Sistemas Operativos

Concurrencia en ADA

Curso 2025

Facultad de Ingeniería, UDELAR

Agenda

1. Introducción
2. Tareas (tasks)
3. Comunicación entre tareas
4. Entry call selectiva
5. Array de entradas

Introducción

Introducción a la concurrencia en ADA

- Lenguaje de alto nivel que incluye primitivas para programación concurrente como parte de su especificación.
  - Fue uno de los primeros lenguajes en hacer esto.
- Nombrado en honor a Ada Lovelace (1815–1852)
- El término Task (tarea) modela una actividad concurrente.
- Cada tarea representa un hilo de ejecución independiente.
- Las tareas son iniciadas automáticamente al comienzo de la ejecución de un programa.
- Un programa termina cuando finaliza la ejecución de todas sus tareas.

Tareas (tasks)

Declaración de tareas

- Las tareas son declaradas en dos partes: especificación y cuerpo.

```
procedure EJEMPLO is
  task CALCULADORA is
    begin
        ...
    end CALCULADORA;
    task body CALCULADORA is
    begin
        ...
  end CALCULADORA;
begin
    ...
end EJEMPLO;
```

Declaración de tareas

- Los tipos de tareas permiten iniciar más de una tarea con la misma lógica (si no se crean instancias del tipo no hace nada)

```
procedure EJEMPLO is
  task type CALCULADORA is
    begin
        ...
    end CALCULADORA;
    task body CALCULADORA is
    begin
        ...
  end CALCULADORA;
  hilos: array(2) of CALCULADORA;
begin
  ...
end EJEMPLO;
```

Comunicación entre tareas

Entradas de comunicación entre tareas

- Comunicación implementada en base a entradas (entry).
- En la especificación de cada tarea se deben declarar sus entradas.

```
task CALCULADORA is
    entry SUM(A,B: Integer, out C: Integer);
end CALCULADORA;
```

- Cualquier otra tarea puede realizar una entry call e invocar la entrada de otra tarea.

```
CALCULADORA.SUM(3, 6, result);
```

- Si fuera un task type como en la diapositiva 5:

```
hilos[1].SUM(3, 6, result);
```

Aceptando invocaciones a entradas

- El cuerpo de la tarea debe definir la lógica para aceptar las **entry call** mediante la sentencia accept.
- La sentencia **accept** acepta una **entry call** por vez.

```
task body CALCULADORA is
begin
    loop
        accept SUM(A,B: Integer, out C: Integer) do
            C := A + B;
        end SUM;
    end loop;
end CALCULADORA
```

Rendezvous entre tareas

- Las sentencias **entry call** y **accept** son bloqueantes y sincronizan las tareas invocada e invocadora.
- La tarea invocadora permanecerá bloqueada en la **entry call** hasta que su invocación sea aceptada y finalice la ejecución del código dentro del **accept** que la aceptó.
- La tarea invocada permanecerá bloqueada en un **accept** hasta recibir una **entry call** en la entrada especificada.
- Esta sincronización entre dos tareas tiene en nombre de rendezvous (cita o encuentro).

Seleccionando entre más de una entrada

- La sentencia **select** espera simultáneamente por **entry calls** en más de un **accept** y responde a la que llega primero.

```
loop
    select
        accept SUM(A,B: Integer, out C: Integer) do
            C := A + B;
        end SUM;
        -- otro código opcional
    or
        accept MULT(A,B: Integer, out C:Integer) do
            C := A * B;
        end MULT;
    end select;
end loop;
```

Guardas condicionales en la selección de entradas

- Las guardas (clausulas when) son condiciones booleanas que abren o cierran los **accept** de un **select**.

```
loop
    select
        when CANT < 5 =>
            accept ENTRAR();
            CANT := CANT + 1;
    or
        when CANT > 0 =>
            accept SALIR();
            CANT := CANT - 1;
    end select;
    ...
end loop;
```

Guardas condicionales en la selección de entradas

- Todas las guardas son evaluadas al comenzar el **select** y luego permanecen inmutables.
- Debe existir al menos una guarda que evalúe verdadero o un **accept** sin guarda.
    - De lo contrario el programa abortará.
- Las **entry call** de una misma entrada siempre son aceptadas en orden de llegada.
- Sin embargo no se garantiza orden de atención para **entry call** de diferentes entradas.
    - En este caso la atención se realiza de forma no determinista.

Guardas condicionales en la selección de entradas

- El atributo Count de una entrada indica cuantas **entry call** están esperando atención en esa entrada.
- Permite implementar un orden entre entradas.
- Solo aplica para las entradas del propio task.

```
select
    when MULT'Count = 0 =>
        accept SUM(A,B: Integer, out C: Integer) do
            C := A + B;
        end SUM;
or
    accept MULT(A,B: Integer, out C: Integer) do
        C := A * B;
    end MULT;
end select;
```

Alternativa ELSE en la selección de entradas

- Acepta un encuentro en una de las entradas con guarda abierta si hay una **entry call** esperando atención.
- En caso de no haber una **entry call** esperando atención, se ejecuta inmediatamente el código de la clausula else.
  - Ojo con busy waiting

```
select
    accept SUM(A,B:int,out C:int) do
        ...
    else
        ...
end select;
```

Alternativa OR DELAY en la selección de entradas

- Espera X segundos por una **entry call** en alguna de las entradas con guarda abierta.
- Pasado ese tiempo se ejecuta el código de la clausula or delay.

```
select
    accept SUM(A,B:int,out C:int) do
        ...
or
    delay 3.0 -- espera 3 segundos
    ...
end select;
```

Entry call selectiva

Entry call selectiva: ELSE

- Intenta una **entry call**. En caso que no se acepte inmediatamente el redezvous, se continúa con el código else.

```
select
    DOCTOR.CONSULTA();
    ...
else
    tomar_una_aspirina();
    ir_a_la_cama();
end select;
```

Entry call selectiva: OR DELAY

- Intenta una **entry call** y se bloquea a lo más durante X segundos en espera de que se produzca el redezvous.

```
select
    DOCTOR.CONSULTA();
    ...
or
    delay 86400.0; -- 24 horas
    ir_al_hospital();
end select;
```

Resumen Select

1. Se evalúan todas las guardas para determinar cuáles entradas están abiertas.
   - La entradas sin guarda siempre están abiertas.
2. Si hay entradas abiertas con al menos un pedido de encuentro pendiente se selecciona una cualquiera de ellas y se atiende su primer **entry call**.
   - Dentro de una misma entrada el orden es FIFO.
3. Si no hay pedidos de encuentro en las entradas abiertas:
   3.1 Si hay ELSE entonces se sale por la clausula ELSE.
   3.2 Si hay DELAY X entonces se esperan X segundos por un encuentro y si no ocurre se sale por la clausula DELAY.
   3.3 Si hay no hay ELSE ni DELAY X, se bloquea hasta el primero pedido de encuentro.
4. Si no hay entradas abiertas se aborta el programa.

Array de entradas

Array de entradas

- Se pueden definir arrays de entradas si se quiere tener varias entradas distintas pero relacionadas y con los mismos parámetros
- En el ejemplo siguiente se tiene un task que modela 3 depósitos

```
task DEPOSITOS is
     entry AGREGAR (1..3) (A: Objeto);
     entry SACAR (1..3) (A: Objeto);
end DEPOSITOS;
task body DEPOSITOS is
     ...
     accept AGREGAR(1)(A: Objeto)
     ...
     accept AGREGAR(2)(A: Objeto)
     ...
     accept AGREGAR(3)(A: Objeto)
     ...
end DEPOSITOS
```

Array de entradas

- Solo se pueden aceptar entradas de una entry específica (no vale **`accept`** `AGREGAR(i)` donde `i` no está definida)
- Al invocar se usa `DEPOSITOS.AGREGAR(2)(obj)`
- En este ejemplo puede parecer igual a un array de tasks pero:
  - Esta solución permite tener información compartida entre todos los depósitos (ej. una cantidad máxima total de objetos entre los 3)
  - Solo hay un hilo de ejecución