MESSAGE-BASED SYNCHRONISATION AND COMMUNICATION

Goals
- To understand the requirements for communication and synchronisation based on message passing
- To understand:
  - the Ada extended rendezvous
  - selective waiting
  - POSIX message queues
  - Remote procedure calls
Message-Based Communication and Synchronisation

- Use of a single construct for both synchronisation and communication
- Three issues:
  - the model of synchronisation
  - the method of process naming
  - the message structure

![Diagram showing communication and synchronisation between Process P1 and Process P2]
Variations in the process synchronisation model arise from the semantics of the send operation

- Asynchronous (or no-wait) (e.g. POSIX)
  - Requires buffer space. What happens when the buffer is full?
Process Synchronisation

- **Synchronous** (e.g. CSP, occam2)
  - No buffer space required
  - Known as a rendezvous

![Diagram of process synchronisation](image)
Process Synchronisation

- Remote invocation (e.g. Ada)
  - Known as an extended rendezvous
- Analogy:
  - The posting of a letter is an asynchronous send
  - A telephone is a better analogy for synchronous communication

![Diagram of process synchronisation](#)
Asynchronous and Synchronous Sends

- Asynchronous communication can implement synchronous communication:

  P1
  asyn_send (M)
  wait (ack)

  P2
  wait (M)
  asyn_send (ack)

- Two synchronous communications can be used to construct a remote invocation:

  P1
  syn_send (message)
  wait (reply)

  P2
  wait (message)
  ... 
  construct reply 
  ... 
  syn_send (reply)
Disadvantages of Asynchronous Send

- Potentially infinite buffers are needed to store unread messages
- Asynchronous communication is out-of-date; most sends are programmed to expect an acknowledgement
- More communications are needed with the asynchronous model, hence programs are more complex
- It is more difficult to prove the correctness of the complete system
- Where asynchronous communication is desired with synchronised message passing then buffer processes can easily be constructed; however, this is not without cost
Process Naming

- Two distinct sub-issues
  - direction versus indirection
  - symmetry
- With direct naming, the sender explicitly names the receiver:
  
  send <message> to <process-name>

- With indirect naming, the sender names an intermediate entity (e.g. a channel, mailbox, link or pipe):
  
  send <message> to <mailbox>

- With a mailbox, message passing can still be synchronous
- Direct naming has the advantage of simplicity, whilst indirect naming aids the decomposition of the software; a mailbox can be seen as an interface between parts of the program
Process Naming

- A naming scheme is symmetric if both sender and receiver name each other (directly or indirectly)
  
  send <message> to <process-name>
  wait <message> from <process-name>
  
  send <message> to <mailbox>
  wait <message> from <mailbox>

- It is asymmetric if the receiver names no specific source but accepts messages from any process (or mailbox)
  wait <message>

- Asymmetric naming fits the client-server paradigm

- With indirect the intermediary could have:
  – a many-to-one structure  
  – a many-to-many structure
  – a one-to-one structure  
  – a one-to-many
Message Structure

- A language usually allows any data object of any defined type (predefined or user) to be transmitted in a message.
- Need to convert to a standard format for transmission across a network in a heterogeneous environment.
- OS allow only arrays of bytes to be sent.
The Ada Model

- Ada supports a form of message-passing between tasks
- Based on a client/server model of interaction
- The server declares a set of services that it is prepared to offer other tasks (its clients)
- It does this by declaring one or more public entries in its task specification
- Each entry identifies the name of the service, the parameters that are required with the request, and the results that will be returned
Entries

entry_declarations ::= 
  entry defining_identifier[(discrete_subtype_definition)]
  parameter_profile;

entry Syn;
entry Send(V : Value_Type);
entry Get(V : out Value_Type);
entry Update(V : in out Value_Type);
entry Mixed(A : Integer; B : out Float);
entry Family(Boolean)(V : Value_Type);
Example

task type Telephone_Operator is
  entry Directory_Enquiry(
    Person : in Name;
    Addr : Address;
    Num : out Number);
  -- other services possible
end Telephone_Operator;

An_Op : Telephone_Operator;

-- client task executes
  "11 Main, Street, York"
  Stuarts_Number);
Accept Statement

accept_statement ::= 

    accept entry_direct_name[(entry_index)]
    parameter_profile [do
        handled_sequence_of_statements
    end [entry_identifier]];

 accept Family(True)(V : Value_Type) do
    -- sequence of statements
exception
    -- handlers
end Family;
task body Telephone_Operator is
begin
    ...
    loop
        -- prepare to accept next call
        accept Directory_Enquiry (...) do
            -- look up telephone number
            exception
                when Illegal_Number =>
                    -- propagate error to client
                    end Directory_Enquiry;
            -- undertake housekeeping
        end loop;
    ...
end Telephone_Operator;
task type Subscriber;
task body Subscriber is
begin
  ...
  loop
    ...
    ...
  end loop;
  ...
end Subscriber;
Protocol

```
T.E(A,B)
```

```
task T is ...

accept E(X : int; Y: out int) do
  -- use X
  -- undertake computation
  -- produce Y
  -- complete computation
end E;
```
Synchronisation

- Both tasks must be prepared to enter into the communication
- If one is ready and the other is not, then the ready one waits for the other
- Once both are ready, the client's parameters are passed to the server
- The server then executes the code inside the accept statement
- At the end of the accept, the results are returned to the client
- Both tasks are then free to continue independently
task type Bus_Driver (Num : Natural) is
  entry Get_Ticket (R: in Request, M: in Money;
                   G : out Ticket) ;
    -- money given with request, no change given!
end Bus_Driver;

task body Bus_Driver is
begin
  loop
    accept Get_Ticket (R: Request,
                        M: Money; G : out Ticket) do
      -- take money
      G := Next_Ticket(R);
    end Get_Ticket;
  end loop;
end Bus_Driver;
type Bus_T (N : Natural) is
  record
    ....
    Driver : Bus_Driver(N);
  end record;

Number31 : Bus_T(31);
Number60 : Bus_T(60);
Number70 : Bus_T(70);
Shop Keeper Example

task Shopkeeper is
  entry Serve(X : Request; A: out Goods);
  entry Get_Money(M : Money; Change : out Money);
end Shopkeeper;

task body Shopkeeper is
begin
  loop
    accept Serve(X : Request; A: out Goods) do
      A := Get_Goods;
    end Serve;
    accept Get_Money(M : Money; Change : out Money) do
      -- take money return change
    end Get_Money;
  end loop;
end Shopkeeper;

What is wrong with this algorithm?
task Customer;

task body Customer is

begin
  -- go to shop
  Shopkeeper.Serve(Weekly_Shoping, Trolley);
  -- leave shop in a hurry!

end Customer;
task type Rider;
task body Rider is
begin
  ...
  -- go to bus stop and wait for bus
  while Bus /= Number31 loop
    -- moan about bus service
  end loop;
  Bus.Bus_Driver.Get_Ticket(Heslington, Fiftyp, Ticket);
    -- get in line
    -- board bus, notice three more number 31 buses
  ...
end Rider;
Other Facilities

- 'Count gives number of tasks queued on an entry
- Entry families allow the programmer to declare, in effect, a single dimension array of entries
- Nested accept statements allow more than two tasks to communicate and synchronise
- A task executing inside an accept statement can also execute an entry call
- Exceptions not handled in a rendezvous are propagated to both the caller and the called tasks
- An accept statement can have exception handlers
Restrictions

- Accept statements can only be placed in the body of a task
- Nested accept statements for the same entry are not allowed
- The 'Count attribute can only be accessed from within the task that owns the entry
- Parameters to entries cannot be access parameters but can be parameters of an access type
task Multiplexer is
    entry Channel(1..3)(X : Data);
end Multiplexer;

task body Multiplexer is
begin
    loop
        for I in 1..3 loop
            accept Channel(I)(X : Data) do
                -- consume input data on channel I
                end Channel;
        end loop;
    end loop;
end Multiplexer;
**Tesco**

```vhdl
type Counter is (Meat, Cheese, Wine);
task Tesco_Server is
  entry Serve(Counter)(Request: . . .);
end Tesco_Server;

task body Tesco_Server is
begin
  loop
    accept Serve(Meat)(. . .) do . . . end Serve;
    accept Serve(Cheese)(. . .) do . . . end Serve;
    accept Serve(Wine)(. . .) do . . . end Serve;
  end loop
end Tesco_Server;
```

- What happens if all queues are full?
- What happens if the Meat queue is empty?
task body Controller is
begin
  loop
    accept Doio (I : out Integer) do
      accept Start;
      accept Completed (K : Integer) do
        I := K;
      end Completed;
    end Doio;
  end loop;
end Controller;
task Shopkeeper is
  entry Serve_Groceries(. . .);
  entry Serve_Tobacco( . . .);
  entry Serve_Alcohol(. . .);
end Shopkeeper;

task body Shopkeeper is
begin
  . . .
  accept Serve_Groceries ( . . . ) do
    -- no change for a £10 note
    accept Serve_Alcohol(. . .) do
      -- serve another Customer,
      -- get more change
      end Serve_Alcohol
    end Serve_Groceries
  . . .
end Shopkeeper;
Entry Call within Accept Statement

task Car_Spares_Server is
  entry Serve_Car_Part(Number: Part_ID; . . .);
end Car_Spares_Server ;

task body Car_Spares_Server is
begin
  . . .
  accept Serve_Car_Part(Number: Part_ID; . . .) do
    -- part not is stock
    Dealer.Phone_Order(. . .);
  end Serve_Car_Part;
  . . .
end Car_Spares_Server;
Exceptions

accept Get(R : out Rec; Valid_Read : out Boolean) do
loop
  begin
  Put("VALUE OF I?"); Get(R.I);
  Put("VALUE OF F?"); Get(R.F);
  Put("VALUE OF S?"); Get(R.S);
  Valid_Read := True;
  return;
  exception
  when Ada.Text_IO.Data_Error =>
    Put("INVALID INPUT: START AGAIN");
  end;
end loop;
exception
  when Ada.Text_IO.Mode_Error =>
    Valid_Read := False;
end Get;
Private Entries

- Public entries are visible to all tasks which have visibility to the owning task's declaration.
- Private entries are only visible to the owning task:
  - if the task has several tasks declared internally; these tasks have access to the private entry.
  - if the entry is to be used internally by the task for requeuing purposes.
  - if the entry is an interrupt entry, and the programmer does not wish any software task to call this entry.
task type Telephone_Operator is
    entry Report_Fault(N : Number);
private
    entry Allocate_Repair_Worker(N : out Number);
end Telephone_Operator;

task body Telephone_Operator is
    Failed : Number;
    task type Repair_Worker;
    Work_Force: array (1.. Num_Workers) of Repair_Worker;
    task body Repair_Worker is
        Job : Number:
        begin
            ...Telephone_Operator.Allocate_Repair_Worker(Job);...
        end Repair_Worker;
begin

loop

accept Report_Fault(N : Number) do
  Failed := N;
end Report_Fault;
-- log faulty line

if New_Fault(Failed) then -- new fault

  accept Allocate_Repair_Worker(N : out Number) do
    N := Failed;
  end Allocate_Repair_Worker;

end if;

end loop;

end Telephone_Operator;
Selective Waiting

- So far, the receiver of a message must wait until the specified process, or mailbox, delivers the communication.
- A receiver process may actually wish to wait for any one of a number of processes to call it.
- Server processes receive request messages from a number of clients; the order in which the clients call being unknown to the servers.
- To facilitate this common program structure, receiver processes are allowed to wait selectively for a number of possible messages.
- Based on Dijkstra’s guarded commands.
The select statement comes in four forms:

\[
\text{select\_statement} ::= \\
\text{selective\_accept} \mid \\
\text{conditional\_entry\_call} \mid \\
\text{timed\_entry\_call} \mid \\
\text{asynchronous\_select}
\]
Selective Accept

The selective accept allows the server to:

- wait for more than a single rendezvous at any one time
- time-out if no rendezvous is forthcoming within a specified time
- withdraw its offer to communicate if no rendezvous is available immediately
- terminate if no clients can possibly call its entries
Syntax Definition

selective_accept ::= 
    select
    [guard]
    selective_accept_alternative
{ or
    [guard]
    selective_accept_alternative
[ else
    sequence_of_statements ]
end select;

guard ::= when <condition> =>
Syntax Definition II

selective_accept_alternative ::= 
    accept_alternative | 
    delay_alternative  | 
    terminate_alternative

accept_alternative ::= 
    accept_statement [ sequence_of_statements ]

delay_alternative ::= 
    delay_statement [ sequence_of_statements ]

terminate_alternative ::= 
    terminate;
task Server is
  entry S1(...);
  entry S2(...);
end Server;

task body Server is
  ...
begin
  loop
    select
      accept S1(...) do
        -- code for this service
        end S1;
      or
      accept S2(...) do
        -- code for this service
        end S2;
    end select;
  end loop;
end Server;
task type Telephone_Operator is
  entry Directory_Enquiry (P : Name; A : Address;
    N : out Number);
  entry Directory_Enquiry (P : Name; PC : Postal_Code;
    N : out Number);
  entry Report_Fault(N : Number);
private
  entry Allocate_Repair_Worker (N : out Number);
end Telephone_Operator;
Example II

```vhdl

task body Telephone_Operator is
    Failed : Number;
end task type Repair_Worker;

Work_Force : array(1.. Num_Workers) of Repair_Worker;

end task body Repair_Worker is separate;
```

© Alan Burns and Andy Wellings, 2001
begin
  loop
    select
    accept Directory_Enquiry( ... ; A: Address...) do
      -- look up number based on address
      end Directory_Enquiry;
    or
    accept Directory_Enquiry( ... ;
                          PC: Postal_Code...) do
      -- look up number based on ZIP
      end Directory_Enquiry;
    or
or

accept Report_Fault(N : Number) do
  ...
end Report_Fault;

if New_Fault(Failed) then
  accept Allocate_Repair_Worker (N : out Number) do
    N := Failed;
  end Allocate_Repair_Worker;
end if;
end select;
end loop;
end Telephone_Operator;
Note

- If no rendezvous are available, the select statement waits for one to become available.
- If one is available, it is chosen immediately.
- If more than one is available, the one chosen is implementation dependent (RT Annex allows order to be defined).
- More than one task can be queued on the same entry; default queuing policy is FIFO (RT Annex allows priority order to be defined).
type Counter is (Meat, Cheese, Wine);
task Tesco_Server is
    entry Serve(Counter) (Request: . . .);
end Tesco_Server;
task body Tesco_Server is
begin
    loop
        select
            accept Serve(Meat)(. . .) do . . . end Serve;
        or
            accept Serve(Cheese)(. . .) do . . . end Serve;
        or
            accept Serve(Wine)(. . .) do . . . end Serve;
        end select
    end loop
end Tesco_Server;

- What happens if all queues are full?
- What happens if the Meat queue is empty?
What is the difference between

select
  accept A;
  B;
or
  accept C;
end select

and

select
  accept A do
    B;
  end A;
or
  accept C;
end select
Guarded Alternatives

- Each select accept alternative can have an associated guard
- The guard is a boolean expression which is evaluated when the select statement is executed
- If the guard evaluates to true, the alternative is eligible for selection
- If it is false, the alternative is not eligible for selection during this execution of the select statement (even if client tasks are waiting on the associated entry)
Example Usage

```plaintext
select
  when Boolean_Expression =>
    accept S1(...) do
      -- code for service
    end S1;
    -- sequence of statements
  or
    ...
end select;
```
task body Telephone_Operator is
begin
  ...
  select
    accept Directory_Enquiry (...) do ... end;
  or
    accept Directory_Enquiry (...) do ... end;
  or
    when Workers_Available =>
      accept Report_Fault (...) do ... end;
  end select;
end Telephone_Operator;
Corner Shop

type Counter is (Tobacco, Alcohol, Groceries);
task Shopkeeper is
  entry Serve(Counter) (Request: . . .);
end Shopkeeper;

task body Shopkeeper is
begin
  loop
    select
      when After_7pm =>
        accept Serve(Alcohol)(. . .) do . . . end Serve;
      or
      when Customers_Age > 16 =>
        accept Serve(Tobacco)(. . .) do . . . end Serve;
      or
        accept Serve(Groceries)(. . .) do . . . end Serve;
    end select
  end loop
end Shopkeeper;

■ Are these guards OK?
Delay Alternative

- The delay alternative of the select statement allows the server to time-out if an entry call is not received within a certain period.
- The timeout is expressed using a delay statement, and therefore can be relative or absolute.
- If the relative time is negative, or the absolute time has passed, the delay alternative becomes equivalent to the else alternative.
- More than one delay is allowed.
Consider a task which reads a sensors every 10 seconds, however, it may be required to change its periods during certain modes of operation

```vhdl
task Sensor_Monitor is
    entry New_Period(P : Duration);
end Sensor_Monitor;
```
task body Sensor_Monitor is
    Current_Period : Duration := 10.0;
    Next_Cycle : Time := Clock + Current_Period;
begin
    loop
        -- read sensor value etc.
        select
            accept New_Period(P : Duration) do
                Current_Period := P;
            end New_Period;
            Next_Cycle := Clock + Current_Period;
        or
            delay until Next_Cycle;
            Next_Cycle := Next_Cycle + Current_Period;
        end select;
    end loop;
end Sensor_Monitor;
Delay Alternative: Error Detection

- Used to program timeouts

```plaintext
task type Watchdog is
  entry All_Is_Well;
end Watchdog;
```
Watchdog

task body Watchdog is
    Client_Failed : Boolean := False;
begin
    loop
        select
            accept All_Is_Well;
        or
            delay 10.0;
            -- signal alarm
            Client_Failed := True;
        end select;
    end loop;
end Watchdog;
The Else Part

task body Sensor_Monitor is
  Current_Period : Duration := 10.0;
  Next_Cycle : Time := Clock + Current_Period;
begin
  loop
    -- read sensor value etc.
    select
      accept New_Period(P : Duration) do
        Current_Period := P;
      end New_Period;
    else -- cannot be guarded
      null;
    end select;
    Next_Cycle := Clock + Current_Period;
    delay until Next_Cycle;
  end loop;
end Sensor_Monitor;
The Delay and the Else Part

- Cannot mix else part and delay in the same select statement.
- The following are equivalent

```plaintext
select
  accept A;
or
  accept B;
else
  C;
end select;
```

```plaintext
select
  accept A;
or
  accept B;
or
  delay 0.0;
  C;
end select;
```
More on Delay

select
  accept A;
  delay 10.0;
end select;

or

select  
  accept A;
  delay 10.0;
end select;

else
  delay 10.0;
end select;

select
  accept A; 
  delay 5.0;
end select;

or

delay 5.0;
end select;

What is the difference?
The Terminate Alternative

- In general a server task only needs to exist when there are clients to serve.
- The very nature of the client server model is that the server does not know the identity of its clients.
- The terminate alternative in the select statement allows a server to indicate its willingness to terminate if there are no clients that could possibly request its service.
- The server terminates when a master of the server is completed and all its dependants are either already terminated or are blocked at a select with an open terminate alternative.
Primes by Sieve
procedure Primes_By_Sieve is
  task type Sieve is
    entry Pass_On(Int : Integer);
  end Sieve;

  task Odd;

type Sieve_Ptr is access Sieve;

function Get_New_Sieve return Sieve_Ptr is
begin
  return new Sieve;
end Get_New_Sieve;

  task body Odd is ...
task body Sieve is ...

begin null; end Primes_By_Sieve;
task body Odd is
  Limit : constant Positive := ...;
  Num  : Positive;
  S : Sieve_Ptr := new Sieve;
begin
  Num := 3;
  while Num < Limit loop
    S.Pass_On(Num);
    Num := Num + 2;
  end loop;
end Odd;
task body Sieve is
    New_Sieve : Sieve_Ptr;
    Prime, Num : Positive;
begin
    accept Pass_On(Int : Integer) do
        Prime := Int;
    end Pass_On;
    -- Prime is a prime number, could output
    loop
        select
            accept Pass_On(Int : Integer) do
                Num := Int;
            end Pass_On;
            or
                terminate;
        end select;
        exit when Num rem Prime /= 0;
    end loop;
Primes by Sieve V

New_Sieve := Get_New_Sieve;
New_Sieve.Pass_On(Num);
loop
  select
    accept Pass_On(Int : Integer) do
      Num := Int;
    end Pass_On;
  or
    terminate;
  end select;
if Num rem Prime /= 0 then
  New_Sieve.Pass_On(Num);
end if;
end loop;
end Sieve;
**Last Wishes**

- Last Wishes can be programmed using controlled types
- Example: count the number of times two entries are called

```ada
with Ada.Finalization; use Ada;
package Counter is
  type Task_Last_Wishes is new Finalization.Limited_Controlled
    with record
      Count1, Count2 : Natural := 0;
    end record;
  procedure Finalize(Tlw : in out Task_Last_Wishes);
end Counter;
```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Ada.Text_IO; use Ada.Text_IO;
package body Counter is
    procedure Finalize(Tlw : in out Task_Last_Wishes) is
        begin
            Put("Calls on Service1:");
            Put(Tlw.Count1);
            Put(" Calls on Service2:");
            Put(Tlw.Count2);
            New_Line;
            end Finalize;
end Counter;
task body Server is
   Last_Wishes : Counter.Task_Last_Wishes;
begin
   -- initial housekeeping
   loop
      select
         accept Service1(...) do
            ...
         end Service1;
         Last_Wishes.Count1 := Last_Wishes.Count1 + 1;
      or
         accept Service2(...) do
            ...
         end Service2;
         Last_Wishes.Count2 := Last_Wishes.Count2 + 1;
      or
         terminate;
      end select;
   end loop;
end Server;

As the task terminates the finalize procedure is executed
Program Error

- If all the accept alternatives have guards then there is the possibility in certain circumstances that all the guards will be closed.
- If the select statement does not contain an else clause then it becomes impossible for the statement to be executed.
- The exception `Program_Error` is raised at the point of the select statement if no alternatives are open.
Sample Exam Question

A server task has the following Ada specification.

```ada
task Server is
  entry Service_A;
  entry Service_B;
  entry Service_C;
end Server;
```

Write the body of the `Server` task so that

- If client tasks are waiting on all the entries, the `Server` should service the clients in a cyclic order, that is accept first a `Service_A` entry, and then a `Service_B` entry, and then a `Service_C`, so on

- If not all entries have a client task waiting, the `Server` should service the other entries in a cyclic order. The `Server` tasks should not be blocked if there are clients still waiting for a service

- If the `Server` task has no waiting clients then it should NOT busy-wait; it should block waiting for a client's request to be made

- If all the possible clients have terminated, the `Server` should terminate

Assume that client tasks are not aborted and issue simple entry calls only
The Selective Accept: Summary

- A selective accept must contain at least one accept alternative (each possibly guarded)

- A selective accept may contain one and only one of the following:
  - a terminate alternative (possibly guarded), or
  - one or more delay alternatives (each possibly guarded), or
  - an else part
A select alternative is 'open' if it does not contain a guard or if the boolean condition associated with the guard evaluates to true; otherwise the alternative is 'closed'.

On execution: all guards, open delay expressions, and open entry family expressions are evaluated.

A choice is made from the open alternatives.
Non-determinism and Selective Waiting

- Concurrent languages make few assumptions about the execution order of processes.
- A scheduler is assumed to schedule processes non-deterministically.
- Consider a process P that will execute a selective wait construct upon which processes S and T could call.
Non-determinism and Selective Waiting

- P runs first; it is blocked on the select. S (or T) then runs and rendezvous with P.
- S (or T) runs, blocks on the call to P; P runs and executes the select; a rendezvous takes place with S (or T).
- S (or T) runs first and blocks on the call to P; T (or S) now runs and is also blocked on P. Finally P runs and executes the select on which T and S are waiting.
- The three possible interleavings lead to P having none, one or two calls outstanding on the selective wait.
- If P, S and T can execute in any order then, in latter case, P should be able to choose to rendezvous with S or T — it will not affect the programs correctness.
Non-determinism and Selective Waiting

- A similar argument applies to any queue that a synchronisation primitive defines.
- Non-deterministic scheduling implies all queues should release processes in a non-deterministic order.
- Semaphore queues are often defined in this way; entry queues and monitor queues are specified to be FIFO.
- The rationale here is that FIFO queues prohibit starvation but if the scheduler is non-deterministic then starvation can occur anyway!
Timed Entry Calls

- A timed entry call issues an entry call which is cancelled if the call is not accepted within the specified period (relative or absolute)

- Note that only one delay alternative and one entry call can be specified.

```
  task type Subscriber;
```
task body Subscriber is
    Stuarts_Number : Number;
begin
    loop
      ...
      select
                                 "10 Main Street, York", Stuarts_Number);
        -- log the cost of a directory enquiry call
      or
        delay 10.0;
        -- phone up Stuart's parents and ask them;
        -- log the cost of a long distance call
      end select;
      ...
    end loop;
end Subscriber;
task body Telephone_Operator is
  ...
begin
  loop
    -- prepare to accept next request
    select
      accept Directory_Enquiry(Person : Name;
        Addr : Address; Num : out Number) do
        delay 3600.0; -- take a lunch break
        end Directory_Enquiry; or
      ...
    end select;
  ...
end loop;
end Telephone_Operator;

Time-out is on the start of the rendezvous not the finish
task type Shopper;

task body Shopper is
begin
  . . .
  -- enter shop
  select
    shopkeeper.Serve_Groceries(. . .)
  or
    delay10.0;
    -- moan about queues;
  end select;
  -- leave shop
  . . .
end Shopper;

WARNING
accept Serve_Groceries(. . .) do
  -- go to lunch
end Serve_Groceries;
The conditional entry call allows the client to withdraw the offer to communicate if the server task is not prepared to accept the call immediately.

It has the same meaning as a timed entry call where the expiry time is immediate.

```plaintext
select
else
  null;  -- assume they are on already
end select;
```
A conditional entry call should only be used when the task can genuinely do other productive work, if the call is not accepted.

Care should be taken not to program polling, or busy-wait, solutions unless they are explicitly required.

Note, the conditional entry call uses an `else`, the timed entry call an `or`.
They cannot be mixed, nor can two entry call statements be included.

A client task cannot therefore wait for more than one entry call to be serviced.

The asynchronous select statement allows some of these restrictions to be overcome.
procedure Dining_Philosophers is
    package Activities is
        procedure Think;
        procedure Eat;
    end Activities;

    N : constant := 5;  -- number of philosophers
    type Philosophers_Range is range 0..N-1;

    task type Phil(P : Philosophers_Range);
    type Philosopher is access Phil;

    task type Chopstick_Control is
        entry Pick_Up;
        entry Put_Down;
    end Chopstick_Control;
task Deadlock_Prevention is
    entry Enters;
    entry Leaves;
end Deadlock_Prevention;

Chopsticks : array(Philosophers_Range) of Chopstick_Control;
Philosophers : array(Philosophers_Range) of Philosopher;

package body Activities is separate;
task body Phil is separate;
task body Chopstick_Control is separate;
task body Deadlock_Prevention is separate;

begin
for P in Philosophers_Range loop
    Philosophers(P) := new Phil(P);
end loop;
edendining_Philosophers;
Dining Philosophers III

separate (Dining_Philosophers)
task body Chopstick_Control is
begin
  loop
    accept Pick_Up;
    accept Put_Down;
  end loop;
end Chopstick_Control;
Dining Philosophers IV

```
separate (Dining_Philosophers)
task body Deadlock_Prevention is
    Max : constant Integer := N - 1;
    People_Eating : Integer range 0..Max := 0;
begin
    loop
        select
            when People_Eating < Max =>
                accept Enters;
                People_Eating := People_Eating + 1;
            or
                accept Leaves;
                People_Eating := People_Eating - 1;
        end select;
    end loop;
end Deadlock_Prevention;
```
Dining Philosophers V

separate (Dining_Philosophers)
task body Phil is
    Chop_Stick1, Chop_Stick2 : Philosophers_Range;
begin
    Chop_Stick1 := P;
    Chop_Stick2 := (Chop_Stick1 + 1) mod N;
    loop
        Think;
        Deadlock_Prevention.Enters;
        Chopsticks(Chop_Stick1).Pick_Up;
        Chopsticks(Chop_Stick2).Pick_Up;
        Eat;
        Chopsticks(Chop_Stick1).Put_Down;
        Chopsticks(Chop_Stick2).Put_Down;
        Deadlock_Prevention.Leaves;
    end loop;
end Philosopher;
Exercises

- Modify the code so that the program terminates after each philosopher has taken 32 meals
- Make your solution resilient to a task failing
- Replace the control tasks with protected objects
POSIX Message Queues

- POSIX supports asynchronous, indirect message passing through the notion of message queues
- A message queue can have many readers and many writers
- Priority may be associated with the queue
- Intended for communication between processes (not threads)
- Message queues have attributes which indicate their maximum size, the size of each message, the number of messages currently queued etc.
- An attribute object is used to set the queue attributes when the queue is created
POSIX Message Queues

- Message queues are given a name when they are created
- To gain access to the queue, requires an `mq_open` name
- `mq_open` is used to both create and open an already existing queue (also `mq_close` and `mq_unlink`)
- Sending and receiving messages is done via `mq_send` and `mq_receive`
- Data is read/written from/to a character buffer.
- If the buffer is full or empty, the sending/receiving process is blocked unless the attribute O_NONBLOCK has been set for the queue (in which case an error return is given)
- If senders and receivers are waiting when a message queue becomes unblocked, it is not specified which one is woken up unless the priority scheduling option is specified
A process can also indicate that a signal should be sent to it when an empty queue receives a message and there are no waiting receivers.

In this way, a process can continue executing whilst waiting for messages to arrive or one or more message queues.

It is also possible for a process to wait for a signal to arrive; this allows the equivalent of selective waiting to be implemented.

If the process is multi-threaded, each thread is considered to be a potential sender/receiver in its own right.
typedef enum {xplane, yplane, zplane} dimension;

void move_arm(int D, int P);

#define DEFAULT_NBYTES 4
int nbytes = DEFAULT_NBYTES;

#define MQ_XPLANE   "/mq_xplane" -- message queue name
#define MQ_YPLANE   "/mq_yplane" -- message queue name
#define MQ_ZPLANE   "/mq_zplane" -- message queue name
#define MODE . . . /* mode info for mq_open */
/* names of message queues */
void controller(dimension dim) {
    int position, setting;
    mqd_t my_queue; /* message queue */
    struct mq_attr ma; /* attributes */
    char buf[DEFAULT_NBYTES];
    ssize_t len;

    position = 0;
    switch(dim) { /* open appropriate message queue */
        case xplane:
            my_queue = MQ_OPEN(MQ_XPLANE, O_RDONLY, MODE, &ma);
            break;
        case yplane:
            my_queue = MQ_OPEN(MQ_YPLANE, ...);
            break;
        case zplane:
            my_queue = MQ_OPEN(MQ_ZPLANE, ...);
            break;
        default:
            return;
    };
}
while (1) {
    /* read message */
    len = mq_receive(my_queue, &buf[0], nbytes, null);
    setting = *((int *)&buf[0]);
    position = position + setting;
    move_arm(dim, position);
};

Now the main program which creates the controller processes and passes the appropriate coordinates to them:
void (*C)(dimension dim) = &controller;

int main(int argc, char **argv) {
    mqd_t mq_xplane, mq_yplane, mq_zplane;
    struct mq_attr ma; /* queue attributes */
    int xpid, ypid, zpid;
    char buf[DEFAULT_NBYTES];

    /* set message queues attributes*/
    ma.mq_flags = 0;    /* No special behaviour */
    ma.mq_maxmsg = 1;
    ma.mq_msgsize = nbytes;
    mq_xplane = MQ_OPEN(MQ_XPLANE,
                        O_CREAT|O_EXCL, MODE, &ma);
    mq_yplane = ...;
    mq_zplane = ...;
/ * Duplicate the process to get three controllers */  
switch (xpid = FORK()) {
  case 0: controller(xplane); exit(0); /* child */
  default:    /* parent */
    switch (ypid = FORK()) {
      case 0: controller(yplane); exit(0);
      default:    /* parent */
        switch (zpid = FORK()) {
          case 0: controller(zplane); exit(0);
          default:    /* parent */
            break;
        }
    }
}

while (1) {
  /* set up buffer to transmit each co-ordinate 
     to the controllers, for example */
  MQ_SEND(mq_xplane, &buf[0], nbytes, 0);
}
Summary

- The semantics of message-based communication are defined by three issues:
  - the model of synchronisation
  - the method of process naming
  - the message structure

- Variations in the process synchronisation model arise from the semantics of the send operation.
  - asynchronous, synchronous or remote invocation
  - Remote invocation can be made to appear syntactically similar to a procedure call

- Process naming involves two distinct issues; direct or indirect, and symmetry
Summary

- Ada has remote invocation with direct asymmetric naming.
- Communication in Ada requires one task to define an entry and then, within its body, accept any incoming call. A rendezvous occurs when one task calls an entry in another.
- Selective waiting allows a process to wait for more than one message to arrive.
- Ada’s select statement has two extra facilities: an else part and a terminate alternative.
- POSIX message queues allow asynchronous, many to many communication.