Programación concurrente en Ada. Breve introducción

Joaquín EZPELETA
Dept. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza

Índice

- Declaración y uso de tareas Ada
- Dependencia y activación de tareas Ada
- Sincronización por citas en Ada
- La instrucción **`Select`**
- Tipo **`Duration`** y acción **`Delay`**
- Paquete **`Calendar`** y tipo **`Time`**
- Uso de **`Select`** y citas
- Manejo de excepciones
- Más sincronización en ADA: tipos protegidos
- La “buena educación” en el uso de objetos protegidos

Declaración y uso
d

```
Vars en1:Bool:=FALSE
     en2:Bool:=FALSE
     ult:Ent:=1
P1::
Mq TRUE
   en1:=TRUE
   ult:=1
   Mq en2 ∧ ult=1
      seguir
   FMq
   secCrit1
   en1:=FALSE
   secNoCrit1
FMq
```

```
with Ada.Text_IO; use Ada.Text_IO;
procedure peterson is
   en1: boolean := FALSE;
   en2: boolean := FALSE;
   ult: integer := 1;

   task P1;
   task P2;
-----------------------------------
   task body P1 is
   begin
      while TRUE loop
         en1 := TRUE
         ult := 1
         while en2 and ult=1 loop
            null;
         end loop;
         --put_line("P1 en CS");
         en1 := FALSE;
         --put_line("P1 en NCS");
      end loop;
   end P1;
```

```
---
begin
   null;
end peterson;
```

Declaración y uso de tareas Ada

```
with Ada.Text_IO; use Ada.Text_IO;
procedure peterson is
   en1: boolean := FALSE;
   en2: boolean := FALSE;
   ult: integer := 1;

   task P1;
   task P2;
-----------------------------------
   task body P1 is
   begin
      while TRUE loop
         en1 := TRUE
         ult := 1
         while en2 and ult=1 loop
            null;
         end loop;
         --put_line("P1 en CS");
         en1 := FALSE;
         --put_line("P1 en NCS");
      end loop;
   end P1;
```

declaración

```
   task body P2 is
      begin
         while TRUE loop
            en2 := TRUE
            ult := 2
            while en1 and ult=2 lo
               null;
            end loop;
            --put_line("P2 en CS")
            en2 := FALSE;
            --put_line("P2 en NCS"
         end loop;
   end P2;
-----------------------------------
begin
   null;
end peterson;
```

Declaración y uso de tareas Ada

```
task type T(...) is
  ...
end  T;
type R is record
   laTarea: T;
   ...
end record;
type ptTask is access T;
```

**tipo tarea*
**parametrizada*
(sólo discretos y punteros)

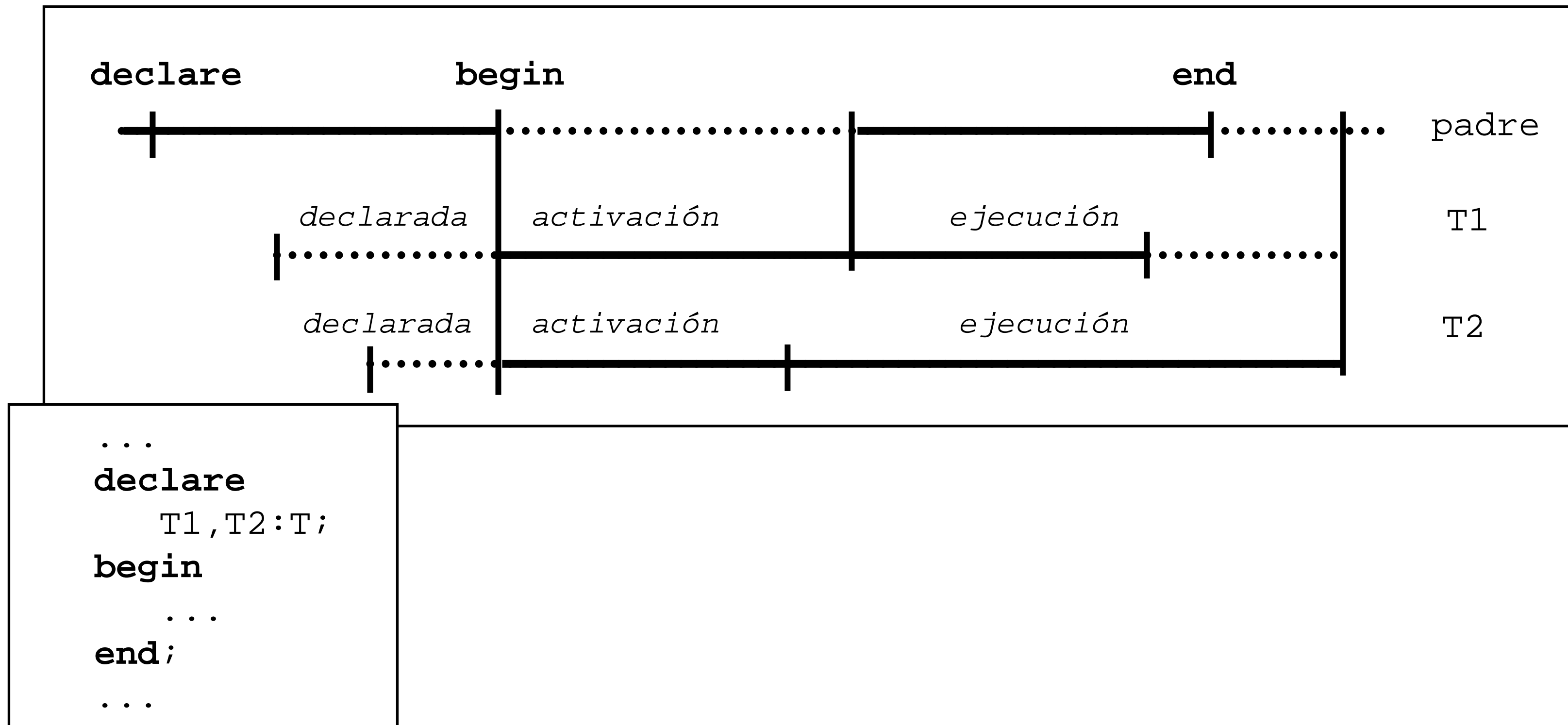
```
   vectorDeTareas: array(1..10) of T;
   vectorDeRegistros: array(1..3) of R;
   reg: R;
   pt: ptTask;
begin
   ...
   declare
      T1:T(v);
      T2:T(v');
   begin
      ...
   end;
   pt := new T(v'');
   ...
end;
```

** vectores de tareas*
** punteros a tareas*

Dependencia y activación de tareas Ada

- Una tarea declarada en un procedimiento, bloque u otra tarea (en la parte de implementación) **depende** de dicha unidad de programa.
  - La **activación** de una tarea es **automática** al comenzar la ejecución de la unidad en la que está declarada.
  - Esta unidad **no acaba** hasta que **todas las tareas** declaradas en ella **han concluido su ejecución**
  - Una tarea creada dinámicamente se activa en el momento de su creación, y depende del bloque que contiene la declaración
- Comunicación entre tareas:
  - mediante **variables globales** compartidas entre varias tareas
  - mediante el mecanismo de **cita** (***rendez-vous***).
    - una cita puede estar parametrizada para permitir la comunicación entre tareas

Dependencia y activación de tareas Ada



Sincronización por citas en Ada

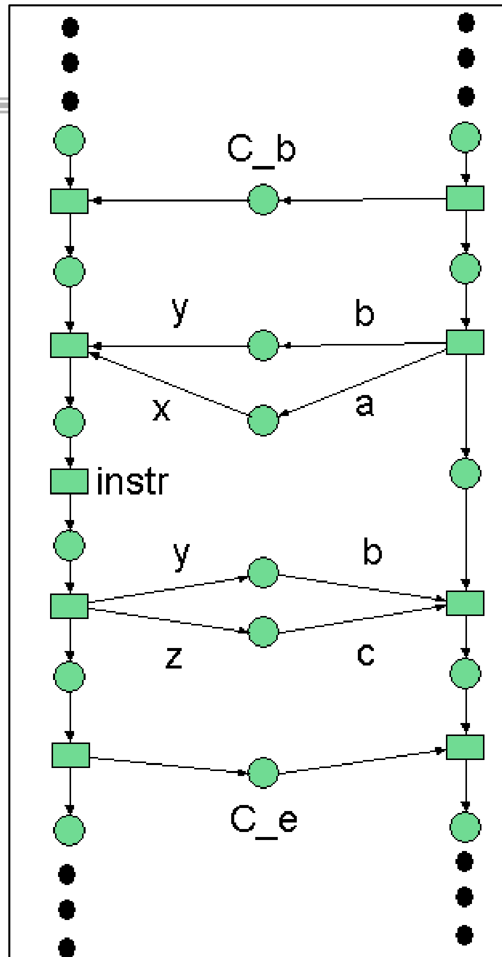
- Ejemplos de uso de citas:

```
task T1 is
   entry C1;
   entry C2(n: in integer;
            r: out real);
   ...
end T1;
task body T1 is
begin
   ...
   accept C2(n,r) do
      ...
   end C2;
   ...
   accept C1;
   ...
end T1;
```

```
...
task body T2 is
   ...
   T1.C1;
   ...
end T2;
```

```
...
task body T3 is
   ...
   T1.C2(h,s);
   ...
end T3;
```

citas en Ada




```
  entry C(x: in ...;
          y: in out ...;
          z: out ...);
   ...
task body T1 is
begin
   ...
   accept C(x,y,z) do
      instr
   end C2;
   ...
end T1;
```

```
task body T2 is
begin
   ...
   T1.C(a,b,c);
   ...
end T1;
```

La instrucción `Select`

¡¡Dos problemas!!

```
task buffer is
   entry leer(D: out Dato);
   entry escribir(D: in Dato);
end buffer;
task T1;
task T2;
task body buffer is
   N: constant := 100;   --maxDatos
   X: array(1..N) of Dato;
   i,j: integer range 1..N := 1;
   nDatos: integer range 0..N := 0;
begin
   loop
      ...
   end loop;
end buffer;
```

```
select
   accept escribir(D: in Dato) do
      X(i):=D;
      i:=i mod N+1;
      nDatos:=nDatos+1;
   end escribir;
or
   accept leer(D: out Dato) do
      D:=X(j);
      j:=j mod N+1;
      nDatos:=nDatos-1;
   end leer;
end select;
```

parámetros actuales se evalúan antes

```
...
buffer.leer(d1)
...
```
T1

```
...
buffer.escribir(d2)
...
```
T2

La instrucción **Select**

```
task buffer is
   entry leer(D: out Dato);
   entry escribir(D: in Dato);
end buffer;

task body buffer is
   N: constant := 100;    --maxDatos
   X: array(1..N) of Dato;
   i,j: integer range 1..N := 1;
   nDatos: integer range 0..N := 0;
begin
   loop
      ...
   end loop;
end buffer;
```

```
select
   accept escribir(D: in Dato) do
      X(i):=D;
   end escribir;
   i:=i mod N+1;
   nDatos:=nDatos+1;
or
   accept leer(D: out Dato) do
      D:=X(j);
   end leer;
   j:=j mod N+1;
   nDatos:=nDatos-1;
end select;
```

¡¡Un problema!!

La instrucción **Select**

```
task buffer is
   entry leer(D: out Dato);
   entry escribir(D: in Dato);
end buffer ;

task body buffer is
   N: constant := 100;    --maxDatos
   X: array(1..N) of Dato;
   i,j: integer range 1..N := 1;
   nDatos: integer range 0..N := 0;
begin
   loop
      ...
   end loop;
end buffer;
```

```
select
   when nDatos<N  =>
     accept escribir(D: in Dato) do
        X(i):=D;
     end escribir;
   i:=i mod N+1;
   nDatos:=nDatos+1;
or
   when nDatos>0  =>
     accept leer(D: out Dato) do
        D:=X(j);
     end leer;
   j:=j mod N+1;
   nDatos:=nDatos-1;
end select;
```

Tipo **Duration** y acción **Delay**

- Ada tiene predefinido el tipo **`Duration`**
  - un tipo numérico de coma fija
  - un dato de este tipo representa intervalos de tiempo medidos en segundos.
- El rango de valores representables depende de la implementación
  - al menos, debe posibilitar representar valores positivos y negativos de un día: **`[-86400.0 .. 86400.0]`**
  - Precisión: **`Duration'small`** no debe superar los 20 ms, aunque se recomienda que no sobrepase los 50 µs.
- Ada ofrece la posibilidad de programar esperas mediante la acción

```
delay <tiempoDeEspera>
```

  - **`<tiempoDeEspera>`** de tipo Duration, medido en segundos
  -
- Ejemplo:

```
delay 2.0
```

  - provoca la suspensión de la tarea durante, ***al menos***, 2.0 segundos.

¿Por qué "*al menos*"?

Paquete **`Calendar`** y tipo **`Time`**

- El tipo **`Time`** es un TAD, especificado en el módulo predefinido **`Calendar`**, que representa *valores absolutos del tiempo*.

```
package calendar is   --  especificación
   type TIME is private;
   subtype year_number is integer range 1901 .. 2099;
   subtype month_number is integer range 1 .. 12;
   subtype day_number is integer range 1 .. 31;
   subtype day_duration is duration range 0.0 .. 86400.0;

   function clock return time;
     -- devuelve el valor del tiempo actual

...
```

Paquete **Calendar** y tipo **Time**

```
function year(date: time) return year_number;
   --  devuelve el año de date
function month(date: time) return  month_number;
   --  devuelve el mes de date
function day(date: time) return  day_number;
   --  devuelve el dia de date
function seconds(date: time) return  day_duration;
   --  devuelve el número de segundos
   --  transcurridos en el día de date
procedure split(date: in time;year: out year_number;
           month: out month_number;
           day: out day_number; second: out day_duration);
   --  descompone date
function time_of(year: in year_number;
           month: in month_number;
           day: in day_number;
           second: in day_duration) return time;
--  compone un tiempo y devuelve su valor
```

Paquete **Calendar** y tipo **Time**

```
function "+"(left:time; right: duration) return time;
function "+"(left:duration; right: time) return time;
function "-"(left:time; right: duration) return time;
function "-"(left:duration; right: time) return duration;
function "<"(left:time; right: time) return boolean;
function "<=“(left:time; right: time) return boolean;
function ">"(left:time; right: time) return boolean;
function ">="(left:time; right: time) return boolean;
Time_error: exception;
```

Paquete **Calendar** y tipo **Time**

```
with calendar;  use calendar;

task rutina;
task body rutina is
   periodo: constant
            duration := 10.0;
begin
   loop
      delay periodo;
      -- acción a iterar
      ...
   end loop;
end rutina;
```

```
with calendar;  use calendar;
task  rutina;
task body rutina is
   periodo: constant duration := 10.0;
   siguiente: time;
begin
   siguiente := clock+periodo;
   loop
      delay siguiente-clock;
      -- acción a iterar
      ...
      siguiente := siguiente+periodo;
   end loop;
end rutina;
```

Uso de **Select** y citas

```
select
   accept cita1(...) do
      -- acción a ejecutar
      -- durante la cita1
      ...
   end cita1;
   ...
or
   accept cita2(...) do
      -- acción a ejecutar
      -- durante la cita2
      ...
   end cita1;
   ...
or
   delay 5.0; -- timeout
   ...
end select;
```

Si en 5.0 sgs. no recibo petición...

```
select
   accept cita1(...) do
      -- acción a ejecutar
      -- durante la cita1
      ...
   end cita1;
   ...
or
   accept cita2(...) do
      -- acción a ejecutar
      -- durante la cita2
      ...
   end cita1;
   ...
else
   -- acción a ejecutar
   -- alternativamente
   ...
end select;
```

```
or
   delay 0.0;
   ...
```

Si no tengo petición...

Uso de `Select` y citas

```
operador.llamar("apagar fuego");
select
   accept confirmacion;
or
   delay 100.0;
   bomberos.dar_aviso;
end select;
```

```
operador.llamar("apagar fuego");
select
   accept confirmacion;
else
   delay 100.0;
   bomberos.dar_aviso;
end select;
```

```
select
   operador.llamar("apagar fuego");
or
   delay 100.0;
   bomberos.dar_aviso;
end select;
```

Si en 100.0 sgs. no me atienden...

```
select
   operador.llamar("apagar fuego");
else
   delay 100.0;
   bomberos.dar_aviso;
end select;
```

Si no me atienden ya..

Uso de **`Select`** y citas

```
select
   accept cita1(...) do
      -- acción a ejecutar
      -- durante la cita1
      ...
   end cita1;
or
   accept cita2(...) do
      -- acción a ejecutar
      -- durante la cita2
      ...
   end cita1;
or
   ...
or
   terminate;
end select;
```

- La alternativa **`terminate`**, termina la ejecución de la tarea, y se ejecuta cuando:
  - la unidad de la que depende la tarea ha alcanzado su final y
  - las restantes tareas dependientes o han acabado o también están en disposición de ejecutar una alternativa terminate.

Manejo de excepciones

- Una **excepción** es una situación anómala en la ejecución de un programa
  - División por cero, acceso a un elemento de un vector cuyo índice esté fuera de rango, desbordamiento en un cálculo, error en la gestión dinámica de la memoria, problemas al crear o acceder a una tarea, etc.
- **`Exception`** es un tipo predefinido en Ada

```
errorDeParidad: exception;
alarma: exception;
exc1, exc2: exception;
```

Manejo de excepciones

- Ada tiene ya prededefinidas algunas excepciones de aparición frecuente:

```
Constraint_Error: problemas en rangos, índices, ...

Numeric_Error: división por cero, desbordamiento, ...

Program_Error: acceso indebido a procedimientos, módulos,
etc., uso indebido de estructuras de selección, ...

Storage_Error: problemas en la gestión de la memoria (datos
dinámicos, procedimientos, etc.)

Tasking_Error: problemas en la sincronización, comunicación y
ejecución de tareas
```

Manejo de excepciones

- Las excepciones se manejan en la parte final del bloque, cuerpo de subprograma, cuerpo de paquete, tarea, instrucción **accept**
  - tienen acceso a todos los objetos declarados en dicha unidad

```
...
begin
   --sec. de instrucciones
exception
   when Constraint_Error | Program_Error =>
      --lo que haya que hacer específico para estos casos
   when Storage_Error =>
      --lo que haya que hacer específico para este caso
   when others =>
      --lo que haya que hacer para el resto de casos
      --no es obligatorio completar las excepciones a tratar
end;
```

Manejo de excepciones

- Una excepción se crea explícitamente mediante **`raise`**:

```
errorDeParidad: exception;
...
if sumaDeBits /= paridadEsperada then
   raise errorDeParidad;
end if;
```

```
...
exception when errorDeParidad =>
   Put_Line("Me estás engañando");
end;
```

- Cuando ocurre una excepción, la ejecución normal del programa se abandona, pasando al manipulador de la excepción
- La excepciones se propagan "hacia arriba", siguiendo la *cadena dinámica* de invocación
  - si en un nivel determinado no se encuentra un manipulador para una excepción, se propaga "hacia arriba"
- Conviene usarlas sólo en condiciones "excepcionales"

Manejo de excepciones

```
with ada.Exceptions;
  use ada.Exceptions;
```

```
procedure pruebaExcepciones is
   procedure p1 is
      miEx1: exception;
   begin
      put_line("En p1");
      raise miEx1;
   end p1;

   procedure p2 is
      miEx2: exception;
   begin
      put_line("En p2");
      raise miEx2;
   end p2;
begin
   p1;
   p2;
   exception
      when others =>
         put_line("main detecta incontrolada";
end;
```

```
En p1
main detecta incontrolada
```

Manejo de excepciones

```
with ada.Exceptions;
  use ada.Exceptions;
```

```
procedure pruebaExcepciones is
   procedure p1 is
      miEx1: exception;
   begin
      put_line("En p1");
      raise miEx1;
      exception
         when miEx1 => put_line("P1 caza miEx1");
   end p1;

   procedure p2 is
      miEx2: exception;
   begin
      put_line("En p2");
      raise miEx2;
   end p2;
begin
   p1;
   p2;
   exception
      when others =>
         put_line("main detecta incontrolada");
end;
```

```
En p1
P1 caza miEx1
En p2
main detecta incontrolada
```

Manejo de excepciones

```
with ada.Exceptions;use ada.Exceptions;
procedure pruebaExcepciones is
   procedure p1 is
      miEx1: exception;
   begin
      put_line("En p1");
      raise miEx1;
   end p1;
begin
   p1;
   exception
      when Event: others =>
         put("main detecta ");
         put_line(Exception_Name(Event));
end;
```

```
En p1
main detecta PRUEBAEXCEPCIONES.P1.MIEX1
```

Más sincronización en ADA: tipos protegidos

- Métodos de sincronización en ADA:
  - variables compartidas
  - citas
  - objetos protegidos
- Objeto protegido
  - encapsula datos
  - restringe el acceso a los procs/funcs protegidos y a las "entry" protegidas
  - ADA asegura
    - que *la ejecución de los procs/funcs y "entry" se lleva a cabo en exclusión mutua*
      - implica acceso en mutex a lectura/escritura sobre las variables
    - funciones: no pueden modificar las variables
      - varios accesos concurrentes a funciones (respetando lo anterior)

Más sincronización en ADA: tipos protegidos

- Ejemplo: un entero con acceso en exclusión mutua

```
protected type shared_integer(initial_value: integer) is
   function read return integer;
   procedure write(new_value: integer);
   procedure increment(by: integer);
private
   the_data: integer :=
                initial_value;
end shared_integer;
```

```
protected body shared_integer is
   function read return integer is
   begin
      return the_data;
   end;
   procedure write(new_value: integer) is
   begin
      the_data:= new_value;
   end;
   procedure increment(by: integer)is
   begin
      the_data:= the_data + by;
   end;
end shared_integer;
```

- es un tipo “limited” (no “=“ “:=“)
- interfaz:
  - procs, funcs, “entries”
  - discriminante: discreto/puntero
- ¿Orden de servicio?
  - el ARM no lo especifica
- Todos los datos en la parte “privada” (no en el body)

Más sincronización en ADA: tipos protegidos

- Sobre las "entries" protegidas:
  - también se ejecutan en mutex
  - puede leer/escribir sobre las variables privadas
  - siempre está guardada (barrera)
    - si barrera no activa, la tarea que llama se queda "en suspenso" hasta que se haga cierta y no haya otra tarea activa en el cuerpo
    - implementación natural para la sincronización por condición
  - una estrategia FIFO (salvo cambio forzado, claro)
- Un ejemplo: un buffer limitado compartido por varias tareas

Más sincronización en ADA: tipos protegidos

```
buffer_size: constant integer := 10;
type index is mod buffer_size;
subtype count is natural range 0..buffer_size;

type buffer is array(index) of data_item;

protected type bounded_buffer is
   entry get(item: out data_item);
   entry put(item: in data_item);
private
   first: index := index'first;
   last: index := index'last;
   number_in_buffer: count := 0;
   buf: buffer;
end bounded_buffer;
```

```
my_buffer: bounded_buffer

.....
my_buffer.Put(my_item)
.....
```

```
protected body bounded_buffer is
   entry get(item: out data_item)
                when number_in_buffer /= 0 is
   begin
      item := buf(first);
      first := first + 1;
      number_in_buffer := number_in_buffer - 1;
   end;

   entry put(item: in data_item)
                when number_in_buffer < buffer_size is
   begin
      last := last + 1;
      buf(last) := item;
      number_in_buffer := number_in_buffer + 1;
   end;
end bounded_buffer;
```

Más sincronización en ADA: tipos protegidos

- Las barreras de las entradas protegidas se (re) evalúan cuando:
  - una tarea invoca una entrada protegida cuya barrera usa una variable que ha podido cambiar desde que se evaluó por última vez
  - una tarea termina de ejecutar un procedimiento o entrada protegida y quedan tareas en la cola de alguna entrada cuya barrera usa alguna variable modificada desde la última vez que se evaluó
  - ¿Qué pasa en el caso de las funciones?

Más sobre uso de objetos protegidos

- ¿Qué pasa cuando una tarea invoca un procedimiento/entrada protegida de un obj. protegido que está siendo usado por otra tarea?
  - reglas definidas por el ARM (ver [Burns-Wellings])
- Si alguien está ejecutando una función, el objeto protegido tiene un "*cerrojo de lectura*" (*read lock*) activo
- Si alguien está ejecutando un proc. o entrada protegida, el objeto protegido tiene un "*cerrojo de lectura-escritura*" (read-*write lock*) activo
- Lo que sigue, en orden, es una "película" de lo que ocurre cuando una tarea trata de invocar algo de un objeto protegido:

Más sobre uso de objetos protegidos

```
1) si el objeto tiene un cerrojo de lectura activo, y se invoca una función,
la función se ejecuta, y nos vamos al punto 14
2) si el objeto tiene un cerrojo de lectura activo y se invoca un proc. o
entrada protegida, la ejecución de la llamada se retrasa hasta que no haya
ninguna tarea usando el objeto
3) si el objeto tiene un cerrojo de lectura-escritura activo, la tarea se
retrasa mientras hay tareas con requerimientos de acceso en conflicto con él
4) si el objeto no tiene ningún cerrojo activo, y se invoca una función, se
activa el cerrojo de lectura, y vamos a 5
5) la función se ejecuta, y vamos a 14
6) si el objeto no tiene ningún cerrojo activo y se invoca un proc. o entrada
protegida, se activa el cerrojo de lectura-escritura, y vamos a 7
7) si la llamada es un proc., se ejecuta, y vamos a 10
8) si la llamada es una entrada, la barrera asociada se evalúa; si da cierto,
el cuerpo se ejecuta, y vamos a 10
9) si la barrera da falso, la llamada va a la cola asociada a la entrada, y
vamos a 10
```

Más sobre uso de objetos protegidos

```
10) todas las barreras con tareas en espera y que referencian variables que
han podido cambiar desde la última vez que se evaluaron, son re-evaluadas, y
vamos a 11
11) si hay varias barreras abiertas, se elige una, se ejecuta el cuerpo
asociado, y vamos a 10
12) si no hay barreras abiertas con tareas en la cola, vamos a 13
13) si hay tareas esperando para acceder al objeto protegido, o bien una única
que necesite el cerrojo de lectura-escritura accede al objeto, o bien todas
que deseen ejecutar una función se activan, ejecutándose los pasos 5), 7) u
8), según sea el caso. Si no hay tareas esperando, el protocolo termina
14) si no hay ninguna tarea activa usando el objeto protegido vamos a 13. En
caso contrario, el protocolo termina.
```

Implementación de semáforos mediante objetos protegidos

```
protected type semaforo (vI: natural) is
   entry wait;
   procedure send;
private
   val: natural := vI
end semaforo;

protected body semaforo is
begin
   entry wait when val>0 is
   begin
      val := val-1;
   end;
   procedure send is
   begin
      val := val+1;
   end;
end semaforo;
```

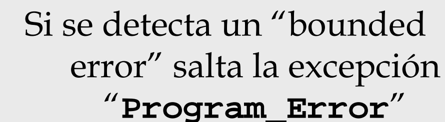
```
s: semaforo(3);

...
s.wait;
...
s.send;
...
```

- ¿tarea u objeto protegido?
  - tarea: elemento activo
    - terminación dependiendo de estructura
  - objeto: elemento pasivo
    - desaparece al terminar el bloque en que se declara

La “buena educación” en el uso de objetos protegidos

- Recomendación: el código a ejecutar en un proc/func/entry de un objeto protegido debe ser tan pequeño como se pueda
  - durante su ejecución, otras tareas están esperando
- Se considera un “bounded error” ejecutar operaciones “potencialmente bloqueantes”:
  - instrucción “select”
  - instrucción “accept”
  - llamada a un “entry”
  - instrucción “delay”
  - creación o activación de tareas
  - llamar a un subprograma con instrucciones potencialmente bloqueantes



La “buena educación” en el uso de objetos protegidos

- También es de mala educación llamar a un proc. externo que invoque a un proc. interno
  - el cerrojo de lectura-escritura ya está activo, por lo que seguro que se bloquea

- “**Bounded errors**: The language rules define certain kinds of errors that need not be detected either prior to or during run time, but if not detected, the range of possible effects shall be bounded. The errors of this category are called bounded errors. The possible effects of a given bounded error are specified for each such error, but in any case one possible effect of a bounded error is the raising of the exception Program_Error.”

ARM95

La “buena educación” en el uso de objetos protegidos

```
procedure malaEducacion is

protected type uno is
   procedure p;
end;
u: uno;

protected type dos is
   entry e;
private
   n: integer := 1;
end;
d: dos;

protected body uno is
   procedure p is
   begin
      d.e;
   end;
end;
```

```
protected body dos is
   entry e when n>3 is
   begin
      null;
   end;
end;

begin
   u.p;
end;
```

```
$> gnatmake malaEducacion
gcc -c malaeducacion.adb
malaeducacion.adb:27:18: warning: potentially
         blocking operation in protected operation
ganatbind -x malaeducacion.ali
gnatlink malaeducacion.ali
```