Esta práctica tiene como objetivo entender más conceptos fundamentales de concurrencia usando diferentes tipos de tareas. Para ello veremos tareas que incluyan selección, guardas, temporización y terminación controlada. Posteriormente se tratará el tema de los objetos protegidos en tareas. Se aplicarán estos conceptos de forma progresiva a través de ejercicios.

1 El problema del buffer limitado

El problema del buffer limitado, también conocido como el problema del productor-consumidor, es un ejemplo clásico que plantea la problemática del acceso concurrente al mismo recurso -un buffer-, que tiene una capacidad limitada, por parte de una serie de productores y consumidores de información.

En este contexto, podemos imaginar dos tipos de tareas: por un lado, los productores, que generan datos y los colocan en el buffer; y por otro lado, los consumidores, que extraen esos datos del buffer para procesarlos. La dificultad surge precisamente porque el buffer no puede crecer indefinidamente: tiene un número fijo de espacios disponibles.

Esto genera dos situaciones problemáticas si no se gestiona correctamente la sincronización entre tareas. La primera cuestión a considerar es que se ha de controlar el acceso exclusivo a la sección crítica, es decir, al propio buffer. La segunda cuestión importante reside en controlar dos situaciones: i) un productor no puede insertar un elemento cuando el buffer está lleno, lo que provocaría pérdida de datos o sobrescritura y ii) un consumidor no puede extraer un elemento cuando el buffer está vacío, lo que daría lugar a bloqueos o lecturas inválidas.

La clave para resolver este problema está en garantizar que las tareas accedan al buffer únicamente cuando las condiciones sean las adecuadas: los productores deben esperar si no hay espacio libre, y los consumidores deben esperar si no hay datos disponibles.

A continuación trataremos de dar solución modelando poco a poco el problema.

2 Modelando el problema

Una posible implementación del problema del buffer limitado podría ser hacer uso de semáforos. Sin embargo en Ada, este problema se puede modelar mediante tareas concurrentes. Vamos a tratar de abordar el problema considerando que gestionaremos un único elemento mediante soluciones incrementales paso a paso que incluyan:

- Tareas sincronizadas simples.
- Tareas sincronizadas con selección.
- Tareas sincronizadas con selección y guardas.
- Tareas sincronizadas con selección, guardas y temporización
- Tareas sincronizadas con selección, guardas, temporización y terminación controlada

Este problema no solo es un buen ejercicio académico: tiene muchas aplicaciones reales. Se encuentra, por ejemplo, en sistemas de impresión donde hay que gestionar una cola de trabajos, en la transmisión de datos en redes, en sistemas multimedia como el streaming de audio o vídeo, y en la organización de tareas dentro de una base de datos o un sistema operativo.

2.1 Uso tareas sincronizadas simples

En el problema del buffer limitado, tanto el propio buffer en el que se almacenarán los datos como los procesos productor y consumidor se modelarán mediante tareas.

La gestión del acceso concurrente al buffer, así como la inserción y extracción de datos, se realizará mediante dos `entries`, denominados respectivamente `Insertar` y `Escribir`. En este caso, asumimos que nuestro buffer solo tiene capacidad para un elemento.

A continuación se muestra la implementación de la tarea `Buffer`. Como se puede apreciar, el acceso al contenido del buffer se realiza mediante `Insertar` y `Extraer`. El parámetro de este último `entry` es de salida, ya que es la operación de extracción de datos.

```
task type Buffer is
  entry Insertar(D: in Integer);
  entry Extraer(D: out Integer);
end Buffer;

task body Buffer is
  Dato: Integer:= 0;
begin
  loop
    accept Insertar(D: in Integer) do
       Dato:= D;
       Put_Line("Insertado " & Integer'Image(D));
    end Insertar;
    accept Extraer(D : out Integer) do
       D := Dato;
       Dato:=0;
       Put_Line("Extraído " & Integer'Image(D));
    end Extraer;
  end loop;
end Buffer;
```

A continuación se muestra una posible implementación, a modo de ejemplo, de la tarea `Consumidor`.

```
task type Consumidor;

task body Consumidor is
  V : Integer;
begin
  for I in 1 .. 3 loop
     delay 1.5;
     B.Extraer(V);
  end loop;
end Consumidor;
```

Ahora se muestra una posible implementación, a modo de ejemplo, de la tarea `Productor`.

```
task type Productor;

task body Productor is
begin
  for I in 1 .. 3 loop
     delay 1.0;
     B.Insertar(I);
  end loop;
end Productor;
```

El resultado final sería el siguiente:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Etapa_1_Accept is
   task type Buffer is
      entry Insertar(D: in Integer);
      entry Extraer(D: out Integer);
   end Buffer;

   task body Buffer is
      Dato: Integer:= 0;
   begin
      loop
        accept Insertar(D: in Integer) do
           Dato:= D;
           Put_Line("Insertado " & Integer'Image(D));
        end Insertar;
        accept Extraer(D : out Integer) do
           D := Dato;
           Dato:=0;
           Put_Line("Extraído " & Integer'Image(D));
        end Extraer;
      end loop;
   end Buffer;
   B : Buffer;

   task type Productor;
   task type Consumidor;

   task body Productor is
   begin
      for I in 1 .. 3 loop
         delay 1.0;
         B.Insertar(I);
      end loop;
   end Productor;

   task body Consumidor is
      V : Integer;
   begin
      for I in 1 .. 3 loop
         delay 1.5;
         B.Extraer(V);
      end loop;
   end Consumidor;

   P : Productor;
   C : Consumidor;
begin
   Put_Line("=== Etapa 1: Accept sin control de estado ===");
   delay 6.0;
end Etapa_1_Accept;
```

Ejercicio:

¿Qué sucede si el bucle que se encuentra en el Productor es de 2 en vez de 3? Razonar la respuesta.

2.2 Uso tareas sincronizadas con selección

A la vista de lo sucedido, y debido a la necesidad de proporcionar distintos servicios en el buffer, Ada proporciona la instrucción `select ... or` con el objetivo de que esta pueda atender múltiples `entries`, en el orden que sea.

Por tanto, procedemos a modificar la implementación de la tarea `Buffer`.

```
task type Buffer is
  entry Insertar(D: in Integer);
  entry Extraer(D: out Integer);
end Buffer;

task body Buffer is
  Dato: Integer:= 0;
begin
  loop
     select
        accept Insertar(D: in Integer) do
           Dato:= D;
           Put_Line("Insertado " & Integer'Image(D));
        end Insertar;
     or
        accept Extraer(D : out Integer) do
           D := Dato;
           Dato:=0;
           Put_Line("Extraído " & Integer'Image(D));
        end Extraer;
     end select;
  end loop;
end Buffer;
```

Las tareas relacionadas con el `Productor` y `Consumidor` se mantienen. El resultado final sería el siguiente:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Etapa_2_Select_Accept is
   task type Buffer is
      entry Insertar(D: in Integer);
      entry Extraer(D: out Integer);
   end Buffer;

   task body Buffer is
      Dato: Integer:= 0;
   begin
      loop
```

```
         select
            accept Insertar(D: in Integer) do
               Dato:= D;
               Put_Line("Insertado " & Integer'Image(D));
            end Insertar;
         or
            accept Extraer(D : out Integer) do
               D := Dato;
               Dato:=0;
               Put_Line("Extraído " & Integer'Image(D));
            end Extraer;
         end select;
      end loop;
   end Buffer;
   B : Buffer;

   task type Productor;
   task type Consumidor;

   task body Productor is
   begin
      for I in 1 .. 3 loop
         delay 1.5;
         B.Insertar(I);
      end loop;
   end Productor;

   task body Consumidor is
      V : Integer;
   begin
      for I in 1 .. 3 loop
         delay 1.5;
         B.Extraer(V);
      end loop;
   end Consumidor;

   P : Productor;
   C : Consumidor;
begin
   Put_Line("=== Etapa 2: Select + Accept sin control de estado ===");
   delay 6.0;
end Etapa_2_Select_Accept;
```

Ejercicio:

¿Qué sucede si el bucle que se encuentra en el Productor es de 2 en vez de 3 o si el delay en el Productor aumenta a 2.0? Razonar la respuesta.

2.3 Uso tareas sincronizadas con selección y guardas

Las guardas en Ada son condiciones booleanas que controlan si una entrada (`entry`) está disponible para ser aceptada dentro de una construcción `select` o en un objeto protegido. Son una herramienta clave para introducir comportamiento condicional y sincronización entre tareas.

Una guarda actúa como un filtro lógico. Una entrada solo se habilita si su condición (guarda) es verdadera. Esto es especialmente útil para:

- evitar que una tarea consuma datos si no hay nada disponible (por ejemplo, en un buffer).
- impedir que una tarea produzca datos si el espacio está lleno.
- coordinar varias tareas sin necesidad de estructuras adicionales (como semáforos).

Por tanto, procedemos a modificar, de nuevo, la implementación de la tarea `Buffer`.

```
task type Buffer is
  entry Insertar(D: in Integer);
  entry Extraer(D: out Integer);
end Buffer;

task body Buffer is
    Dato: Integer := 0;
begin
  loop
     select
        when Dato = 0 =>
        -- Solo se puede insertar cuando el buffer está vacío (Dato = 0)
        accept Insertar(D: in Integer)  do
           Dato := D;
           Put_Line("Insertado " & Integer'Image(D));
        end Insertar;
     or
        when Dato /= 0 =>
        -- Solo se puede extraer cuando el buffer no está vacío (Dato != 0)
        accept Extraer(D: out Integer)  do
           D := Dato;
           Dato := 0;  -- Después de extraer, el buffer se vacía
           Put_Line("Extraído " & Integer'Image(D));
        end Extraer;
     end select;
  end loop;
end Buffer;
```

Las tareas relacionadas con el `Productor` y `Consumidor` se mantienen. El resultado final sería el siguiente:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Etapa_3_Guarda_Select_Accept is
   task type Buffer is
      entry Insertar(D: in Integer);
      entry Extraer(D: out Integer);
   end Buffer;

   task body Buffer is
        Dato: Integer := 0;
   begin
      loop
         select
            when Dato = 0 =>
            -- Solo se puede insertar cuando el buffer está vacío (Dato = 0)
            accept Insertar(D: in Integer)  do
               Dato := D;
               Put_Line("Insertado " & Integer'Image(D));
            end Insertar;
         or
            when Dato /= 0 =>
            -- Solo se puede extraer cuando el buffer no está vacío (Dato != 0)
            accept Extraer(D: out Integer)  do
               D := Dato;
               Dato := 0;
               Put_Line("Extraído " & Integer'Image(D));
            end Extraer;
         end select;
      end loop;
   end Buffer;

   B : Buffer;

   task type Productor;
   task type Consumidor;

   task body Productor is
   begin
      for I in 1 .. 3 loop
         delay 1.5;
         B.Insertar(I);
      end loop;
   end Productor;

   task body Consumidor is
      V : Integer;
   begin
      for I in 1 .. 3 loop
         delay 1.5;
         B.Extraer(V);  -- Intenta extraer un valor del buffer
      end loop;
   end Consumidor;

   P : Productor;  -- Instancia de la tarea Productor
   C : Consumidor;  -- Instancia de la tarea Consumidor

begin
   Put_Line("=== Etapa 3: Guarda + Select + Accept con control de estado ===");
   delay 6.0;
end Etapa_3_Guarda_Select_Accept;
```

2.4 Uso de tareas sincronizadas con selección, guardas y temporización

La temporización en Ada sirve para varias cosas:

1. Evitar bloqueos innecesarios:

   La temporización permite que el sistema no quede bloqueado esperando indefinidamente a que ocurra alguna acción. Si un recurso o tarea no está disponible o no se puede realizar una acción en ese momento, en lugar de que el proceso espere sin hacer nada (lo que podría bloquear otros procesos), el sistema puede "dormir" por un tiempo (mediante delay) y luego volver a intentarlo o realizar otras acciones. Esto mantiene la concurrencia fluida.

   Por ejemplo, si el buffer está vacío y el consumidor intenta extraer datos sin éxito, la temporización le da al sistema tiempo para realizar otras acciones en lugar de quedarse bloqueado esperando una operación que no puede ejecutarse en ese momento.

2. Prevención de "Hambre" de tareas:

   Sin temporización, las tareas podrían no tener suficiente tiempo de ejecución si otras tareas siempre están bloqueando el acceso a recursos. Al incluir temporización, se da un pequeño "respiro" a las tareas, lo que ayuda a evitar la "hambre" (starvation), donde una tarea nunca puede acceder a los recursos porque otras siempre los están ocupando.

3. Proporcionar oportunidades de respuesta:

   Si un proceso o tarea está esperando una condición o un evento, en lugar de esperar indefinidamente, el uso de temporización permite que el sistema haga otras cosas (como otras operaciones de entrada/salida, tareas de mantenimiento, etc.) antes de volver a comprobar si la condición deseada se ha cumplido.

4. Balanceo de Tareas Concurrentes:

   Esta estrategia también es útil cuando se quiere balancear el tiempo de ejecución entre diferentes tareas, evitando que alguna de ellas monopolice el procesador.

Por tanto, procedemos a modificar, de nuevo, la implementación de la tarea `Buffer`.

```
task type Buffer is
  entry Insertar(D: in Integer);
  entry Extraer(D: out Integer);
end Buffer;

task body Buffer is
    Dato: Integer := 0;  -- Buffer inicialmente vacío
begin
  loop
     select
        when Dato = 0 =>
        -- Solo se puede insertar cuando el buffer está vacío (Dato = 0)
        accept Insertar(D: in Integer)  do
           Dato := D;
           Put_Line("Insertado " & Integer'Image(D));
        end Insertar;
     or
        when Dato /= 0 =>
        -- Solo se puede extraer cuando el buffer no está vacío (Dato != 0)
```

```
        -- Solo se puede extraer cuando el buffer no está vacío (Dato != 0)
        accept Extraer(D: out Integer)  do
           D := Dato;
           Dato := 0;
           Put_Line("Extraído " & Integer'Image(D));
        end Extraer;
     or
        -- Si no hay actividad en el buffer, esperar 1 segundo
        delay 1.0;
        Put_Line("Esperando actividad...");
     end select;
  end loop;
end Buffer;
```

Las tareas relacionadas con el `Productor` y `Consumidor` se mantienen. El resultado final sería el siguiente:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Etapa_4_Temp_Guarda_Select_Accept is
   task type Buffer is
      entry Insertar(D: in Integer);
      entry Extraer(D: out Integer);
   end Buffer;

   task body Buffer is
        Dato: Integer := 0;  -- Buffer inicialmente vacío
   begin
      loop
         select
            when Dato = 0 =>
            -- Solo se puede insertar cuando el buffer está vacío (Dato = 0)
            accept Insertar(D: in Integer)  do
               Dato := D;
               Put_Line("Insertado " & Integer'Image(D));
            end Insertar;
         or
            when Dato /= 0 =>
            -- Solo se puede extraer cuando el buffer no está vacío (Dato != 0)
            accept Extraer(D: out Integer)  do
               D := Dato;
               Dato := 0;
               Put_Line("Extraído " & Integer'Image(D));
            end Extraer;
         or
            -- Si no hay actividad en el buffer, esperar 1 segundo
            delay 1.0;
            Put_Line("Esperando actividad...");
         end select;
      end loop;
   end Buffer;
```

```
   B : Buffer;

   task type Productor;
   task type Consumidor;

   task body Productor is
   begin
      for I in 1 .. 3 loop
         delay 1.5;
         B.Insertar(I);
      end loop;
   end Productor;

   task body Consumidor is
      V : Integer;
   begin
      for I in 1 .. 3 loop
         delay 1.5;
         B.Extraer(V);  -- Intenta extraer un valor del buffer
      end loop;
   end Consumidor;

   P : Productor;  -- Instancia de la tarea Productor
   C : Consumidor;  -- Instancia de la tarea Consumidor

begin
   Put_Line("=== Etapa 4: Temp + Guarda + Select + Accept con control de estado ===");
   delay 6.0;
end Etapa_4_Temp_Guarda_Select_Accept;
```

Ejercicio:

¿Qué sucede si el bucle que se encuentra en el Productor es de 2 en vez de 3? ¿Termina en algún momento?

2.5 Uso de tareas sincronizadas con selección, guardas, temporización y terminación

En algunos casos puede ser necesario finalizar la tarea de forma explícita cuando ya no tiene más trabajo que hacer o cuando se cumple una determinada condición. Para ello se usa `terminate`.

Por tanto, procedemos a modificar, de nuevo, la implementación de la tarea `Buffer`.

```
task type Buffer is
  entry Insertar(D: in Integer);
  entry Extraer(D: out Integer);
end Buffer;
```

```
task body Buffer is
    Dato: Integer := 0;  -- Buffer inicialmente vacío
begin
  loop
     select
        when Dato = 0 =>
        -- Solo se puede insertar cuando el buffer está vacío (Dato = 0)
        accept Insertar(D: in Integer)  do
           Dato := D;
           Put_Line("Insertado " & Integer'Image(D));
        end Insertar;
     or
        when Dato /= 0 =>
        -- Solo se puede extraer cuando el buffer no está vacío (Dato != 0)
        accept Extraer(D: out Integer)  do
           D := Dato;
           Dato := 0;
           Put_Line("Extraído " & Integer'Image(D));
        end Extraer;
     or
        -- Terminamos cuando ambas tareas han completado su trabajo
        terminate;  -- Aquí se termina de inmediato
     end select;
  end loop;
end Buffer;
```

Las tareas relacionadas con el `Productor` y `Consumidor` se mantienen. El resultado final sería el siguiente:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Etapa_5_Term_Temp_Guarda_Select_Accept is
   task type Buffer is
      entry Insertar(D: in Integer);
      entry Extraer(D: out Integer);
   end Buffer;

   task body Buffer is
        Dato: Integer := 0;  -- Buffer inicialmente vacío
   begin
      loop
         select
            when Dato = 0 =>
            -- Solo se puede insertar cuando el buffer está vacío (Dato = 0)
            accept Insertar(D: in Integer)  do
               Dato := D;
               Put_Line("Insertado " & Integer'Image(D));
            end Insertar;
```

```
         or
            when Dato /= 0 =>
            -- Solo se puede extraer cuando el buffer no está vacío (Dato != 0)
            accept Extraer(D: out Integer)  do
               D := Dato;
               Dato := 0;
               Put_Line("Extraído " & Integer'Image(D));
            end Extraer;
         or
            -- Terminamos cuando ambas tareas han completado su trabajo
            terminate;  -- Aquí se termina de inmediato
         end select;
      end loop;
   end Buffer;

   B : Buffer;  -- Instanciamos el buffer

   task type Productor;
   task type Consumidor;

   task body Productor is
   begin
      for I in 1 .. 3 loop
         delay 1.5;
         B.Insertar(I);  -- Intenta insertar un valor en el buffer
      end loop;
   end Productor;

   task body Consumidor is
      V : Integer;
   begin
      for I in 1 .. 3 loop
         delay 1.5;
         B.Extraer(V);  -- Intenta extraer un valor del buffer
      end loop;
   end Consumidor;

   P : Productor;  -- Instancia de la tarea Productor
   C : Consumidor;  -- Instancia de la tarea Consumidor

begin
   Put_Line("=== Etapa 5: Term + Temp + Guarda + Select + Accept con control de estado ===");
   delay 6.0;  -- Para permitir que las tareas Productor y Consumidor trabajen un poco
end Etapa_5_Term_Temp_Guarda_Select_Accept;
```

Ejercicio:

¿Qué sucede si se añade a la tarea buffer lo siguiente?:

```
    or
        -- Si no hay actividad en el buffer, esperar 1 segundo
        delay 1.0;
        Put_Line("Esperando actividad...");
```

2.6 Los objetos protegidos

Hasta ahora hemos visto como en el caso del buffer limitado de la anterior sección, no es necesario utilizar ningún tipo de mecanismo de exclusión mutua ya que su contenido nunca será accedido de manera concurrente, gracias a la propia naturaleza de los `entries`.

En Ada, un objeto protegido es un tipo de módulo protegido que encapsula una estructura de datos y exporta subprogramas o funciones. Estos subprogramas operan sobre la estructura de datos bajo una exclusión mutua automática. En Ada es posible definir barreras o guardas en las propias entradas, que deben evaluarse a verdadero antes de que a una tarea se le permita entrar. Este planteamiento se basa en la definición de regiones críticas condicionales

Vamos a verlo paso a paso con un caso práctico que consiste en un contador compartido entre varias tareas:

1. Declarar el objeto protegido

```
protected Contador_Protegido is
   procedure Incrementar;
   function Valor return Integer;
private
   C : Integer := 0;
end Contador_Protegido;
```

2. Cuerpo del objeto protegido

```
protected body Contador_Protegido is
   procedure Incrementar is
   begin
      C := C + 1;
   end Incrementar;

   function Valor return Integer is
   begin
      return C;
   end Valor;
end Contador_Protegido;
```

3. Crear las tareas que usan el contador

```
task type Trabajadora;

task body Trabajadora is
begin
   for I in 1 .. 10 loop
      delay 0.1;
      Contador_Protegido.Incrementar;
   end loop;
end Trabajadora;
```

Creamos varias instancias de esta tarea:

```
T1, T2, T3 : Trabajadora;
```

4. Programa principal para lanzar todo

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Demo_Contador_Protegido is
   -- Declaraciones anteriores van aquí (Contador_Protegido, Trabajadora, etc.)
   T1, T2, T3 : Trabajadora;
begin
   Put_Line("Iniciando tareas...");
   delay 2.0; -- Esperamos a que las tareas terminen
   Put_Line("Valor final del contador: " & Integer'Image(Contador_Protegido.Valor));
end Demo_Contador_Protegido;
```

El código completo se encuentra aquí:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Demo_Contador_Protegido is
    protected Contador_Protegido is
        procedure Incrementar;
        function Valor return Integer;
    private
        C : Integer := 0;
    end Contador_Protegido;

    protected body Contador_Protegido is
        procedure Incrementar is
        begin
            C := C + 1;
        end Incrementar;
        function Valor return Integer is
        begin
            return C;
        end Valor;
    end Contador_Protegido;

    task type Trabajadora;
    task body Trabajadora is
    begin
        for I in 1 .. 10 loop
            delay 0.1;
            Contador_Protegido.Incrementar;
        end loop;
    end Trabajadora;

    T1, T2, T3 : Trabajadora;
begin
    Put_Line("Iniciando tareas...");
    delay 2.0; -- Esperamos a que las tareas terminen
    Put_Line("Valor final del contador: " & Integer'Image(Contador_Protegido.Valor));
end Demo_Contador_Protegido;
```

3 Entregables

Tarea 1: Responder a las preguntas planteadas en la práctica.

Tarea 2: Implementar en el problema del buffer limitado considerando que existe un único dato (siguiendo la práctica) usando objetos protegidos.

Tarea 3: Implementar el problema del buffer limitado considerando que ahora el buffer es un array de 5 enteros. Realizar los pasos de forma incremental.

Tarea 4: Implementar el problema del buffer limitado usando objetos protegidos (considerando el array de 5 elementos).