Programación Concurrente 2016

Clase 9

Facultad de Informática
UNLP

Resumen de la clase anterior

Pasaje de mensajes asincrónico

- **Análisis de la resolución de un mismo problema sobre diferentes arquitecturas**
- **Pares: Cálculo de la topología de una red**
- **MPI**

Pasaje de mensajes sincrónico

- **Concurrencia y deadlock**
- **CSP – comunicación guardada**
- **Ejemplos de filtros, C/S y pares**
- **Occam**

Programación Paralela con el concepto de *Bag of Tasks*

Idea: tener una “bolsa” de tareas que pueden ser compartidas por procesos “worker”.

C/ worker ejecuta un código básico

```
while (true) {
    obtener una tarea de la bolsa
    if (no hay más tareas)
        BREAK;    # exit del WHILE
    ejecutar tarea (incluyendo creación de tareas);
 }
```

Puede usarse p/ resolver problemas con un n° fijo de tareas y p/ soluciones recursivas con nuevas tareas creadas dinámicamente.

El paradigma de “bag of tasks” es sencillo, escalable (aunque no necesariamente en performance) y favorece el balance de carga entre los procesos.

Multiplicación de matrices con *Bag of Tasks*

Multiplicación de 2 matrices a y b de nxn $\Rightarrow$ **n^2** productos internos e/ filas de a y columnas de b.
C/ producto interno es independiente y se puede realizar en paralelo.

Sup. máquina con PR procesadores (PR < n)$\Rightarrow$ PR procesos *worker*.

Para balancear la carga, c/u debería computar casi el mismo número de productos internos (buscando trabajo cuando no tiene).

Si PR << n, un buen tamaño de tarea sería una o unas pocas filas de la matriz resultado *c*. Por simplicidad, suponemos 1 fila.
Inicialmente, la bolsa contiene n tareas, una por fila.

Pueden estar ordenadas de cualquier manera $\Rightarrow$ se puede representar la bolsa, simplemente contando filas: **INT proxfila = 0;**
Un worker saca una tarea de la bolsa con: **<fila=proxfila; proxfila++; >** donde fila es una variable local (Fetch and Add).

Multiplicación de matrices con *Bag of Tasks*

```
INT proxfila =0              # "bolsa" de tareas
DOUBLE a[n,n], b[n,n], c[n,n];

PROCESS Worker [w=1 TO PR]  {
    INT fila;
    DOUBLE suma;
    WHILE (true) {
            # obtener una tarea
            < fila = proxfila; proxfila++; >
             IF (fila >= n)
                 BREAK;
             Calcular los productos internos para c[fila, *] ;
     }
}
```

Para terminar el proceso se pueden contar los BREAK, al llegar a n se tiene la matriz c completa y se puede finalizar el cálculo.

LINDA. Primitivas para concurrencia

LINDA ⇒ aproximación distintiva al procesamiento concurrente que combina aspectos de MC y PMA.

NO es un lenguaje de programación, sino un conjunto de 6 primitivas que operan sobre una MC donde hay "**tuplas nombradas**" (*tagged tuples*) **que pueden ser *pasivas* (datos) o *activas* (tareas)**.

Puede agregarse como biblioteca a un lenguaje secuencial.

El núcleo de LINDA es el *espacio de tuplas* compartido (TS) que puede verse como un único canal de comunicaciones compartido, *pero en el que no existe orden*:

- Depositar una tupla (**OUT**) funciona como un **SEND**.
- Extraer una tupla (**IN**) funciona como un **RECEIVE.**
- **RD** permite "leer"como un RECEIVE pero sin extraer la tupla de TS.
- **EVAL** permite creación de procesos (tuplas activas) dentro de TS.
- Por último **INP y RDP** permiten hacer IN y RD no bloqueantes.

Si bien hablamos de MC, TS puede estar físicamente distribuida en una arq. multiprocesador (más coomplejo) ⇒ puede usarse para almacenar estr. de datos distribuidas, y ≠ procesos pueden acceder concurrentemente diferentes elementos de las mismas

LINDA. Primitivas para concurrencia

TS $\Rightarrow$ colección no ordenada de tuplas pasivas y activas.

- Tuplas de datos (pasivas): registros tagged que contienen el estado compartido de una computación.
- Tuplas activas (de proceso): rutinas que ejecutan asincrónicamente.
- Cuando una tupla proceso termina se convierte en tupla de datos

Interacción $\Rightarrow$ leyendo, escribiendo y generando tuplas de datos.

Cuando una tupla proceso termina, se convierte en tupla de datos.

Cada tupla de datos en TS tiene la forma:

("tag", value$_1$, ..., value$_n$)

"tag" es un string literal para distinguir entre tuplas que representan distintas estructuras de datos.

Las primitivas básicas (*y atómicas*) para manipular tuplas de datos son OUT, IN y RD

LINDA. Primitivas para depositar y extraer del TS

Un proceso deposita una tupla en TS ejecutando:

out("tag", expr$_1$, ..., expr$_n$)

- La ejecución de out termina cuando las expresiones fueron evaluadas y la tupla de datos resultante fue depositada en TS.
- Similar a send, pero la tupla se almacena en un TS no ordenado en lugar de ser agregada a un canal específico.

Un proceso extrae una tupla de datos de TS ejecutando:

in("tag", field$_1$, ..., field$_n$)

- Cada field$_i$ o es una expresión o un parámetro formal de la forma *?var* donde *var* es una vble local en el proceso ejecutante.
- Los argumentos p/ in son llamados *template*. El proc que ejecuta in se demora hasta que TS contenga al menos 1 tupla que matchee el template, y luego remueve una de TS.
- El *tag* actúa como un nombre de canal.

LINDA. Primitivas para depositar y extraer del TS

Ejemplo 1: semáforo
La cantidad de tuplas “sem” en TS da el valor del semáforo “sem”)

OUT(“sem”) ≡ V(sem)
IN(“sem”) ≡ P(sem)

Ejemplo 2: arreglo de semáforos

IN(“forks”, i) ≡ P(forks[i])
OUT(“forks”,i) ≡ V(forks[i])

rd se usa para examinar tuplas de datos.

- Si ***t*** es un template, **rd(t)** demora al proceso hasta que TS contiene una tupla de datos que matchee. Como con in, las vbles en *t* son asignadas con los valores de los campos de la tupla de datos.
- La tupla permanece en TS.

LINDA. Primitivas para examinar tuplas de datos

inp y rdp son variantes no bloqueantes de **in** y **rd.**

- Son predicados que retornan true si hay una tupla matching en TS, y false en otro caso.
- Si retornan true, tienen el mismo efecto que in y rd respectivamente.

Brindan una manera de que un proceso haga polling sobre TS.

Ejemplo: contador barrera para **n** procesos.

```
OUT("barrier", 0);        # inicialización por parte de algún proceso

IN ("barrier", ?counter)          # toma la tupla barrier
OUT("barrier", counter+1)         # pone el nuevo valor
RD("barrier", n);                 # espera a que lleguen los n
```

LINDA. Primitivas para tuplas proceso

eval crea tuplas proceso. Tiene la misma forma que **out** y también produce una nueva tupla:

eval("tag", $expr_1$, ..., $expr_n$)

- Al menos una expresión es un call a función o procedimiento. Todos los campos de **eval** pueden ser tratados concurrentemente por procesos diferentes y el proceso que escribe el **eval** no espera.

- Cuando todos los campos son tratados (incluida la ejecución del o los procedimientos/funciones) la tupla se convierte en pasiva (quedan sólo datos) y se agrega a TS.

***eval* provee el medio para incorporar concurrencia en un programa Linda.**

LINDA. Primitivas para tuplas proceso

Ejemplo: sea la sentencia concurrente:

co [i := 1 to n] a[i] := f(i);

Evalúa *n* llamados de f en paralelo y asigna los resultados al arreglo compartido *a*.

El código *C-Linda* es:

```
for ( i = 1;  i ≤ n; i++ )
  eval ("a", i, f(i));
```

Dispara (forks) n tuplas proceso; c/u evalúa un llamado de f(i).

Cuando una tupla proceso termina, se convierte en tupla de datos con: nombre del arreglo, índice y valor de ese elemento del arreglo ***a*.**

Conceptos de RPC y Rendezvous

El PM se ajusta bien a problemas de filtros y pares que interactúan, ya que se plantea la ***comunicación unidireccional***.

Para resolver C/S la comunicación bidireccional obliga a especificar 2 tipos de canales (requerimientos y respuestas).

Además, cada cliente necesita un canal de reply distinto...

RPC (Remote Procedure Call) y Rendezvous $\Rightarrow$ técnicas de comunicación y sincronización entre procesos que suponen ***un canal bidireccional*** $\Rightarrow$ ideales para programar aplicaciones C/S

RPC y Rendezvous combinan una interfaz “tipo monitor” con operaciones exportadas a través de llamadas externas (CALL) con mensajes sincrónicos (demoran al llamador hasta que la operación llamada se termine de ejecutar y se devuelvan los resultados).

Conceptos de RPC y Rendezvous. Diferencias

Difieren en la manera de servir la invocación de operaciones:

- Un enfoque es declarar un procedure p/ c/ operación y crear un nuevo proceso (al menos conceptualmente) p/ manejar c/ llamado (RPC porque el llamador y el cuerpo del procedure pueden estar en distintas máquinas)
P/ el cliente, durante la ejecución del servicio, es como si tuviera en su sitio el proceso remoto que lo sirve (Ej: JAVA)

- El segundo enfoque es hacer *rendezvous* con un proceso existente.
Un rendezvous es servido por una sentencia de E/ (o accept) que espera una invocación, la procesa y devuelve los resultados (Ej:Ada)

Rendezvous *extendido*, para diferenciarlo del *simple* de PMS.
Modelo de CSP extendido p/ tratar procesos dinámicos y bidireccionales

Remote Procedure Call

Los programas se descomponen en **módulos** (con procesos y procedures), que pueden residir en espacios de direcciones distintos.

Los procesos de un módulo pueden compartir variables y llamar a procedures de ese módulo.

Un proceso en un módulo puede comunicarse con procesos de otro módulo sólo invocando procedimientos exportados por éste.

Los módulos tienen especificación e implementación de procedures

module *Mname*
  headers de procedures exportados (visibles)
body
  declaraciones de variables
  código de inicialización
  cuerpos de procedures exportados
  procedures y procesos locales
end

Remote Procedure Call

Los procesos locales son llamados *background* para distinguirlos de las operaciones exportadas.

Header de un procedure visible::
op *opname* (formales) [**returns** result]

El cuerpo de un procedure visible es contenido en una declaración proc:
proc *opname*(identif. formales) **returns** identificador resultado
declaración de variables locales
sentencias
end

Un proceso (o procedure) en un módulo llama a un procedure en otro ejecutando: **call *Mname.opname* (argumentos)**

Para un llamado local, el nombre del módulo se puede omitir.

Remote Procedure Call

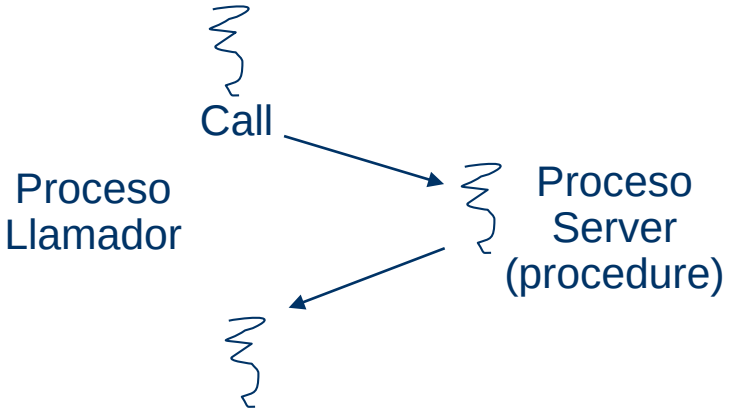
La implementación de un llamado intermódulo es distinta que p/ uno local, ya que los dos módulos pueden estar en distintos espacios: un **nuevo proceso** sirve el llamado, y los argumentos son pasados como mensajes e/ el llamador y el proceso server.

El llamador se demora mientras el proceso servidor ejecuta el cuerpo del procedure que implementa *opname*.

Cuando el server vuelve de *opname* envía los resultados al llamador y termina. Después de recibir los resultados, el llamador sigue.

Si el proceso llamador y el procedure están en el mismo espacio de direcciones, es posible evitar crear un nuevo proceso.

En gral, un llamado será remoto ⇒ se debe crear un proceso server o alocarlo de un pool preexistente.



Remote Procedure Call. Sincronización en Módulos

Por sí mismo, RPC es solo un mecanismo de comunicación.

Aunque un proceso llamador y su server sincronizan, el único rol del server es actuar en nombre del llamador (como si éste estuviera ejecutando el llamado $\Rightarrow$ la sincronización entre ambos es implícita).

Necesitamos que los procesos en un módulo sincronicen (procesos server ejecutando llamados remotos y procesos del módulo).
Esto comprende EM y SxC.

Existen dos enfoques para proveer sincronización, dependiendo de si los procesos en un módulo ejecutan ***con exclusión mutua (un solo proceso por vez) o concurrentemente***.

Remote Procedure Call. Sincronización en Módulos

Si ejecutan con EM las VC son protegidas automáticamente contra acceso concurrente, pero es necesario programar SxC.

Si pueden ejecutar concurrentemente necesitamos mecanismos para programar EM y SxC **(c*/* módulo es un programa concurrente)**
$\Rightarrow$ podemos usar cualquier método ya descripto (semáforos, monitores, o incluso rendezvous).

Es más general asumir que los procesos pueden ejecutar concurrentemente (más eficiente en un multiprocesador de MC).

Asumimos que procesos en un módulo ejecutan concurrentemente, usando por ej. time slicing.
Por ahora, usamos semáforos para programar EM y SxC; luego veremos el agregado de rendezvous y comunicación guardada.

Remote Procedure Call. Ejemplo: Time Server

Módulo que brinda servicios de timing a procesos cliente en otros módulos.
Dos operaciones visibles: ***get_time*** **y** ***delay*****(*****interval*****)**
Un proceso interno que continuamente inicia un timer por hardware, luego incrementa el tiempo al ocurrir la interrupción de timer.

```
module TimeServer
    op get_time( ) returns INT;    # recupera la hora del día
    op delay(INT interval );       # ticks del intervalo de demora

  body
    inicializaciones…

    implementaciones de get_time y delay…

   implementación de procesos locales (Clock en este caso)

end TimeServer;
```

Remote Procedure Call. Ejemplo: Time Server

```
module TimeServer
   op get_time( ) returns INT;      # recupera la hora del día
   op delay(INT interval );         # ticks del intervalo de demora
body
   INT tod = 0;                     # hora del día
   SEM m= 1;                        # semáforo para exclusión mutua
   SEM d[n] = ([n] 0);              # semáforos de demora privados
   QUEUE of (INT waketime, INT proces_id) napQ;

proc get_time ( ) returns time {
      time := tod;
   }
proc delay(interval)  {
    # asumimos interval > 0
    INT waketime = tod + interval;
    P(m)
    insert (waketime, myid) en napQ;
    V(m);
    P(d[myid]);  }   # espera a ser despertado

      Process Clock  {
        Inicia timer por hardware;
        WHILE (true) {
          Esperar interruptción y rearrancar timer
          tod := tod + 1;
          P(m);
          while tod ≥ min waketime en napQ {
            remove (waketime, id) de napQ;
            V(d[id]);  }# despierta al proceso Id
         V(m);   }
        }

    end TimeServer;
```

Remote Procedure Call. Ejemplo: Time Server

Múltiples clientes pueden llamar a *get_time* y a *delay* a la vez $\Rightarrow$ múltiples procesos “servidores” estarían atendiendo los llamados concurrentemente.

Los pedidos de *get_time* se pueden atender concurrentemente porque sólo significan leer la variable tod.

Pero, *delay* y clock necesitan ejecutarse con EM porque manipulan *napQ*, la cola de procesos cliente "durmiendo”.

El valor de *myid* en *delay* se supone un entero único e/ 0 y n-1. Se usa p/ indicar el semáforo privado sobre el cual está esperando un cliente.

RPC. Manejo de caches en un sistema de archivos distribuido

Versión simplificada de un problema que se da en sistemas de archivos y BD distribuidos.

Procesos de aplicación que ejecutan en una WS, y archivos de datos almacenados en un FS.
Los pgms de aplicación que quieren acceder a datos del FS, llaman procedimientos read y write del módulo local *FileCache*.
Leen o escriben arreglos de caracteres

Los archivos se almacenan en el FS en bloques de 1024 bytes, fijos.
El módulo FileServer maneja el acceso a bloques del disco; provee dos procedimientos (ReadBlk y WriteBlk).

El módulo *FileCache* mantiene en cache los bloques recientemente leídos. Al recibir pedido de *read*, *FileCache* 1ro chequea si los bytes solicitados están en su cache. Sino, llama al proc *readblock* del *FileServer.* Algo similar ocurre con los *write*.

RPC. Manejo de caches en un sistema de archivos distribuido. Módulo *FileCache*

```
Module FileCache     # ubicado en cada workstation
    op read (INT count ;  result CHAR buffer[ *] );
    op write (INT count; CHAR buffer[*] );
body
    declaración del cache de N bloques de archivo;
    variables para la descripción de los registros de cada file;
    declaración de semáforos para sincronización de acceso al cache;
    proc read (count, buffer)   {
        IF (los datos pedidos no están en el cache)   {
                seleccionar los bloques del cache a usar;
                IF (se necesita vaciar parte del cache);
                    FileServer.writeblk(....);
                FileServer.readblk(....);            }
        buffer= número de bytes requeridos del cache;      }
    proc write(count, buffer)   {
        IF (los datos apropiados no están en el cache)   {
                seleccionar los bloques del cache a usar;
                IF (se necesita vaciar parte del cache);
                    FileServer.writeblk(....);       }
        bloquedeCache= número de bytes desde buffer;   }
end FileCache;
```

RPC. Manejo de caches en un sistema de archivos distribuido.

Los llamados de los programas de aplicación de las WS son locales a su *FileCache*, pero desde estos módulos se invocan los procesos remotos de *FileServer*.
FileCache es un server para procesos de aplicación; *FileServer* es un server para múltiples clientes *FileCache,* uno por WS

Si existe un *FileCache* por pgm de aplicación, no se requiere sincronización interna entre los read y write, porque sólo uno puede estar activo. Si múltiples programas de aplicación usaran el mismo *FileCache*, tendríamos que usar semáforos para implementar la EM en el acceso a *FileCache*.

En cambio en *FileServer* se requiere sincronización interna, ya que atiende múltiples *FileCache* y contiene un proceso *DiskDriver* (la sincronización no se muestra en el código).

RPC. Manejo de caches en un sistema de archivos distribuido. Módulo *Fileserver*.

```
Module FileServer     # ubicado en el servidor
    op readblk (INT fileid, offset;  result CHAR blk[1024] );
    op writeblk (INT fileid, offset; CHAR blk[1024] );
body
    declaración del cache de bloques del disco;
    cola de pedidos pendientes de acceso al disco;
    declaración de semáforos para acceso al cache y a la cola;
    # a continuación no se ve el código de sincronización.

    proc readblk (fileid, offset, blk)   {
            IF (los datos pedidos no están en el cache)   {
                almacenar el pedido de lectura en la cola del disco;
                esperar que la operación de lectura sea procesada;
            }
            blk= bloques pedidos del disco;
    }
```

RPC. Manejo de caches en un sistema de archivos distribuido. Módulo *Fileserver*.

```
proc writeblk (fileid, offset, blk)   {
            Ubicar el bloque en el cache;
            IF (es necesario grabar físicamente en el disco)   {
              almacenar el pedido de escritura en la cola del disco;
              esperar que la operación de escritura sea procesada;
              }
            bloque cache = blk;
    }

process DiskDriver   {
    WHILE (true)   {
            esperar por un pedido de acceso físico al disco;
            arrancar una operación física; esperar interrupción;
            despertar el proceso que está esperando completar el
                request;
     }
}
end FileServer;
```

RPC. Implementación de filtros y peers

El problema de construir una red de filtros (por ejemplo para la ordenación) es más complicado con RPC que con mensajes.
Esto se debe a que no es un problema “tipo Cliente-Servidor”.

Es necesario introducir paths de comunicación dinámicos por medio de *capabilities* (punteros a operaciones en otros módulos).
TAREA: Leer el ejemplo del libro de Andrews

El intercambio de valores entre pares, por ejemplo el problema que resolvimos en clases anteriores con mensajes asincrónicos también resulta algo más complejo con RPC.
Si dos procesos en distintos módulos deben intercambiar valores, cada uno debe exportar un procedimiento que el otro módulo llamará.
TAREA: Leer el ejemplo del libro de Andrews

RPC. Implementación de filtros y peers. Intercambio de valores

Si 2 procesos de distintos módulos deben intercambiar valores, cada módulo debe exportar un procedimiento que el otro módulo llamará.

```
module Intercambio [i = 1 to 2]
    op depositar(int);
body
    int otrovalor;
    sem listo = 0;              # usado para señalizar
 proc depositar(otro)  {        # llamado por otro módulo
    otrovalor = otro;           # salva el valor del otro
    V(listo);          }        # deja que el Worker lo tome

 process Worker {
    int mivalor;
    call Intercambio[3-i].depositar(mivalor);     # envía al otro
    P(listo);                   # espera a recibir el valor del otro
    .......
   }
end Intercambio
```

RPC. Implementación de filtros y peers. Filtros Merge

```
optype stream = (int);        # tipo de las operaciones de stream de datos
module Merge[i = 1 to n]
    op in1 stream, in2 stream;        # streams de entrada
    op inicializar(cap stream);       # link a stream de salida
body
    int v1, v2;                  # valores de entrada desde los streams 1 y 2
    cap stream out;              # capability para stream de salida
    sem vacio1 = 1, lleno1 = 0, vacio2 = 1, lleno2 = 0;
 proc inicializar(salida) {out = salida; }
 proc in1(valor1) {P(vacio1); v1 = valor1; V(lleno1); }
 proc in2(valor2) {P(vacio2); v2 = valor2; V(lleno2); }
 process M {
    P(lleno1); P(lleno2);      # espera los dos valores de entrada
    while (v1 != EOS and v2 != EOS)
            if (v1 <= v2)  { call out(v1); V(vacio1); P(lleno1); }
               else { call out(v2); V(vacio2); P(lleno2); }
    if (v1 == EOS)   while (v2 != EOS) { call out(v2); V(vacio2); P(lleno2); }
       else while (v1 != EOS) { call out(v1); V(vacio1); P(lleno1); }
    call out(EOS); # agrega el valor centinela}
end Merge
```

JAVA. Remote Method Invocation (RMI)

Java soporta el uso de RPC en programas distribuidos mediante la invocación de métodos remotos (RMI).

Una aplicación que usa RMI tiene 3 componentes:
- Una interfase que declara los headers para métodos remotos
- Una clase server que implementa la interfase
- Uno o más clientes que llaman a los métodos remotos

El server y los clientes pueden residir en máquinas diferentes

Tarea: ver los ejemplos en el libro de Andrews

Rendezvous

RPC por si mismo sólo brinda un mecanismo de comunicación intermódulo.
Dentro de un módulo es necesario programar la sincronización.

Además, a veces son necesarios procesos extra sólo para manipular los datos comunicados por medio de RPC (ej: Merge)

Rendezvous combina ***comunicación*** **y** ***sincronización***:

- Como con RPC, un proceso cliente **invoca** una operación por medio de un ***call***, pero esta operación **es servida por un proceso existente** en lugar de por uno nuevo.

- Un proceso servidor usa una *sentencia de entrada* para esperar por un call y actuar.

- Las operaciones se atienden una por vez más que concurrentemente

Rendezvous

La especificación de un módulo contiene declaraciones de los headers de las operaciones exportadas, pero el cuerpo consta de un único proceso que sirve operaciones

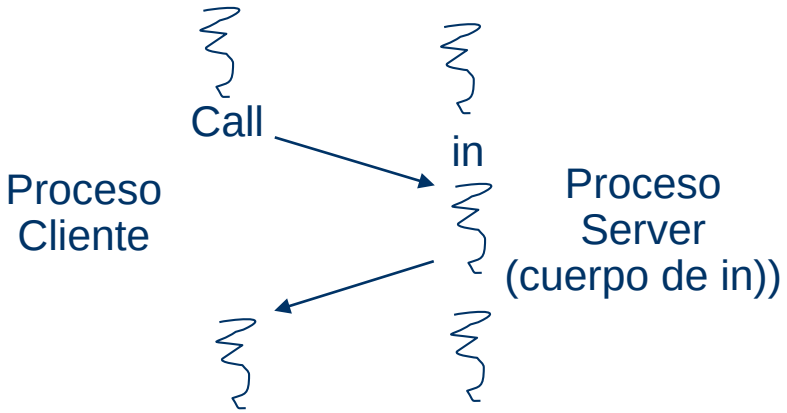
Si un módulo exporta ***opname***, el proceso server en el módulo realiza *rendezvous* con un llamador de **opname** ejecutando una sentencia de entrada:

in *opname*(identif. formales) → **S; ni**

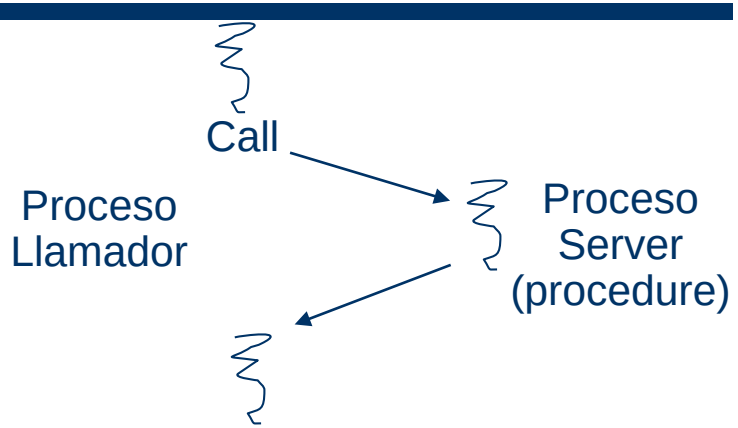
Las partes entre **in** y **ni** se llaman *operación guardada*.

Una sentencia de E/ demora al proceso server hasta que haya al menos un llamado pendiente de **opname**; luego elige el llamado pendiente más viejo, copia los argumentos en los parámetros formales, ejecuta **S** y finalmente retorna los parámetros de resultado al llamador. Luego, **ambos** procesos pueden continuar.

Rendezvous



Rendezvous: a diferencia de RPC, el server es un proceso activo



RPC

Combinando comunicación guardada con rendezvous:

in $\mathbf{op_1}$ (formales$_1$) **and** $\mathbf{B_1}$ **by** $\mathbf{e_1 \rightarrow S_1}$**;**

- **...**
- $\mathbf{op_n}$ (formales$_n$) **and** $\mathbf{B_n}$ **by** $\mathbf{e_n \rightarrow S_n}$**;**

ni

Las B_i son *expresiones de sincronización* opcionales
Los e_i son *expresiones de scheduling* opcionales.
En ambos casos, pueden referenciar a los parámetros formales.

Rendezvous.
Ejemplo: Buffer limitado

```
module BufferLimitado
   op depositar (typeT), retirar (result typeT);
body
   process Buffer {
      typeT buf[n];
      int ocupado = 0, libre = 0, cantidad = 0;
      while (true)
         in depositar (item) and cantidad < n →
                              buf[libre] = item;
                              libre = (libre+1) mod n;
                              cantidad = cantidad + 1;
          retirar (item) and cantidad > 0 →
                              item = buf[ocupado];
                              ocupado = (ocupado+1) mod n;
                              cantidad = cantidad - 1;
         ni
   }
end BufferLimitado
```

Rendezvous.
Ejemplo: Filósofos Centralizado

```
module Mesa
    op tomar_ten(int), dejar_ten(int);
body
    process Mozo {
            bool comiendo[5] =([5] false);
            while (true)
                in tomar_ten(i) and not (comiendo[izq(i)] or comiendo[der(i)] →
                                    comiendo[i] = true;
                □ dejar_ten(i) →
                                    comiendo[i] = false;
                ni      }
end Mesa
Process Filosofo [i = 0 to 4] {
    while (true) {
            call tomar_ten(i);
            come;
            call dejar_ten(i);
            piensa;
      }
  }
```

Rendezvous.
Ejemplo: Time Server

```
Module TimeServer
    op get_time () returns int;
    op delay (int);
    op tick ();  # llamada x el manejador de interrupción del clock

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

A diferencia de la solución con RPC, *waketime* hace referencia a la hora en que debe despertarse

Rendezvous.
Ejemplo: Alocador SJN

```
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
```

Tareas:

-Ver en el libro de Andrews los ejemplos de Red de filtros e Intercambio de valores con Rendezvous

Rendezvous. Intercambio de valores

```
module Intercambio[i = 1 to 2]
    op depositar(int);
body
  process Worker {
    int mivalor, otrovalor;
    if (i == 1) {
       call Intercambio[2].depositar(myvalue);
       in depositar(otrovalor) → skip; ni
     }
        else {
           in depositar(otrovalor) → skip; ni
           call Intercambio[1].depositar(mivalor);
     }
  .......
}
end Intercambio
```

Rendezvous. Filtros Merge

```
optype stream = (int);        # tipo de las operaciones de stream de datos
module Merge[i = 1 to n]
    op in1 stream, in2 stream;   # streams de entrada
    op inicializar(cap stream);   # link a stream de salida
body
  process Filtro {
    int v1, v2;                        # valores desde los streams de entrada
    cap stream out;                    # capability para stream salida
    in inicializar(c) → out = c ni
    in in1(v) → v1 = v; ni
    in in2(v) → v2 = v; ni
    while (v1 != EOS and v2 != EOS)
            if (v1 <= v2)  { call out(v1); in in1(v) → v1 = v; ni }
               else { call out(v2); in in2(v) → v2 = v; ni }
    if (v1 == EOS)   while (v2 != EOS) { call out(v2); in in2(v) → v2 = v; ni }
       else while (v1 != EOS) { call out(v1); in in1(v) → v1 = v; ni }
    call out(EOS);
  }
end Merge
```

Rendezvous. El lenguaje ADA

Desarrollado por Dpto de Defensa de USA para que sea el standard en programación de aplicaciones de defensa (desde STR a grandes sistemas de información)

Desde el punto de vista de la concurrencia, un pgm Ada tiene “tasks” (tareas o procesos) que pueden ejecutar independientemente y que contienen primitivas de sincronización.

Los puntos de invocación (entrada) a una task 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 ***tipo task***, y luego crear instancias de procesos (tareas) identificado con dicho tipo (arreglo, puntero, instancia simple).

Tasks en ADA

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 en ADA.
Call: Entry call.

El rendezvous es el principal mecanismo de sincronización en Ada y también es el mecanismo de comunicación primario.
Las declaraciones de entry son similares a las de op):

entry identificador (formales)

Los parámetros del entry pueden ser IN, OUT o IN OUT.
También soporta arreglos de entries, llamados *familias* de entry.

Entry call: Si el task T declara el entry E, otras tasks en el alcance de la especificación de T pueden invocar a E con:

call T.E (parámetros reales)

La ejecución del call demora al llamador hasta que la operación E terminó (o abortó o alcanzó una excepción).

Sincronización en ADA.
Call: Entry call.

Se usa *entry call condicional* si una tarea quiere hacer *polling* de otra:

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

Elige el entry call si puede ejecutarse de inmediato; sino, elige *else*

Se usa *entry call temporal* (*timed entry call)* si una tarea llamadora quiere esperar a otra a lo sumo un intervalo de tiempo:

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

Sincronización en ADA. Sentencia de Entrada: *Accept*

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

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

Demora la tarea hasta que haya una invocación, copia los parámetros reales en los formales, y ejecuta las sentencias.
Cuando termina, los parámetros formales de S/ son copiados a los parámetros reales.
Luego, el llamador y el proceso ejecutante continúan.

accept E1(parámetros formales) **do** cuerpo de E1 **end** E1**;**
accept E2(parámetros formales) **do** cuerpo de E2 **end** E2**;**
accept E3(parámetros formales) **do** cuerpo de E3 **end** E3**;**

Especifica que se *espera un pedido por el entry E1,* luego de atendido se espera un pedido por E2 y luego un pedido por E3.

Sincronización en ADA. Sentencia de Entrada: *Accept*

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 (salvo la última) se llama *alternativa*.

B_i son expr. booleanas, y las cláusulas when son opcionales.

Una alternativa está *abierta* si B_i es true o se omite el when.

Las B_i no pueden referenciar parámetros del entry call, y tampoco hay expresión de scheduling.

Demora al proceso hasta que el accept en alguna alternativa abierta pueda ejecutarse (hay una invocación pendiente del entry).

Puede contener una alternativa **else** (que se elige si no se puede elegir otra alternativa), **or delay** (se elige si transcurrió el intervalo, como un *timeout*), **or terminate** (se elige si todas las tareas que hacen RV con esta terminaron o están esperando un terminate)

Ejemplo. Mailbox para un mensaje

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

A, B, C : Mailbox;    -- Se declaran 3 instancias del tipo Mailbox.

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

Podemos utilizar estos mailbox para manejar mensajes:

```
call A.Depositar(x1);
call B.Depositar(x2);
call 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;
```

Sincronización en ADA. 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
```

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 C/S, pero difícil programar algoritmos filtros o peers que intercambian información (para éstos es mejor PMA)

⇒ **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 (sentencias in). La elección la toma el programador del módulo.

La *Notación de Primitivas Múltiples*

Invocación	***Servicio***	***Efecto***
call	**proc**	llamado a procedimiento
call	**in**	rendezvous
send	**proc**	creación dinámica de proceso
send	**in**	PMA

Un llamado a procedure es local si el llamador y el proc están en el mismo módulo; sino, es remoto. Una operación no puede ser servida tanto por proc como por in pues el significado no sería claro

Pero 1 operación puede ser servida por más de una sentencia de E/, quizás en más de un proceso en el módulo que la declara. En este caso, los procesos comparten la cola de invocaciones pendientes

TAREAS:
-Leer los ejemplos en el libro de Andrews.
-Leer las características del lenguaje MPD