

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 estandar 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  $E_1$ ; sentencias1  
or    ...  
or    when  $B_n \Rightarrow$  accept  $E_n$ ; sentenciasn  
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 (Buffer Limitado)

- ♦ Solución vista para Rendezvous general

```
module BufferLimitado
  op depositar (typeT), retirar (result typeT);
body
  process Buffer
  { queue buf;
    int cantidad = 0;

    while (true)
      { in depositar (item) and cantidad < n → push (buf, item);
        cantidad = cantidad + 1;
        □ retirar (item) and cantidad > 0 → pop (buf, item);
          cantidad = cantidad - 1;
        ni
      }
  }
end BufferLimitado
```


Ejemplo

Mailbox para N mensajes (Buffer Limitado)

♦ Solución en ADA

TASK Mailbox IS

ENTRY Depositar (msg: IN mensaje);

ENTRY Retirar (msg: OUT mensaje);

END Mailbox;

TASK BODY Mailbox IS

buf: queue;

cantidad integer := 0;

BEGIN

LOOP

SELECT

WHEN cantidad < N => ACCEPT Depositar (msg: IN mensaje) DO

push (buf, msg);

cantidad := cantidad + 1;

END Depositar;

OR

WHEN cantidad > 0 => ACCEPT Retirar (msg: OUT mensaje) DO

pop (buf, msg);

cantidad = cantidad - 1;

END Retirar;

END SELECT;

END LOOP;

END Mailbox;

Ejemplo

Filósofos Centralizado

♦ Solución vista para Rendezvous general

```
module Mesa
  op tomar(int), dejar(int);
body
  process Mozo
    { bool comiendo[5] =([5] false);
      while (true)
        in tomar(i) and not (comiendo[izq(i)] or comiendo[der(i)]) → comiendo[i] = true;
        □ dejar(i) → comiendo[i] = false;
        ni
      }
  end Mesa

Process Filosofo [i = 0 to 4]
  { while (true)
    { call tomar(i);
      come;
      call dejar(i);
      piensa;
    }
  }
```

Ejemplo

Filósofos Centralizado

♦ Solución en ADA – Múltiples entry

TASK Mesa IS

```
ENTRY Tomar0; ENTRY Tomar1; ENTRY Tomar2; ENTRY Tomar3; ENTRY Tomar4;  
ENTRY Dejar (id: IN integer);
```

END Mesa;

TASK BODY Mesa IS

```
Comiendo: array (0..4) of bool := (0..4=> false);
```

BEGIN

```
For i in 0..4 loop
```

```
    Filósofos(i).identificacion(i);
```

```
end loop;
```

LOOP

SELECT

```
    when (not (comiendo(4) or comiendo(1)) => ACCEPT Tomar0; comiendo(0) := true;
```

```
OR when (not (comiendo(0) or comiendo(2)) => ACCEPT Tomar1; comiendo(1) := true;
```

```
OR when (not (comiendo(1) or comiendo(3)) => ACCEPT Tomar2; comiendo(2) := true;
```

```
OR when (not (comiendo(2) or comiendo(4)) => ACCEPT Tomar3; comiendo(3) := true;
```

```
OR when (not (comiendo(3) or comiendo(0)) => ACCEPT Tomar4; comiendo(4) := true;
```

```
OR ACCEPT Dejar(id: IN integer) do
```

```
    comiendo(id) := false;
```

```
    end Dejar;
```

```
END SELECT;
```

```
END LOOP;
```

END Mesa;

Ejemplo

Filósofos Centralizado

♦ Solución en ADA – Múltiples entry

TASK TYPE Filosofo IS

ENTRY Identificar (identificacion: IN integer);

END Filosofo;

Filosophos: array (0..4) of Filosofo;

TASK BODY Filosofo IS

id: integer;

BEGIN

ACCEPT Identificar (identificacion : IN integer) do

id := identificación;

End Identificar;

LOOP

if (id = 0) then Mesa.Tomar0;

else if (id = 1) then Mesa.Tomar1;

else if (id = 2) then Mesa.Tomar2;

else if (id = 3) then Mesa.Tomar3;

else if (id = 4) then Mesa.Tomar4;

//Come

Mesa.Dejar(id);

//Piensa

END LOOP;

END Mesa;

Ejemplo

Filósofos Centralizado

♦ Solución en ADA – Encolar pedidos

TASK TYPE Filosofo IS

ENTRY Identificar (identificacion: IN integer);

ENTRY Comer;

END Filosofo;

TASK Mesa IS

ENTRY Tomar (id: IN integer);

ENTRY Dejar (id: IN integer);

END Mesa;

Filosofos: array (0..4) of Filosofo;

TASK BODY Filosofo IS

id: integer;

BEGIN

ACCEPT Identificar (identificacion : IN integer) do id := identificación; End Identificar;

LOOP

Mesa.Tomar(id);

Accept Comer;

//Come

Mesa.Dejar(id);

//Piensa

END LOOP;

END Mesa;

Ejemplo

Filósofos Centralizado

♦ Solución en ADA – Encolar pedidos

TASK BODY Mesa IS

Comiendo: array (0..4) of bool := (0..4=> false);

QuiereC: array (0..4) of bool := (0..4=> false);

aux: integer;

BEGIN

For i in 0..4 loop Filósofos(i).identificacion(i); end loop;

LOOP

SELECT

ACCEPT Tomar(id: IN integer) do aux := id; END Tomar;

if (not (comiendo((aux+1) mod 5) or comiendo((aux-1) mod 5))) then

comiendo(aux) = true;

Filósofos(aux).Comer;

else QuiereC (aux) := true; end if;

OR ACCEPT Dejar(id: IN integer) do aux := id; end Dejar;

comiendo(aux) := false;

for i in 0..4 loop

if (QuiereC(i) and not (comiendo((i+1) mod 5) or comiendo((i-1) mod 5))) then

comiendo(i) = true;

QuiereC(i) := false;

Filósofos(i).Comer;

end if;

end loop;

END SELECT;

END LOOP;

END Mesa;

Ejemplo

Time Server

- ◆ Solución vista para Rendezvous general

```
module TimeServer
  op get_time () returns int;
  op delay (int);
  op tick ();

body TimeServer
  process Timer
    { int tod = 0;
      while (true)
        in get_time () returns time → time = tod;
        □ delay (waketime) and waketime <= tod by waketime → skip;
        □ tick () → tod = tod + 1; reiniciar timer;
        ni
    }
end TimeServer
```

Ejemplo

Time Server

♦ Solución en ADA

PROCEDURE DESPERTADORES IS

Task TimeServer is

entry get_time (hora: OUT int); entry delay (hd, id: IN int); entry tick;

End TimeServer;

Task Reloj;

Task Type Cliente Is

entry Identificar (identificacion: IN integer); entry seguir;

End Cliente;

ArrClientes: array (1..C) of Cliente;

Task Body Cliente Is

id: integer; hora: integer;

BEGIN

ACCEPT Identificar (identificacion : IN integer) do id := identificación; End Identificar;

TimeServer.get_time(hora);

TimeServer.delay(hora+.....);

ACCEPT seguir;

End Cliente;

Task Body Reloj is

BEGIN

loop delay(1); TimeServer.tick; end loop;

End Reloj;

Ejemplo

Time Server

♦ Solución en ADA

Task Body TimeServer is

```
    actual: integer := 0;
    dormidos: colaOrdenada;
    auxId, auxHora: integer;
BEGIN
    LOOP
        SELECT
            when (tick'count = 0) => ACCEPT get_time (hora: OUT integer) do  hora := actual; END get_time;
        OR when (tick'count = 0) => ACCEPT delay(hd, id: IN integer) do agregar(dormidos, (id,hd)); END delay;
        OR ACCEPT tick;
            actual := actual + 1;
            while (not empty (dormidos)) and then (VerHoraPrimero(dormidos) <= actual)) loop
                sacar(dormidos, (auxId, auxHora);
                ArrClientes[auxId].seguir;
            end loop;
        END SELECT;
    END LOOP;
End TimeServer;

BEGIN
    for i in 1..C loop
        ArrClientes(i).identificacion(i);
    end loop;
END DESPERTADORES;
```

Ejemplo

Alocador SJN

- ♦ Solución vista para Rendezvous general

```
module Alocador_SJN
  op pedir(int tiempo), liberar;

body
  process SJN
    { bool libre = true;
      while (true)
        in pedir (tiempo) and libre by tiempo → libre = false;
        □ liberar ( ) → libre = true;
      ni
    }
end SJN_Allocator
```

Ejemplo

Alocador SJN

♦ Solución en ADA

PROCEDURE SchedulerSJN IS

Task Alocador_SJN is

 entry pedir (tiempo, id: IN int);
 entry liberar;

End Alocador_SJN ;

Task Type Cliente Is

 entry Identificar (identificacion: IN integer);
 entry usar;

End Cliente;

ArrClientes: array (1..C) of Cliente;

Task Body Cliente Is

 id: integer; tiempo: integer;

BEGIN

 ACCEPT Identificar (identificacion : IN integer) do id := identificación; End Identificar;

 loop

//trabaja y determina el valor de tiempo

 Alocador_SJN.pedir(id, tiempo);

 Accept usar;

//Usa el recurso

 Alocador_SJN.liberar;

 end loop;

End Cliente;

Ejemplo

Alocador SJN

♦ Solución en ADA

Task Body Alocador_SJN is

```
libre: boolean := true;  
espera: colaOrdenada;  
tiempo, aux: integer;
```

Begin

```
loop  
  aux := -1;  
  select  
    accept Pedir (tiempo, id: IN integer) do  
      if (libre) then libre:= false; aux := id;  
      else agregar(espera, (id, tiempo)); end if;  
    end Pedir;  
  or accept liberar;  
    if (empty (espera)) then libre := true;  
    else sacar(espera, (aux, tiempo)); end if;  
  end select;  
  if (aux <> -1) then ArrClientes[aux].usar; end if;  
end loop;
```

End Alocador_SJN ;

BEGIN

```
for i in 1..C loop  
  ArrClientes(i).identificacion(i);  
end loop;
```

END SchedulerSJN;

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*