Part IV
Other Systems: II
Ada Tasking: A Brief Review

Programs must be written for people to read, and only incidentally for machines to execute.

Harold Abelson and Gerald Jay Sussman
The Development of Ada: 1/2

- 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.
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).
Ada Task Type and Body: 1/2

- A task requires two components: a task **type** (definition) and a task **body** (implementation).

```ada
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
      -- other statements
    end put;
    -- other statements
  end;
```

The entries are used to access the task.
Ada Task Type and Body: 2/2

- Static tasks can be declared as follows:

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

- Or, tasks can be dynamically allocated:

```ada
type access_to_myTask is access myTask;
to_myTask : access_to_myTask;
-- other statements
to_myTask := new myTask;
```
Task Execution: 1/3

- Tasks run independently until
  - an ACCEPT statement
    - wait for someone to call this entry, then proceed to the rendezvous section. After this, both tasks execute their ways.
  - an ENTRY call
    - wait for the corresponding task reaches it ACCEPT statement, then proceed to the rendezvous section. After this, both tasks execute their ways.
Task Execution: 2/3

- 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 call resumes once the **ACCEPT** completes.
Task Execution: 3/3

- Thus, the ENTRY-ACCEPT pair looks like 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.
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

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

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

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

**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 and print
**end loop;**
**end** CONSUMER;
**A 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;

Mutex is a task
Selective Wait: 1/4

- The **select** statement has the following purposes:
  1. Wait for more than a single rendezvous at any one time;
  2. Time out if no rendezvous is forthcoming within a specified period;
  3. Withdraw its offer to communicate if no rendezvous is immediately available;
  4. Terminate if no other tasks can possibly call its entries.
Selective Wait: 2/4

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

Each `select_alternative` is an `accept`, or a `delay` statement followed by other statements, or a `terminate` statement.

At most one `terminate` can be used in a selective wait.

One and only one `accept` in `select` or `or` will be executed.

`or` and `else` are optional
Selective Wait: 3/4

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;

now the task can terminate
Each `select_alternative` can have a **guard**: “`when condition =>`”

These are the guards:

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

It is a program error if all guards are FALSE. One and only one guards whose conditions are true will be selected.
Dining Philosophers: 1/2

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

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

task type Chopstick is
  entry Pick_Up;
  entry Put_Down;
end Chopstick;

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

task body Chopstick is
begin
  loop
    select
      accept Pick_Up;
      accept Put_Down;
    or
      terminate;
    end select;
  end loop;
end Chopstick;
Dining Philosophers: 2/2

\textbf{task body} Philosopher \textbf{is}

\begin{align*}
& \text{i : ID;} \\
& \text{limit :: \textbf{constant} := 100\_100;} \\
& \text{count : integer := 0;} \\
& \text{left, right : ID;} \\
\end{align*}

\textbf{begin}

\begin{align*}
& \text{\textbf{accept} Get\_ID(k: \textbf{in} ID) \textbf{do}} \\
& \hspace{1em} \text{i := k;} \\
& \text{\textbf{end Get\_ID;} } \\
& \text{left := i; right := i \textbf{mod} 5 + 1;} \\
\end{align*}

\textbf{while} \text{count \textbar= limit \textbf{loop}}

\begin{align*}
& \text{Chop(left).Pick\_Up;} \\
& \text{Chop(right).Pick\_Up;} \\
& \quad \textbf{\textit{-- eating}} \\
& \text{Chop(right).Put\_Down;} \\
& \text{Chop(left).Put\_Down;} \\
& \text{count := count + 1;} \\
& \text{\textbf{end loop;} } \\
\end{align*}

\textbf{end Philosopher;} \\

\textbf{begin} \textbf{-- the main} \\
\textbf{for k in ID \textbf{loop}} \\
\hspace{1em} \text{Philo(k).Get\_ID(k); \textbf{-- assign ID}} \\
\textbf{end loop;} \\
\textbf{end} DiningPhilosophers;
task type GetChair is
  entry Enter;
  entry Exit;
end GetChair;

this is a counting semaphore

task body GetChair is
  i : integer := 0;
begin
  loop
    select
      when i < 4 =>
        accept Enter;
        i := i + 1;
      or
        accept Exit;
        i := i - 1;
    or
      terminate;
  end select;
  end loop
end GetChair;
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;
  loop
    select
      when Count > 0 =>
        accept Wait do
          Count := Count – 1;
        end Wait;
      or
        accept Signal;
        Count := Count + 1;
    end select;
  end loop;
end CountingSemaphore;
A conditional entry has the following syntax:

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

When execution reaches the `select` statement and the other party is not ready for a rendezvous, 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, do some local thing and try again.

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