dit
UPM

Concurrencia en Ada

Juan Antonio de la Puente
DIT/UPM

Objetivos

Veremos el modelo de concurrencia de Ada

- tareas
- comunicación y sincronización con objetos protegidos
- otras formas de comunicación (citas)

Concurrencia en lenguajes de programación

- La concurrencia puede estar soportada por el lenguaje o solo por el sistema operativo
  - *Ada y Java tienen concurrencia*
  - *C no incluye la concurrencia en el lenguaje*

Ventajas

- Programas más legibles y fáciles de mantener
- Programas más portátiles
- Se pueden detectar errores al compilar
- No hace falta sistema operativo

Inconvenientes

- Distintos modelos de concurrencia según lenguajes
- Puede ser difícil de realizar eficientemente
- Se puede mejorar la portabilidad con SO normalizados

Tareas en Ada

- En Ada, las actividades concurrentes reciben el nombre de **tareas**
  - se tienen que declarar explícitamente
- La tareas se pueden declarar en cualquier zona declarativa
  - Las tareas se crean implícitamente cuando se elabora su declaración
  - Las tareas se empiezan a ejecutar antes que el cuerpo correspondiente a la parte declarativa donde se declaran
- Las tareas se pueden comunicar y sincronizar mediante diversos mecanismos
  - variables compartidas
  - objetos protegidos
  - citas

Tareas y tipos tarea

- Las tareas se pueden declarar como objetos únicos o como objetos de un tipo tarea
  - en el primer caso son de un tipo anónimo

- Las declaraciones de tareas tienen dos partes:
  - **Especificación**: contiene la interfaz visible de la tarea
    - » nombre, parámetros, elementos de comunicación y otras cosas
  - **Cuerpo**: contiene las instrucciones que ejecuta la tarea

- Las tareas y tipos tarea son unidades de programa, pero no de compilación.
  - deben estar declarados en una unidad de compilación (paquete o subprograma).

Ejemplo

```
procedure Example is
   task A; -- especificación
   task B;

   task body A is -- cuerpo
      -- declaraciones locales
   begin
      -- secuencia de instrucciones;
   end A;

   task body B is
      -- declaraciones locales
   begin
      -- secuencia de instrucciones;
   end B;

begin -- A y B empiezan a ejecutarse concurrentemente aquí
   -- secuencia de instrucciones de Example
   -- se ejecuta concurrentemente con A y B
end Example; -- no termina hasta que terminen A y B
```

Declaración de tipos tarea

```
task type A_Type;
task type B_Type;

A : A_Type;  -- se pueden declarar objetos en cualquier punto
B : B_Type;  -- donde sea visible la declaración

task body A_Type is
   ...
end A_Type;

task body B_Type is
   ...
end B_Type;
```

Estructuras

```
task type T;

A, B : T;

type Long is array (1 .. 100) of T;

type Mixture is
record
   Index  : Integer;
   Action : T;
end record;

task body T is
   ...
end T;
```

Discriminantes

```
procedure Main is
   type Dimension is (Xplane, Yplane, Zplane);
   task type Control (Dim : Dimension);
     -- Dim es un discriminante;
     -- solo puede ser de un tipo discreto o de acceso
   C1 : Control (Xplane);
   C2 : Control (Yplane);
   C3 : Control (Zplane);

   task body Control is
      Position : Integer; -- posición absoluta
      Setting  : Integer; -- movimiento relativo
   begin
      Position := 0;
      loop
         New_Setting (Dim, Setting);
         Position := Position + Setting;
         Move_Arm (Dim, Position);
      end loop;
   end Control;
begin
   null;
end Main;
```

Tareas dinámicas

```
procedure Example is

   task type T;
   type A is access T;

   P : A;
   Q : A := new T; -- aquí se crea una nueva tarea
                   -- que se ejecuta inmediatamente;
                   -- la nueva tarea es Q.all
   task body T is ...

begin
   ...
   P := new T;       -- se crea otra tarea
   Q := new T;       -- y otra más
   -- la antigua Q.all sigue ejecutándose (pero es anónima)
   ...
end Example;
```

Identificación de tareas

- Los tipos acceso permiten dar un nombre a las tareas
- Los tipos tarea son privados y limitados

```
task type T;
type A is access T;

P : A := new T;
Q : A := new T;
...
P.all := Q.all; -- ilegal
P     := Q;     -- legal
```

- A veces se hace complicado
  - un mismo nombre puede designar tareas diferentes
- Es útil tener un *identificador* único para cada tarea

Ada.Task_Identification

```
package Ada.Task_Identification is

   -- definido en el anexo C - programación de sistemas

   type Task_Id is private;
   Null_Task_Id : constant Task_Id;

   function  "=" (Left, Right : Task_ID) return Boolean;

   function  Image (T : Task_ID) return String;

   function  Current_Task return Task_ID;

   procedure Abort_Task   (T : in out Task_ID);

   function  Is_Terminated(T : Task_ID) return Boolean;
   function  Is_Callable  (T : Task_ID) return Boolean;

private
   ...
end Ada.Task_Identification;
```

Además, *T'Identity* proporciona la identidad de cualquier tarea

Activación, ejecución, finalización

- La ejecución de una tarea pasa por tres fases
  - Activación: se elabora la zona declarativa del cuerpo de la tarea
    - » se crean y se inician los objetos locales
  - Ejecución: se ejecuta la secuencia de instrucciones del cuerpo de la tarea
  - Finalización: se ejecutan operaciones finales en los objetos locales

Activación

- Consiste en la elaboración de las declaraciones locales de una tarea
  - las tareas estáticas se activan justo antes de empezar la ejecución del padre
  - las tareas dinámicas se activan al ejecutarse el operador *new*
- El padre espera a que termine la activación de las tareas que ha creado
  - pero los hijos no se esperan unos a otros
- Si se produce una excepción durante la activación, se eleva *Tasking_Error* en el padre

Activación de tareas dinámicas

- Las tareas dinámicas se activan inmediatamente después de la evaluación del operador **new** que las crea
- La tarea que ejecuta **new** espera también a que termine la activación de sus hijos dinámicos


```
declare
   task type T;
   type A is access T;
   P : A;
   task body T is ...
begin
   P := new T;       -- Se crea y se activa P.all
   ...               -- Las siguientes instrucciones no se ejecutan
                     -- hasta que se ha activado P.all
end;
```

Terminación

- Una tarea esta *completada* cuando termina la ejecución de su secuencia de instrucciones
- Una tarea *termina* cuando deja de existir
- Una tarea puede terminar de varias formas
  - Completando la ejecución de su cuerpo, normalmente o por una excepción sin manejar
  - Poniéndose de acuerdo con otras tareas para terminar conjuntamente (lo veremos más adelante)
  - Siendo abortada por otra tarea o por sí misma
    - » mediante la instrucción **abort** T

- El atributo booleano *T'Terminated* es verdadero cuando *T* ha terminado

Terminación de tareas dinámicas

- El *tutor* (en Ada se llama *master* o *amo*) de una tarea dinámica es el bloque donde se declara el tipo acceso
  - la tarea dinámica es un *pupilo* de su tutor
- Una tarea no puede terminar hasta que han terminado todos sus pupilos

```
declare
   task type T;
   type A is access T;
begin
   ...
   declare -- bloque interior
      X : T;           -- el amo de X es el bloque interior
      Y : A := new T; -- el amo de Y.all es el bloque exterior
   begin
      ...
   end;                -- espera a que termine X, pero no Y.all
   ...                 -- Y.all es anónima aquí
end;                   -- ahora espera a que termine Y.all
```

Aborto

- Una tarea puede abortar otra tarea cuyo nombre es visible mediante la instrucción

```
abort T;
```

- Cuando se aborta una tarea, también se abortan todos sus pupilos
- Problema: no se puede abortar una tarea anónima
- Se puede abortar una tarea identificada mediante el procedimiento *Ada.Task_Identification.Abort_Task*

Tareas y unidades de biblioteca

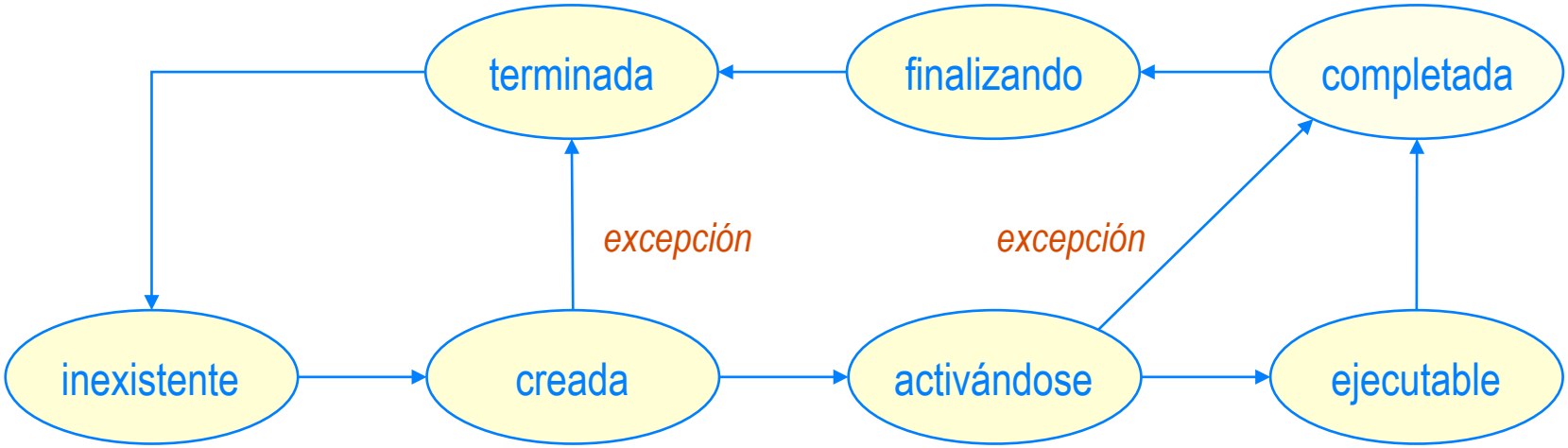
- El amo de las tareas declaradas en paquetes de biblioteca es la *tarea de entorno*
  - es la tarea inicial que elabora todos los paquetes de biblioteca y llama al procedimiento principal
- El amo de las tareas dinámicas cuyo tipo de acceso está declarado en un paquete de biblioteca también es la tarea de entorno
- El procedimiento principal de un programa Ada (y el programa en sí) no puede terminar hasta que hayan terminado todas las tareas de biblioteca

Ejemplo

```
package P is                    -- paquete de biblioteca
  task type T;
  A : T;                        -- tarea de biblioteca
end P;

package body P is
  task body T is ...;
  B : T;                        -- otra tarea de biblioteca
  ...
end P;
-- A y B se activan al elaborarse el paquete
-- por la tarea de entorno
```

```
with P;
procedure Main is
begin
  null;                         -- se ejecutan concurrentemente A y B
end Main;                       -- el programa no termina hasta que
                                -- terminan A y B
```

Estados de las tareas en Ada



Objetos protegidos en Ada

- Un objeto protegido es un objeto compuesto cuyas operaciones se ejecutan en exclusión mutua
  - se pueden declarar tipos protegidos u objetos protegidos únicos
  - en ambos casos hay *especificación* y *cuerpo*.
    - » la especificación contiene la interfaz visible desde otras unidades de programa.
  - los tipos y objetos protegidos son unidades de programa, pero no de compilación.
    - » deben estar declarados en una unidad de compilación (paquete o subprograma).

Ejemplo: entero compartido (1)

```
protected type Shared_Integer(Initial_Value : Integer) is

  function Value return Integer;
  procedure Change(New_Value : Integer);

private
  Data : Integer := Initial_Value;
end Shared_Integer;
```

- Las operaciones (subprogramas) se declaran en la parte pública
- Los detalles de implementación del tipo van en la parte privada
  - solo son visibles en el cuerpo
  - no se pueden declarar tipos de datos
- Es parecido a un registro
- Puede tener discriminantes
  - tienen que ser de tipos discretos o acceso

Ejemplo: entero compartido (2)

```
protected body Shared_Integer is

   function Value return Integer is
   begin
      return Data;
   end Value;

   procedure Change(New_Value : Integer) is
   begin
      Data := New_Value;
   end Change;

end Shared_Integer;
```


- Los cuerpos de las operaciones van en el cuerpo del tipo protegido
- No se pueden declarar tipos ni objetos en el cuerpo

Ejemplo: entero compartido (3)

```
declare
   X : Shared_Integer(0);
   Y : Shared_Integer(1);
begin
   X.Change(Y.Value + 1); -- ahora X.Value es 2
end;
```

- No hay cláusula *use* para los objetos protegidos
- Las operaciones deben ir siempre cualificadas con el nombre del objeto

Subprogramas protegidos

- Con los objetos protegidos solo se pueden efectuar *operaciones protegidas*
  - Un *procedimiento protegido* da acceso exclusivo en lectura y escritura a los datos privados.
  - Una *función protegida* da acceso concurrente en lectura sólo a los datos privados
- Se pueden ejecutar concurrentemente varias llamadas a una función protegida, pero no a un procedimiento protegido, ni a ambos a la vez.
- El núcleo de ejecución realiza la exclusión mutua mediante mecanismos más primitivos
  - ¡ojo! una tarea que está intentando acceder a un objeto protegido no se considera suspendida
    - » la operaciones protegidas son cortas y no se pueden suspender

Entradas protegidas y sincronización condicional

- Una *entrada* es una operación protegida con una interfaz semejante a la de un un procedimiento

```
entry E (...);
```

- En el cuerpo se le asocia una *barrera* booleana

```
entry E (...) when B is ...
```

  - si la barrera es falsa cuando se invoca la operación, la tarea que llama se suspende en una cola de espera
  - cuando la barrera es verdadera, la tarea puede continuar
- Las entradas se usan para realizar sincronización condicional

Ejemplo: productor y consumidor (1)

```
-- tampón limitado

Size   : constant Positive := 32;
...
protected type Bounded_Buffer is
  entry Put(X : in  Item);
  entry Get(X : out Item);
private
  First   : Index := Index'First;
  Last    : Index := Index'Last;
  Number  : Count := 0;
  Store   : Buffer_Store;
end Bounded_Buffer;

Buffer : Bounded_Buffer;
```

Productor y consumidor (2)

```
protected body Bounded_Buffer is

  entry Put (X : in Item) when Number < Size is
  begin
    Last         := Last + 1;
    Store(Last)  := X;
    Number       := Number + 1;
  end Put;

  entry Get (X : out Item) when Number > 0 is
  begin
    X            := Store(First);
    First        := First + 1;
    Number       := Number - 1;
  end Get;

end Bounded_Buffer;
```

Productor y consumidor (3)

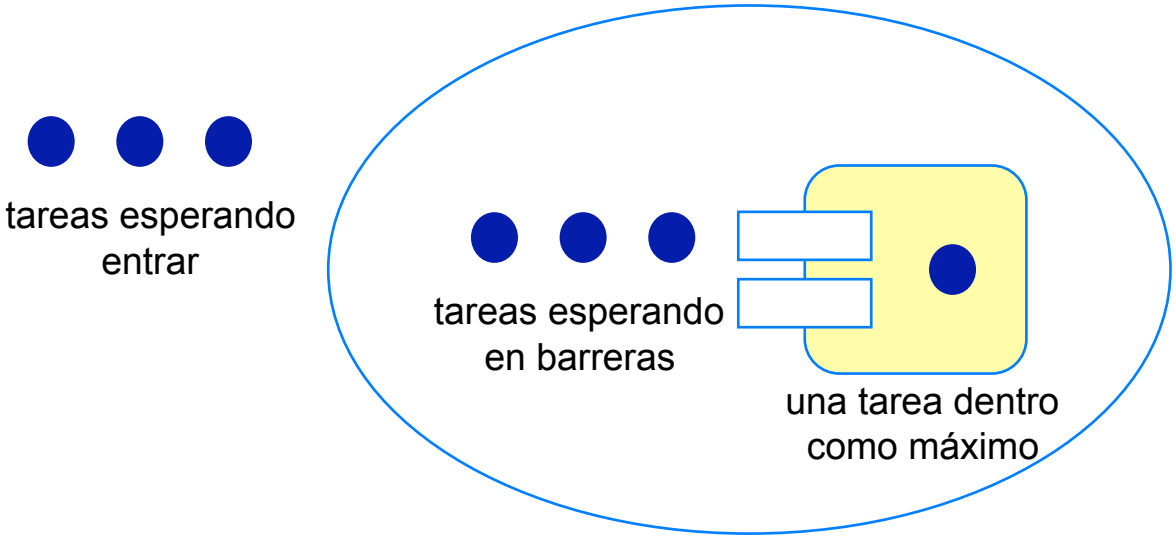
```
procedure Producer_Consumer is

   task Producer;
   task Consumer;

   task body Producer is
     X : Item;
   begin
     loop
       Produce(X);
       Buffer.Put(X);
     end loop;
   end Producer;

   task body Consumer is
     X : Item;
   begin
     loop
       Buffer.Get(X);
       Consume(X);
     end loop;
   end Consumer;

end Producer_Consumer;
```

Evaluación de las barreras

- Las barreras se evalúan cuando
  - una tarea llama a una entrada y la barrera hace referencia a una variable que puede haber cambiado desde la última evaluación
  - una tarea termina la ejecución de un procedimiento o entrada y hay tareas esperando en entradas cuyas barreras hacen referencia a variables que pueden haber cambiado desde la última evaluación
- No se deben usar variables globales en las barreras
- La corrección de un programa no debe depender del momento en que se evalúan las barreras (se puede hacer con más frecuencia de lo indicado).

Exclusión mutua y barreras

- Las tareas esperando en barreras tienen preferencia sobre las que esperan acceder al objeto
  - de esta forma se evitan condiciones de carrera en la implementación
- Este modelo se llama *cáscara de huevo* (*eggshell*)

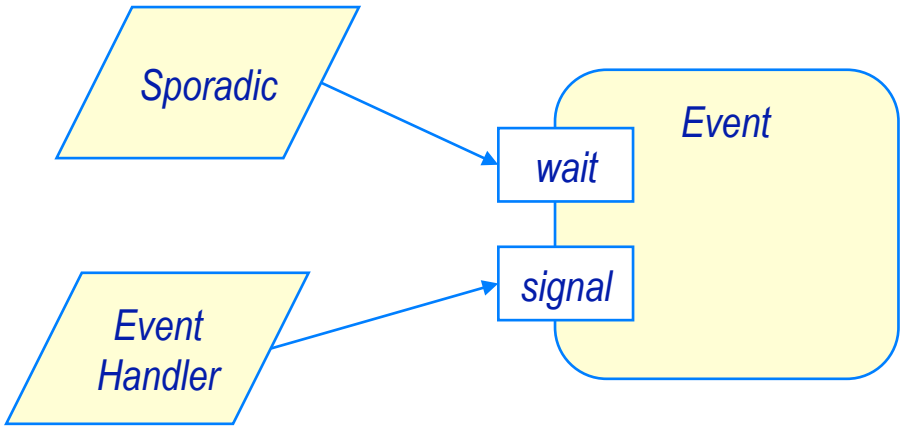


Restricciones

- En el cuerpo de una operación protegida no se pueden ejecutar operaciones "potencialmente bloqueantes":
  - llamadas a entradas
  - retardos
  - creación o activación de tareas
  - llamadas a subprogramas que contengan operaciones potencialmente bloqueantes
- Tampoco puede haber llamadas al mismo objeto

- El objetivo de estas restricciones es evitar que una tarea se quede suspendida indefinidamente en el acceso a un subprograma protegido

Ejemplo: tarea esporádica



Ejemplo: tarea esporádica (2)

```
task Sporadic;
task Event_Handler;

protected Event is
  entry Wait;    -- uno solo esperando como máximo
  procedure Signal;
private
  Occurred : Boolean := False;
end Event;

protected body Event is
  entry Wait when Occurred is
  begin
    Occurred := False;
  end Wait;

  procedure Signal is
  begin
    Occurred := True;
  end Signal;
end Event;
```

Ejemplo: tarea esporádica (3)

```
task body Sporadic is
begin
   loop
      Event.Wait;
      -- acción esporádica
   end loop;
end Sporadic;

task body Event_Handler is
begin
   ...
 Event.Signal;
   ...
end Event_Handler;
```

Objetos de suspensión

- Proporcionan una funcionalidad similar

```
package Ada.Synchronous_Task_Control is
   type Suspension_Object is limited private;
   procedure Set_True  (S : in out Suspension_Object);
   procedure Set_False (S : in out Suspension_Object);
   function  Current_State (S : Suspension_Object)
     return Boolean;
   procedure Suspend_Until_True (S : in out Suspension_Object);
   -- Raises Program_Error if more than one task tries to
   -- suspend on S at once
   -- Sets S to False
private
   ...
end Ada.Synchronous_Task_Control;
```

Ejemplo: tarea esporádica

```
...
use  Ada.Synchronous_Task_Control;
Event : Suspension_Object;

task body Sporadic is
begin
   loop
      Suspend_Until_True (Event);
      -- acción esporádica
   end loop;
end Sporadic;

task body Event_Handler is
begin
   ...
   Set_To_True (Event);
   ...
end Event_Handler;
```

Reencolamiento

- Las barreras no pueden depender de los parámetros de la llamada
- Para conseguir el mismo efecto se permite que una llamada a una entrada que ya ha sido aceptada pueda volver a encolarse en otra entrada mediante una instrucción **requeue**:

```
requeue entrada [with abort];
```

  - La nueva entrada tiene que tener un perfil de parámetros compatible con la llamada en curso, o bien no tener parámetros
  - La primera llamada se completa al hacer el *requeue*
  - La cláusula *with abort* permite que se cancele la llamada cuando hay temporizaciones, o cuando se aborta la tarea que llama

Ejemplo: suceso radiado (1)

```
protected type Event is
  entry Wait;   -- ahora puede haber varios esperando
  entry Signal;
private
  entry Reset;
  Occurred : Boolean := False;
end Event;
```

Suceso radiado (2)

```
protected body Event is

  entry Wait when Occurred is
  begin
    null;  -- sólo sincronización
  end Wait;

  entry Signal when True is -- barrera obligatoria
  begin
    if Wait'Count > 0 then
      Occurred := True;
      requeue Reset;
    end if;
  end Signal;

  entry Reset when Wait'Count = 0 is
  begin
    Occurred := False;
  end Reset;

end Event;
```

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

```
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

- 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);
Operator.Directory_Enquiry("Juan Pérez", No_de_Juan);
```

- 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)

Aceptación (2)

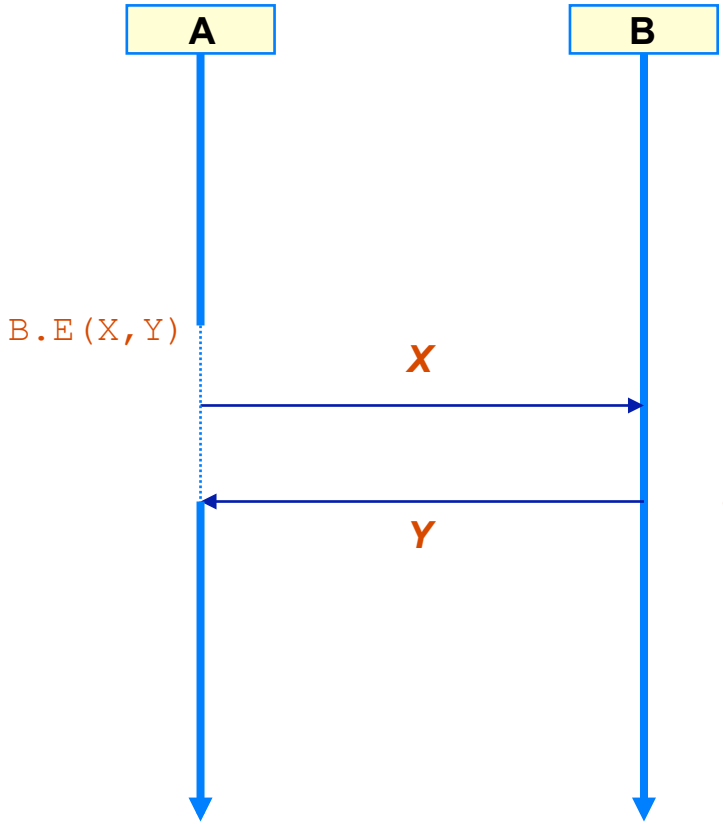
- Una instrucción *accept* se puede poner en cualquier lugar del cuerpo de una tarea
  - en particular, se puede poner dentro de otro *accept* (siempre que sea de distinta entrada)
  - no se pude poner en un procedimiento
- El cuerpo del *accept* especifica las acciones que se ejecutan cuando se acepta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

```
accept E;
```

Ejecución de una cita extendida

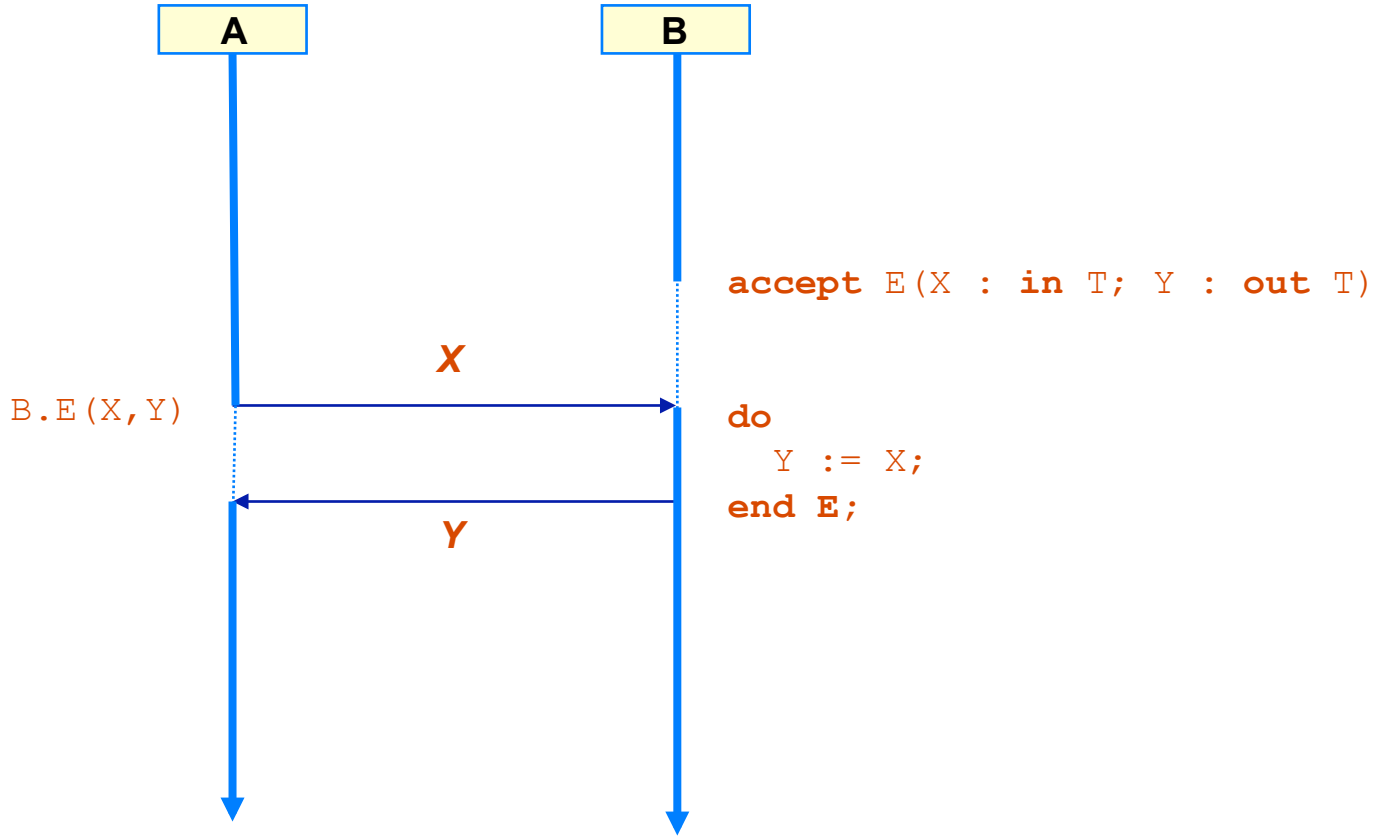
- Las dos tareas deben estar listas para realizar la comunicación.
  - la que llega primero a la cita se suspende hasta que la otra ejecuta la instrucción complementaria (llamada o aceptación)
- Cuando las dos están listas
  - se pasan los parámetros de entrada a la tarea llamada
  - se ejecuta el cuerpo 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 eleva 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 eleva 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

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 estructura 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

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

Transferencia de control asíncrona

- Es una forma especial de select:

```
select
   suceso;
   secuencia de instrucciones
then abort
   secuencia de instrucciones
end select;
```

- El suceso puede ser una llamada a una entrada
  - entrada protegida o entrada de tarea

El perfil de Ravenscar

- El modelo de concurrencia de Ada es muy extenso
  - flexible, pero complejo
- El perfil de Ravenscar es un subconjunto de la parte concurrente de Ada para aplicaciones críticas
  - estándar en Ada 2005
- Estrategia:
  - eliminar elementos con tiempo de ejecución excesivo o imprevisible
  - permitir el análisis temporal del sistema
  - facilitar la implementación de la concurrencia mediante un núcleo de tiempo real pequeño, eficiente y fiable

Modelo de tareas de Ravenscar

- Tareas y objetos protegidos estáticos
  - no hay creación dinámica ni declaraciones anidadas
  - las tareas no terminan
- Objetos protegidos con una entrada, como máximo, con
  - barrera simple (variable booleana declarada en el mismo objeto)
  - una tarea como máximo esperando que se abra la barrera
- Control de tareas síncrono (objetos de suspensión)
- No hay citas, ni ningún tipo de instrucción *select*

La adecuación al modelo se puede comprobar al compilar mediante restricciones (excepto terminación y colas)

Restricciones del perfil de Ravenscar

```
No_Task_Hierarchy
No_Task_Allocators
No_Task_Termination
No_Local_Protected_Objects
No_Protected_Type_Allocators
Simple_Barriers
Max_Entry_Queue_Length => 1
Max_Task_Entries => 0
Max_Protected_Entries => 1
No_Select_Statements
No_Requeue_Statements
```

Pragma *Profile*

- Resumen de todas las restricciones

```
pragma Profile (Ravenscar)
```

equivale a todas las restricciones anteriores (y algunas más que veremos más adelante)

- Es un *pragma de configuración*
  - afecta a todo el programa
  - en GNAT se coloca en un fichero especial (*gnat.adc*)

Resumen

- Ada tiene un modelo de tareas abstracto, flexible y completo
  - Las tareas se pueden sincronizar y comunicar con objetos protegidos que encapsulan datos compartidos
  - También pueden efectuar citas
  - Los mecanismos de aceptación selectiva, llamada condicional y transferencia asíncrona de control añaden flexibilidad
- El perfil de Ravenscar define un modelo más sencillo
  - comportamiento temporal previsible
  - fácil de implementar con un núcleo reducido