed by some other statements, or a terminate.

A select_alternative shall contain at least one accept.
In addition, it can contain (1) at most one terminate, (2) one or more delay, or (3) an else.
Note that these three possibilities are mutually exclusive.

If several accept blocks are available, one of them is selected arbitrarily.

If the corresponding entry already has queued calls, one will be selected based on the queuing policy.

If there is an else, it means this select does not have delay nor terminate!
Selective Accept: 1/2

A selective accept statement shall contain at least one accept. Additionally, it can contain:
- only one terminate
- one or more delay
- an else
These three are mutually exclusive.

task type Example is
entry Task_1(....);
entry Task_2(....);
entry Other_Task(....);
end Example;

task body Example is
......
begin
loop
select
accept Task_1(....) do
  -- statements
end Task_1;
or
accept Task_2(....) do
  -- statements
end Task_2;
or
accept Other_Task(....) do
  -- statements
end Other_Task;
or
delay expr;
end select;
end loop;
end Example;
Selective Accept: 2/2

**task body** CONSUMER is

X : character;

**begin**

**loop**

**select**

**accept** REC(C: in character) do

X := C;  -- retrieve the input character

end REC;

PUT(UPPER(X));  -- convert to upper case and print

**or**

**terminate;**

end select;

end loop;

end CONSUMER;

if no one calls REC(), the execution goes to the or part and terminates

now the task can terminate as no entry calls
Dining Philosophers: 1/3

**task type** Chopstick is

**entry** Pick_Up;

**entry** Put_Down;

end Chopstick;

**task body** Chopstick is

begin

  loop

  select

    **accept** Pick_Up;
    **accept** Put_Down;

    or

    terminate;

  end select;

end loop;

end Chopstick;
Dining Philosophers: 2/3

**task type** Philosopher is
**entry** Get_ID(k: in ID);
**end** Philosopher;

**task body** Philosopher is
  i : ID;
  limit :: constant := 100_100;
  count : integer := 0;
  left, right : ID;
**begin**
  accept Get_ID(k: in ID) do
    i := k;
  end Get_ID;
  left := i; right := i \mod 5 + 1;
**while** count /= limit **loop**
  Chop(left).Pick_Up;
  Chop(right).Pick_Up;
  -- eating
  Chop(right).Put_Down;
  Chop(left).Put_Down;
  count := count + 1;
**end loop**;
**end** Philosopher;

This solution is not deadlock-free!
Dining Philosophers: 3/3

procedure DiningPhilosophers is
  subtype ID is integer range 1..5;

-- task Philosopher ……
-- task Chopstick ……

-- local variables

Chop : array(ID) of Chopstick;  -- the 5 chopsticks
Philo : array(ID) of Philosopher;  -- the 5 philosophers

begin – procedure DiningPhilosophers
  for k in ID loop
    Philo(k).Get_ID(k);  -- assign ID
  end loop;
end DiningPhilosophers;
Selective Accept with Guards

- Each `select_alternative` can have a `guard`:

  “`when condition =>`”

```
loop
  select
    when condition_1 =>
      accept xyz(...) do
        -- statements in accept
      end xyz;
    or
      when condition_2 =>
        accept abc(...) do
          -- statements in accept
        end abc;
    or
      terminate;
  end select;
end loop;
```

These are the guards:

It is a program error if all guards are `FALSE`.

One and only one guards whose conditions are true will be selected.

The rules for using `delay, terminate` and `else` are the same as those without guards.
Dining Philosophers – 4 Chairs

**task type** GetChair **is**

**entry** Enter;

**entry** Exit;

**end** GetChair;

**task body** GetChair **is**

`i : integer := 0;
begin
  loop
    select
      when `i < 4` =>
        accept Enter;
        `i := i + 1;`
      -- if there is a free chair
      or when `i = 4` =>
        accept Exit;
        `i := i − 1;`
      -- if no Enter call, accept Exit
      or
        terminate;  -- terminate or delay for some time
    end select;
  end loop;
end GetChair;

this is a counting semaphore
Counting Semaphores

**task type** CountingSemaphore **is**

**entry** Initialize(N: *in* Natural);
**entry** Wait;
**entry** Signal;
**end** CountingSemaphore;

**task body** CountingSemaphore **is**

Count : Natural; -- non-negative integer

begin

accept Initialize(N : *in* Natural) do
  Count := N;
end Initialize;

end Initialize;

loop

  select

  when Count > 0 =>
    accept Wait do
      Count := Count – 1;
    end Wait;
  or when Count <= 0 =>
    accept Signal;
      Count := Count + 1;
  end select;

end loop;

end CountingSemaphore;
Timed Entry Call

- A timed entry call has the following syntax:

```plaintext
select
  entry_call;
  other statements
end select;
```

- Or

```plaintext
delay expr;
  sequence of statements (optional)
end select;
```

- If the call is not selected before the expiration time is reached, the entry call is cancelled.
- If the call is queued and not selected before the expiration time is reached, an attempt to cancel the call is made.
- If the call completes due to the cancellation or completes normally, the sequence of statements is executed.
A conditional entry has the following syntax:

```
select
  entry_call;
  other statements
else
  sequence of statements
end select;
```

When execution reaches the `select` statement and the other party is not ready for a rendezvous immediately, the call is cancelled and the `else` part is executed.

In other words, there is no waiting at the entry call if the other party is not ready.
The following does:

- Loops until a character can be read from the buffer
- If a character can be read, process it and break the loop
- If a character cannot be read immediately, do some local things and try again later.

```plaintext
loop
  select
    BUFFER.GET(C);
    -- process the retrieved character
    exit;
  else
    -- do some other local computation
  end select;
end loop;
```
The End