Programación Concurrente ATIC
Redictado de Programación Concurrente

Clase 7

Facultad de Informática

UNLP

Links al archivo con audio

La teoría con los audios está en formato MP4. Debe descargar los archivos comprimidos de los siguientes links:

- Programación concurrente en memoria distribuida: https://drive.google.com/uc?id=1Xdh_8do7D0XmRQBNDHmHf6MA3usfZaUS&export=download

- Pasaje de Mensajes Asincrónicos (PMA): https://drive.google.com/uc?id=1h3mD73WydEjYSlGXMxQCiJLwsyM3QEbt&export=download

Programación concurrente en memoria distribuida

Conceptos generales

- Arquitecturas de memoria distribuida ⇒ *procesadores* + *memo local* + *red de comunicaciones* + ***mecanismo de comunicación / sincronización*** ⇒ ***intercambio de mensajes.***
- ***Programa distribuido:*** programa concurrente comunicado por mensajes. Supone la ejecución sobre una arquitectura de memoria distribuida, aunque puedan ejecutarse sobre una de memoria compartida (o híbrida).
- ***Primitivas de pasaje de mensajes:*** interfaz con el sistema de comunicaciones ⇒ semáforos + datos + sincronización.
- Los procesos *SOLO* ***comparten canales*** (físicos o lógicos). Variantes para los canales:
  - Mailbox, input port, link.
  - Uni o bidireccionales.
  - Sincrónicos o asincrónicos.

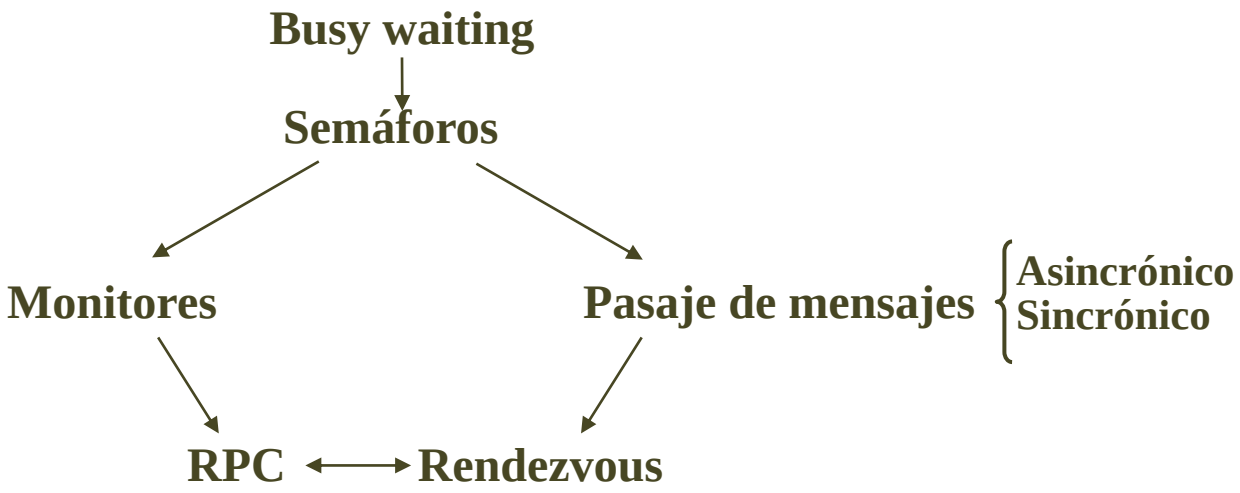
Características

- Los canales son lo único que comparten los procesos
  - Variables locales a un proceso (“cuidador”).
  - La exclusión mutua no requiere mecanismo especial.
  - Los procesos interactúan comunicándose.
  - Accedidos por primitivas de envío y recepción.
- Mecanismos para el Procesamiento Distribuido:
  - Pasaje de Mensajes Asincrónicos (PMA)
  - Pasaje de Mensajes Sincrónico (PMS)
  - Llamado a Procedimientos Remotos (RPC)
  - Rendezvous
- La sincronización de la comunicación interproceso depende del patrón de interacción:
  - Productores y consumidores (Filtros o pipes)
  - Clientes y servidores
  - Pares que interactúan

Cada mecanismo es más adecuado para determinados patrones

Relación entre mecanismos de sincronización

- ***Semáforos*** ⇒ mejora respecto de ***busy waiting***.
- ***Monitores*** ⇒ combinan Exclusión Mutua implícita y señalización explícita.
- ***PM*** ⇒ extiende semáforos con datos.
- ***RPC*** y ***rendezvous*** ⇒ combina la interface procedural de monitores con PM implícito.



Pasaje de Mensajes Asincrónicos (PMA)

Uso de canales en PMA

- ***PMA*** $\Rightarrow$ canales = colas de mensajes enviados y aún no recibidos.
- ***Declaración de canales*** $\rightarrow$ **chan** ***ch*** **(** id_1 **:** $tipo_1$**, ... ,** id_n **:** $tipo_n$ **)**
  - **chan** entrada (char);
  - **chan** acceso_disco (INT cilindro, INT bloque, INT cant, CHAR* buffer)**;**
  - **chan** resultado[n] (INT);
- ***Operación Send*** $\rightarrow$ un proceso agrega un mensaje al final de la cola ("ilimitada") de un canal ejecutando un *send*, que no bloquea al emisor:

  send ch(expr1, ... , exprn);
- ***Operación Receive*** $\rightarrow$ un proceso recibe un mensaje desde un canal con *receive*, que demora ("bloquea") al receptor hasta que en el canal haya al menos un mensaje; luego toma el primero y lo almacena en variables locales:

  receive ch(var_1***, ... ,*** var_n***);***

  Las variables del receive deben tener los mismos tipos que la declaración del canal.

  Receive es una primitiva ***bloqueante***, ya que produce un delay. **Semántica**: el proceso NO hace nada hasta recibir un mensaje en la cola correspondiente al canal. ***NO*** es necesario hacer polling.

Características de los canales

- Acceso a los contenidos de cada canal: atómico y respeta orden FIFO.
- En principio los canales son ilimitados, aunque las implementaciones reales tendrán un tamaño de buffer asignado.
- Se supone que los mensajes NO se pierden ni modifican y que todo mensaje enviado en algún momento puede ser "leído".
- ***empty(ch)*** → determina si la cola de un canal está vacía. Útil cuando el proceso puede hacer trabajo productivo mientras espera un mensaje, **pero debe usarse con cuidado**.
  - O podría ser ***false***, y no haber más mensajes cuando sigue ejecutando (si no en el único en recibir por ese canal).
  - La evaluación de ***empty*** podría ser ***true***, y sin embargo existir un mensaje al momento de que el proceso reanuda la ejecución.

Características de los canales

Los canales son declarados globales a los procesos, ya que pueden ser compartidos. Según la forma en que se usan podría ser:

- Cualquier proceso puede enviar o recibir por alguno de los canales declarados. En este caso suelen denominarse ***mailboxes***.
- En algunos casos un canal tiene un solo receptor y muchos emisores (***input port***).
- Si el canal tiene un único emisor y un único receptor se lo denomina ***link***: provee un "camino" entre el emisor y sus receptores.

Ejemplo

```
chan entrada(char), salida(char [CantMax]);

Process Carac_a_Linea
{ char linea [CantMax], int i = 0;
  WHILE (true)
     { receive entrada (linea[i]);
       WHILE (linea[i] ≠ CR and i < CantMax)
          {  i := i + 1;
               receive entrada (linea[i]);
          }
       linea [i] := EOL;
       send salida(linea);
       i := 0;
     }
 }
```

```
Process Proceso_1
{  char a;
   WHILE (true)
      {  leer_carácter_por_teclado(a);
         send entrada(a);
      }
 }

Process Proceso_2
{  char res[CantMax];
   WHILE (true)
      { receive salida(res);
        imprimir_en_pantalla(res);
      }
 }
```

Productores y consumidores (*filtro)*

Red de Ordenación

- ***Filtro:*** proceso que recibe mensajes de uno o más canales de entrada y envía mensajes a uno o más canales de salida. La salida de un filtro es función de su estado inicial y de los valores recibidos.
- Esta función del filtro puede especificarse por un predicado que relacione los valores de los mensajes de salida con los de entrada.
- ***Problema:*** ordenar una lista de *N* números de modo ascendente. Podemos pensar en un filtro *Sort* con un canal de entrada (*N* números desordenados) y un canal de salida (*N* números ordenados).

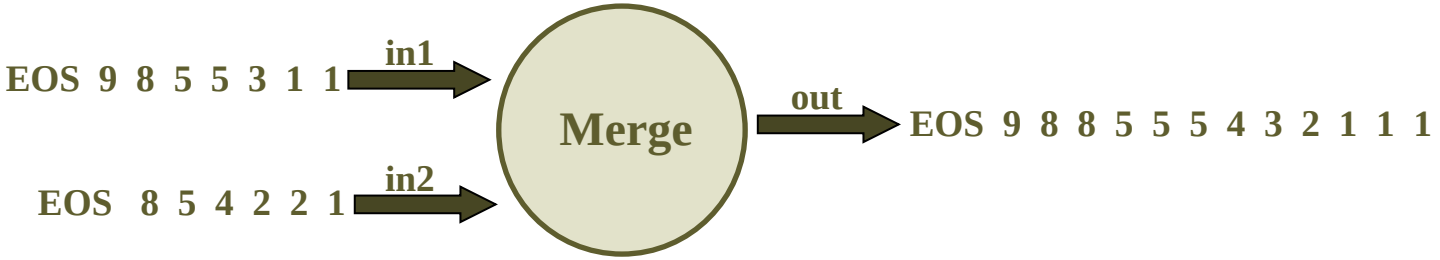
```
Process Sort
        {  receive todos los números del canal entrada;
           ordenar los números;
           send de los números ordenados por el canal OUTPUT;
         }
```

- ¿Cómo determina *Sort* que recibió todos los números?
  - conoce *N*.
  - envía *N* como el primer elemento a recibir por el canal *entrada*.
  - cierra la lista de *N* números con un valor especial o “centinela”.

Productores y consumidores (*filtro)*

Red de Ordenación

- Solución más eficiente que la "secuencial" ⇒ red de pequeños procesos que ejecutan en paralelo e interactúan para "armar" la salida ordenada ***(merge network).***
- ***Idea:*** mezclar repetidamente y en paralelo dos listas ordenadas de *N1* elementos cada una en una lista ordenada de *2*N1* elementos.
- Con ***PMA***, pensamos en *2* canales de entrada por cada canal de salida, y un carácter especial *EOS* cerrará cada lista parcial ordenada.
- La red es construida con filtros ***Merge***:
  - Cada *Merge* recibe valores de dos *streams* de entrada ordenados, ***in1*** e ***in2***, y produce un *stream* de salida ordenado, ***out***.
  - Los *streams* terminan en ***EOS***, y *Merge* agrega ***EOS*** al final.
  - ¿Cómo implemento *Merge*?. Comparar repetidamente los próximos dos valores recibidos desde ***in1*** e ***in2*** y enviar el menor a ***out***.



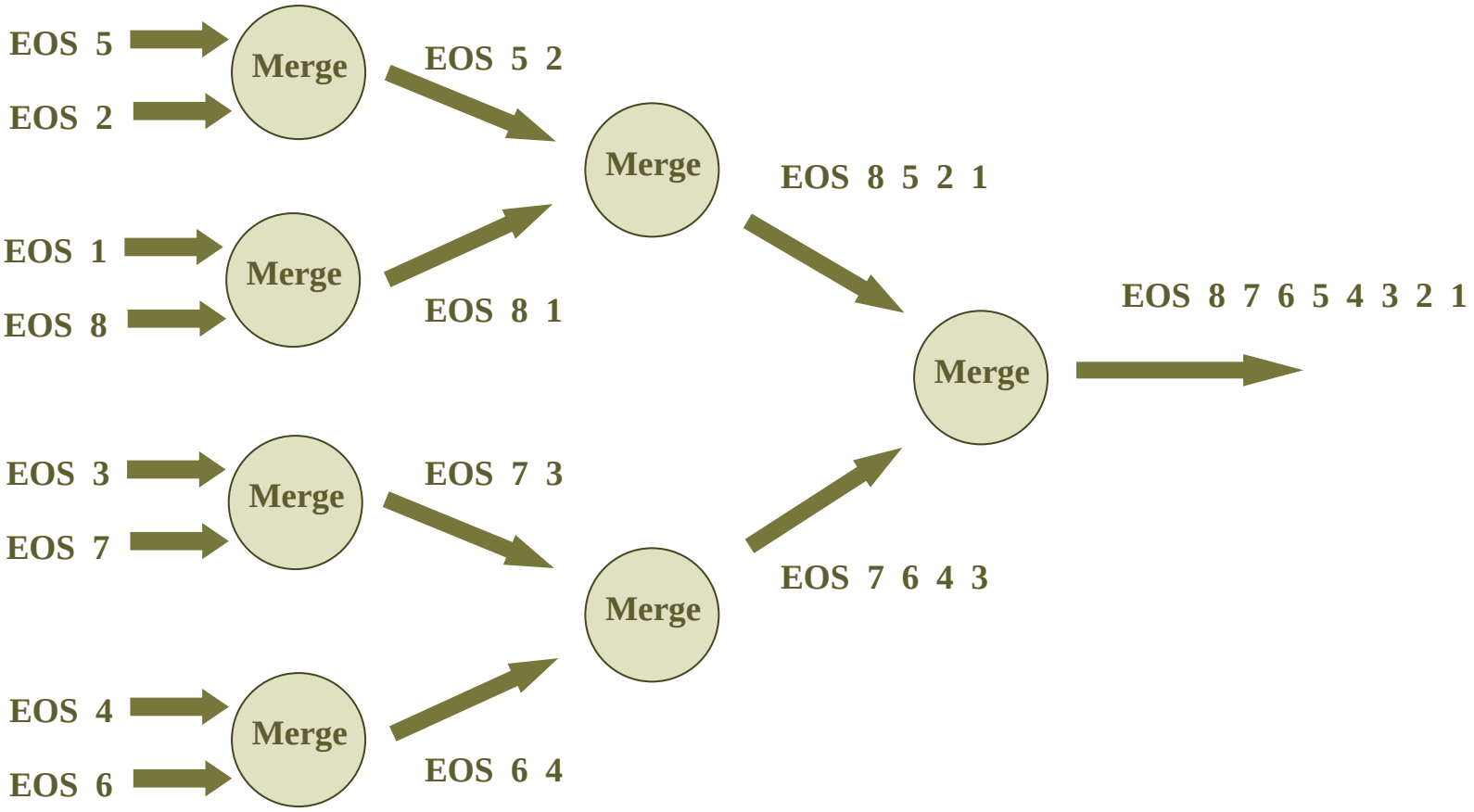
Productores y consumidores (*filtro)*

Red de Ordenación

```
chan  in1(int), in2(int), out(int);

Process Merge
{ int v1, v2;

  receive in1(v1);
  receive in2(v2);
  while (v1 ≠ EOS) and (v2 ≠ EOS)
    { if  (v1 ≤ v2) { send out(v1); receive in1(v1); }
      else  { send out(v2); receive in2(v2); }
    }

  if  (v1 == EOS) while (v2 ≠ EOS) {send out(v2); receive in2(v2);}
  else  while (v1 ≠ EOS) {send out(v1); receive in1(v1);}

  send out (EOS);
}
```

Productores y consumidores (*filtro)*

Red de Ordenación



Productores y consumidores (*filtro)*

Red de Ordenación

- *n*-1 procesos; el ancho de la red es $\log_2 n$.
- Canales de entrada y salida compartidos
- Puede programarse usando:
  - ***Static naming*** (arreglo global de canales, y cada instancia de *Merge* recibe desde 2 elementos del arreglo y envía a otro ⇒ embeber el árbol en un arreglo).
  - ***Dynamic naming*** (canales globales, parametrizar los procesos, y darle a cada proceso 3 canales al crearlo; todos los *Merge* son idénticos, pero se necesita un coordinador).
- Los filtros podemos conectarlos de distintas maneras. Solo se necesita que la salida de uno cumpla las suposiciones de entrada del otro ⇒ pueden reemplazarse si se mantienen los comportamientos de entrada y salida.

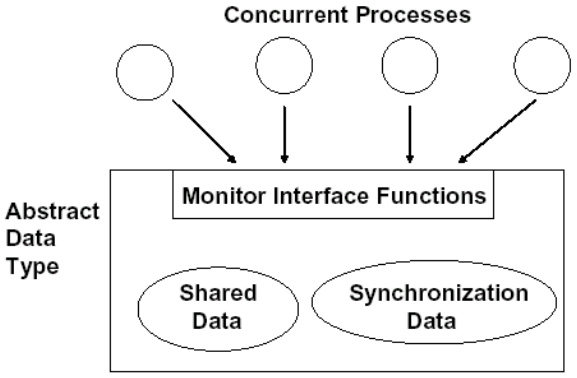
Clientes y Servidores.
Monitores Activos

- ***Servidor:*** proceso que maneja pedidos (*"requests"*) de otros procesos ***clientes***. ¿Cómo implementamos *C/S* con *PMA*?

- Dualidad entre monitores y PM: cada uno de ellos puede simular al otro.

Monitor ⇒ *manejador de recurso.* Encapsula variables permanentes que registran el estado, y provee un conjunto de procedures. Los simulamos, usando procesos servidores y PM, como procesos activos en lugar de como conjuntos pasivos de procedures.

```
monitor Mname
 {   declaración de  variables permanentes;
     código de inicialización;
     procedure op(formales) { cuerpo de op;}
 }
```



Clientes y Servidores.
Monitores Activos

- Un ***Servidor*** es un proceso que maneja pedidos (requerimientos) de otros procesos clientes. Veremos cómo implementar Cliente/Servidor con PMA.

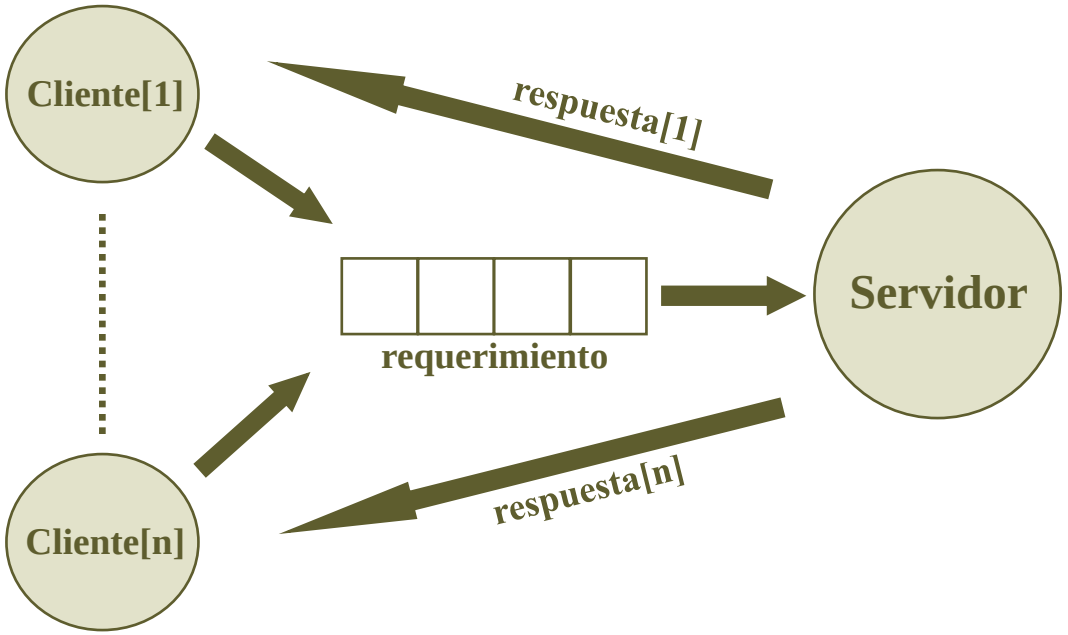
- Un proceso ***Cliente*** que envía un mensaje a un canal de requerimientos general, luego recibe el resultado desde un canal de respuesta propio (¿por qué?).

- En un sistema distribuido, lo natural es que el proceso ***Servidor*** resida en un procesador físico y M procesos ***Cliente*** residan en otros N procesadores (N<= M).

Clientes y Servidores.

Monitores Activos – 1 operación

- Para simular ***Mname***, usamos un proceso server ***Servidor***.
- Las variables permanentes serán variables locales de ***Servidor***.
- Llamado: un proceso ***cliente*** envía un mensaje a un canal de ***requerimiento***.
- Luego recibe el resultado por un canal de ***respuesta*** propio



Clientes y Servidores.

Monitores Activos – 1 operación

```
chan requerimiento (int idCliente, tipos de los valores de entrada );
chan respuesta[n] (tipos de los resultados );

Process Servidor
      {  int idCliente;
         declaración de variables permanentes;
         código de inicialización;
         while (true)
            { receive requerimiento (IdCliente, valores de entrada);
              cuerpo de la operación op;
              send respuesta[IdCliente] (resultados);
            }
      }

Process Cliente [i = 1 to n]
      {  send requerimiento (i, argumentos);
         receive respuesta[i] (resultados);
      }
```

Clientes y Servidores.

Monitores Activos – Múltiples operaciones

- Podemos generalizar esta solución de *C/S* con una única operación para considerar múltiples operaciones.
- El ***IF*** del *Servidor* será un ***CASE*** con las distintas clases de operaciones.
- El cuerpo de cada operación toma datos de un canal de entrada en ***args*** y los devuelve *al cliente adecuado* en resultados.

type clase_op = enum(op_1, ..., op_n);
type tipo_arg = union(arg_1 : $tipoAr_1$, ..., arg_n : $tipoAr_n$);
type tipo_result = union(res_1 : $tipoRe_1$, ..., res_n : $tipoRe_n$);

chan request(int idCliente, clase_op, tipo_arg);
chan respuesta[n](tipo_result);

Process Servidor .....
Process Cliente [i = 1 to n] .....

Clientes y Servidores.

Monitores Activos – Múltiples operaciones

```
Process Servidor
{  int IdCliente; clase_op oper; tipo_arg args;
   tipo_result resultados;
   código de inicialización;
   while ( true)
      {  receive request(IdCliente, oper, args);
         if ( oper == op1 ) { cuerpo de op1;}
            .....
         elsif ( oper == opn ) { cuerpo de opn;}
         send respuesta[IdCliente](resultados);
      }
}

Process Cliente [i = 1 to n]
{  tipo_arg mis_args;
   tipo_result  mis_resultados;
   send request(i, opk, mis_args);
   receive respuesta[i] (mis_resultados);
}
```

Clientes y Servidores.

Monitores Activos – Múltiples operaciones y variables condición

- Hasta ahora el monitor no requería variables condición ya que el *Servidor* no requería demorar la atención de un pedido de servicio. ***Caso general:*** monitor con múltiples operaciones y con sincronización por condición. Para los clientes, la situación es transparente ⇒ ***cambia el servidor.***

- Consideramos un caso específico de manejo de múltiples unidades de un recurso (ejemplos: *bloques de memoria, impresoras*).
  - Los clientes "adquieren" y devuelven unidades del recurso.
  - Las unidades libres se insertan en un "conjunto" sobre el que se harán las operaciones de INSERTAR y REMOVER.
  - El número de unidades disponibles es lo que "controla" nuestra variable de sincronización por condición.

```
Monitor Administrador_Recurso
{   int disponible = MAXUNIDADES;
    set unidades = valores iniciales;
    cond libre;

    procedure adquirir( int  *Id )
            {  if (disponible == 0)  wait(libre)
               else  disponible --;
               remove(unidades, id);
            }

    procedure liberar(int id )
            {   insert(unidades, id);
               if (empty(libre)) disponible ++
               else signal(libre);
            }
 }
```

Clientes y Servidores.

Monitores Activos – Múltiples operaciones y variables condición

- Caso en que el servidor tiene dos operaciones:
  - Si no hay unidades disponibles, el servidor no puede esperar hasta responder al pedido ⇒ debe salvarlo y diferir la respuesta.
  - Cuando una unidad es liberada, atiende un pedido salvado (si hay) enviando la unidad.

```
type clase_op = enum(adquirir, liberar);
chan request(int idCliente, claseOp oper, int idUnidad );
chan respuesta[n] (int id_unidad);

Process Administrador_Recurso
{   int disponible = MAXUNIDADES;
    set unidades = valor inicial disponible;
    queue pendientes;
    while (true)
     {  receive request (IdCliente, oper, id_unidad);
        if (oper == adquirir)
           {   if (disponible > 0)
                 {  disponible = disponible - 1;
                    remove (unidades, id_unidad);
                    send respuesta[IdCliente] (id_unidad);
                 }
               else  push (pendientes, IdCliente);
           }
        else
           { if empty (pendientes)
                 {  disponible= disponible + 1;
                    insert(unidades, id_unidad);
                 }
               else
                 {  pop (pendientes, IdCliente);
                    send respuesta[IdCliente](id_unidad);
                 }
           }
     }  //while
}    //process  Administrador_Recurso

Process Cliente[i = 1 to n]
{   int id_unidad;
    send request(i, adquirir, 0);
    receive respuesta[i](id_unidad);
     //Usa la unidad
    send request(i, liberar, id_unidad);
}
```

Clientes y Servidores.

Monitores Activos – Múltiples operaciones y variables condición

- El monitor y el Servidor muestran la dualidad entre monitores y PM: hay una correspondencia directa entre los mecanismos de ambos.
- La eficiencia de monitores o PM depende de la arquitectura física de soporte:
  - Con MC conviene la invocación a procedimientos y la operación sobre variables condición.
  - Con arquitecturas físicamente distribuidas tienden a ser más eficientes los mecanismos de PM.
- Dualidad entre Monitores y Pasaje de Mensajes

Programas con Monitores		***Programas basados en PM***
• Variables permanentes	↔	Variables locales del servidor
• Identificadores de procedures	↔	Canal *request* y tipos de operación
• Llamado a procedure	↔	*send request*( )*; receive respuesta*
• Entry del monitor	↔	*receive request*( )
• Retorno del procedure	↔	*send respuesta*( )
• Sentencia *wait*	↔	Salvar pedido pendiente
• Sentencia *signal*	↔	Recuperar/ procesar pedido pendiente
• Cuerpos de los procedure	↔	Sentencias del “case” de acuerdo a la clase de operación.

Clientes y Servidores.

Sentencia de Alternativa Múltiple

- Resolución del mismo problema con sentencias de alternativa múltiple. Ventajas y desventajas

```
chan pedido (int idCliente);
chan liberar (int idUnidad);
chan respuesta[n] (int idUnidad);

Process Administrador_Recurso
{   int disponible = MAXUNIDADES;
    set unidades = valor inicial disponible;
    int id_unidad, idCliente;
    while (true)
      {  if  ( (not empty(pedido) and (disponible >0) ) →
               receive pedido (idCliente);
               disponible = disponible - 1;
               remove (unidades, id_unidad);
               send respuesta[idCliente] (id_unidad);
          □ (not empty(liberar)) →
               receive liberar (id_unidad);
               disponible= disponible + 1;
               insert(unidades, id_unidad);
         } //if
       }  //while
   }    //process  Administrador_Recurso
```

```
Process Cliente[i = 1 to n]
 {  int id_unidad;

    send pedido (i);
    receive respuesta[i](id_unidad);
     //Usa la unidad
    send liberar (id_unidad);
 }
```

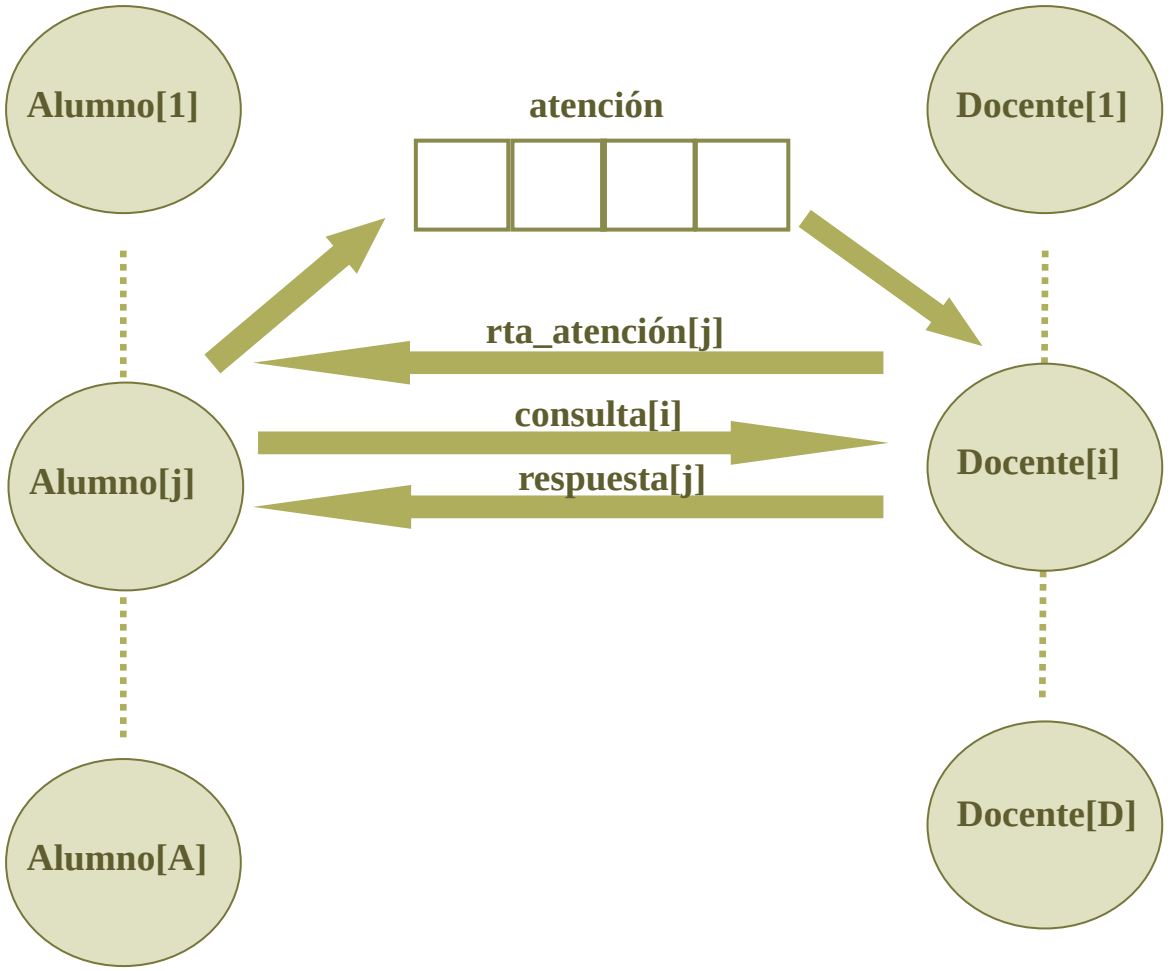
Clientes y Servidores.
Continuidad Conversacional

- Existen ***A*** alumnos que hacen consultas a ***D*** docentes.
- El alumno espera hasta que un docente lo atiende, y a partir de ahí le comienza a realizar las diferentes consultas hasta que no le queden dudas.
- Los alumnos son los procesos "*clientes*", y los docentes los procesos "*Servidores*". Los procesos servidores son idénticos, y cualquiera de ellos que esté libre puede atender un requerimiento de un alumno.

*Todos los alumnos pueden pedir atención por un **canal global** y recibirán respuesta de un docente dado por un **canal propio**. ¿Por qué?*

Clientes y Servidores.

Continuidad Conversacional



Clientes y Servidores.
Continuidad Conversacional

```
chan atención (int);
chan consulta[D] (string);
chan rta_atención[A](int);
chan respuesta[A] (string);

Process Alumno [a = 1 to A]
    {  int idDocente;
       string preg, res;
       send atención (a);
       receive rta_atención[a] (idDocente);
       while (tenga consultas para hacer)
          { send consulta[idDocente](preg);
            receive respuesta[a](res);
          }
       send consulta [idDocente] ('FIN');
     }
```

```
Process Docente [d = 1 to D]
{    string preg, res;
     int idAlumno;
     bool seguir = false;

     while (true)
       {  receive atención (idAlumno);
          send rta_atención[idAlumno](d);
          seguir = true;
          while (seguir)
             { receive consulta[d](preg);
               if (preg == 'FIN')  seguir = false
               else
                    { res = resolver la pregunta (preg)
                      send respuesta [idAlumno](res);
                    }
             }
       }
}
```

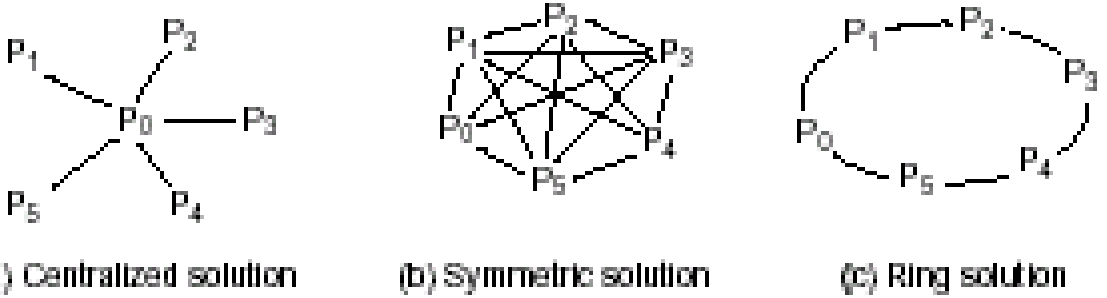
- Este ejemplo de interacción entre clientes y servidores se denomina ***continuidad conversacional*** (desde la solicitud de atención hasta la última consulta).
- ***atención*** es un canal compartido por el que cualquier *Docente* puede recibir. Si cada canal puede tener un solo receptor, se necesita otro proceso intermedio. ¿Para qué?.

Pares (peers) interactuantes.

Intercambio de Valores

Problema: cada proceso tiene un dato local *V* y los *N* procesos deben saber cuál es el menor y cuál el mayor de los valores.

Ejemplo donde los procesadores están conectados por tres modelos de arquitectura: ***centralizado***, ***simétrico*** y en ***anillo circular***.



(a) Centralized solution (b) Symmetric solution (c) Ring solution

Pares (peers) interactuantes.

Intercambio de Valores – Solución centralizada

- Cada proceso tiene un valor ***V*** local. Al final todos los procesos deben conocer el mínimo y máximo valor de todo el sistema.
- La arquitectura centralizada es apta para una solución en que todos envían su dato local ***V*** al procesador central, éste ordena los ***N*** datos y reenvía la información del mayor y menor a todos los procesos $\Rightarrow$ ***2(N-1)*** mensajes.

```
chan valores(int),   resultados[n-1] (int minimo, int maximo);
```

```
Process P[0]
{  int v; int nuevo, minimo = v, máximo = v;
   for [i=1 to n-1]
      {     receive valores (nuevo);
            if (nuevo < minimo)  minimo = nuevo;
            if (nuevo > maximo)  maximo = nuevo;
       }
   for [i=1 to n-1]
        send resultados [i-1] (minimo, maximo);
}
```

```
Process P[i=1 to n-1]
{  int v;  int minimo, máximo;
   send valores (v);
   receive resultados[i-1](minimo, maximo);
}
```

- ¿Se puede usar un único canal de resultados?

Pares (peers) interactuantes.

Intercambio de Valores – Solución simétrica

- En la arquitectura simétrica o "full conected" hay un canal entre cada par de procesos. Todos los procesos ejecutan el mismo algoritmo.
- Cada proceso trasmite su dato local ***V*** a los ***N-1*** restantes procesos. Luego recibe y procesa los ***N-1*** datos que le faltan, de modo que en paralelo toda la arquitectura está calculando el mínimo y el máximo y toda la arquitectura tiene acceso a los ***N*** datos.
- Ejemplo de solución *SPMD*: cada proceso ejecuta el mismo programa pero trabaja sobre datos distintos $\Rightarrow$***N(N-1)*** mensajes.
- Si disponemos de una primitiva de ***broadcast***, serán nuevamente ***N*** mensajes.

```
chan valores[n] (int);
Process P[i=0 to n-1]
{ int v=…, nuevo, minimo = v, máximo=v;
  for [k=0 to n-1  st  k <> i ]
      send valores[k] (v);
  for [k=0 to n-1  st  k <> i ]
    { receive valores[i] (nuevo);
      if (nuevo < minimo) minimo = nuevo;
      if (nuevo > maximo) maximo = nuevo;
    }
 }
```

- ¿Se puede usar un único canal?

Pares (peers) interactuantes.

Intercambio de Valores – Solución en anillo circular

- Un tercer modo de organizar la solución es tener un anillo donde ***P[i]*** recibe mensajes de ***P[i-1]*** y envía mensajes a ***P[i+1]***. ***P[n-1]*** tiene como sucesor a ***P[0]***.
- Esquema de 2 etapas. En la primera cada proceso recibe dos valores y los compara con su valor local, trasmitiendo un máximo local y un mínimo local a su sucesor. En la segunda etapa todos deben recibir la circulación del máximo y el mínimo global.
  - ***P[0]*** deberá ser algo diferente para "arrancar" el procesamiento.
  - Se requerirán (***2N)-1*** mensajes.
  - Notar que si bien el número de mensajes es lineal (igual que en la centralizada) los tiempos pueden ser muy diferentes. ¿Por qué?.

```
chan valores[n] (int minimo, int maximo);
Process P[0]
{ int v=…, minimo = v, máximo=v;
  send valores[1] (minimo, maximo);
  receive valores[0] (minimo, maximo);
  send valores[1] (minimo, maximo);
}

Process P[i=1 to n-1]
{ int v=…, minimo, máximo;
  receive valores[i] (minimo, maximo);
  if (v<minimo) minimo = v;
  if (v> maximo) maximo = v;
  send valores[(i+1) mod n] (minimo, maximo);
  receive valores[i] (minimo, maximo);
  if (i < n-1) send valores[i+1] (minimo, maximo);
 };
```

Pares (peers) interactuantes.
Comentarios sobre las soluciones

- ***Simétrica*** es la más corta y sencilla de programar, pero usa el mayor número de mensajes (si no hay broadcast). Pueden transmitirse en paralelo si la red soporta transmisiones concurrentes, pero el overhead de comunicación acota el speedup.
- ***Centralizada y anillo*** usan n° lineal de mensajes, pero tienen distintos patrones de comunicación que llevan a distinta performance:
  - En ***centralizada***, los mensajes al coordinador se envían casi al mismo tiempo ⇒ sólo el primer *receive* del coordinador demora mucho.
  - En ***anillo***, todos los procesos son *productores y consumidores*. El último tiene que esperar a que todos los otros (uno por vez) reciban un mensaje, hacer poco cómputo, y enviar su resultado.

    Los mensajes circulan 2 veces completas por el anillo ⇒ Solución inherentemente lineal y lenta para este problema.