Programación Concurrente ATIC
Redictado Programación Concurrente

Clase 8

Facultad de Informática

UNLP

Conceptos Básicos

- El Pasaje de Mensajes 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 ⇒ técnicas de comunicación y sincronización entre procesos que suponen ***un canal bidireccional*** ⇒ 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).

Diferencias entre RPC y Rendezvous

- Difieren en la manera de servir la invocación de operaciones.
  - Un enfoque es declarar un *procedure* para cada operación y crear un nuevo proceso (al menos conceptualmente) para manejar cada llamado (RPC porque el llamador y el cuerpo del *procedure* pueden estar en distintas máquinas). Para 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 Entrada* (o accept) que espera una invocación, la procesa y devuelve los resultados *(Ej:Ada).*

RPC

Remote Procedure Call (RPC)

- 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 (RPC)

- 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

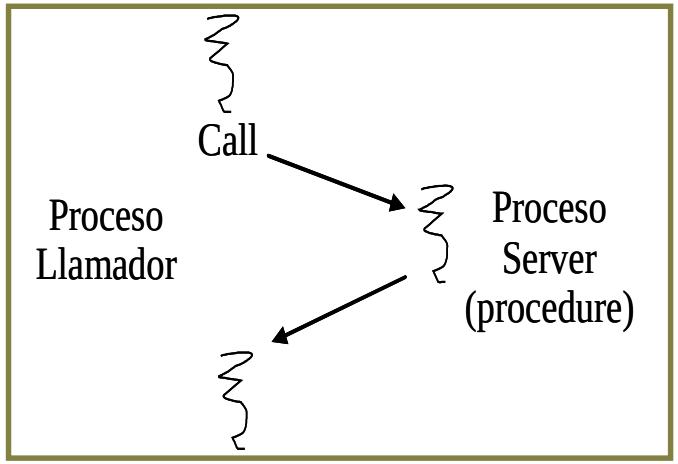
- Un proceso (o procedure) en un módulo llama a un procedure en otro ejecutando:

  call ***Mname.opname*** **(argumentos)**

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

Remote Procedure Call (RPC)

- La implementación de un llamado intermódulo es distinta que para 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 entre 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 general, un llamado será remoto ⇒ se debe crear un proceso server o alocarlo de un pool preexistente.



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 ⇒ 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 Exclusión Mutua y Sincronización por Condición.
- Existen dos enfoques para proveer sincronización, dependiendo de si los procesos en un módulo ejecutan:
  - ***Con exclusión mutua (un solo proceso por vez).***
  - ***Concurrentemente***.

Sincronización en módulos

- *Si ejecutan con Exclusión Mutua* las variables compartidas son protegidas automáticamente contra acceso concurrente, pero es necesario programar sincronización por condición.

- *Si pueden ejecutar concurrentemente* necesitamos mecanismos para programar exclusión mutua y sincronización por condición ***(cada módulo es un programa concurrente)*** ⇒ 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 memoria compartida). Asumimos que procesos en un módulo ejecutan concurrentemente, usando por ejemplo *time slicing*.

Ejemplo Cliente/Servidor

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;
    op delay(INT interval, INT myid);
  body
    INT tod = 0;
    SEM m= 1;
    SEM d[n] = ([n] 0);
    QUEUE of (INT waketime, INT id's) napQ;

   proc get_time ( ) returns time
     {   time := tod; }

   proc delay(interval, myid)
     {   INT waketime = tod + interval;
         P(m);
         insert ((waketime, myid) napQ);
         V(m);
         P(d[myid]);
     }

Process Clock
  { Inicia timer por hardware;
    WHILE (true)
      { Esperar interrupción,
        luego rearrancar timer;
        tod := tod + 1;
        P(m);
        WHILE tod ≥ min(waketime, napQ)
            { remove ((waketime, id), napQ);
              V(d[id]);
            }
        V(m);
      }
  }

end TimeServer;
```

Ejemplo Cliente/Servidor

Time Server

- Múltiples clientes pueden llamar a *get_time* y a *delay* a la vez ⇒ 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 Exclusión Mutua porque manipulan ***napQ***, la cola de procesos cliente "durmiendo”.
- El valor de ***myid*** en *delay* se supone un entero único entre 0 y n-1. Se usa para indicar el semáforo privado sobre el cual está esperando un cliente.

Ejemplo Cliente/Servidor

Manejo de caches en un sistema de archivos distribuido

- Versión simplificada de un problema que se da en sistemas de archivos y BD distribuidos.
- Suponemos procesos de aplicación que ejecutan en una WS, y archivos de datos almacenados en un FS. Los programas 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*** primero chequea si los bytes solicitados están en su cache. Sino, llama al procedimiento ***readblock*** del ***FileServer***. Algo similar ocurre con los ***write***.

Ejemplo Cliente/Servidor

Manejo de caches en un sistema de archivos distribuido

```
Module FileCache     # ubicado en cada workstation
    op read (INT count ;  result CHAR buffer[ *] );
    op write (INT count; CHAR buffer[*] );
body
   cache de N bloques; descripción de los registros de cada file; semáforos para sincronizar 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;
```

Ejemplo Cliente/Servidor

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 programa 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).

Ejemplo Cliente/Servidor

Manejo de caches en un sistema de archivos distribuido

```
Module FileServer     # ubicado en el servidor
    op readblk (INT fileid, offset;  result CHAR blk[1024] );
    op writeblk (INT fileid, offset; CHAR blk[1024] );
body
   cache de bloques; cola de pedidos pendientes; semáforos para acceso al cache y a la cola;
    proc readblk (fileid, offset, blk)
       {  IF (los datos pedidos no están en el cache) {encola el pedido;  esperar que la lectura sea procesada;}
          blk= bloques pedidos del disco;
       }
    proc writeblk (fileid, offset, blk)
       {   Ubicar el bloque en el cache;
          IF (es necesario grabar físicamente en disco) {encola el pedido; esperar que la 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;
```

Ejemplo Pares Interactuantes

Intercambio de valores

- Si dos procesos de diferentes 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;

 proc depositar(otro)
    { otrovalor = otro;
      V(listo);
    }

 process Worker
   {  int mivalor;
      call Intercambio[3-i].depositar(mivalor);
      P(listo); .....
   }

end Intercambio
```

RPC en 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.

Rendezvous

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***:

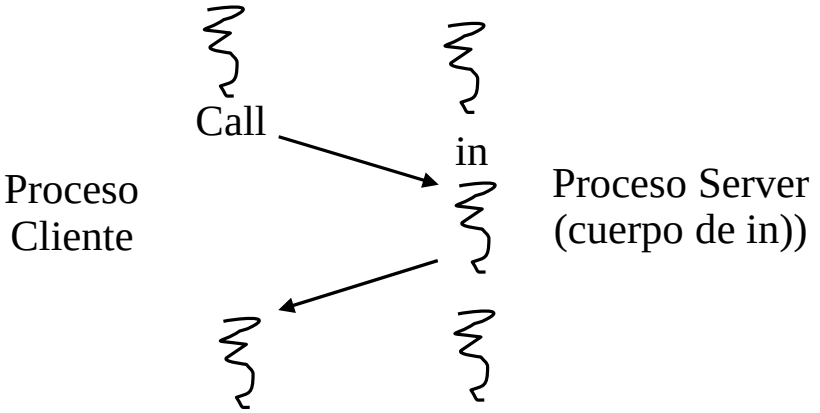
  in *opname* (parámetros formales) → **S; ni**

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

- Una sentencia de entrada 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

- A diferencia de RPC el server es un proceso activo.



- Combinando comunicación guardada con rendezvous:

in op$_1$ (formales$_1$) **and B$_1$ by e$_1$ → S$_1$;**
□ **...**
□ **op$_n$** (formales$_n$) **and B$_n$ by e$_n$ → S$_n$;**
ni

  - Los B_i son ***expresiones de sincronización*** opcionales.
  - Los e_i son ***expresiones de scheduling*** opcionales.

} Pueden referenciar a los parámetros formales.

Ejemplo
Buffer limitado

```
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

Filósofos Centralizado

```
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
Time Server

- A diferencia del ejemplo visto para RPC, *waketime* hace referencia a la hora que debe despertarse.

```
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
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
```

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);*