dit
UPM

Comunicación mediante mensajes

Juan Antonio de la Puente
DIT/UPM

Objetivos

- Comprender los problemas relacionados con la comunicación entre procesos basada en el interambio de mensajes
- Estudiar la forma de realizar este tipo de comunicación en Ada y C/POSIX

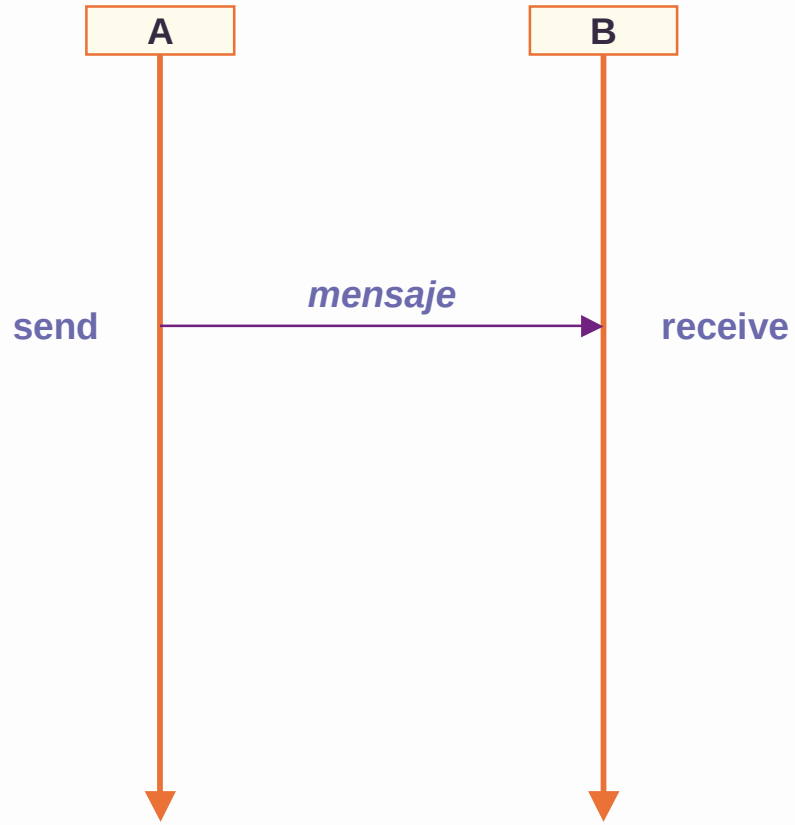
Índice

- **Comunicación mediante mensajes**
- Comunicación entre tareas en Ada
  - cita extendida
  - espera selectiva
  - llamada selectiva
- Comunicación entre *threads* en C/POSIX

Comunicación mediante mensajes

- Las tareas se pueden comunicar y sincronizar mediante mensajes
- Esta forma de comunicación no necesita memoria común
- Se usan los mismos mecanismos para la sincronización y la comunicación
- Hay tres aspectos de interés:
  - Sincronización
  - Identificación del proceso emisor y receptor
  - Estructura de los mensajes

Diagramas de secuencia de mensajes

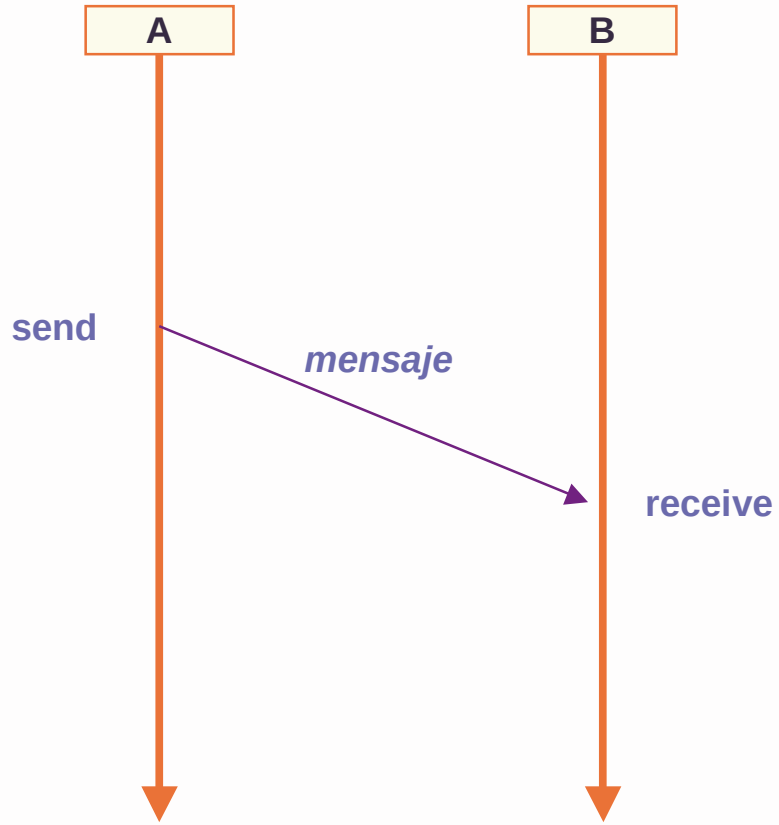


- Representan la interacción entre procesos mediante el intercambio de mensajes
- El tiempo va hacia abajo

Sincronización en el envío de mensajes

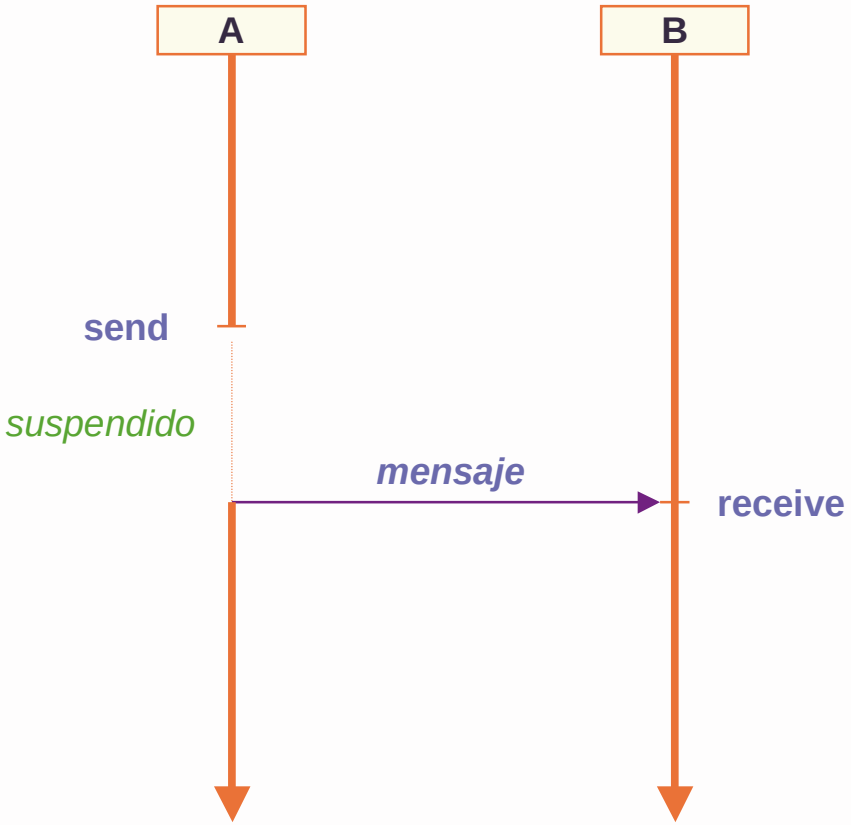
- El proceso receptor siempre espera si el mensaje no ha llegado todavía.
- Para el proceso emisor hay tres modelos básicos:
  - *Comunicación asíncrona*: el emisor continúa su ejecución
  - *Comunicación síncrona (cita)*: el emisor espera a que el receptor reciba el mensaje
  - *Invocación remota (cita extendida)*: el emisor espera a que el receptor reciba el mensaje, y la respuesta de éste

Comunicación asíncrona



- Hace falta usar un tampón
  - capacidad potencialmente ilimitada
  - si es limitado puede bloquearse el emisor
- El emisor puede requerir un reconocimiento
  - diseño complicado

Comunicación síncrona

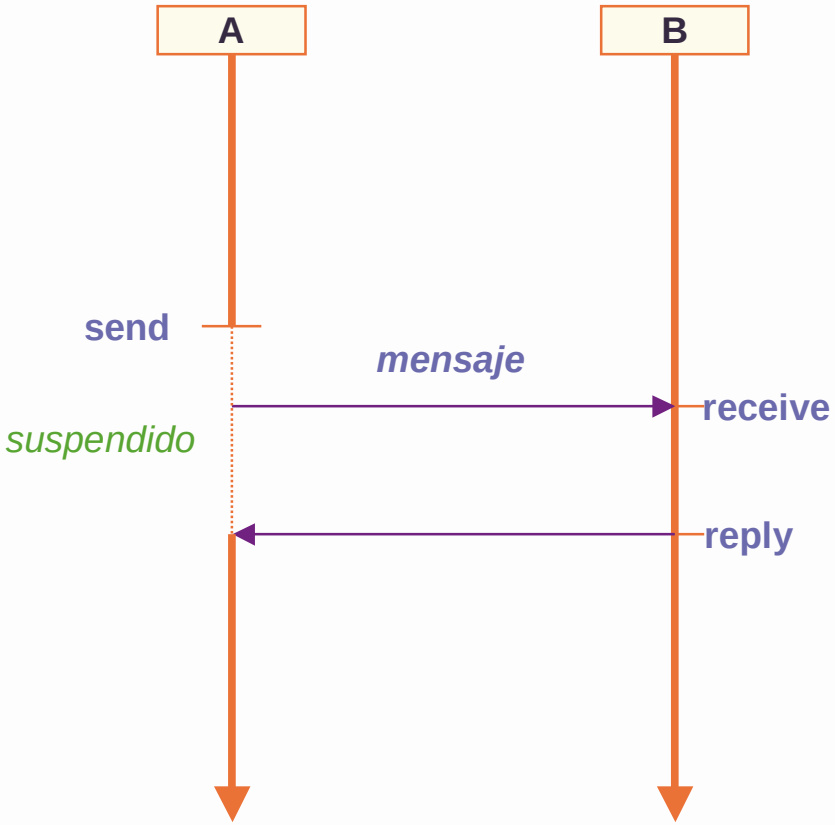


- No hace falta tampón
- Se puede construir a partir de la comunicación asíncrona:

```
A
send(m);
receive(ack);
```

```
B
receive(m);
send(ack);
```

Invocación remota



- No hace falta tampón
- Se puede construir a partir de la comunicación síncrona:

A	B
`s_send(m);`	`s_receive(m);`
	`preparar r`
`s_receive(r);`	`s_send(r);`

Identificación del emisor y el receptor

- Identificación directa o indirecta
  - » *directa*: el emisor identifica explíctamente el receptor

    ```
    send mensaje to proceso
    ```

  - » *indirecta*: se utiliza un intermediario (buzón, canal, tubería, etc.)

    ```
    send mensaje to buzón
    ```

- Simetría
  - » *Comunicación simétrica*: el emisor identifica el receptor, y viceversa

    ```
    send mensaje to proceso (buzón)
    receive mensaje from proceso (buzón)
    ```

  - » *Comunicación asimétrica*: el receptor acepta mensajes de cualquier emisor o buzón
  - » Este tipo de comunicación es adecuado para realizar servidores

Estructura de los mensajes

- Deberían poderse enviar objetos de cualquier tipo
- Es difícil de conseguir en un entorno distribuido
  - diferentes representaciones de tipos de datos
  - problemas con punteros
- Se puede hacer con bibliotecas estándar de "aplanamiento" de tipos
  - convierten cualquier tipo en una secuencia de octetos

Índice

- Comunicación mediante mensajes
- **Comunicación entre tareas en Ada**
  - **cita extendida**
  - espera selectiva
  - llamada selectiva
- Comunicación entre *threads* en C/POSIX

Comunicación entre tareas en Ada

- Se basa en un mecanismo de ***cita extendida***
  - invocación remota directa y asimétrica
- Una tarea puede recibir mensajes a través de ***entradas*** declaradas en su especificación
  - la especificación de una entrada es similar a la de un procedimiento

    Ejemplo:

```
task type Screen is
   entry Put (Char : Character; X,Y : Coordinate);
end Screen;

Display : Screen;
```

  - otras tareas pueden *llamar* a la entrada

```
Display.Put('A',50,24);
```

Entradas

- Puede haber entradas homónimas, siempre que tengan distintos parámetros
  - También puede haber entradas homónimas con subprogramas
- Se pueden definir *familias de entradas* con un discriminante discreto

```
type Channel is range 0..7;
 task Multiplexor is
   entry Get(Channel)(Data : Input_Data);
 end Multiplexor;
```

- Puede haber entradas privadas

```
 task type Telephone_Operator is
   entry Directory_Enquiry (Person : in  Name;
                            Phone  : out Number);
   entry Report_Fault      (Phone  : in  Number);
 private
   entry Allocate_Repair_Worker (Id : out Worker_Id);
 end Telephone_Operator;
```

Llamada

- Para llamar a una entrada hay que identificar la tarea receptora (no hay cláusula *use*)

```
Display.Put('A',50,24);
Multiplexor.Get(3)(X);
Operator.Directory_Enquiry("Juan Pérez", No_de_Juan);
```

- Se puede renombrar una entrada como un procedimiento

```
procedure Enquiry (Person : in  Name; Phone : out Number)
  renames Operator.Directory_Enquiry;
```

- Si se llama a una entrada de una tarea que no está activa, se eleva la excepción `Tasking_Error`

Aceptación (1)

- Para que se lleve a cabo una cita, la tarea receptora debe *aceptar* la llamada al punto de entrada correspondiente

```
accept Put(Char : Character; X,Y : Coordinate) do
  -- escribir Char en la posición (X,Y)
end Put;
```

```
accept Get(3)(Data : Input_Data) do
   -- leer Data del canal 3
end Get;
```

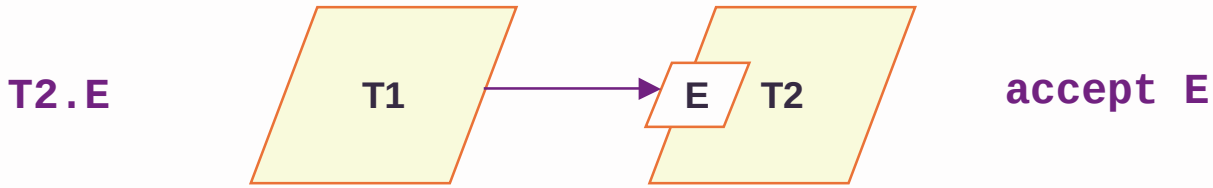
  - Debe haber al menos un *accept* por cada entrada (puede haber más)
  - El índice identifica cuál de las entradas de una familia se acepta
    - » debe haber un *accept* para cada miembro de la familia

Aceptación (2)

- Una instrucción *accept* se puede poner en cualquier lugar del cuerpo de una tarea
  - en particular, se puede poner dentro de otro *accept* (siempre que sea de distinta entrada)
  - no se pude poner en un procedimiento
- El cuerpo del *accept* especifica las acciones que se ejecutan cuando se acepta la llamada
  - La secuencia de instrucciones puede incluir manejadores de excepciones
- Si el cuerpo es nulo, se puede usar una forma simplificada:

```
accept E;
```

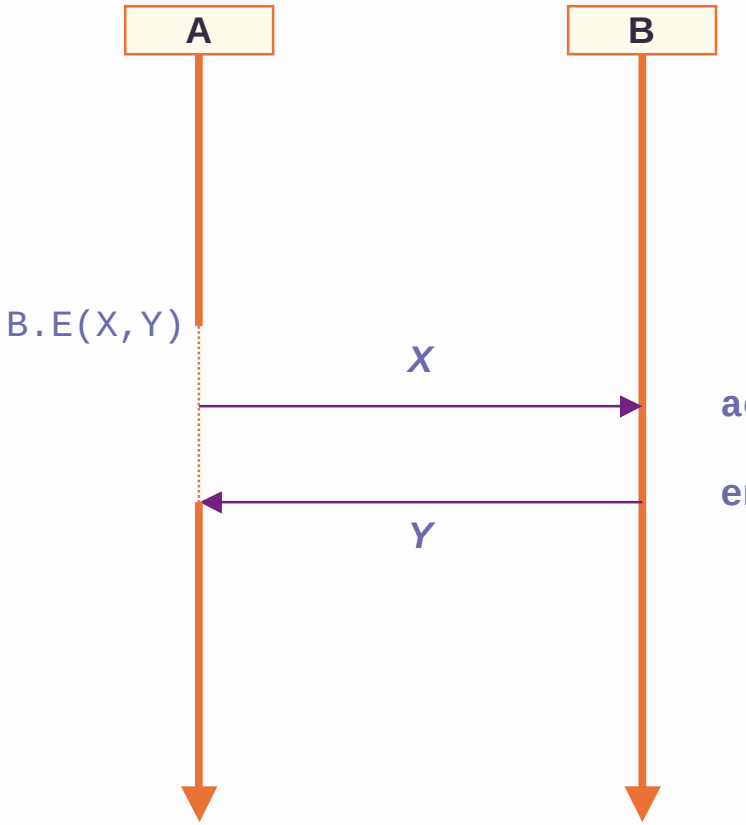
Diagrama de procesos



Ejecución de una cita extendida

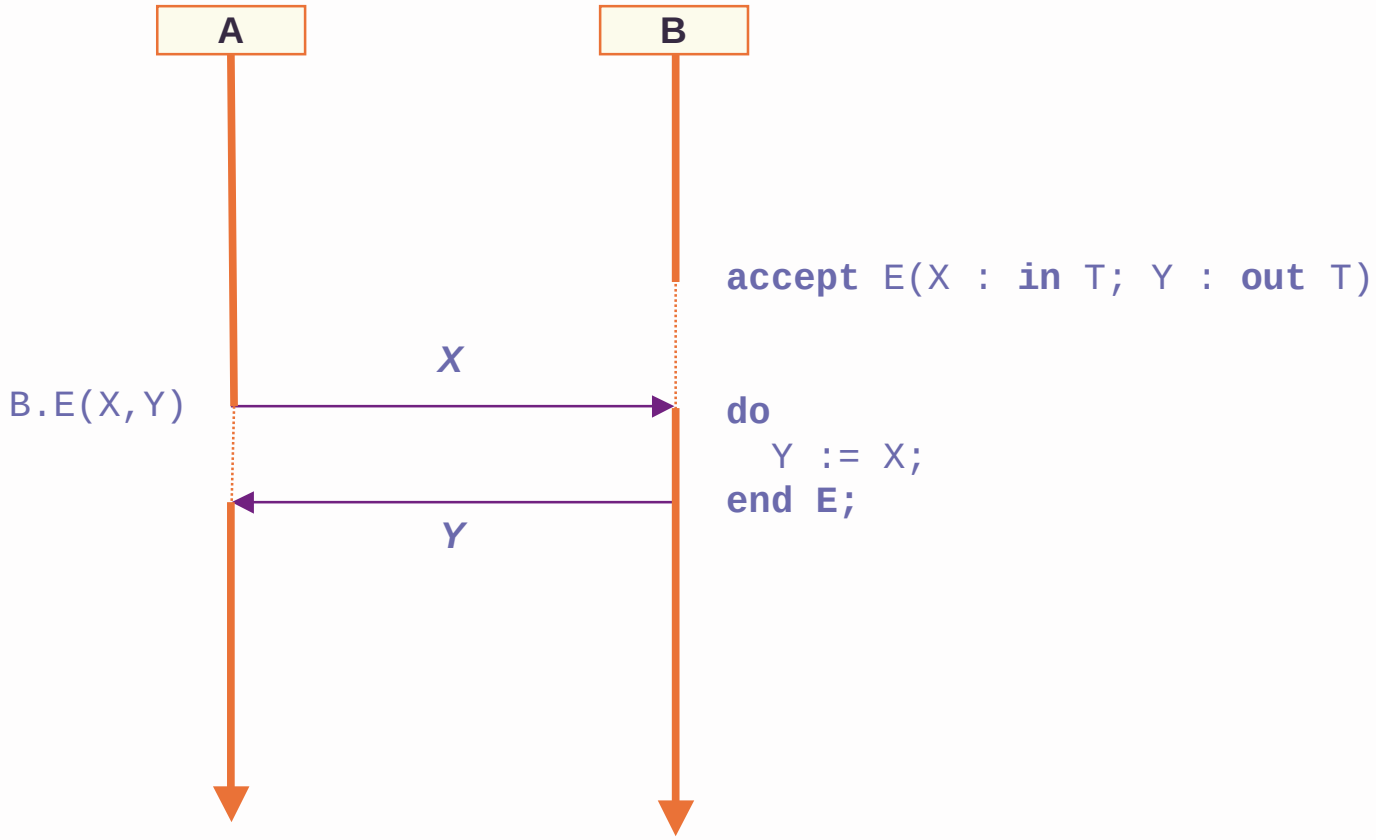
- Las dos tareas deben estar listas para realizar la comunicación.
  - la que llega primero a la cita se suspende hasta que la otra ejecuta la instrucción complementaria (llamada o aceptación)
- Cuando las dos están listas
  - se pasan los parámetros de entrada a la tarea llamada
  - se ejecuta el cuerpo del accept
  - se copian los parámetros de salida al cliente
- A continuación, las dos continúan su ejecución asíncronamente.
- Si varias tareas invocan el mismo punto de entrada de otra tarea, se colocan en una cola
- Una tarea que espera para poder realizar una cita permanece suspendida durante el tiempo que dura la espera

Sincronización (1)




```
accept E(X : in T; Y : out T)  do
  Y := X;
end E;
```

Sincronización (2)



Excepciones en citas

- Puede elevarse una excepción cuando se está ejecutando una cita
  - si hay un manejador en el cuerpo del *accept*, la cita termina normalmente
  - si la excepción no se maneja dentro del *accept*,
    - » la cita termina inmediatamente
    - » la excepción se vuelve a elevar en las dos tareas (puede ser anónima en la que llama)

Ejemplo

```
accept Directory_Enquiry(Person : in  Name;
                         Phone  : out Number) do
  Data_Base.Lookup (Person, Phone_Record);
  Phone := Phone_Record.Phone_Number;
exception
  when Data_Base.Not_Found => Phone := No_Phone;
end Directory_Enquiry;
```

- Si durante la ejecución de `Lookup` se produce la excepción `Not_Found`, se recupera el error dando un valor nulo al parámetro `Phone` y se termina la cita
- El cliente y el servidor continúan normalmente
- Si se produce cualquier otra excepción, la cita termina y la excepción se propaga en los dos, inmediatamente después de la llamada en el cliente, y de la aceptación en el servidor

Índice

- Comunicación mediante mensajes
- **Comunicación entre tareas en Ada**
  - cita extendida
  - **espera selectiva**
  - llamada selectiva
- Comunicación entre *threads* en C/POSIX

Espera selectiva

- A menudo no es posible prever en qué orden se van a invocar las distintas entradas de una tarea
- Esto ocurre cuando sobre todo en las tareas servidoras
  - Un *servidor* es una tarea que acepta llamadas a una o más entradas, y ejecuta un *servicio* para cada una de ellas
  - un *cliente* es una tarea que solicita servicios llamando a las entradas de un servidor
  - Los servidores no saben en qué orden les van a llamar los clientes
    - » deben estar dispuestos a aceptar cualquier llamada cuando no están ocupados
- Es necesario que una tarea pueda esperar simultáneamente llamadas en varias entradas

Aceptación selectiva en Ada

- Es una estructuraa de control que permite la espera selectiva en varias alternativas

```
select
  accept entrada_1 do  -- alternativa_1
      ...
    end entrada_1;
    [secuencia_de_instrucciones]
or
  accept entrada_2 do  -- alternativa 2
      ...
    end entrada_2;
    [secuencia_de_instrucciones]
or
  ...
end select;
```

Ejemplo

```
task body Telephone_Operator is
begin
  loop
    select
      accept Directory_Enquiry (Person : in  Name;
                                Phone  : out Number) do
        -- buscar el número y asignar el valor a Phone
      end Directory_Enquiry;
    or
      accept Report_Fault (Phone : Number) do
        -- avisar al servicio de mantenimiento
      end Report_Fault;
    end select;
  end loop;
end Telephone_Operator;
```

Alternativas guardadas

- A veces es necesario que alguna de las alternativas de una selección se acepte sólo en determinadas condiciones.
- Se pueden poner *guardas* en las alternativas.
- Una guarda es una expresión booleana.

```
when condición => alternativa
```

- Las guardas se evalúan al ejecutar el *select*
  - Las alternativas cuyas guardas son verdaderas se tienen en cuenta para la selección. Se dice que estas alternativas están *abiertas*
  - Las alternativas cuyas guardas son falsas se ignoran. Se dice que estas alternativas están *cerradas*
  - Se considera un error que todas alternativas estén cerradas

Ejemplo

```
task body Telephone_Operator is
begin
  loop
    select
      accept Directory_Enquiry(Person : in  Name;
                               Phone  : out Number) do
        -- buscar el número y asignar el valor a Phone
      end Directory_Enquiry;
    or
      when Today in Weekday =>
        accept Report_Fault (Phone : Number) do
        -- avisar al servicio de mantenimiento
        -- (sólo en días laborables)
        end Report_Fault;
    end select;
  end loop;
end Telephone_Operator;
```

Selección condicional

- Una instrucción *select* puede tener una parte final de la forma:

```
select
  alternativa
{or
  alternativa}
else
  secuencia_de_instrucciones
end select;
```

  - La parte *else* se ejecuta si al llegar al *select* no se puede aceptar inmediatamente ninguna otra alternativa
  - No puede haber parte else y alternativas temporizadas en un mismo *select*
  - La parte *else* no es una alternativa y, por tanto, no puede estar guardada

Alternativa terminate

- Una de las alternativas de un select puede tener la forma:

```
terminate;
```

- Esta alternativa se selecciona cuando
  - el tutor de la tarea ha completado su ejecución
  - todas las tareas que dependen del mismo dueño están terminadas o esperando en un *select* con una alternativa *terminate*
    - » En este caso terminan todas ellas simultáneamente
- Es conveniente que las tareas servidoras terminen así
- La alternativa terminate puede estar guardada
- Es incompatible con las alternativas temporizadas y con la parte *else*

Resumen de la aceptación selectiva

- Se evalúan las guardas; sólo se consideran las alternativas abiertas (guardas verdaderas)
  - si todas las alternativas están cerradas se eleva *Program_Error*
- Si hay llamadas en una o más alternativas abiertas, se elige una de forma indeterminista
  - se ejecuta el *accept* y la secuencia que le sigue, y termina el *select*
- Si no hay llamadas pendientes
  - si hay parte *else* se ejecuta inmediatamente y se termina el *select*
  - si no, la tarea se suspende hasta que llegue una llamada a una de las alternativas abiertas
  - si hay alternativa *terminate* y ya no se pueden recibir más llamadas, termina la tarea

Índice

- Comunicación mediante mensajes
- **Comunicación entre tareas en Ada**
  - cita extendida
  - espera selectiva
  - **llamada selectiva**
- Comunicación entre *threads* en C/POSIX

Llamada condicional

- La llamada condicional permite que un cliente retire su petición si no es aceptada inmediatamente

```
select
   llamada_a_entrada;
   [secuencia_de_instrucciones]
else
   secuencia de instrucciones
end select;
```

  - Si la llamada no se acepta inmediatamente, se abandona y se ejecuta la parte *else*
  - Aquí tampoco puede haber más de una alternativa
  - Sólo se debe usar si la tarea puede realizar trabajo útil cuando no se acepta la llamada

Tareas y exclusión mutua

- Una tarea sólo puede ejecutar una instrucción *accept* en un momento dado.
  - Los cuerpos de las instrucciones *accept* están en exclusión mutua
  - Esto asegura la integridad de los recursos de las tareas servidoras
  - Se puede usar para emular un objeto protegido en Ada 83

Ejemplo

```
task type Server is -- gestiona un recurso global
  entry S1(...);    -- servicio S1
  entry S2(...);    -- servicio S2
  ...               -- etc.
end Server;
```

Esquema de servidor

```
task body Server is
  -- estructura de datos del recurso
begin
  loop
    select
      accept S1 (...) do ... end S1;
    or
      accept S2 (...) do ... end S2; ...
    or
      ...
    end select;
    ...
  end loop;
end Server;
```

Índice

- Comunicación mediante mensajes
- Comunicación entre tareas en Ada
  - cita extendida
  - espera selectiva
  - llamada selectiva
- **Comunicación entre *threads* en C/POSIX**

Mensajes en C/POSIX.1c

- Las *colas de mensajes* permiten efectuar comunicación asíncrona, indirecta y simétrica entre *threads* o procesos
- Una cola de mensajes puede tener varios escritores y lectores
- Las colas tienen nombre
- Cada cola tiene un tampón asociado
  - si se llena, se bloquea el emisor (se puede anular)
  - el receptor se bloquea si el tampón está vacío
  - Si hay varios *threads* bloqueados se reanuda uno arbitrariamente
    - » se puede especificar que se reanude el más prioritario
- Hay más opciones, bastante complicadas

Definiciones

```
/* mqueue.h */

typedef ... mqd_t;  /* message queue descriptor */
struct mq_attr {
  long mq_flags;    /* message queue flags */
  long mq_maxmsg;   /* maximum number of messsages */
  long mq_msgsize;  /* maximum message size */
  long mq_curmsgs   /* number of messages queued */
}

mqdt_t mq_open(const char *name, int oflag);
int    mq_close(mqd_t mqdes);

int mq_send(mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned int msg_prio);
sssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                    size_t msg_len, unsigned int *msg_prio);
```

Resumen

- En Ada las tareas pueden comunicarse por medio de citas, según un modelo de clientes y servidores
  - Una tarea puede tener entradas, que se pueden llamar desde otras tareas
  - La tarea receptora debe aceptar la llamada para que se produzca la cita
  - La tarea que llama espera si la llamada todavía no se ha aceptado
- La aceptación selectiva permite:
  - Esperar simultáneamente llamadas en varias entradas
  - Ejecutar acciones alternativas cuando no hay llamadas pendientes
  - Terminar la ejecución de un servidor cuando ya no es necesario
- La llamada condicional permite evitar que una tarea se quede esperando la aceptación de una cita
- Las colas de mensajes de POSIX permiten efectuar comunicación asíncrona, indirecta y simétrica