



# Resumen Segundo Parcial

## Programación concurrente con memoria distribuida

Arquitecturas de memoria distribuida  $\Rightarrow$  procesadores + memoria 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 memoria compartida.

Los procesos SÓLO comparten canales (físicos o lógicos). Variantes para los canales:

- Mailbox, input port, link.
- Uni o bidireccionales.
- Sincrónicos o asincrónicos.

La exclusión mutua no requiere mecanismo especial. Los procesos interactúan comunicándose. Accedidos por envío y recepción.

Mecanismos para el procesamiento distribuido:

- Pasaje de Mensajes Asincrónicos (PMA).

- Pasaje de Mensajes Sincrónico (PMS).
- Llamado a Procedimientos Remotos (RPC).
- Rendezvous.

Cada mecanismo es más adecuado para determinados patrones.

## Pasaje de Mensajes Asincrónicos (PMA)

PMA ⇒ canales = colas de mensajes enviados y aún no recibidos.

Declaración de canales: *chan nombreCanal(tipoDato)*

- **chan** entrada (char);
- **chan** acceso\_disco (int, int, string, char);
- **chan** resultado[N] (int);

**Send** → un proceso agrega un mensaje al final de la cola de un canal ejecutando un send, que no bloquea al emisor: *send nombreCanal(datoMandado)*

**Receive** → un proceso recibe un mensaje desde un canal con receive, que demora al receptor hasta que en el canal haya al menos un mensaje, luego toma el primero y lo almacena en variables locales: *receive nombreCanal(variableDondeSeGuarda)*

Acceso a los contenidos de cada canal: atómico y respeta orden FIFO (First In, First Out).

*empty(nombreCanal)* → determina si la cola de un canal está vacía. La evaluación del empty podría ser true, y sin embargo existir un mensaje al momento de que el proceso reanuda la ejecución.

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

- Mailbox: cualquier proceso puede enviar o recibir por alguno de los canales declarados.
- Input port: Un solo receptor y muchos emisores.
- Link: único emisor y único receptor.

La eficiencia de monitores o de PM depende de la arquitectura física de soporte:

- Con memoria compartida conviene la invocación a procedimientos y la operación sobre variables condición.

- Con arquitecturas físicamente distribuidas tienden a ser más eficientes los mecanismos de 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 y receive
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

Los programas se componen SÓLO de procesos y canales (no existen las variables compartidas).

Los canales actúan como colas de mensajes enviados y no recibidos. Son de tipo mailbox (todos los procesos los pueden usar para enviar o recibir mensajes).

Los procesos interactúan entre ellos únicamente por medio del envío de mensajes (tanto para comunicación como para sincronización por condición).

No se requiere sincronización por exclusión mutua ya que no existen las variables compartidas.

El uso de cada canal es atómico, por lo que no se harán al mismo tiempo 2 operaciones (send y/o receive sobre el mismo canal).

La operación del envío no es bloqueante, deposita el mensaje al final del canal y continúa la ejecución.

La operación de la recepción es bloqueante, si el canal está vacío se demora hasta que haya al menos un mensaje en él, luego saca el primer mensaje del canal (más viejo).

La función empty consulta por mensajes pendientes, y retorna un booleano que indica si el canal está o no vacío.

El uso de sentencias de alternativa múltiple (if no determinístico) y alternativas iterativas múltiple (do no determinístico) puede generar busy waiting, que en PMA está permitido pero hay que tratar de evitar.

### Ejemplos:

- En una empresa de software hay N personas que prueban un nuevo producto para encontrar errores, cuando encuentran uno generan un reporte para que un empleado corrija el error y esperan la respuesta del mismo. El empleado toma los reportes de acuerdo al orden de llegada, los evalúan, hace las correcciones necesarias y le responde a la persona que hizo el reporte.

```

Chan Reportes(int, string);
Chan Respuestas[N](string);

Process Persona[id:0..N-1] {
    string reporte, string respuesta;
    while (true) {
        Reporte = generarReporte();
        send Reportes(id, Reporte);
        receive Respuestas[id](Respuesta);
    }
}

Process Empleado {
    string res;
    while (true) {
        receive Reportes(idPersona, reporte);
        res = generarRespuesta();
        send Respuestas[idPersona](res);
    }
}

```

- En una empresa de software hay N personas que prueban un nuevo producto para encontrar errores, cuando encuentran uno generan un reporte para que uno de los 3 empleados corrija el error (las personas no deben recibir ninguna respuesta). Los empleados toman los reportes de acuerdo al orden de llegada, los evalúan y hacen las correcciones necesarias, cuando no hay reportes para atender los empleados se dedican a leer durante 10 minutos.

```

Chan Reportes(string);
Chan Siguiente(int);
Chan Trabajo[3](string);

Process Persona[id: 0..N-1] {
    while (true) {
        string r = generarReporte();
        send Reportes(r);
    }
}

Process empleado[id:0..2] {
    string tarea;
    while (true) {

```

```

        send Siguiente(id);
        receive Trabajo[id](tarea);
        if (tarea == 'leer')
            delay(600);
        else
            corregir(tarea);
    }
}

Process coordinador {
    string reporte;
    int idEmpleado;
    while (true) {
        receive Siguiente(idEmpleado);
        if (!empty(Reportes))
            receive Reportes(reporte);
        else
            send reporte = 'leer';
            send Trabajo[idEmpleado](reporte);
    }
}

```

- Simular la atención en un locutorio con 10 cabinas telefónicas, el cual tiene un empleado que se encarga de atender a N clientes. Al llegar, cada cliente espera hasta que el empleado le indique a qué cabina ir, la usa y luego se dirige al empleado para pagarle. El empleado atiende a los clientes en el orden en que hacen los pedidos, pero siempre dando prioridad a los que terminaron de usar la cabina. A cada cliente se le entrega un ticket factura. Nota: maximizar la concurrencia; suponga que hay una función Cobrar() llamada por el empleado que simula que el empleado le cobra al cliente.

```

Chan llegada(int);
Chan cabina[N](int);
Chan pedido(boolean);
Chan tickets[N](Ticket);

Process cliente[id:0..N-1] {
    int cabinaUsar;
    Ticket miTicket;
    send llegada(id);
    send pedido(true);
    receive cabina[id](cabinaUsar);
    usarCabina(cabinaUsar);
    send salida(id, cabinaUsar);
    send pedido(true);
    receive tickets[id](miTicket);
}

Process empleado {
    Cola cabinas;
}

```

```

boolean hay;
int idPersona, cabinaLibre;
for (int i = 0; i < 10; i++)
    cabinas.push(i);

for (int i = 0; i < N * 2; i++) {
    receive pedido(hay);
    if (!empty(salida) || cabinas.isEmpty()) {
        receive salida(idPersona, cabinaLibre);
        cabinas.push(cabinaLibre);
        Ticket entregar = Cobrar();
        send tickets[idPersona](entregar);
    } else {
        receive llegada(idPersona);
        send cabina[idPersona](cabinas.pop());
    }
}
}

```

## Pasaje de Mensajes Sincrónicos (PMS)

Los canales son de tipo link: 1 emisor y 1 receptor.

La diferencia entre PMA y PMS es la transmisión del send: en PMS es bloqueante:

- El transmisor queda esperando que el mensaje sea recibido por el receptor.
- La cola de mensajes asociada con un send sobre un canal se reduce a 1 mensaje: menos memoria.
- El grado de concurrencia se reduce respecto a la sincronización por PMA ya que los emisores se bloquean.

Hay mayor concurrencia en PMA. Para lograr el mismo efecto en PMS se debe interponer un proceso buffer. Otra desventaja del PMS es la mayor probabilidad de deadlock. El programador debe ser cuidadoso de que todas las sentencias de send y receive hagan matching.

Formas generales de las sentencias de comunicación:

**Destino ! port(dato1, dato2,...,datoN)**

**Fuente ? port(dato1, dato2,...,datoN)**

Destino y Fuente nombran un proceso, o un elemento de un arreglo de procesos.

Fuente puede nombrar cualquier elemento de un arreglo: Fuente[\*]

port son etiquetas que se usan para distinguir entre distintas clases de mensajes que un proceso podría recibir.

Dos procesos se comunican cuando ejecutan sentencias de comunicación que hagan matching:

**A ! canaluno(dato);**

**B ? canaluno(resultado);**

## Comunicación guardada

Las sentencias ? y ! tienen limitaciones ya que son bloqueantes. Hay problemas si un proceso quiere comunicarse con otros sin conocer el orden en que los otros quieren hacerlo con él. Las operaciones de comunicación pueden ser guardadas, es decir hacer un Await hasta que una condición sea verdadera. Para esto, se pueden utilizar *do* e *if* de guarda.

Las sentencias de comunicación guardada soportan comunicación no determinística:

*B; C → S;*

- B puede omitirse y se asume true.
- B y C forman la guarda.
- La guarda tiene éxito si B es true y ejecutar C no causa demora.
- La guarda falla si B es falsa.
- La guarda se bloquea si B es true pero C no puede ejecutarse inmediatamente.

*if*

*B1; comunicación 1 → S 1;*

*B2; comunicación 2 → S 2;*

...

*Bi; comunicación i → S i;*

*fi*

Ejecución:

1. Se evalúan las guardas:

- Si todas las guardas fallan, el if termina sin efecto.
- Si al menos una guarda tiene éxito, se elige una de ellas de forma no determinística.

- Si algunas guardas se bloquean, se espera hasta que alguna de ellas tiene éxito.
2. Luego de elegir una guarda exitosa, se ejecuta la sentencia de comunicación de la guarda elegida.
  3. Se ejecuta la sentencia S i.

La ejecución del do es similar (se repite hasta que todas las guardas fallen).

Los programas se componen sólo de procesos (no existen variables compartidas).

Los canales son de tipo link (un único emisor y un único receptor) y sincrónicos. Son estructuras implícitas y no se deben declarar.

Los procesos interactúan entre ellos únicamente por medio del envío de mensajes (tanto para comunicación como para sincronización por condición).

No se requiere sincronización por exclusión mutua ya que no existen las variables compartidas.

No existen las estructuras de canales, la comunicación se realiza nombrando al proceso con el cual se realiza la comunicación.

- Envío (!): la operación es bloqueante y sincrónica, se demora hasta que la recepción haya terminado.
  - *destino!port(mensaje);*
  - *destino[i]!port(mensaje);*
- Recepción?: la operación es bloqueante y sincrónica.
  - *origen?port(variable);*
  - *origen[i]?port(variable);*
  - *origen[\*]?port(variable);*

El uso de comunicación guardada (if y do) permiten seleccionar de forma no determinística entre varias alternativas de comunicación en base a las condiciones del proceso y los mensajes que están listos para ser recibidos.

*B; C → S;*

- **B:** condición booleana que puede no estar (en ese caso se considera true), e indica si el proceso está o no en condiciones de procesar el mensaje recibido en C.
- **C:** sentencia de comunicación (sólo recepción) que seguro debe estar.

- **S:** conjunto de sentencias que se ejecutarán en caso de ser elegida la guarda.

Evaluación de una guarda:

- **Éxito:** la condición booleana es verdadera (o no la tiene) y la comunicación se puede realizar sin producir demora (el emisor ya está esperando para hacer la comunicación).
- **Fallo:** la condición booleana es falsa, sin importar lo que ocurra con la sentencia de comunicación.
- **Bloqueo:** la condición booleana es verdadera (o no la tiene) pero la comunicación no se puede realizar sin producir demora (el emisor aún no llegó a la sentencia de envío).

En el if guardado se evalúan todas las guardas y en base a eso:

Si una o más son exitosas se selecciona una de ellas en forma no determinística, se ejecuta la sentencia de recepción que forma parte de la guarda, y posteriormente el conjunto de sentencias asociadas a la guarda.

Si todas las guardas fallan no se selecciona ninguna y se sale del if sin realizar ninguna acción.

Si no hay ninguna guarda exitosa pero hay una o más con estado bloqueo entonces el proceso se demora en el if hasta que haya una guarda exitosa. En ese momento se ejecuta igual que en el primer caso.

El do guardado funciona de la misma manera que el if, con la única diferencia que en lugar de hacerlo una vez repite el mecanismo hasta que todas las guardas fallan.

### Ejemplos:

- En una empresa de software hay un empleado Testeo que prueba un nuevo producto para encontrar errores, cuando encuentra uno genera un reporte para que otro empleado Mantenimiento corrija el error (no requiere una respuesta para seguir trabajando) y continúa trabajando. El empleado Mantenimiento toma los reportes para evaluarlos y hacer las correcciones necesarias.

```

Process testeo {
    while (true) {
        String reporte = encontrarError();
        administrador ! reportes(reporte);
    }
}

Process mantenimiento {
    while (true) {

```

```

        administrador ! pedido();
        administrador ? corregir(reporte);
        corregirError(reporte);
    }
}

Process administrador {
    Cola r;
    String encontrado;
    do
        administrador ? reportes(encontrado) => r.push(encontrado);
        (!r.isEmpty()); administrador ? pedido() => mantenimiento ! corregir(r.pop());
    od
}

```

- En un estadio de fútbol hay una máquina expendedora de gaseosas que debe ser usada por E Espectadores de acuerdo al orden de llegada. Cuando el espectador accede a la máquina en su turno usa la máquina y luego se retira para dejar al siguiente. Nota: cada Espectador una sólo una vez la máquina.

```

Process espectador[id:0..E-1] {
    administrador ! llegada(id);
    maquina ? usar();
    usarMaquina();
    maquina ! salir();
}

Process maquina {
    for (int i = 0; i < E; i++) {
        administrador ! pedido();
        administrador ? proximo(idProximo);
        espectador[idProximo] ! usar();
        espectador[idProximo] ? salir();
    }
}

Process administrador {
    int idPersona;
    Cola personas;
    do
        espectador[*] ? llegada(idPersona) => personas.push(idPersona);
        !personas.isEmpty(); maquina ? pedido() => maquina ! proximo(personas.pop());
    od
}

```

- En un laboratorio de genética veterinaria hay 3 empleados. El primero de ellos continuamente prepara las muestras de ADN; cada vez que termina, se la envía al segundo empleado y vuelve a su trabajo. El segundo empleado toma cada muestra de ADN preparada, arma el set de análisis que se deben realizar con

ella y espera el resultado para archivarlo. Por último, el tercer empleado se encarga de realizar el análisis y devolverle el resultado al segundo empleado.

```
Process empleado1 {
    while (true) {
        String m = preparar();
        administrador ! muestra(m);
    }
}

Process empleado2 {
    String muestraADN, archivar;
    while (true) {
        administrador ! pedido();
        administrador ? resultado(muestraADN);
        Set s = armarSet(muestraADN);
        empleado3 ! trabajar(s);
        empleado3 ? result(archivar);
    }
}

Process empleado3 {
    Set setAnalisis;
    while (true) {
        empleado2 ? trabajar(setAnalisis);
        String res = realizarAnalisis(setAnalisis);
        empleado 2 ! result(res);
    }
}

Process administrador {
    Cola cola;
    String muestraP;
    do
        empleado1 ? muestra(muestraP) => cola.push(muestraP);
        !cola.isEmpty(); empleado2 ? pedido => empleado2 ! resultado(cola.pop());
    od
}
```

## Rendezvous

Técnica de comunicación y sincronización entre procesos que suponen un canal bidireccional.

Mensajes sincrónicos: demoran al llamador hasta que la operación llamada se termine de ejecutar y se devuelvan los resultados. Un rendezvous es servido por una sentencia de entrada que espera una invocación, la procesa y devuelve los resultados.

Combina comunicación y sincronización:

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

Las operaciones se atienden una por vez más que concurrentemente.

Una sentencia de entrada demora al proceso hasta que haya al menos un llamado pendiente, luego elige el llamado pendiente más viejo, copia los argumentos en los parámetros formales, ejecuta las sentencias y finalmente retorna los parámetros de resultado al llamador. Luego, ambos procesos pueden continuar.

## ADA

Tasks (tareas) que se pueden ejecutar independientemente y contienen 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 operación accept.

Se puede declarar un type task y luego crear instancias de procesos identificado con dicho tipo.

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

TASK BODY nombre IS
    declaraciones locales
begin
    sentencias
end nombre;
```

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

Entry:

- Parámetros IN, OUT y IN OUT.
- Entry call: La ejecución demora al llamador hasta que la operación terminó.

- Entry call condicional: Si la entry call no es atendida inmediatamente, se ejecutan otras sentencias.
- Entry call temporal: Si la entry call no es atendida en el tiempo determinado, se ejecutan otras sentencias.

### Accept:

La tarea que declara un entry atiende los llamados a esos entry con la sentencia accept.

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. Soporta la comunicación guardada.

```
select
  when B 1 => accept EntryCall 1; sentencias 1;
or
  when B 2 => accept Entry Call 2; sentencias 2;
...
or
  when B N => accept Entry Call N; sentencias N;
end select;
```

Cada línea se llama alternativa. Las cláusulas when son opcionales.

Pueden contener una alternativa else, o or delay.

Una aplicación en ADA es un único programa con un cuerpo principal, y para que trabaje concurrente se desarrollan TASKS dentro de ese programa que pueden ejecutar independientemente y contienen formas de sincronizarse y comunicarse.

Se pueden implementar directamente TASK, que es una única instancia de esa tarea, o se puede declarar un TASK TYPE para crear varias instancias iguales de ese tipo (variables, arreglos).

Cada TASK / TASK TYPE tienen una especificación donde se declaran las operaciones exportadas (llamadas ENTRY) y un BODY donde se implementa la tarea.

### Estructura del programa:

```
Procedure nombre is
  Especificacion de un TASK
  Task nombreT1 is
    Declaracion de los ENTRY de la tarea
```

```

end nombreT1;

Especificacion de un tipo TASK
Task Type nombreTipoT is
    Declaracion de los ENTRY del tipo tarea
end nombreTipoT;
Declaracion de variables de tipo nombreTipoT

Cuerpo de las tareas
Task body nombreT1 is
    implementacion de la tarea
end nombreT1;

Task body nombreTipoT is
    implementacion del tipo tarea
end nombreTipoT;
begin
    cuerpo principal del programa
end nombre;

```

Las tareas pueden exportar o no operaciones, si lo hacen deben definir los entrys correspondientes en su especificacion. Los entrys pueden tener o no parámetros, si los tiene se debe declarar el nombre, el tipo de dato y el tipo de parámetro (IN, OUT o IN OUT).

El cuerpo de las tareas es donde se implementa la tarea. En esa parte no hay diferencia entre las Task y los Task Type.

Si se hubiese definido un arreglo de un tipo tarea, se generarán varias tareas idénticas de ese tipo. La única manera de identificar una de otra es por su posición en el arreglo, pero la tarea por si misma no conoce ese dato. Si lo necesita saber, otra tarea o el cuerpo principal del programa se lo deberá comunicar para que guarde ese identificador en una variable.

El rendezvous es el principal mecanismo de sincronización de Ada. No se pueden utilizar variables compartidas.

La comunicación entre 2 tareas se realiza por medio de un Entry Call por parte del proceso que realiza el pedido y un Accept por parte del que resuelve ese pedido. Ambas tareas deben esperar a que el Accept termine su ejecución para poder continuar (se mantienen sincronizados durante el rendezvous).

La comunicación es bidireccional, y es fundamental que esta característica se aproveche.

3 opciones de Entry Call que se diferencian por el tiempo de demora:

- Entry Call Simple: la ejecución demora al llamador hasta que la otra tarea termine de ejecutar el accept correspondiente.

*Tarea2.nombreEntry1;*

*Tarea3.nombreEntry2(5, x, z);*

*ArregloTareas(posicion).nombreEntry3(w, 44);*

- Entry Call Condicional: no espera a que le acepten el pedido, si la otra tarea no está lista para realizar el accept a su pedido inmediatamente, entonces lo cancela y realiza otra cosa.

*select*

*tarea3.nombreEntry(variable);*

*sentencias S1 a realizar sólo si se completó el entry call*

*else*

*sentencias S2 a realizar sólo si se canceló el entry call*

*end select;*

- Entry Call Temporal: Espera a lo sumo un tiempo a que la otra tarea realice el accept a su pedido, pasado el tiempo cancela el entry call y realiza otra cosa.

*select*

*tarea4.nombreEntry2();*

*sentencias S1 a realizar sólo si se completó el entry call*

*or delay 600*

*sentencias S2 a realizar sólo si no se completó el entry call*

*end select;*

Por cada entry declarado en la especificación de una tarea, en el cuerpo se debe hacer al menos un accept para dicho entry.

Por cada entry existe en la tarea una cola implícita de entry calls pendientes.

El accept puede tener un cuerpo, en cuyo caso hasta que no se termine de ejecutar el mismo no se pierde la sincronización con el entry call. O puede no tenerlo, en cuyo caso inmediatamente termina la sincronización con el entry call.

El accept demora la tarea hasta que haya al menos un entry call en la cola implícita asociada, saca al primero de ellos, y dependiendo del tipo de accept hace:

Si no tiene cuerpo: termina inmediatamente el accept y ambas tareas continúan con su ejecución.

Si tiene cuerpo: copia los parámetros reales del llamado en los parámetros formales (si el entry call tiene parámetros), ejecuta las sentencias que están en el cuerpo, cuando termina los parámetros formales de salida son copiados a los parámetros reales (si tiene parámetros de tipo OUT o IN OUT). Luego ambas tareas continúan con su ejecución.

ADA brinda la posibilidad de implementar wait selectivos por medio de comunicación guardada. Select para los Accept. Permite tener varias alternativas de Accept que pueden o no tener asociada una condición booleana (no se pueden utilizar los parámetros del entry como parte de la condición) utilizando la cláusula when.

*select*

*accept nombreEntry1;*

*sentencias S1 a realizar sólo si se completó el accept nombreEntry1;*

*or*

*when (condición) ⇒*

*accept nombreEntry2 IS*

*sentencias S2 a realizar sólo si se completó el accept nombreEntry2;*

*end nombreEntry2;*

*sentencias S3 a realizar sólo si se completó el accept nombreEntry3;*

*or delay tiempo*

*sentencias S4 a realizar sólo si pasó tiempo sin poder aceptar ninguna alternativa*

*end select;*

Cada entry tiene asociado un atributo que puede ser consultado sólo por la tarea a la que pertenece el entry que indica la cantidad de entry call pendientes:

*NombreEntry'count*

Este atributo puede ser usado en los when, por ejemplo para dar prioridad a un entry sobre otro.

### Ejemplos:

- Se debe modelar la atención en un banco por medio de un único empleado. Los clientes llegan y son atendidos de acuerdo al orden de llegada.

```

Procedure Banco is

    Task empleado is
        entry llegada(datos: IN string, resultado: OUT string);
    end empleado;

    Task type cliente
        arrayClientes: array(1..N) of cliente

    Task body cliente is
        datos, resultado: string;
    begin
        empleado.llegada(datos, resultado);
    end cliente;

    Task body empleado is
        datos, resultado: string;
    begin
        for:=1..N loop
            accept llegada(datos: IN string, resultado: OUT string) do
                resultado:= resolverPedido();
            end llegada;
        end loop;
    end empleado;
begin
end banco;

```

El cliente no debe hacer nada entre que hace el pedido y recibe la respuesta. Por el otro lado, el emplead cuando acepta un pedido inmediatamente lo resuelve y le envía el resultado al cliente. Por lo tanto la resolución del pedido se debe hacer en el cuerpo del accept. Se asegura la atención en orden de llegada ya que los entry call se almacenan en una cola implícita.

- Se dispone de un sistema compuesto por 1 central y 2 procesos periféricos, que se comunican continuamente. Se requiere modelar su funcionamiento considerando las siguientes condiciones: La central siempre comienza su ejecución tomando una señal del proceso 1; luego toma aleatoriamente señales de cualquiera de los dos indefinidamente. Los procesos periféricos envían señales continuamente a la central. La señal del proceso 1 será considerada vieja (se desecha) si en 2 minutos no fue recibida. Si la señal del proceso 2 no puede ser recibida inmediatamente, entonces espera 1 minuto y vuelve a mandarla (no se desecha).

```

Procedure sistema is

    Task central is
        entry perif1(s: IN integer);
        entry perif2(s: IN integer);
    end central;

    Task periferico1;

    Task periferico2;

    Task body central is

        begin
            accept perif1();
            loop
                select
                    accept perif1(s: IN integer);
                    or
                    accept perif2(s: IN integer);
                end select;
            end loop;
        end central;

    Task body periferico1 is
        senial:integer;
    begin
        loop
            senial:= generarSenial();
            select
                central.perif1(senial);
            or delay 120
                null;
        end loop
    end periferico1;

    Task body periferico2 is
        senial: integer;
        nueva: boolean;
    begin
        nueva:= true;
        loop
            if (nueva) then
                senial:=generarSenial();
                nueva:= false;
            end if;
            select
                central.perif2();
                nueva:=true;
            else
                delay(60);
            end loop;
        end periferico2;

        begin

```

```
    null;  
end sistema;
```

- En un sistema para acreditar carreras universitarias, hay UN Servidor que atiende pedidos de U Usuarios de a uno a la vez y de acuerdo con el orden en que se hacen los pedidos. Cada usuario trabaja en el documento a presentar, y luego lo envía al servidor; espera la respuesta de este que le indica si está todo bien o hay algún error. Mientras haya algún error, vuelve a trabajar con el documento y a enviarlo al servidor. Cuando el servidor le responde que está todo bien, el usuario se retira. Cuando un usuario envía un pedido espera a lo sumo 2 minutos a que sea recibido por el servidor, pasado ese tiempo espera un minuto y vuelve a intentarlo (usando el mismo documento).

```
Procedure universidad is  
  
Task type usuario;  
arrUsuarios: array(1..N) of usuario;  
  
Task servidor is  
    entry documento(d: IN string; aprobado: OUT boolean);  
end servidor;  
  
Task body usuario is  
    d: string;  
    aprobado, cambiar, seguir: boolean;  
begin  
    seguir:= true;  
    cambiar:= true;  
    while (seguir) loop  
        if (cambiar) then  
            d:=trabajar();  
            cambiar:=false;  
        end if;  
        select  
            servidor.documento(d, aprobado);  
            if (aprobado) then  
                seguir:=false  
            else  
                cambiar:=true;  
            end if;  
        or delay 120  
            delay(60);  
        end loop;  
    end usuario;  
  
Task body servidor is  
    aprobado: boolean;  
    d: string;  
begin  
    for:=1..U loop
```

```
accept.documento(d: IN string; aprobado: OUT boolean) do
    aprobado:= corregir();
end documento;
end loop;
end servidor;

begin
    null;
end universidad;
```