Programación Concurrente ATIC
Redictado Programación Concurrente

Clase 9

Facultad de Informática

UNLP

ADA– Lenguaje con Rendezvous

El lenguaje ADA

- Desarrollado por el Departamento de Defensa de USA para que sea el estandard en programación de aplicaciones de defensa (desde sistemas de Tiempo Real a grandes sistemas de información).
- Desde el punto de vista de la concurrencia, un programa Ada tiene ***tasks*** (tareas) que pueden ejecutar independientemente y contienen primitivas de sincronización.
- Los puntos de invocación (entrada) a una tarea se denominan ***entrys*** y están especificados en la parte visible (header de la tarea).
- Una tarea puede decidir si acepta la comunicación con otro proceso, mediante la primitiva ***accept***.
- Se puede declarar un ***type task***, y luego crear instancias de procesos (tareas) identificado con dicho tipo (arreglo, puntero, instancia simple).

Tasks

- La forma más común de especificación de task es:

  TASK *nombre* **IS**
  declaraciones de ENTRYs
  end;

- La forma más común de cuerpo de task es:

  TASK BODY *nombre* **IS**
  declaraciones locales
  BEGIN
  sentencias
  END *nombre***;**

- Una especificación de TASK define una única tarea.
- Una instancia del correspondiente ***task body*** se crea en el bloque en el cual se declara el TASK.

Sincronización
Call: *Entry Call*

- El ***rendezvous*** es el principal mecanismo de sincronización en Ada y también es el mecanismo de comunicación primario.

- ***Entry:***
  - Declaración de ***entry simples y familia de entry*** (parámetros IN, OUT y IN OUT)**.**
  - ***Entry call.*** La ejecución demora al llamador hasta que la operación E terminó (o abortó o alcanzó una excepción).
  - ***Entry call condicional***:

```
select entry call;
        sentencias adicionales;
else
        sentencias;
end select;
```

  - ***Entry call temporal:***

```
select entry call;
        sentencias adicionales;
or delay tiempo
        sentencias;
end select;
```

Sincronización

Sentencia de Entrada: *Accept*

- La tarea que declara un entry sirve llamados al entry con *accept*:

  accept *nombre* **(parámetros formales) do** sentencias **end *nombre*;**

- Demora la tarea hasta que haya una invocación, copia los parámetros reales en los parámetros formales, y ejecuta las sentencias. Cuando termina, los parámetros formales de salida son copiados a los parámetros reales. Luego ambos procesos continúan.

- La ***sentencia wait selectiva*** soporta comunicación guardada.

  select *when* $B_1 \Rightarrow$ ***accept*** E_1; sentencias$_1$
  or ...
  or *when* $B_n \Rightarrow$ ***accept*** E_n; sentencias$_n$
  end select;

  - Cada línea se llama ***alternativa***. Las cláusulas ***when*** son opcionales.
  - Puede contener una alternativa ***else, or delay, or terminate.***
  - Uso de atributos del entry: ***count, calleable.***

Ejemplo

Mailbox para 1 mensajes

```
TASK TYPE Mailbox IS
      ENTRY Depositar (msg: IN mensaje);
      ENTRY Retirar (msg: OUT mensaje);
END Mailbox;

A, B, C : Mailbox;

TASK BODY Mailbox IS
    dato: mensaje;
BEGIN
      LOOP
         ACCEPT Depositar (msg: IN mensaje) DO dato :=  msg;  END Depositar;
         ACCEPT Retirar (msg: OUT mensaje) DO msg := dato;  END Retirar;
      END LOOP;
END Mailbox;
```

Podemos utilizar estos mailbox para manejar mensajes: *A.Depositar(x1); B.Depositar(x2); C.Retirar(x3);*

Ejemplo

Mailbox para N mensajes

```
TASK Mailbox IS
      ENTRY Depositar (msg: IN mensaje);
      ENTRY Retirar (msg: OUT mensaje);
END Mailbox;

TASK BODY Mailbox IS
       datos: array (0..N-1) of mensaje;
       cant, pri, ult integer := 0;
BEGIN
      LOOP
            SELECT
                  WHEN cant < N => ACCEPT Depositar (msg: IN mensaje) DO
                                                ult := (ult MOD N); datos[ult] := msg; cant := cant +1;
                              END Depositar;
            OR
                  WHEN cant > 0 => ACCEPT Retirar (msg: OUT mensaje) DO
                                          msg := datos[pri]; pri := (pri MOD N); cant := cant -1;
                              END Retirar;
            END SELECT;
      END LOOP;
END Mailbox;
```

Ejemplo
Lectores-Escritores

```
Procedure Lectores-Escritores is
   Task Sched IS
            Entry  InicioLeer;
            Entry  FinLeer;
            Entry  InicioEscribir;
            Entry  FinEscribir;
   End Sched;
   Task type Lector;
   Task body Lector is
       Begin
           Loop
               Sched.InicioLeer; … Sched.FinLeer;
            End loop;
   End Lector;
   Task type Escritor;
   Task body Escritor is
       Begin
           Loop
               Sched.InicioEscribir; … Sched.FinEscribir;
            End loop;
   End Lector;
   VecLectores: array (1..cantL) of Lector;
   VecEscritores: array (1..cantE) of Escritor;
```

```
Task body Sched is
        numLect: integer :=0;
    Begin
        Loop
            Select
                When InicioEscribir'Count = 0 =>
                    accept  InicioLeer;
                    numLect := numLect+1;
            or   accept FinLeer;
                 numLect := numLect-1;
            or   When numLect = 0 =>
                    accept InicioEscribir;
                    accept FinEscribir;
                    For i in 1..InicioLeer‘count loop
                        accept InicioLeer;
                        numLect:= numLect +1;
                    End loop;
            End select;
       End loop;
End Sched;

Begin
     Null;
End Lectores-Escritores
```

Primitivas Múltiples

La Notación de Primitivas Múltiples

- RPC y rendezvous → un proceso inicia la comunicación con un ***call***, que bloquea al llamador hasta que la operación es servida y se retornan los resultados. Ideales para interacciones Cliente/Servidor, pero difícil programar algoritmos filtros o peers que intercambian información.
- **Notación de Primitivas Múltiples:** combina **RPC**, **Rendezvous** y **PMA** en un paquete coherente.
  - ✓ Brinda gran poder expresivo combinando ventajas de las 3 componentes, y poder adicional.
  - ✓ Programas → colecciones de módulos. Una operación visible se declara en la especificación del módulo. Puede ser invocada por procesos de otros módulos, y es servida por un proceso o procedure del módulo que la declara.
  - ✓ También se usan operaciones ***locales***, que son declaradas, invocadas y servidas dentro del cuerpo de un único módulo.

La Notación de Primitivas Múltiples

- Una operación puede ser invocada por ***call sincrónico*** o por ***send asincrónico***:

  call *Mname.op* (argumentos)

  send *Mname.op* (argumentos)

  - ✓ El call termina cuando la operación fue servida y los resultados fueron retornados.
  - ✓ El send termina tan pronto como los argumentos fueron evaluados.

- Una operación es servida por UN procedure (proc) o por rendezvous (puede ser servida por más de 1 sentencia IN – Comparten la cola de pendientes). La elección la toma el programador del módulo.

Invocación	*Servicio*	*Efecto*
call	**proc**	Llamado a procedimiento
call	**in**	Rendezvous
send	**proc**	Creación dinámica de proceso
send	**in**	PMA