Sistemas en Tiempo Real – Práctica ADA 2

Ejercicio 1

Vamos a implementar el objeto de sincronización denominado barrera (ver la Práctica Posix 1 para los detalles del objeto). Para ello, se deberán cumplir las siguientes características:

- Crear el paquete *paquetebarrera* que implemente la barrera. Dentro del paquete se definirán:
  - El objeto protegido *barrera_t* (se implementará usando abstracción de datos, ver el anexo) que constará de los siguientes elementos:
    - Elementos privados
      - Número de tareas que se bloquearán en la barrera (positivo).
      - Número de tareas actualmente bloqueadas en la barrera (natural).
      - Bandera que indicará si las tareas pueden continuar (verdadero) o bloquearse (falso) (booleano).
    - Elementos públicos
      - Procedimiento *Sincronizar* sin parámetros → Incrementa el número de tareas que hay dentro de la barrera y, en caso de ser la última tarea en llegar, desbloqueará a todas las tareas bloqueadas.
      - Entrada *Wait* sin parámetros → Bloqueará a la tarea invocante mientras la bandera no sea verdadera. Una vez que una tarea quede desbloqueada, decrementará el número de tareas que están bloqueadas en la barrera y, en caso de ser la última en salir, pondrá la bandera a falso. Este proceso dejará la barrera lista para volver a utilizarse tras haber salido todas las tareas.
      - Procedimiento *Resetear* con un parámetro de entrada de tipo positivo → Si no hay ninguna tarea bloqueada en la barrera, cambiará el número de tareas que se bloquearán por el dato indicado en el parámetro de entrada del procedimiento.
  - Procedimiento *SincronizarEnBarrera* con un parámetro de entrada salida de tipo puntero a *barrera_t* → Se encargará de invocar a los dos métodos del objeto protegido, incrementando el número de tareas en la barrera y bloqueándose en la misma.
  - Procedimiento *ResetearBarrera* con un parámetro de entrada de tipo positivo → Se encargará de invocar al método *Resetear* del objeto protegido.
  - Al utilizar estos dos procedimientos en lugar de los métodos del objeto protegido, se garantiza el uso sencillo de la barrera y permite modificar la implementación de la misma sin afectar a los usuarios del paquete.
- Definir una tarea tipo que simplemente invocará al procedimiento *SincronizarEnBarrera* para sincronizarse con el resto de tareas y mostrará dos mensajes por pantalla, uno antes y otro después de entrar en la barrera.
- Crear e invocar 10 tareas usando el tipo tarea anterior.

Ejercicio 2

Implementar un programa que cree una tarea que espere la señal SIGINT (ver el anexo para saber cómo capturar la señal SIGINT en ADA). Dicha tarea deberá esperar hasta que el número de ocurrencias de la señal sea 10, momento en el que terminará la tarea y, con ella, el programa.

La práctica se comprimirá en un único archivo que será entregado a través del campus virtual usando la tarea creada para tal efecto.

ANEXO – Ayuda para ADA

Creación de paquetes

Se crearán dos ficheros en la carpeta src del proyecto:

- nombre_paquete.ads: definición del paquete.
- nombre_paquete.adb: implementación del paquete.

El nombre de ambos ficheros, así como el nombre del paquete, deben ser el mismo y no se podrán usar mayúsculas.

Paso de parámetros de tipo puntero a tareas

Para poder pasar parámetros de tipo puntero a las tareas debemos usar la palabra reservada *access* delante del tipo (sería similar al * en C++). Para evitar posibles problemas, se definirá un tipo puntero al tipo original, que será el usado en la tarea.

Sin embargo, como lo que queremos es pasar la dirección de una variable estática (lo que en C++ sería el &), usaremos *access all* en lugar de sólo *access* en la definición del tipo de los parámetros de la tarea.

Además, cuando definamos la variable que queremos pasar por declaración, usaremos la palabra *aliased* delante del nombre del tipo de la variable (esto permite acceder a la dirección de la misma) y, posteriormente usaremos el atributo '*Access* para acceder a la dirección de la variable.

Por último, en el caso de necesitar usar luego ese parámetro como una variable estática (no como puntero), por ejemplo a la hora de pasar parámetros a métodos que reciban el tipo estático (sin puntero), usaremos el campo *.all* que tienen todos los punteros. Esto hace referencia al contenido del puntero, no a la dirección.

Resumiendo:

```
Variable: aliased TipoNormal;--Variable de tipo no puntero
TipoPuntero is access all TipoNormal;--Nuevo tipo puntero
task type TareaTipo(Parametro: TipoPuntero);--Parámetro de la
                                        --tarea de tipo puntero
task body TareaTipo is begin … end;--Cuerpo de la tarea
Tarea: TareaTipo(Variable'Access);--Creación de la tarea pasando
                     --por parámetro la dirección de la variable
```

Algunas funciones de utilidad

- Put (paquete GNAT.IO): Muestra por pantalla una cadena de caracteres o un entero.
- New_Line (paquete GNAT.IO): Imprime un salto de línea.
- Put_Line (paquete GNAT.IO): Imprime por pantalla una cadena de caracteres y la termina con un salto de línea.
- integer'Image(Dato): Devuelve el valor del entero *Dato* en forma de cadena de caracteres.
- integer'Value(Dato): Devuelve el valor de la cadena de caracteres *Dato* en forma de entero.
- &: Operador de concatenación.

Abstracción de datos en ADA

La abstracción de datos busca que el usuario de un paquete no conozca cómo está implementada una parte de dicho paquete, por ejemplo, los objetos protegidos (como es el caso de esta práctica). Haciendo esto se consigue que, en caso de cambiar un objeto protegido de un paquete (incluso añadiendo o eliminando métodos) no repercuta en la utilización del mismo por parte del usuario, ya que el objeto es usado por métodos definidos en el paquete, y serán estos métodos los utilizados por el usuario. Los pasos a seguir son los siguientes:

- Archivo .ads:
  - Definición abstracta y pública del objeto protegido:
    ```
    type TipoAbstracto is limited private;
    ```
    Es necesario predefinir el tipo antes para poder ser utilizado tanto por el usuario del paquete como para cualquier otro elemento que se defina antes de la cláusula *private* y que necesite usar dicho tipo.
  - Definición de los procedimientos y funciones que usarán los métodos del objeto protegido (pueden usarlos porque están dentro del mismo paquete):
    ```
    procedure P1(Param: in out TipoAbstracto[; otros parámetros]);
    procedure P2(Param: in out TipoAbstracto[; otros parámetros]);
    function F(Param: in TipoAbstracto[; otros parámetros]);
    ```
    Obsérvese que los procedimientos usan el parámetro como entrada salida, mientras que en las funciones se usa únicamente como entrada. Esto es debido a que las funciones se usan únicamente para acceso de lectura, mientras que los procedimientos y entradas se usan para modificar el contenido del objeto, por lo que deben ser de entrada/salida.
  - Definición privada del objeto protegido:
    ```
    private
        protected type TipoAbstracto is
            procedure Procedimiento (dato: TipoDato);
            entry Entrada (dato: out TipoDato);
            function Funcion;
            ...
        private
            -- Atributos privados del objeto
            ...
        end;
    ```
  - Cualquier otra definición necesaria en el paquete.

- Archivo .adb:
  - Implementación de los procedimientos/funciones. Estos métodos incluirán invocaciones a los métodos del objeto protegido.
    ```
    procedure Procedimiento(TA: in out TipoAbstracto[; otros parámetros]) is begin
        ...
        TA.P(...);
        ...
    end;
    procedure Entrada(TA: in out TipoAbstracto[; otros parámetros]) is begin
        ...
        TA.E(...);
        ...
    end;
    ```

```
function Funcion(TA: in TipoAbstracto[; otros parámetros]) is begin
    ...
    return TA.F(...);
end;
```

- Implementación de los métodos del objeto protegido:

```
protected body TipoAbstracto is
    procedure Procedimiento ([parámetros]); is begin
        ...
    end;
    entry Entrada ([parámetros]) when Condicion is begin
        ...
    end;
    function Funcion([parámetros]) return Tipo is begin
        ...
    end;
    ...
end TipoAbstracto;
```

- Cualquier otra implementación del paquete

Captura de SIGINT en ADA

Para capturar la señal SIGINT en ADA debemos crear un manejador de evento. Para ello debemos crear un objeto protegido (sin la cláusula *type*) con dos métodos, un procedimiento, que será el manejador de la señal, y una entrada, que usaremos para esperar a la ocurrencia de la señal en la tarea esporádica.

Dentro de la definición del objeto protegido hay que incluir los siguientes pragmas (deben incluirse tras definir el procedimiento para que no de error de compilación por no encontrarlo):

```
-- Pragma para indicar que ProcedimientoControlador es un
-- manejador de interrupciones
pragma Interrupt_Handler(ProcedimientoControlador);

-- Pragma para indicar que ProcedimientoControlador va a
-- controlar la señal SIGINT
pragma Attach_Handler(ProcedimientoControlador, SIGINT);

-- Pragma necesario para que el programa pueda manejar las
-- señales
pragma Unreserve_All_Interrupts;
```

Para poder utilizar señales, debemos incluir los paquetes `Ada.Interrupts` y `Ada.Interrupts.Names`.