dit
UPM

Comunicación y sincronización con datos comunes

Juan Antonio de la Puente
DIT/UPM

Objetivos

- Comprender los problemas relacionados con la comunicación entre procesos concurrentes
- Revisar los mecanismos de sincronización que se utilizan para la comunciación mediante datos compartidos entre tareas
- Estudiar la forma de realizar este tipo de comunicación en Ada

Índice

Comunicación y sincronización

- Raras veces los procesos de un sistema son independientes unos de otros

  Más a menudo **cooperan** para un fin común o **compiten** por la utilización de recursos.
- Para ello es necesario realizar operaciones de **comunicación** y **sincronización** entre procesos
  - Dos procesos *se comunican* cuando hay una transferencia de información de uno a otro
  - Dos procesos *están sincronizados* cuando hay restricciones en el orden en que ejecutan algunas de sus acciones
- Los dos conceptos están relacionados
- Hay diferentes formas de abordar el problema:
  - **Datos comunes**
  - **Mensajes**.

Comunicación con datos comunes

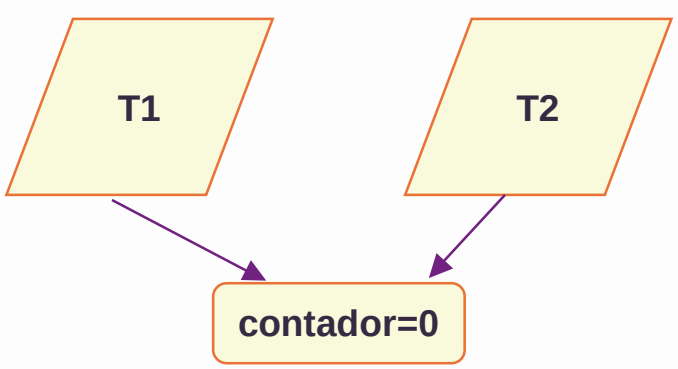
- En un sistema monoprocesador, la forma más directa de comunicación entre dos o más procesos consiste en *compartir datos comunes*

- Sin embargo, el acceso incontrolado a variables comunes puede producir resultados anómalos
  - Se dice que hay una *condición de carrera* cuando el resultado de la ejecución depende del orden en que se intercalan las instrucciones de dos o más procesos
  - Se trata de una situación anómala que hay que evitar

Ejemplo

Incrementar contador

```
LDA contador
INC
STA contador
```



Incrementar contador

```
LDA contador
INC
STA contador
```

- El resultado final puede ser 1 ó 2
  - Depende de las velocidades relativas de los procesos
  - Para evitar condiciones de carrera hay que asegurar que las operaciones con variables comunes se ejecutan de forma *atómica* (indivisible)

Exclusión mutua

- Una secuencia de instrucciones que se debe ejecutar de forma indivisible se denomina *sección crítica*
  - Las operaciones de dos secciones críticas respecto a la misma variable no se entremezclan

- La forma de sincronización que se usa para proteger una sección crítica se llama *exclusión mutua*
  - Supondremos que el acceso a una celda de memoria es atómico (hay exclusión mutua entre operaciones que solo hacen un acceso a memoria)

Sincronización condicional

- Cuando una acción de un proceso solo se puede ejecutar si otro proceso está en determinado estado o ha ejecutado ciertas acciones, hace falta un tipo de sincronización denominada *sincronización condicional*
- Ejemplo: Productor y consumidor con tampón limitado

  - No se debe hacer *Put* cuando el tampón está lleno.
  - No se debe hacer *Get* cuando el tampón está vacío
  - Además, hay exclusión mutua en el acceso al tampón

Índice

Espera ocupada

- Una forma de realizar la exclusión mutua es usar un *indicador* compartido
  - Se puede hacer si el acceso al indicador sea atómico (por ejemplo, mediante una instrucción *test-and-set*)

```
while Test_and_Set(Flag) loop
   null;
end loop;
-- sección crítica
Flag := False;
```

  - Si no, hay que usar algoritmos complejos
- Es un método poco eficiente y complicado
- No es fácil imponer un orden de acceso cuado hay varios procesos esperando

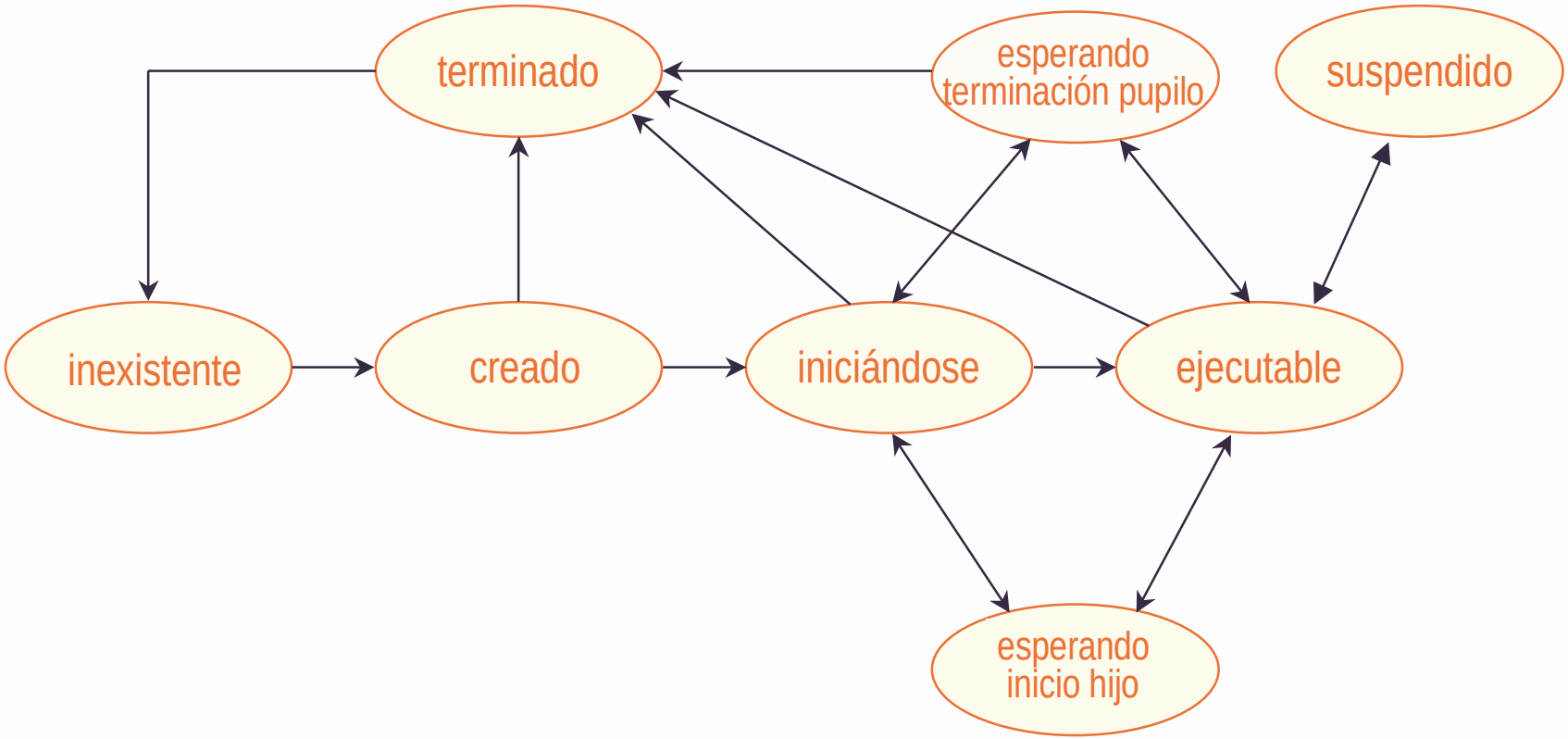
Suspensión síncrona

- Una solución más eficiente consiste en disponer de un indicador que permita que un proceso se detenga o continúe su ejecución de forma atómica
  - la atomicidad se asegura en el núcleo de multiprogramación
- Ejemplo

```
if not Flag then
   suspend;
end if;
-- sección crítica
Flag := False;
```

  - ¡tiene una condición de carrera!
- Es mejor para sincronización condicional

Estados de un proceso



Semáforos

- Un *semáforo* es una variable que toma valores enteros no negativos
- Además de asignarle un valor inicial, sólo se pueden hacer dos operaciones con un semáforo S
  - ***Wait***: Si S > 0, decrementa S en 1
    Si S ≤ 0, la tarea se suspende hasta que S > 0
  - ***Signal***: Incrementa S en 1

  Wait y *Signal* son indivisibles

- Otros nombres comunes para las operaciones de un semáforo son:
  - Down / Up
  - P / V *(passieren / vrijmachen)*
  - Secure / Release,

Ejemplo

- Supongamos una definición como la siguiente (ojo, no está predefinida en Ada)

```
package Semaphores is
   type Semaphore (Initial : Integer := 1)
     is limited private;
   procedure Wait  (S : in out Semaphore);
   procedure Signal(S : in out Semaphore);
private
   ...
end Semaphores;
```

La implementación debe garantizar las propiedades de **`Wait`** y **`Signal`**

Exclusión mutua con semáforos

```
Mutex : Semaphore; -- iniciado a 1 por defecto
```

```
task T1;
task body T1 is
begin
  loop
    Wait(Mutex);
    -- sección crítica 1
    Signal(Mutex);
    ...
   end loop;
 end T1;
```

```
task T2;
task body T2 is
begin
  loop
    Wait(Mutex);
    -- sección crítica 2
    Signal(Mutex);
    ...
   end loop;
 end T2;
```

Mutex es un *semáforo binario*

Sincronización condicional

```
Condition : Semaphore(0);
```

```
task T1; -- espera
task body T1 is
begin
  loop
    S1A;
    Wait(Condition);
    S1B;
   end loop;
 end T1;
```

```
task T2; -- avisa
task body T2 is
begin
  loop
    S2A; -- establece condición
    Signal(Condition);
    S2B;
   end loop;
 end T2;
```

S1B no se ejecuta hasta que *T2* avisa que se cumple la condición

Productor y consumidor (1)

```
package Buffer is  -- objeto abstracto
  procedure Put(X: in  Item);
  procedure Get(X: out Item);
end Buffer;
```

```
package body Buffer is

  Size   : constant Positive := 32; -- por ejemplo
  subtype Index is mod Size;
  type    Store is array (Index) of Item;

  Box          : Store;
  First, Last  : Index := Index'First;
  Mutex        : Semaphore(1);    -- exclusión mutua
  Full;        : Semaphore(0);    -- elementos ocupados
  Empty        : Semaphore(Size); -- elementos libres
```

Productor y consumidor (2)

```
   procedure Put (X : in Item) is
   begin
     Wait(Empty);
     Wait(Mutex);
     Box(Last) := X;
     Last        := Last + 1;
     Signal(Mutex);
     Signal(Full);
   end Put;

   procedure Get (X : out Item) is
   begin
     Wait(Full);
     Wait(Mutex);
     X     := Box(First);
     First := First + 1;
     Signal(Mutex);
     Signal(Empty);
    end Get;
end Buffer;
```

Problemas de los semáforos

- Es fácil cometer errores
  - Un solo *wait* o *signal* mal colocado puede hacer que el programa funcione incorrectamente

    Ejemplos:

```
Signal(S); ... Signal(S);      -- no hay exclusión mutua
Wait(S);   ... Wait(S);        -- interbloqueo
Signal(S); ... Wait(S);        -- depende
```

- Es mejor usar mecanismos más abstractos y fiables
  - regiones críticas
  - monitores
  - objetos protegidos

Interbloqueo *(deadlock)*

- Dos procesos están *interbloqueados* si cada uno de ellos espera un condición o un recurso que depende del otro

```
X,Y : Semaphore;
```

```
task T1;
task body T1 is
begin
  loop
    Wait(X);
    Wait(Y);
    -- usar recursos
    Signal(Y));
    Signal(X);
 end loop;
 end T1;
```

```
task T2;
task body T2 is
begin
  loop
    Wait(Y);
    Wait(X);
    -- usar recursos
    Signal(X);
    Signal(Y);
  end loop;
 end T2;
```

Interbloqueos, bloqueos “vivos” e inanición

- Se pueden evitar los interbloqueos asignando los recursos de forma ordenada
  - no siempre es posible
- Se puede intentar asignar los recursos de forma más flexible
  - pueden aparecer otros problemas:
    - » **bloqueo “vivo”** *(livelock)*
      - los procesos se ejecutan, pero ninguno consigue recursos
    - » **inanición** *(starvation)*
      - los procesos se ejecutan, pero algunos no consiguen el recurso nunca

Regiones críticas condicionales

- Una *región crítica* es una secuencia de instrucciones que se ejecuta en exclusión mutua
  - el compilador produce el código de bajo nivel necesario
  - se identifica la variable compartida a la que se accede en una sección crítica
- La sincronización condicional se consigue mediante *guardas* en las regiones críticas
  - solo se puede entrar en la región crítica cuando la guarda es verdadera

Ejemplo (ojo, no es Ada):

```
V : shared T;
   ...
region V when condición is
   ...
end region;
```

Productor y consumidor (1)

```
package body Buffer is

  Size   : constant Positive := 32; -- por ejemplo
  subtype Index is mod Size;
  subtype Count is Natural range 0..Size;
  type    Store is array (Index) of Item;

  type Buffer_Type is
    record
      Box          : Store;
      First, Last  : Index := Index'First;
      Number       : Count := 0;
    end record;

  Data : shared Buffer_Type;  -- ¡no es Ada!
```

Productor y consumidor (2)

```
   procedure Put (X : in Item) is
   begin
     region Data when Data.Number < Size do  -- ¡no es Ada!
       Data.Box(Data.Last) := X;
       Data.Last           := Data.Last + 1;
       Data.Number         := Data.Number + 1;
     end region;
   end Put;

   procedure Get (X : out Item) is
   begin
     region Data when Data.Number > 0 do     -- ¡no es Ada!
       X           := Data.Box(Data.First);
       Data.First  := Data.First + 1;
       Data.Number := Data.Number - 1;
     end region;
   end Get;

end Buffer;
```

Problemas de las regiones críticas

- Implementación complicada
  - cada vez que un proceso sale de una región crítica hay que volver a evaluar las guardas de los procesos que están esperando en otras regiones con la misma variable
  - para ello hay que reanudar el proceso que espera, que se puede volver a suspender si la guarda es falsa
  - potencialmente, muchos cambios de contexto
- No es obligatorio que estén confinadas en módulos
  - el programa puede ser difícil de comprobar y mantener
- Una solución mejor son los *monitores*

Monitores

- Un *monitor* es un módulo cuyas operaciones se ejecutan en exclusión mutua
  - el compilador produce el código de bajo nivel necesario
- La sincronización condicional se obtiene con *variables de condición*
  - dos operaciones primitivas: *signal* y *wait*
    - » ***wait*** suspende el proceso de forma incondicional y lo pone en una cola asociada a la condición que espera
    - » ***signal*** reanuda el primer proceso que espera en la señal
    - » no hay memoria: si no espera nadie, la señal se pierde
  - un proceso que espera una condición libera el monitor

Productor y consumidor (1)

```
monitor Buffer is                          -- ¡ojo! no es Ada
  procedure Put(X: in  Item);
  procedure Get(X: out Item);
end Buffer;
```

```
monitor body Buffer is                     -- ¡ojo! no es Ada

  Size   : constant Positive := 32; -- por ejemplo
  subtype Index is mod Size;
  subtype Count is Natural range 0..Size;
  type    Store is array (Index) of Item

  Box          : Store;
  First, Last  : Index := Index'First;
  Number       : Count := 0;
  Non_Full     : Condition := True;    -- ¡ojo! no es Ada
  Non_Empty    : Condition := False;   -- ¡ojo! no es Ada
```

Productor y consumidor (2)

```
   procedure Put (X : in Item) is
   begin
     if Number = Size then
       Wait(Non_Full);                    -- ¡ojo! no es Ada
     end if;
     Box(Last) := X;
     Last         := Last + 1;
     Number       := Number + 1;
     Signal(Non_Empty);                   -- ¡ojo! no es Ada
   end Put;
   procedure Get (X : out Item) is
   begin
     if Number = 0 then
       Wait(Non_Empty);                   -- ¡ojo! no es Ada
     end if;
     X      := Box(First);
     First  := First + 1;
     Number := Number - 1;
     Signal(Non_Full);                    -- ¡ojo! no es Ada
    end Get;
end Buffer;
```

Problemas de los monitores

- Las variables de condición son un mecanismo de bajo nivel

- La semántica de *signal* presenta problemas:
  - Cuando un proceso hace *signal* puede haber dos procesos activos en el monitor
  - Hay varias formas de evitar que se viole la exclusión mutua
    - » permitir *signal* solo si es la última instruccíón de un subprograma
    - » forzar una salida del monitor al hacer *signal*
    - » suspender el proceso que hace *signal* si se reanuda otro proceso

Índice

- Comunicación con variables comunes
  - exclusión mutua y sincronización condicional
- Mecanismos de sincronización
  - espera ocupada
  - semáforos
  - regiones críticas condicionales
  - monitores
- **Objetos protegidos en Ada**
  - ejemplos de programación
- Sincronización en C/POSIX
  - *mutex* y variables de condición

Objetos protegidos en Ada

- Combinación de monitores y regiones críticas condicionales
- 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;
```

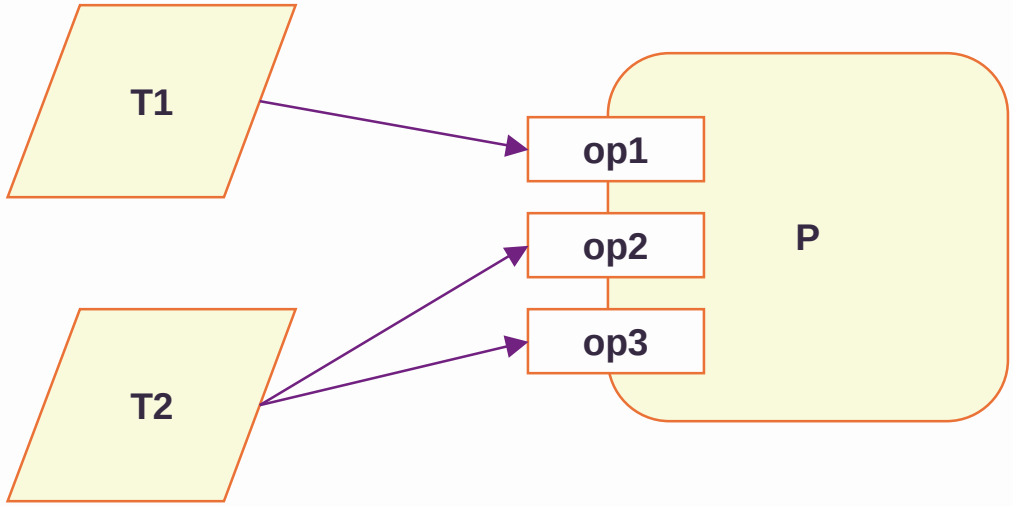
- Los cuerpos de las operaciones van en el cuerpo del tipo protegido
- No se pueden declarar tipos ni objetos

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

Diagrama de procesos



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

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

Productor y consumidor (1)

```
package Buffer is  -- objeto abstracto
  procedure Put(X: in  Item);
  procedure Get(X: out Item);
end Buffer;
```

```
package body Buffer is

  Size   : constant Positive := 32; -- por ejemplo
  subtype Index is mod Size;
  subtype Count is Natural range 0 .. Size;
  type    Store is array (Index) of Item;
```

Productor y consumidor (2)

```
protected type Buffer_Type is
  entry Put(X : in  Item);
  entry Get(X : out Item);
private
  Box          : Store;
  First, Last  : Index := Index'First;
  Number       : Count := 0;
end Buffer_Type;

Data : Buffer_Type;

procedure Put (X : in Item) is
begin
  Data.Put(X);
end Put;

procedure Get (X : out Item) is
begin
  Data.Get(X);
end Get;
```

Productor y consumidor (3)

```
   protected body Buffer_Type is

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

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

   end Buffer_Type;

end Buffer;
```

Productor y consumidor (4)

```
with Buffer; use Buffer;
procedure Producer-Consumer is
  task Producer;
  task Consumer;

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

  task body Consumer is
    X : Item;
  begin
    loop
      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 correción de un programa no debe depender del momento en que se evalúan las barreras (se puede hacer con más frecuencia de lo indicado).

Semáforos (1)

```
package Semaphores is

  type Semaphore (Initial : Integer := 1) is limited
private;
  procedure Wait  (S : in out Semaphore);
  procedure Signal(S : in out Semaphore);

private

  protected type Semaphore (Initial : Integer := 1) is
    entry Wait;
    procedure Signal;
  private
    Value : Natural := Initial;
  end Semaphore;

end Semaphores;
```

Semáforos (2)

```
package body Semaphores is

  procedure Wait  (S : in out Semaphore) is
  begin
    S.Wait;
  end Wait;

  procedure Signal(S : in out Semaphore) is
  begin
    S.Signal;
  end Signal;
```

Semáforos (3)

```
   protected body Semaphore is

     entry Wait when  Value > 0 is
     begin
       Value := Value -1;
     end Wait;

     procedure Signal is
     begin
       Value := Value + 1;
     end Signal;

   end Semaphore;

end Semaphores;
```

Recursos (1)

```
package Resources is

  type Resource is limited private;
  procedure Allocate(R : in out Resource);
  procedure Free    (R : in out Resource);

private

  protected type Resource is
    entry Allocate;
    procedure Free;
  private
    Busy : Boolean := False
  end Resource;

end Resources;
```

Recursos (2)

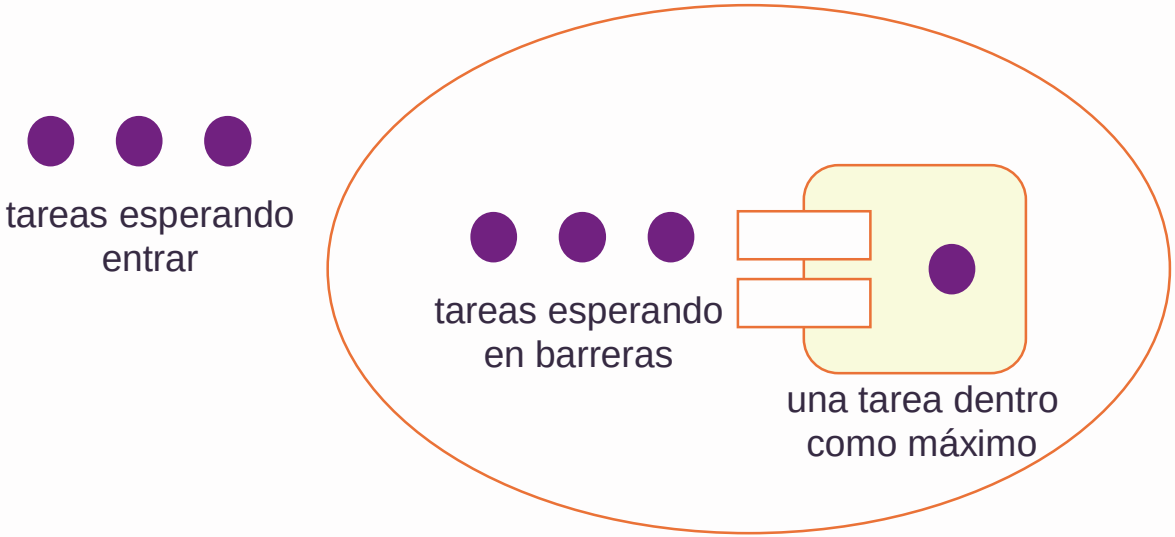
```
package body Resources is

  procedure Allocate(R : in out Resource) is
  begin
    R.Allocate;
  end Wait;

  procedure Free(R : in out Resource) is
  begin
    S.Free;
  end Signal;
```

Recursos (3)

```
   protected body Resource is

     entry Allocate when not Busy is
     begin
       Busy := True;
     end Wait;

     procedure Free is
     begin
       Busy := False;
     end Signal;

   end Resource;

end Resources;
```

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*)



Radiado de mensajes (1)

```
generic
  type Message is private;
package Broadcast is
  procedure Send    (M : in  Message);
  procedure Receive (R : out Message);
end Broadcast;
```

```
package body Broadcast is

  protected Buffer is
    procedure Send    (M : in  Message);
    entry     Receive (M : out Message);
  private
    Data    : Message;
    Arrived : Boolean := False;
  end Buffer;
```

Radiado de mensajes (2)

```
protected body Buffer is

   procedure Send    (M : in  Message) is
   begin
     if Receive'Count > 0 then
       Arrived := True;
       Data    := M;
     end if;
   end Send;

   entry Receive (M : out Message) when Arrived is
   begin
     M := Data;
     if Receive'Count = 0 then   -- ¡el último cierra!
       Arrived := False;
     end if;
   end Receive;

end Buffer;
```

Radiado de mensajes (3)

```
   procedure Send    (M : in  Message) is
   begin
     Buffer.Send(M);
   end Send;
   procedure Receive (R : out Message) is
   begin
     Buffer:Receive(M);
   end Receive;

end Broadcast;
```

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

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

Sucesos (1)

```
package Events is

  type Event is limited private;
  procedure Wait   (E : Event);
  procedure Signal (E : Event);

private
  protected type Event is
    entry Wait;
    entry Signal;
  private
    entry Reset;
    Occurred : Boolean := False;
  end Event;

end Events;
```

Sucesos (2)

```
package body Events is

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

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

Sucesos (3)

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

    procedure Wait (E : Event) is
    begin
      E.Wait;
    end Wait;

    procedure Signal (E : Event) is
    begin
      E.Signal;
    end Signal;

  end Events;
```

Tareas esporádicas

```
with Events; use  Events;
...
Button_Pressed : Event;
...
task body Sporadic is
begin
   loop
      Wait (Button_Pressed);
      -- acción esporádica
   end loop;
end Sporadic;

task body Event_Handler is
begin
   ...
   Signal (Button_Pressed); -- puede activar varias tareas
   ...
end Event_Handler;
```

Objetos de suspensión

- Proporcionan una funcionalidad similar cuando sólo hay una tarea que espera

```
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;
```

Tarea esporádica (2)

```
with Ada.Synchronous_Task_Control;
use  Ada.Synchronous_Task_Control;
...
Button_Pressed : Suspension_Object;

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

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

Índice

Sincronización en C/POSIX.1c

- Hay dos mecanismos que permiten emular monitores con interfaz de procedimiento
  - Un ***mutex*** es una variable que proporciona exclusión mutua mediante dos operaciones, ***lock*** y ***unlock***
  - una ***variable de condición*** proporciona sincronización condicional mediante dos operaciones, ***wait*** y ***signal***
- Ambos tipos de variables pueden tener atributos
- Otras operaciones: crear y destruir, espera temporizada

Definiciones

```
/* pthreads.h */

typedef ... pthread_mutext_t;
typedef ... pthread_cond_t;

...

int pthread_mutex_lock  (pthread_mutext_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_cond_wait(pthread_cond_t *cond ,
                      pthread_mutext_t *mutex);
   /* espera y libera el mutex */
int pthread_cond_signal(pthread_cond_t *cond);
   /* puede reanudar más de un thread */

...
```

Productor y consumidor (1)

```
#include pthreads.h
#define SIZE 32

typedef struct {
  pthread_mutext_t mutex;
  pthread_cond_t non_full;
  pthread_cond_t non_empty;
  int first, last, number;
  int box[SIZE];
} buffer_t;
```

Productor y consumidor (2)

```
int put(int item, buffer_t *buffer) {
  pthread_mutex_lock(&buffer->mutex);
  while (buffer->number == SIZE)
    pthread_cond_wait(&buffer->non_full, &buffer->mutex);
  buffer[last] = item;
  last = (last++)%SIZE; number++;
  pthread_mutex_unlock(&buffer->mutex);
  pthread_cond_signal(&buffer->non_empty);
  return 0;
}

int get(int *item, buffer_t *buffer) {
  pthread_mutex_lock(&buffer->mutex);
  while (buffer->number == 0)
    pthread_cond_wait(&buffer->non_empty, &buffer->mutex);
  item = buffer[first];
  first = (first++)%SIZE; number--;
  pthread_mutex_unlock(&buffer->mutex);
  pthread_cond_signal(&buffer->non_full);
  return 0;
}
```

Resumen

- El acceso de dos o más tareas concurrentes a las variables comunes debe realizarse en exclusión mutua
- Otra forma de sincronización está ligada al cumplimento de una condición
- Algunos mecanismos que permiten conseguir ambas formas de sincronización son:
  - semáforos
  - regiones críticas condicionales
  - monitores
- Los objetos protegidos reúnen las ventajas de las regiones críticas condicionales y los monitores
- Los *mutex* y las variables condicionales de POSIX proporcionan un mecanismo similar a los monitores