Trabajo ADA

PCTR 2024

Rosa María Fernández-Baíllo Callejas

2ºB

Índice

Ejercicio 1

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

procedure main is

begin
    Put_Line("---- Inicio del programa main ----");
    delay 5.0;
    Put_Line("---- Fin del programa main ----");
end main;
```

El ejercicio 1 pedía un programa sencillo que imprimiese unas líneas con una pausa entre ellas. Para ello se utilizan las librerías *Text_IO* para imprimir por pantalla y *Calendar* para la función *delay*.

Ejercicio 2

```
with Ada.Text_IO;
with pkg_procedure;
with pkg_tarea;

procedure Main is
   use Ada.Text_IO;
   use pkg_procedure;
   use pkg_tarea;

   T : tarea_t;
   Numero : Integer;

begin
   Crear_Tarea (T);

   loop
      Estado_Consultas (T);
      Put ("Escriba un numero entero y pulsa ENTER --> ");
      Leer_Entero (Numero);
      if Numero /= 0 then
         Es_Par (T, Numero);
      else
         exit; --salir
      end if;
   end loop;

   abort T;
end Main;
```

Para el ejercicio 2 se pide una solución para crear y gestionar una tarea interactiva que permite al usuario ingresar números enteros y determinar si son pares. Se empieza por importar las librerías proporcionadas de *pkg_procedure* y *pkg_tarea*, y *Text_IO* para el manejo de entrada y salida de texto.

Se declaran las variables T de tipo *tarea_t* para ejecutar y número para guardar el input del usuario.

Se inicia el bucle interactivo que estará ejecutando hasta que el usuario meta el número 0. Dentro del bucle:

1. Se llama al procedimiento *Estado_Consultas* para actualizar el estado de T.
2. Se pide al usuario que ingrese un número entero.
3. Se lee el número ingresado usando *Leer_Entero*.
4. Si el número ingresado no es 0, se llama al procedimiento *Es_Par* para determinar si el número es par.
5. Si el número es 0, se sale del bucle.

Al salir del bucle se termina la tarea T con *abort*.

Ejercicio 3

```
with Ada.Text_IO;
with Ada.Real_Time;
with pkg_tarea;

procedure Main is
   use Ada.Text_IO;
   use Ada.Real_Time;
   use pkg_tarea;

   T1, T2 : Tarea;
   Tiempo_Activacion : Tarea_Periodica := Milliseconds (1000);

begin
   Crear_Tarea (T1, Tiempo_Activacion);
   Crear_Tarea (T2, Tiempo_Activacion);

   T1.Start;
   T2.Start;
   delay 8.0; --primera espera de 8

   -- Aborts
   abort T1;
   abort T2;

   delay 2.0; --segunda espera de 2, final
   Put_Line("Fin del procedimiento main");

end Main;
```

Para este ejercicio 3 se empieza igual importación de la librería *pkg_tarea* y *Text_IO,* pero cambiamos y añadimos Ada.Real.Time para tener las operaciones de tiempo real.

Declaramos las Tareas T1 y T2, y la variable *Tiempo_Activacion* de tipo *Tarea_Periodica* inicializada a 1000 milisegundos.

Luego se llaman los procedimientos *Crear_Tarea* para inicializar las tareas T1 y T2 con el periodo de activación especificado antes y se inician método *Start* en cada una y nos esperamos 8 segundos con *delay* 8.0.

Se terminan las tareas T1 y T2 con *abort* y esperamos otros 2 segundos antes de mostrar el mensaje de finalización.

Este ejercicio ha dado problemas para compilar y salen advertencias de los paquetes importados de tareas, siendo código dado y no encontrando fallos lo he dejado tal cual.

Ejercicio 4

Pkg_semaforo.ads:

```
package pkg_semaforo is
   type Semaforo is new Ada.Finalization.Controlled with private;

   procedure Inicializar (S : in out Semaforo);
   procedure Adquirir (S : in out Semaforo);
   procedure Liberar (S : in out Semaforo);

private
   type Semaforo is new Ada.Finalization.Controlled with null record;

   overriding procedure Initialize (S : in out Semaforo);
   overriding procedure Finalize (S : in out Semaforo);

end pkg_semaforo;
```

Pkg_semaforo.adb

```
package body pkg_semaforo is

   overriding procedure Initialize (S : in out Semaforo) is
   begin
      null;
   end Initialize;

   overriding procedure Finalize (S : in out Semaforo) is
   begin
      null;
   end Finalize;

   procedure Inicializar (S : in out Semaforo) is
   begin
      null;
   end Inicializar;

   procedure Adquirir (S : in out Semaforo) is
   begin
      null;
   end Adquirir;

   procedure Liberar (S : in out Semaforo) is
   begin
      null;
   end Liberar;

end pkg_semaforo;
```

Con este paquete define una estructura básica y esqueleto para un semáforo, pero las implementaciones de los procedimientos Inicializar, Adquirir y Liberar están vacías (null).

El paquete *pkg_semaforo* declara privado como un tipo Semaforo derivado de *Ada.Finalization.Controlled*, lo que permite manejar la inicialización y finalización del semáforo.

Con este paquete define una estructura básica y esqueleto para un semáforo, pero las implementaciones de los procedimientos *Inicializar, Adquirir y Liberar*. Para que el semáforo sea funcional, estos procedimientos deberían ser implementados con la lógica adecuada para manejar la sincronización de tareas. Por ejemplo:

- **Inicializar** podría configurar el estado inicial del semáforo.
- **Adquirir** podría bloquear el semáforo si ya está adquirido, haciendo que la tarea espere hasta que el semáforo se libere.
- **Liberar** podría cambiar el estado del semáforo para indicar que está disponible para otras tareas.

Ejercicio 5

Para resolver este problema tenemos *pkg_tarea.ads* que contiene las definiciones y lógica para las tareas y la ejecución:

```
package pkg_tareas is
   task type Tarea is
      entry Start;
   end Tarea;

   procedure Ejecutivo_Ciclico;

end pkg_tareas;
```

Donde *Ejecutivo_Ciclico* es el procedimiento principal que inicia y gestiona el ciclo de ejecución de las tareas.

Y pkg_tareas.adb:

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

package body pkg_tareas is
   task body Tarea is
   begin
      loop
         select
            accept Start do
               -- Ejecutar tarea
               Put_Line("Ejecutando tarea " &
Character'Image(Character'Val(Self'Identity)));

               -- Esperar tiempo de cómputo
               delay Duration'Value(T'Compute_Time);

               -- Mostrar mensaje de finalización de tarea
               Put_Line("Finalizada tarea " &
Character'Image(Character'Val(Self'Identity)));
            end Start;
         end select;
      end loop;
   end Tarea;

   procedure Ejecutivo_Ciclico is
      T_A, T_B, T_C, T_D, T_E : Tarea;
      Superperiodo : constant Time_Span := Milliseconds(200);
      Periodo_Secundario : constant Time_Span := Milliseconds(25);
```

```
   begin
      -- Crear tareas
      T_A := new Tarea;
      T_B := new Tarea;
      T_C := new Tarea;
      T_D := new Tarea;
      T_E := new Tarea;

      loop
         -- Iniciar nuevo marco
         Put_Line("Inicio de marco");

         -- Iniciar tareas en el orden deseado
         T_A.Start;
         T_B.Start;
         T_C.Start;
         T_D.Start;
         T_E.Start;

         -- Esperar periodo secundario
         delay Duration'Value(Periodo_Secundario);

         -- Mostrar finalización de marco
         Put_Line("Final de marco");

         -- Esperar hasta el próximo marco
         delay Duration'Value(Superperiodo - Periodo_Secundario);
      end loop;

   end Ejecutivo_Ciclico;

end pkg_tareas;
```

Donde cada tarea se modela con un tipo "**Tarea**" en un bucle infinito (loop) para aceptar el inicio de la tarea (Start), ejecutar su cómputo durante el tiempo especificado (T'Compute_Time), y luego finalizar la tarea.

Y **Ejecutivo_Ciclico** es el procedimiento que coordina el ciclo de ejecución de las tareas. Se definen las tareas *T_A, T_B, T_C, T_D y T_E*, cada una inicializada como una nueva instancia de **Tarea**. Dentro de un bucle infinito (loop), se inician secuencialmente las tareas en el orden definido, se espera el periodo secundario (Periodo_Secundario), se muestra el inicio y fin del marco de ejecución, y se espera hasta el próximo superperiodo.

Y desde finalmente tenemos el main.adb:

```
with pkg_tareas;

procedure Main is
   use pkg_tareas;

begin
   Ejecutivo_Ciclico;
end Main;
```

Para tener un uso sencillo y modular del paquete.

Ejercicio 6

Se pedía para este ejercicio modelar el enunciado de los caníbales comensales famoso en la programación concurrente, y se ha resuelto de esta forma:

```
with Ada.Text_IO;
with Ada.Real_Time;
with Ada.Integer_Text_IO;

use Ada.Text_IO;
use Ada.Real_Time;
use Ada.Integer_Text_IO;

procedure Canibales_Comensales is
   N : constant Integer := 5;

   protected Olla is
      entry Servir_Comida;
      function Hay_Raciones return Boolean;
      entry Rellenar_Olla;
   private
      Cantidad_Raciones : Integer := 0;
   end Olla;

   protected body Olla is
      entry Servir_Comida when Cantidad_Raciones > 0 is
      begin
         Cantidad_Raciones := Cantidad_Raciones - 1;
         Put_Line("Se sirvió una ración. Raciones restantes: " &
Integer'Image(Cantidad_Raciones));
      end Servir_Comida;

      function Hay_Raciones return Boolean is
      begin
         return Cantidad_Raciones > 0;
      end Hay_Raciones;

      entry Rellenar_Olla when Cantidad_Raciones = 0 is
      begin
         Cantidad_Raciones := N;
         Put_Line("Olla rellenada. Raciones disponibles: " &
Integer'Image(Cantidad_Raciones));
      end Rellenar_Olla;
   end Olla;

   task type Canibal is
      entry Comer;
      entry Bailar;
   end Canibal;
```

```
   task body Canibal is
   begin
      loop
         accept Bailar do
            Put_Line("El caníbal está bailando");
            delay 2.0; -- Delay de 2 segundos (convertido a milisegundos
por defecto)
         end Bailar;

         accept Comer do
            Put_Line("El caníbal pide de comer");

            if Olla.Hay_Raciones then
               Olla.Servir_Comida;
               Put_Line("El caníbal está comiendo");
               delay 1.0; -- Delay de 1 segundo (convertido a
milisegundos por defecto)
            else
               Put_Line("El caníbal avisa de que no queda comida");
               Olla.Rellenar_Olla;
            end if;
         end Comer;
      end loop;
   end Canibal;

   task type Cocinero is
      entry Rellenar_Olla;
   end Cocinero;

   task body Cocinero is
   begin
      loop
         delay 5.0; -- Delay de 5 segundos (convertido a milisegundos por
defecto)
         Put_Line("El cocinero va a rellenar la olla");
         Olla.Rellenar_Olla;
      end loop;
   end Cocinero;

   Canibal1 : Canibal;
   Canibal2 : Canibal;
   Cocinero1 : Cocinero;

begin
   Canibal1.Bailar;
   Canibal1.Comer;
   Canibal2.Bailar;
   Canibal2.Comer;
```

```
   Cocinero1.Rellenar_Olla;
end Canibales_Comensales;
```

Para la solución se han usado tareas y objetos protegidos para gestionar la sincronización. El objeto protegido **Olla** gestiona el estado de la olla y proporciona operaciones sincronizadas para servir y rellenar la olla:

- *Servir_Comida*: para que un caníbal tome una ración si hay raciones disponibles.
- *Hay_Raciones*: función que devuelve True si hay raciones disponibles en la olla.
- *Rellenar_Olla:* para rellenar la olla cuando esté vacía.

Con la tarea **Canibal** representamos los dos estados que puede tener:

- *Bailar*: Entrada para que el caníbal baile durante un tiempo determinado.
- *Comer*: Entrada para que el caníbal intente comer, sirviendo una ración si está disponible o pidiendo al cocinero que rellene la olla si está vacía.

Y con **Cocinero** añadimos al responsable de *Rellenar_Olla*.

Ejercicio 7

Cuenta_bancaria.ads

```
with Ada.Text_IO;
with Ada.Integer_Text_IO;

use Ada.Text_IO;
use Ada.Integer_Text_IO;

package Cuenta_Bancaria is

   protected type Cuenta is
      entry Consultar_Saldo;
      entry Extraer_Dinero(Cantidad: in Integer);
      entry Realizar_Ingreso(Cantidad: in Integer);
      entry Transferir(Cuenta_Destino: access Cuenta; Cantidad: in
Integer);
   private
      Saldo: Integer := 0;
   end Cuenta;

end Cuenta_Bancaria;
```

Cuenta_bancaria.adb

```
with Ada.Text_IO;
with Ada.Integer_Text_IO;

use Ada.Text_IO;
use Ada.Integer_Text_IO;

package body Cuenta_Bancaria is

   protected body Cuenta is

      entry Consultar_Saldo when True is
      begin
         Put_Line("Saldo actual: " & Integer'Image(Saldo));
      end Consultar_Saldo;

      entry Extraer_Dinero(Cantidad: in Integer) when True is
      begin
         if Saldo >= Cantidad then
            Saldo := Saldo - Cantidad;
            Put_Line("Extracción exitosa. Nuevo saldo: " &
Integer'Image(Saldo));
         else
```

```
            Put_Line("No hay saldo suficiente");
         end if;
      end Extraer_Dinero;

      entry Realizar_Ingreso(Cantidad: in Integer) when True is
      begin
         Saldo := Saldo + Cantidad;
         Put_Line("Ingreso exitoso. Nuevo saldo: " &
Integer'Image(Saldo));
      end Realizar_Ingreso;

      entry Transferir(Cuenta_Destino: access Cuenta; Cantidad: in
Integer) when True is
      begin
         if Saldo >= Cantidad then
            Saldo := Saldo - Cantidad;
            Cuenta_Destino.Realizar_Ingreso(Cantidad);
            Put_Line("Transferencia exitosa. Saldo restante: " &
Integer'Image(Saldo));
         else
            Put_Line("Fondos insuficientes para la transferencia.");
         end if;
      end Transferir;

   end Cuenta;

end Cuenta_Bancaria;
```

Para este ejercicio al igual que antes usemos tipos protegidos para garantizar la sincronización.

El paquete **Cuenta_Bancaria** contiene la definición del tipo protegido **Cuenta**, que encapsula las operaciones permitidas:

- *Consultar_Saldo*: imprime el saldo actual de la cuenta.
- *Extraer_Dinero*: verifica si hay suficiente saldo para la extracción y, si es así, disminuye el saldo en la cantidad especificada.
- *Realizar_Ingreso*: incrementa el saldo de la cuenta en la cantidad especificada.
- *Transferir*: verifica si hay suficiente saldo para la transferencia y, si es así, resta el saldo y realiza un ingreso en la cuenta de destino.