Apunte Final
Programación Concurrente

1) Escriba definición de Concurrencia. Diferencia concurrencia, paralelismo y procesamiento distribuido.

La concurrencia es la capacidad de ejecutar múltiples actividades en paralelo o en forma simultánea.
El paralelismo es la ejecución concurrente sobre diferentes componentes físicos (procesadores).
El paralelismo es un concepto asociado con la existencia de múltiples procesadores ejecutando un algoritmo en forma coordinada y cooperante. Al mismo tiempo se requiere que el algoritmo admita una descomposición en múltiples procesos ejecutables en diferentes procesadores (concurrencia).
El procesamiento distribuido es un conjunto de elementos (heterogéneos) de procesamiento que se interconectan por una red de comunicaciones y cooperan entre ellos para realizar sus tareas asignadas.
Programa concurrente: especifica dos o más procesos que cooperan para realizar una tarea. Cada proceso es un programa secuencial que ejecuta una secuencia de sentencias.
Un programa paralelo es un programa concurrente que típicamente es ejecutado en un multiprocesador. Un programa distribuido es un programa concurrente (o paralelo) en el cual los procesos se comunican por pasaje de mensajes.

2) Desventajas de la programación concurrente

En Programación Concurrente los procesos no son completamente independientes y comparten recursos. La necesidad de utilizar mecanismos de exclusión mutua y sincronización agrega complejidad a los programas → ***menor confiabilidad.***
Los procesos iniciados dentro de un programa concurrente pueden NO estar "vivos". Esta pérdida de la propiedad de **liveness** puede indicar deadlocks o una mala distribución de recursos.
Hay un **no determinismo** implícito en el interleaving de los procesos concurrentes. Esto significa que dos ejecuciones del mismo programa no necesariamente son idénticas → ***dificultad para la interpretación y debug.***
La comunicación y sincronización produce un overhead de tiempo, inútil para el procesamiento. Esto en muchos casos desvirtúa la mejora de perfomance buscada
La mayor complejidad en la especificación de los procesos concurrentes significa que los lenguajes de programación tienen requerimientos adicionales. → ***mayor complejidad en los compiladores y sistemas operativos asociados***
Aumenta el tiempo de desarrollo y puesta a punto respecto de los programas secuenciales. También puede aumentar el costo de los errores → ***mayor costo de los ambientes y herramientas de Ingeniería de Software de sistemas concurrentes.***
La ***paralelización*** de algoritmos secuenciales NO es un proceso directo, que resulte fácil de automatizar.
Para obtener una real mejora de perfomance, se requiere ***adaptar el software concurrente al hardware paralelo.***

3) Cuáles son las tres grandes clases de aplicaciones concurrentes que podemos encontrar?

El primer tipo de aplicaciones se corresponde cuando ejecutamos N procesos independientes sobre M procesadores, con N > M. Un sistema de software de "multithreading" maneja simultáneamente tareas independientes, asignando (por ejemplo por tiempos) los procesadores de que dispone.
Ejemplos típicos:

- ✓ Sistemas de ventanas en PCs o WS.
- ✓ Time sharing en sistemas operativos multiprocesador.
- ✓ Sistemas de tiempo real en plantas industriales

El segundo tipo de aplicaciones es el cómputo distribuido: una red de comunicaciones vincula procesadores diferentes sobre los que se ejecutan procesos que se comunican esencialmente por mensajes. Cada componente del sistema distribuido puede hacer a su vez multithreading.
Ejemplos típicos:

- ✓ Servidores de archivos (recursos) en una red.
- ✓ Sistemas de Bases de datos distribuidas (bancos, reservas de vuelos).
- ✓ Servidores WEB distribuidos.
- ✓ Arquitecturas cliente-servidor.

El procesamiento paralelo es el tercer tipo de aplicaciones. Se trata de resolver un problema en el menor tiempo posible, utilizando una arquitectura multiprocesador en la que se pueda distribuir la tarea global en tareas (independientes? interdependientes?) que puedan ejecutarse en diferentes procesadores.
Paralelismo de datos y paralelismo de procesos.
Ejemplos típicos:

- ✓ Cálculo científico. Modelos de sistemas (meteorología, movimiento planetario).
- ✓ Gráficos, procesamiento de imágenes, realidad virtual, procesamiento de video.
- ✓ Problemas combinatorios y de optimización lineal y no lineal. Modelos econométricos.

4) Diferencie procesamiento secuencial, concurrente y paralelo

Un thread de control, un solo flujo de control → programación secuencial, monoprocesador.
Múltiples threads de control → procesos → concurrencia.
Ejecución de procesos o tareas en una arquitectura multiprocesador → Paralelismo

5) a) Defina Sincronización. Explique los mecanismos de sincronización que conozca.

Sincronización: posesión de información acerca de otro proceso para coordinar actividades.
El estado de un programa concurrente en cualquier punto de tiempo consta de los valores de las variables de programa. Un programa concurrente comienza la ejecución en algún estado inicial. Cada proceso en el programa ejecuta a una velocidad desconocida. A medida que ejecuta, un proceso transforma el estado ejecutando sentencias. Cada sentencia consiste en una secuencia de una o más acciones atómicas que hacen transformaciones de estado indivisibles. La ejecución de un programa concurrente genera una secuencia de acciones atómicas que es algún interleaving de las secuencias de acciones atómicas de cada proceso componente. El trace de una ejecución particular de un programa concurrente puede verse como una *historia.*
El rol de la sincronización es restringir las posibles historias de un programa concurrente a aquellas que son deseables.

Formas de sincronización

- ***Exclusión mutua:*** consiste en asegurar que las secciones críticas de sentencias que acceden a objetos compartidos no se ejecutan al mismo tiempo. La exclusión mutua concierne a combinar las acciones atómicas fine-grained que son implementadas directamente por hardware en secciones críticas que deben ser atómicas para que su ejecución no sea intercalada con otras secciones críticas que referencian las mismas variables.

- ***Sincronización por condición:*** asegura que un proceso se demora si es necesario hasta que sea verdadera una condición dada. La sincronización por condición concierne a la demora de un proceso hasta que el estado conduzca a una ejecución posterior.

Mecanismos de comunicación y sincronización entre procesos

- ***Memoria compartida:*** Los procesos intercambian mensajes sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella. Lógicamente los procesos no pueden operar simultáneamente sobre la memoria compartida, lo que obligará a BLOQUEAR y LIBERAR el acceso a la memoria. La solución más elemental será una variable de control tipo "semáforo" que habilite o no el acceso de un proceso a la memoria compartida.

- ***Pasaje de Mensajes:*** Es necesario establecer un "canal" (lógico o físico) para transmitir información entre procesos. También el lenguaje debe proveer un protocolo adecuado. Para que la comunicación sea efectiva los procesos deben "saber" cuando tienen mensajes para leer y cuando deben trasmitir mensajes.

b) En un programa concurrente pueden estar presentes más de un mecanismo de sincronización?

Si. Por ejemplo, un programa que tiene dos procesos, uno productor y otro consumidor, que comparten un sector de memoria para el pasaje de los datos. Se usaría exclusión mutua para que no accedan a la memoria compartida al mismo tiempo, y sincronización por condición para que un mensaje no sea recibido antes que enviado, y para que un mensaje no sobrescriba a uno que todavía no fue recibido.

6) Describa los siguientes paradigmas de resolución de programas concurrentes: paralelismo iterativo, paralelismo recursivo, productores y consumidores, clientes y servidores y peers.

En el **Paralelismo iterativo** un programa tiene un conjunto de procesos (posiblemente idénticos) cada uno de los cuáles tiene uno o más loops. Es decir cada proceso es un programa iterativo.
La idea es que si estos procesos cooperan para resolver un único problema (ejemplo un sistema de ecuaciones) pueden trabajar independientemente y sincronizar por memoria compartida o envío de mensajes. Ejemplo clásico es la multiplicación de matrices.

En el **Paralelismo recursivo** el problema general (programa) puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos.
Ejemplos clásicos son por ejemplo el sorting by merging o el cálculo de raíces en funciones continuas.

El esquema **productor-consumidor** muestra procesos que se comunican. Es habitual que estos procesos se organicen en pipes a través de los cuáles fluye la información. Cada proceso en el pipe es un filtro que consume la salida de su proceso predecesor y produce una salida para el proceso siguiente. Distintos niveles de SO.

Cliente-servidor es el esquema dominante en las aplicaciones de procesamiento distribuido. Los servidores son procesos que esperan pedidos de servicios de múltiples clientes. Naturalmente unos y otros pueden ejecutarse en procesadores diferentes. Los mecanismos de invocación son variados (rendezvous y RPC por ejemplo). El soporte distribuido puede ser muy simple (LAN) o extendido a toda la WEB. Ejemplo FS.

En los **esquemas de pares** que interactúan los procesos (que forman parte de un programa distribuido) resuelven partes del problema (normalmente mediante código idéntico) e intercambian mensajes para avanzar en la tarea. El esquema permite mayor grado de asincronismo que C-S. Ejemplo multiplicación de matrices distribuida.

7) Analice conceptualmente la resolución de problemas con memoria compartida y memoria distribuida. Compare en términos de facilidad de programación.

En ***memoria compartida*** los procesos intercambian mensajes sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella. Lógicamente los procesos no pueden operar simultáneamente sobre la memoria compartida, lo que obliga a bloquear y liberar el acceso a la memoria. La solución más elemental será una variable de control tipo semáforo que habilite o no el acceso de un proceso a la memoria compartida. Arquitectura monoprocesador o multiprocesador fuertemente acoplada.

En ***memoria distribuida*** los procesos también pueden comunicarse a través de mensajes que llevan datos. Para esto es necesario establecer un canal lógico o físico para transmitir información entre procesos. El pasaje de mensaje puede ser sincrónico o asincrónico y es independiente de la arquitectura.

Por último podemos decir que usar memoria compartida es siempre menos caro que memoria distribuida porque toma menos tiempo escribir en variables compartidas que alocar un buffer de mensajes, llevarlo y pasarlo a otro proceso, además los procesos deben saber cuando tienen mensajes para leer y cuando deben transmitirlos y el lenguaje debe proveer un protocolo adecuado para el manejo de canales así como para el envío y la recepción de mensajes.

8) Analice conceptualmente los modelos de mensajes sincrónicos y asincrónicos. Compárelos en términos de concurrencia y facilidad de programación

Con **pasaje de mensajes asincrónico** (AMP) los canales de comunicación son colas ilimitadas de mensajes. Un proceso agrega un mensaje al final de la cola de un canal ejecutando una sentencia send. Dado que la cola conceptualmente es ilimitada la ejecución de send no bloquea al emisor.
Un proceso recibe un mensaje desde un canal ejecutando la sentencia receive que es bloqueante, es decir, el proceso no hace nada hasta recibir un mensaje en el canal. La ejecución del receive demora al receptor hasta que el canal este no vacío, luego el mensaje al frente del canal es removido y almacenado en variables locales al receptor.
El acceso a los contenidos de cada canal es atómico y se respeta el orden FIFO, es decir, los mensajes serán recibidos en el orden en que fueron enviados. Se supone que los mensajes no se pierden ni se modifican y que todo mensaje enviado en algún momento puede ser leído. Los procesos comparten los canales. Es ideal para algoritmos del tipo HeartBeat o Broadcast.
En el pasaje de **mensajes sincrónico** (SMP) tanto send como receive son primitivas bloqueantes. Si un proceso trata de enviar a un canal se demora hasta que otro proceso este esperando recibir por ese canal. De esta manera, un emisor y un receptor sincronizan en todo punto de comunicación.
La cola de mensajes asociada a un send sobre un canal se reduce a un mensaje. Esto significa menor memoria. Los procesos no comparten los canales. El grado de concurrencia se reduce respecto a AMP. Como contrapartida los casos de falla y recuperación de errores son más fáciles de manejar. Con SMP se tiene mayor probabilidad de deadlock. El programador debe ser cuidadoso para que todos los send y receive hagan matching. Es ideal para algoritmos del tipo Cliente/Servidor o de Filtros.

9) A que se denomina propiedad de programa? Qué son las propiedades de vida y seguridad? Ejemplifique.

- **Propiedad de programa** es un atributo que es verdadero en cualquier posible historia del programa, y por lo tanto de todas las ejecuciones del programa. Cada propiedad puede ser formulada en términos de dos clases especiales de propiedades:
    - **Propiedad de seguridad** asegura que el programa nunca entra en un estado malo (es decir uno en el que algunas variables tienen valores indeseables).
    - **Propiedad de vida** asegura que el programa eventualmente entra en un estado bueno (es decir, uno en el cual todas las variables tiene valores deseables).

Ejemplos de propiedad de seguridad:

1. ***ausencia de demora innecesaria****:* Si un proceso está tratando de entrar a su SC y los otros están ejecutando sus SNC o terminaron, el primero no está impedido de entrar a su SC.

2. ***exclusión mutua****:* Asegura que a lo sumo un proceso a la vez está ejecutando en su sección crítica. El estado malo en este caso sería uno en el cual las acciones en las regiones críticas en distintos procesos fueran ambas elegibles para su ejecución.

3. ***ausencia de deadlock****:* El estado malo es uno en el que todos los procesos están bloqueados, es decir, no hay acciones elegibles. Si 2 o más procesos tratan de entrar a su SC, al menos uno tendrá éxito.

Ejemplos de propiedad de vida:

Terminación*:* Asegura que un programa eventualmente terminará.

Eventual Entrada*.* Un proceso que está intentando entrar a su SC eventualmente lo hará.

10) En qué consiste la propiedad "A los sumo una vez (ASV)"

Atomicidad Fine-Grained

Acción atómica: hace una transformación de estado indivisible. Esto significa que cualquier estado intermedio que podría existir en la implementación de la acción no debe ser visible para los otros procesos. Acción atómica fine-grained es implementada directamente por el hardware sobre el que ejecuta el programa concurrente.

En un programa secuencial, las asignaciones aparecen como atómicas ya que ningún estado intermedio es visible al programa. Sin embargo, esto en general no ocurre en los programas concurrentes, ya que una asignación con frecuencia es implementada por una secuencia de instrucciones de máquina fine-grained. Por ejemplo, consideremos el siguiente programa, y asumamos que las acciones atómicas fine-grained están leyendo y escribiendo las variables:

y := 0; x := 0
co *x := y + z // y := 1; z := 2* ***oc***

Si x := y + z es implementada cargando un registro con y, luego sumándole z, el valor final de x podría ser 0, 1, 2 o 3. Esto es porque podríamos ver los valores iniciales para *y* y *z*, sus valores finales, o alguna combinación, dependiendo de cuán rápido se ejecuta el segundo proceso. Otra particularidad del programa anterior es que el valor final de x podría ser 2, aunque y+z no es 2 en ningún estado de programa.

Variable simple como una variable escalar, elemento de arreglo o campo de registro que es almacenada en una posición de memoria única. **Si una expresión o asignación satisface la siguiente propiedad, la evaluación aún será atómica:**

Propiedad de "a lo sumo una vez". Una expresión ***e*** satisface la propiedad de "a lo sumo una vez" si se refiere a lo sumo a una variable simple ***y*** que podría ser cambiada por otro proceso mientras ***e*** está siendo evaluada, y se refiere a ***y*** a lo sumo una vez.

Una sentencia de asignación **x:=e** satisface esta propiedad si satisface la propiedad y ***x*** no es leída por otro proceso, o si ***x*** es una variable simple y ***e*** no se refiere a ninguna variable que podría ser cambiada por otro proceso.

Para que una sentencia de asignación satisfaga ASV, ***e*** se puede referir a una variable alterada por otro proceso si **x** no es leída por otro proceso (es decir, ***x*** es una variable local). Alternativamente, ***x*** puede ser leída por otro proceso si ***e*** no referencia ninguna variable alterada por otro proceso.

Ninguna asignación en lo siguiente satisface ASV:

co *x := y + 1 // y := x + 1* ***oc***

11) Explique la semántica de la instrucción de grano grueso AWAIT y su relación con instrucciones tipo Test & Set o Fetch & Add.

Es una acción atómica coarse grained, la cual es una secuencia de acciones atómicas fine grained que aparecen como indivisibles. La sentencia await es muy poderosa ya que puede ser usada para especificar acciones atómicas arbitrarias de coarse grained. Esto la hace conveniente para expresar sincronización. Este poder expresivo también hace a await muy costosa de implementar en su forma más general. Sin embargo, hay casos en que puede ser implementada eficientemente utilizando instrucciones especiales que pueden usarse para implementar las acciones atómicas condicionales (test-and-set, fetch-and-add, compare-and-swap).
La instrucción Test-and-Set (TS):

TS(lock): < cc := lock; lock := true; return cc; >

Especificamos sincronización por medio de la sentencia await:

<**await** (B) S;>

La expresión booleana B especifica una condición de demora; S es una secuencia de sentencias secuenciales que se garantiza que termina. Una sentencia await se encierra en corchetes angulares para indicar que es ejecutada como una acción atómica. En particular, se garantiza que B es true cuando comienza la ejecución de S, y ningún estado interno de S es visible para los otros procesos.

Para especificar solo exclusión mutua, abreviaremos una sentencia **await** como sigue:

<S>

Para especificar solo sincronización por condición, abreviaremos una sentencia **await** como:

<**await** (B) >

busy waiting* o *spinning: Si una sentencia **await** cumple los requerimientos de la propiedad de a lo sumo una vez, entonces <**await** B> puede ser implementado como:

do (**not** B) → **skip od**

Cuando la sincronización se implementa de esta manera, un proceso se dice que está en ***busy waiting* o *spinning*,** ya que está ocupado haciendo un chequeo de la guarda.

12) Diferencie acciones atómicas condicionales e incondicionales

Acción atómica incondicional: es una que no contiene una condición de demora B. Tal acción puede ejecutarse inmediatamente.
Acción atómica condicional: es una sentencia await con una guarda B. Tal acción no puede ejecutarse hasta que B sea true. Si B es false, solo puede volverse true como resultado de acciones tomadas por otros procesos.

13) Describa fairness. Relacione dicho concepto con las políticas de scheduling

La mayoría de las propiedades de vida dependen de fairness, la cual trata de garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los otros procesos. Cuando hay varios procesos hay varias acciones atómicas elegibles (es elegible si es la próxima acción atómica en el proceso que será ejecutado). Una política de scheduling determina cuál será la próxima en ejecutarse.

(1) **Fairness Incondicional**. Una política de scheduling es incondicionalmente fair si toda acción atómica incondicional que es elegible eventualmente es ejecutada.

(2) **Fairness Débil**. Una política de scheduling es débilmente fair si es incondicionalmente fair y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada si su guarda se convierte en true y de allí en adelante permanece true.

(3) **Fairness Fuerte**. Una política de scheduling es fuertemente fair si es incondicionalmente fair y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada si su guarda es true con infinita frecuencia.

14) En qué consisten las arquitecturas SIMD y MIMD? Para qué tipo de aplicaciones es más adecuada cada una?

MIMD cada procesador tiene su propio flujo de instrucciones y de datos entonces c/u ejecuta su propio programa. Pueden ser con memoria compartida o distribuida. Logra paralelismo total, en cualquier momento se pueden estar ejecutando diferentes instrucciones con diferentes datos en procesadores diferentes. Ideal para simulaciones, comunicación y diseño.
SIMD tiene múltiples flujos de datos pero sólo un flujo de instrucción. En particular, cada procesador ejecuta exactamente la misma secuencia de instrucciones, y lo hacen en un lockstep. Esto hace a las máquinas SIMD especialmente adecuadas para ejecutar algoritmos paralelos de datos. Logra un paralelismo a nivel de datos, o sea ejecuta la misma instrucción en todos los procesadores.

15) Describa las propiedades que debe cumplir un protocolo de E/S a una sección crítica

(1) **Exclusión mutua**. A lo sumo un proceso a la vez está ejecutando su sección crítica

(2) **Ausencia de Deadlock**. Si dos o más procesos tratan de entrar a sus secciones críticas, al menos uno tendrá éxito.

(3) **Ausencia de Demora Innecesaria**. Si un proceso está tratando de entrar a su SC y los otros están ejecutando sus SNC o terminaron, el primero no está impedido de entrar a su SC.

(4) **Eventual Entrada**. Un proceso que está intentando entrar a su SC eventualmente lo hará.

La primera propiedad es de seguridad, siendo el estado malo uno en el cual dos procesos están en su SC. En una solución busy-waiting, es una propiedad de vida llamada ausencia de livelock. Esto es porque los procesos nunca se bloquean (están vivos) pero pueden estar siempre en un loop tratando de progresar. (Si los procesos se bloquean esperando para entrar a su SC, es una propiedad de seguridad). La tercera propiedad es de seguridad, siendo el estado malo uno en el cual el proceso uno no puede continuar. La última propiedad es de vida ya que depende de la política de scheduling.

16) Cuáles son los defectos que presenta la sincronización por busy waiting? Diferencie esta situación respecto de los semáforos.

Problemas busy waiting: Los protocolos de sincronización que usan solo busy waiting pueden ser difíciles de diseñar, entender y probar su corrección. La mayoría de estos protocolos son bastante complejos, no hay clara separación entre las variables usadas para sincronización y las usadas para computar resultados.
Otra deficiencia es que es ineficiente cuando los procesos son implementados por multiprogramación. Un procesador que está ejecutando un proceso “spinning” podría ser empleado más productivamente por otro proceso. Esto también ocurre en un multiprocesador pues usualmente hay más procesos que procesadores.

Los **semáforos** son herramientas para la sincronización que hacen fácil proteger SC y pueden usarse de manera disciplinada para implementar sincronización por condición. Los semáforos pueden ser implementados de más de una manera. En particular, pueden implementarse con busy waiting.

17)

a. Qué significa que un problema sea de “exclusión mutua selectiva”?

Que un problema sea de exclusión mutua selectiva significa que los procesos tienen que competir por sus recursos no con todos los demás procesos sino con un subconjunto de ellos. Compiten por el acceso a conjuntos superpuestos de variables compartidas. Dos casos típicos de dicha competencia se producen cuando los procesos compiten por los recursos según su ‘tipo’ de proceso o por su proximidad.

b. El problema de los lectores y escritores es de exclusión mutua selectiva? Por qué?

El problema de los lectores y escritores es de exclusión mutua selectiva porque existen distintas clases de procesos que compiten por el acceso a la BD que es compartida por ambos. Los procesos escritores individualmente compiten por el acceso con cada uno de los otros y los procesos lectores como una clase compiten con los escritores.

c. Si en el problema de los lectores y escritores se acepta solo 1 escritor y 1 lector en la BD, tenemos un problema de exclusión mutua selectiva? Por qué?

Si en el problema se acepta solo 1 escritor y 1 lector en la BD el problema deja de ser de exclusión mutua selectiva porque la competencia por acceder a la BD es contra todos.

d. De los problemas de los baños planteados en la teoría, Cuáles podría ser de exclusión mutua selectiva?

De los problemas de los baños planteados en la teoría los que podrían ser de exclusión mutua selectiva son el que tienen un único baño para varones o mujeres (excluyente) sin límite de usuarios y el que tenían un baño único para varones o mujeres (excluyente) con un número máximo de ocupantes simultáneos (que puede ser diferente para varones y mujeres)

18)

a. Por que el problema de los filósofos es de exclusión mutua selectiva?

El problema de los filósofos es de exclusión mutua selectiva porque los procesos compiten por acceder a los tenedores.

b. **Si en lugar de 5 filósofos fueran 3, el problema seguiría siendo de exclusión mutua selectiva? Por qué?**

Si en lugar de 5 filósofos fueran 3 filósofos el problema deja de ser de exclusión mutua selectiva porque todos los filósofos compiten por acceder a las variables compartidas.

c. **El problema de los filósofos resuelto en forma centralizada y sin posiciones fijas es de exclusión mutua selectiva? Por qué**?

El problema de los filósofos resuelto en forma centralizada y sin posiciones fijas no es de exclusión mutua selectiva porque en este caso los procesos filósofos se comunican con un proceso WAITER que decide el acceso o no a los recursos. Al no haber posiciones fijas no hay conjuntos superpuestos.

19)

a. En qué consiste la técnica de "Passing the Baton"? Cuál es su utilidad? Aplique este concepto a la resolución del problema de lectores y escritores. Qué relación encuentra con la técnica de "Passing the Condition"?

La técnica consiste en implementar sincronización por condición arbitraria. Cuando un proceso está dentro de una SC mantiene el baton (testimonio, token) que significa permiso para ejecutar. Cuando el proceso llega a un SIGNAL, pasa el baton (control) a otro proceso. Si algún proceso está esperando una condición que ahora es verdadera el baton pasa a tal proceso, el cual ejecuta su SC y pasa el baton a otro proceso. Si ningún proceso está esperando una condición que sea true, el baton se pasa al próximo proceso que trata de entrar a su SC por primera vez.
Podemos usar semáforos binarios divididos para implementar tanto la EM como la SxC. Sea **e** un semáforo binario cuyo valor inicial es 1 que se usa para controlar la entrada a sentencias atómicas y asociamos un semáforo bj y un contador dj cada uno con guarda semánticamente diferente Bj. El semáforo bj se usa para demorar procesos esperando que Bj se convierta en true y dj es un contador del número de procesos demorados sobre bj.

Para representar <Si>: *P(e)*
Si;
SIGNAL
Para representar <await (Bj) Sj>: *P(e)*
if (not Bj) {dj := dj + 1; V(e); P(bj);}
Sj;
SIGNAL
En ambos fragmentos, SIGNAL es la siguiente sentencia:

```
SIGNAL: if (B1 and d1 > 0){ d1 := d1 - 1; V(b1)}
    [] ...
    [] (Bn and dn > 0){ dn := dn - 1; V(bn)}
    [] else V(e);
fi
```

Las primeras N guardas en SIGNAL chequean si hay algún proceso esperando por una condición que ahora es true. Para el caso que no haya ningún proceso esperando por alguna condición verdadera se ejecutaría la guarda else donde el semáforo de entrar e es señalizado.
Su utilidad es proveer exclusión y controlar cuál proceso demorado es el próximo en seguir y el orden en el cual los procesos son demorados. Puede usarse para implementar cualquier sentencia await tanto <Si> como <await (Bj) Sj >.
Aplicando esta técnica al problema de los lectores y escritores tenemos esta solución:

```
int nr = 0, nw = 0;
sem e = 1, r = 0, w = 0
int dr = 0, dw = 0;
process Lector [i = 1 to M] {
    while(true) {
                P(e);
                if (nw > 0) {       #Si hay un escritor en la BD me "encolo" para ser el próximo
                   dr = dr+1;       #Incremento la variable para avisar que hay un lector esperando
                   V(e);
                   P(r);
                }
                nr = nr + 1;        #Incremento la variable para avisar que hay un lector en la BD
                if (dr > 0) {       #Si hay un lector esperando lo despierto
                    dr = dr – 1;
                    V(r);
                }
                else V(e);          #Sino libero la SC y entro a la BD
                lee la BD;
                P(e);
                nr = nr – 1;        #Decremento la variable para avisar que salí de la BD
                if (nr == 0 and dw > 0) { #Si no hay ningún lector por ingresar y hay escritores dormidos
                  dw = dw – 1;              #Despierto a uno
                  V(w);
                }
                else V(e);
   }
}
```

```
process Escritor [j = 1 to N] {
    while(true) {
        P(e);
        if (nr > 0 or nw > 0) {        #Si hay lectores o escritores en la BD me duermo
           dw = dw+1;
           V(e);
           P(w);
        }
        nw = nw + 1;                   #Incremento la variable para avisar que hay un escritor en la BD
        V(e);
        escribe la BD;
        P(e);
        nw = nw – 1;                   #Decremento la variable para avisar que salí de la SC
        if (dr > 0) {                  #Si hay un lector esperando para entrar lo despierto
            dr = dr - 1;
            V(r);
        }
        Else if (dw > 0) {             #Si hay un escritor esperando para entrar lo despierto
           dw = dw - 1;
           V(w); }
        else V(e);
    }
}
```

La relación que tiene Passing the Baton con Passing de Condition es que ambos determinan cual es el proceso demorado que será despertado para que ejecute su SC.
Como diferencia entre las técnicas podemos decir de que la técnica Passing the Condition cuando un proceso hace signal pasa el control directamente al proceso que estaba esperando para poder entrar a la SC antes que otros procesos que llaman a un procedure dentro del monitor tengan chance de avanzar, a diferencia de la técnica de Passing the Baton que cuando un proceso hace signal es "globalmente visible" por todos los procesos que quieren entrar en la SC.
La técnica Passing the Condition pasa una condición directamente a un proceso despertado en lugar de hacerla globalmente visible.

20) Mencione qué inconvenientes presentan los semáforos como herramienta de sincronización para la resolución de problemas concurrentes

Semáforos Inconvenientes:
-Variables compartidas globales a los procesos
-Sentencias de control de acceso a la SC dispersas en el código
-Al agregar procesos, se debe verificar acceso correcto a las variables compartidas.
-Aunque EM y SxC son conceptos distintos, se programan de forma similar

Los **Monitores** son módulos de programa con más estructura, y que pueden ser implementados tan eficientemente como los semáforos.

Mecanismo de abstracción de datos:
• encapsulan las representaciones de objetos (recursos).
• brindan un conjunto de operaciones que son los únicos medios para manipular la representación

Contiene variables que almacenan el estado del recurso y procedimientos que implementan las operaciones sobre él.

21) Explique sintéticamente los 7 paradigmas de interacción entre procesos planteados en la teoría.

- **Paradigma de servidores replicados**: un server puede ser replicado cuando hay múltiples instancias distintas de un recurso y así cada server manejaría una instancia. La replicación sirve para incrementar la accesibilidad de datos o servicios teniendo más de un proceso que provea el mismo servicio.
- **Paradigma de algoritmos Heartbeat:** Cada proceso ejecuta una secuencia de iteraciones. En cada iteración un proceso envía su conocimiento local de por ejemplo la topología de una red a todos sus vecinos, luego recibe la información de ellos y la combina con la suya. La computación termina cuando todos los procesos aprendieron la topología de red completa. Por lo tanto en estos algoritmos las acciones de cada nodo primero se expanden enviando información, luego se contrae incorporando nueva información.
- **Paradigma algoritmos pipeline**: La información recorre una serie de procesos utilizando alguna forma de receive/send.
- **Paradigma de probes y echos:** se basa en el envío de un mensaje ("probe") enviado por un nodo a su sucesor y la espera posterior del "eco" que es el mensaje de respuesta. Dado que los procesos ejecutan concurrentemente, los probes se envían en paralelo a todos los sucesores. Este paradigma es análogo en programación concurrente a DFS. La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información.
- **Paradigma de BroadCast:** este paradigma consiste en transmitir un mensaje desde un emisor a muchos receptores. Los mensajes broadcast de un proceso se encolan en los canales en el orden de envío, pero broadcast no es atómico y los mensajes enviados por procesos A y B podrían ser recibidos por otros en distinto orden.
- **Paradigma de Token Passing:** se basa en un tipo especial de mensajes o "token" que pueden utilizarse para otorgar un permiso (control) o recoger información global de la arquitectura distribuida.
- **Paradigma Manager/Workers**: representa una implementación distribuida del modelo de Bags of Tasks. Para esto un proceso manager implementara la "bolsa" manejando los tasks, comunicándose con los workers y detectando fin de tareas. Se trata de un esquema Cliente/Servidor.

22)

a) Describa brevemente en que consisten los mecanismos de RPC y Rendezvous. Para qué tipo de problemas son más adecuados?

RPC y rendezvous son ideales para interacciones cliente/servidor. Ambas combinan aspectos de monitores y SMP. Como con monitores, un módulo o proceso exporta operaciones, y las operaciones son invocadas por una sentencia call. Como con las sentencias de salida en SMP, la ejecución de call demora al llamador. La novedad de RPC y rendezvous es que una operación es un canal de comunicación bidireccional desde el llamador al proceso que sirve el llamado y nuevamente hacia el llamador. En particular, el llamador se demora hasta que la operación llamada haya sido ejecutada y se devuelven los resultados.
La diferencia entre RPC y rendezvous es la manera en la cual se sirven las invocaciones de operaciones. Una aproximación es declarar un procedure para cada operación y crear un nuevo proceso (al menos conceptualmente) para manejar cada llamado. La segunda aproximación es rendezvous con un proceso existente. Un rendezvous es servido por medio de una sentencia de entrada (o accept) que espera una invocación, la procesa, y luego retorna resultados.

REMOTE PROCEDURE CALL

Con RPC, usaremos una componente de programa (el módulo) que contiene tanto procesos como procedures. Los procesos dentro de un módulo pueden compartir variables y llamar a procedures declarados en ese módulo. Sin embargo, un proceso en un módulo puede comunicarse con procesos en un segundo módulo solo llamando procedures del segundo módulo. Un módulo tiene dos partes. La parte de especificación (spec) contiene headers de procedures que pueden ser llamados desde otros módulos. El cuerpo implementa estos procedures y opcionalmente contiene variables locales, código de inicialización, y procedures locales y procesos.

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

RENDEZVOUS

Rendezvous combina las acciones de servir un llamado con otro procesamiento de la información transferida por el llamado. Con rendezvous, un proceso exporta operaciones que pueden ser llamadas por otros. Una declaración de proceso tendrá la siguiente forma:

pname:: declaraciones de operación
        declaraciones de variables
        sentencias

Las declaraciones especifican los headers de las operaciones servidas por el proceso.

Un proceso invoca una operación por medio de una sentencia call, la cual en este caso nombra otro proceso y una operación en ese proceso. Pero en contraste con RPC, una operación es servida por el proceso que la exporta. Por lo tanto, las operaciones son servidas una a la vez en lugar de concurrentemente.

Si un proceso exporta una operación *op*, puede rendezvous con un llamador de *op* ejecutando:

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

Llamaremos a las partes entre las palabras claves *operación guardada*. La guarda nombra una operación y sus parámetros formales; el cuerpo contiene una lista de sentencias *S*. El alcance de los formales es la operación guardada entera.

En particular, **in** demora al proceso servidor hasta que haya al menos un llamado pendiente de *op*. Luego elige el llamado pendiente más viejo, copia los argumentos en los formales, ejecuta *S*, y finalmente retorna los parámetros resultados al llamador. En ese punto, ambos procesos (el que ejecuta el **in** y el que llamó a *op*) pueden continuar la ejecución.

Para combinar comunicación guardada con rendezvous, generalizaremos la sentencia **in** como sigue:

in op_1 (formales1) **and** B_1 **by** e1 → S_1
[] .....
[] op_n (formalesn) **and** B_n **by** en → S_n
ni

Cada operación guardada nuevamente nombra una operación y sus formales, y contiene una lista de sentencias. Sin embargo, la guarda también puede contener dos partes opcionales. La segunda parte es una expresión de sincronización (**and** B_i); si se omite, se asume que B_i es true. La tercera parte de una guarda es una expresión de scheduling (**by** e_i); (El lenguaje Ada soporta rendezvous por medio de la sentencia **accept** y comunicación guardada por medio de la sentencia **select**. El **accept** es como la forma básica de **in**, pero el **select** es menos poderoso que la forma general de **in**. Esto es porque **select** no puede referenciar argumentos a operaciones o contener expresiones de scheduling).

Una guarda en una operación guardada tiene éxito cuando (1) la operación fue llamada, y (2) la expresión de sincronización correspondiente es verdadera. La ejecución de **in** se demora hasta que alguna guarda tenga éxito. Como es usual, si más de una guarda tiene éxito, una de ellas es elegida no determinísticamente. Si no hay expresión de scheduling, la sentencia **in** sirve la invocación más vieja que hace que la guarda tenga éxito.

Una expresión de scheduling se usa para alterar el orden de servicio de invocaciones por default (primero la invocación más vieja).

b) Por qué es necesario proveer sincronización dentro de los módulos de RPC? Cómo puede realizarse esta sincronización?

Por sí mismo, RPC es puramente un mecanismo de comunicación. Aunque un proceso llamador y su server sincronizan, el único rol del server es actuar en nombre del llamador. Conceptualmente, es como si el proceso llamador mismo estuviera ejecutando el llamado, y así la sincronización entre el llamador y el server es implícita.

También necesitamos alguna manera para que los procesos en un módulo sincronicen con cada uno de los otros. Estos incluyen tanto a los procesos server que están ejecutando llamados remotos como otros procesos declarados en el módulo. Como es habitual, esto comprende dos clases de sincronización: exclusión mutua y sincronización por condición.

Hay dos aproximaciones para proveer sincronización en módulos, dependiendo de si los procesos en el mismo módulo ejecutan con exclusión mutua o ejecutan concurrentemente. Si ejecutan con exclusión (es decir, a lo sumo hay uno activo por vez) entonces las variables compartidas son protegidas automáticamente contra acceso concurrente. Sin embargo, los procesos necesitan alguna manera de programar sincronización por condición. Para esto podríamos usar automatic signalig (**await** B) o variables condición.

Si los procesos en un módulo pueden ejecutar concurrentemente (al menos conceptualmente), necesitamos mecanismos para programar tanto exclusión mutua como sincronización por condición. En este caso, cada módulo es en sí mismo un programa concurrente, de modo que podríamos usar cualquiera de los métodos descriptos anteriormente. Por ejemplo, podríamos usar semáforos dentro de los módulos o podríamos usar monitores locales. De hecho, como veremos, podríamos usar rendezvous. (O usar MP).

23) Describa las características de comunicación y sincronización de la "notación de primitivas múltiples".

Es una notación que combina RPC, Rendezvous y PMA en un paquete coherente. Provee un gran poder expresivo combinando ventajas de las 3 componentes, y poder adicional.

Como con RCP se estructura los programas con colecciones de módulos.
Una operación visible es especificada en la declaración del módulo y puede ser invocada por procesos de otros módulos pero es servida por un proceso o procedure del módulo que la declara. También se usan operaciones locales, que son declaradas, invocadas y servidas solo por el módulo que la declara.

Una operación puede ser invocada por **call** sincrónico o por **send** asincrónico y las sentencias de invocación son de la siguiente manera:

call Mname.op(argumentos)
send Mname.op(argumentos)

Como por RCP y Rendezvous el call termina cuando la operación fue servida y los resultados fueron retornados. Como con AMP el send termina tan pronto como los argumentos fueron evaluados.
En la notación de primitivas múltiples una operación puede ser servida por un procedure (**proc**) o por rendezvous (sentencias **in**). La elección la toma el programador del modulo que declara la operación. Depende de si el programador quiere que cada invocación sea servida por un proceso diferente o si es más apropiado rendezvous con un proceso existente.
En resumen, hay dos maneras de invocar una operación (call o send) y dos maneras de servir una invocación (proc o in).

24) Describa los mecanismos de comunicación y sincronización provistos por ADA, OCCAM, LINDA y SR.

En ADA el rendezvous es el único mecanismo de sincronización y también es el mecanismo de comunicación primario.
Las declaraciones de entry tienen la forma **entry** identificador (formales). Los parámetros del entry pueden ser in, out o in out.
Ada también soporta arreglos de entries, llamados familias de entry.
Si la task T declara el entry E, otras tasks en el alcance de la especificación de T pueden invocar a E por una sentencia call: **call** T.E(reales). Como es usual, la ejecución de call demora al llamador hasta que la operación E terminó (o abortó o alcanzó una excepción).
La task que declara un entry sirve llamados de ese entry por medio de la sentencia accept. Esto tiene la forma general: accept E(formales) do lista de sentencias end;
La ejecución de accept demora la tarea hasta que haya una invocación de E, copia los argumentos de entrada en los formales de entrada, luego ejecuta la lista de sentencias. Cuando la lista de sentencias termina, los formales de salida son copiados a los argumentos de salida. En ese punto, tanto el llamador como el proceso ejecutante continúan.
Para controlar el no determinismo, Ada provee tres clases de sentencias select: wait selectivo, entry call condicional, y entry call timed.
La sentencia wait selectiva soporta comunicación guardada. La forma más común de esta sentencia es:

select when *B1 → sentencia* ***accept E1****; sentencias1*
or *...*
or ***when*** *Bn → sentencia* ***accept En****; sentenciasn*
end select

Cada línea (salvo la última) se llama alternativa. Las Bi son expresiones booleanas, y las cláusulas when son opcionales. Una alternativa se dice que está abierta si Bi es true o se omite la cláusula when. Esta forma de wait selectivo demora al proceso ejecutante hasta que la sentencia accept en alguna alternativa abierta pueda ser ejecutada, es decir, haya una invocación pendiente del entry nombrado en la sentencia accept. Dado que cada guarda Bi precede una sentencia accept, no puede referenciar los parámetros de un entry call. Además, Ada no provee expresiones de scheduling, lo cual hace difícil resolver algunos problemas de sincronización y scheduling.
La sentencia wait selectiva puede contener una alternativa opcional else, la cual es seleccionada si ninguna otra alternativa puede serlo. En lugar de la sentencia accept, el programador puede también usar una sentencia **delay** o una alternativa **terminate**. Una alternativa abierta con una sentencia delay es seleccionada si transcurrió el intervalo de delay; esto provee un mecanismo de timeout. La alternativa terminate es seleccionada esencialmente si todas las tasks que rendezvous con esta terminaron o están esperando una alternativa terminate.
Estas distintas formas de sentencias wait selectiva proveen una gran flexibilidad, pero también resultan en un número algo confuso de distintas combinaciones. Para "empeorar" las cosas, hay dos clases adicionales de sentencia select.
Un entry call condicional se usa si una task quiere hacer polling de otra. Tiene la forma:

select entry call; sentencias adicionales
else sentencias
end select

El entry call es seleccionado si puede ser ejecutado inmediatamente; en otro caso, se selecciona la alternativa else.
Un entry call timed se usa si una task llamadora quiere esperar a lo sumo un cierto intervalo de tiempo. Su forma es similar a la de un entry call condicional:

select entry call; sentencias
or sentencia de delay; sentencias
end select

En este caso, el entry call se selecciona si puede ser ejecutado antes de que expire el intervalo de delay. Esta sentencia soporta timeout, en este caso, de un call en lugar de un accept.
Ada provee unos pocos mecanismos adicionales para programación concurrente. Las tasks pueden compartir variables; sin embargo, no pueden asumir que estas variables son actualizadas excepto en puntos de sincronización (por ej, sentencias de rendezvous). La sentencia abort permite que una tarea haga terminar a otra. Hay un mecanismo para setear la prioridad de una task. Finalmente, hay atributos que habilitan para determinar cuándo una task es llamable o ha terminado o para determinar el número de invocaciones pendientes de un entry.

En OCCAM no se puede compartir variables es por esto que para comunicarse y sincronizar se usa canales, en particular se usan pasaje de mensajes sincrónicos. Una declaración de canal tiene la forma: **CHAN OF** protocol name. El protocolo define el tipo de valores que son transmitidos por el canal. Pueden ser tipos básicos, arreglos de longitud fija o variable, o registros fijos o variantes.
Los canales son accedidos por los procesos primitivos input (?) y output (!). A lo sumo un proceso compuesto puede emitir por un canal, y a lo sumo uno puede recibir por un canal.

LINDA no es en sí mismo un lenguaje de programación sino un pequeño número de primitivas que son usadas para acceder lo que llamamos espacio de tupla (tuple space, TS). El TS es una memoria compartida, asociativa, consistente de una colección de registros de datos tagged llamados tuplas. TS es como un canal de comunicación compartido simple, excepto que las tuplas no están ordenadas.

Linda generaliza y sintetiza aspectos de variables compartidas y AMP. La operación para depositar una tupla (**out**) es como un send, la operación para extraer una tupla (**in**) es como un receive, y la operación para examinar una tupla (**rd**) es como una asignación desde variables compartidas a locales. Una cuarta operación, **eval**, provee creación de procesos. Las dos operaciones finales, **inp** y **rdp**, proveen entrada y lectura no bloqueante.

En SR hay una variedad de mecanismos de comunicación y sincronización. Los procesos pueden comunicarse y sincronizar también usando semáforos, AMP, RPC, y rendezvous. Así, SR puede ser usado para implementar programas concurrentes tanto para multiprocesadores de memoria compartida como para sistemas distribuidos.
Las operaciones son declaradas en declaraciones **op** como con RCP. Tales declaraciones pueden aparecer en especificaciones de recurso, en cuerpos de recurso, y aún dentro de procesos. Una operación declarada dentro de un proceso es llamada operación local. El proceso declarante puede pasar una capability para una operación local a otro proceso, el cual puede entonces invocar la operación. Esto soporta continuidad conversacional.
Una operación es invocada usando **call** sincrónico o **send** asincrónico. Para especificar qué operación invocar, una sentencia de invocación usa una capability de operación o un campo de una capability de recurso. Dentro del recurso que la declara, el nombre de una operación es de hecho una capability, de modo que una sentencia de invocación puede usarla directamente. Las capabilities de recurso y operación pueden ser pasadas entre recursos, de modo que los paths de comunicación pueden variar dinámicamente.
Una operación es servida o por un procedure (**proc**) o por sentencias de entrada (**in**). Un nuevo proceso es creado para servir cada llamado remoto de un proc. Todos los procesos en un recurso ejecutan concurrentemente, al menos conceptualmente. La sentencia de entrada soporta rendezvous. Puede tener tanto expresiones de sincronización como de scheduling que dependen de parámetros. La sentencia de entrada también puede contener una cláusula opcional else, que es seleccionada si ninguna otra guarda tiene éxito.
Una declaración **process** es una abreviación para una declaración **op** y un **proc** para servir invocaciones de la operación. Una instancia del proceso se crea por un send implícito cuando el recurso es creado. El cuerpo de un process con frecuencia es un loop permanente. (También pueden declararse arreglos de procesos). Una declaración procedure es una abreviación para una declaración op y un proc para servir invocaciones de la operación.
Dos abreviaciones adicionales son la sentencia **receive** y los semáforos. En particular, receive abrevia una sentencia de entrada que sirve una operación y que solo almacena los argumentos en variables locales. Una declaración de semáforo (**sem**) abrevia la declaración de una operación sin parámetros. La sentencia **P** es un caso especial de receive, y la sentencia **V** un caso especial de send.

25) Describa los mecanismos de comunicación y sincronización provistos por MPI, JAVA.

MPI es una biblioteca de comunicaciones a través de pasaje de mensajes que permite comunicar y sincronizar procesos secuenciales escritos en diferentes lenguajes que se ejecutan sobre una arquitectura distribuida. El estilo de programación es SPMD. C/ proceso ejecuta una copia del mismo programa, y puede tomar distintas acciones de acuerdo a su "identidad". Las instancias interactúan llamando a funciones MPI, que soportan comunicaciones proceso a proceso y grupales. MPI_init: inicializar entorno MPI. MPI_finalize: cerrar entorno MPI. MPI_common_size y MPI_common_rank: cantidad de procesos en el comunicador e identificador del proceso dentro del comunicador. MPI_send y MPI_recv: ambos bloqueantes. MPI_Isend y MPI_Irecv: no bloqueantes.

Java soporta concurrencia mediante threads, utiliza métodos sincronizados (monitores). EM implícita provista por la declaración de la palabra "synchronized". EM explicita (sincronización por condición) con los métodos wait, notify y notifyAll. Además provee el uso de RCP en programas distribuidos mediante la invocación de métodos remotos (RMI).

El server y los clientes pueden residir en maquinas diferentes. Una aplicación que usa RMI tiene 3 componentes:

- Una interface que declara los headers para métodos remotos
- Una clase server que implementa la interface
- Uno o más clientes que llaman a los métodos remotos

26) Defina las métricas de speedup y eficiencia. Cuál es el significado de cada una de ellas (qué miden)? Ejemplifique. En qué consiste la "ley de Amdahl".

Ambas técnicas son métricas relativas que representan la fracción de tiempo que los procesadores emplean realizando tareas útiles para la resolución de los algoritmos. Esta métrica caracteriza la efectividad con que el algoritmo paralelo usa los recursos de las computadoras.
La métrica de *speedup* mide el 'tiempo de corrida' o costo de ejecución de un algoritmo. Mide la relación entre el tiempo medio de ejecución del algoritmo ejecutando sobre un procesador y el tiempo medio de ejecución del algoritmo ejecutando sobre p procesadores. Es decir: S = Ts / Tp (siendo Ts el tiempo de ejecución secuencial del algoritmo y Tp el tiempo de ejecución del algoritmo en forma paralela entre p procesadores). En el caso del secuencial debe considerarse el mejor algoritmo secuencial para resolver el problema dado, el cual puede no ser el mismo que el caso paralelo.
El valor máximo de S es en general entre 0 y p. Si S alcanza p se puede afirmar que el algoritmo es totalmente paralelizable para p procesadores, es decir, todos los procesadores trabajan de forma concurrente e inician y finalizan sus tareas en el mismo instante de tiempo, no hay dependencia de datos.
La eficiencia es el porcentaje de tiempo empleado en proceso efectivo. Mide la fracción de tiempo en que los procesadores son útiles para el cómputo. Es el cociente entre el speedup y la cantidad de procesadores. Es decir: E = Ts / pTp.
El valor está entre 0 y 1, dependiendo de la efectividad de uso de los procesadores. Cuando es 1 corresponde al speedup perfecto.

En cualquier programa paralelizado existen dos tipos de código; el código paralelizado y el código secuencial. Existen ciertas secciones de código que ya sea por dependencias, por acceso a recursos únicos o por requerimientos del problema no pueden ser paralelizadas. Estas secciones conforman el código secuencial, que debe ser ejecutado por un solo elemento del procesador.
Entonces, es lógico afirmar que la mejora del speedup de un programa dependerá del tiempo en el que se ejecuta el código secuencial, el tiempo en el que se ejecuta el código paralelizable y el número de operaciones ejecutadas de forma paralela.
La "Ley de Amdahl" enuncia que para cualquier tipo de problema existe un máximo speedup alcanzable que no depende de la cantidad de procesadores que se utilicen para resolverlo. Esto es así porque llega un momento en que no existe manera de aumentar el paralelismo de un programa.

27) Cuál es el objetivo de la programación paralela?

El objetivo principal de la programación paralela es reducir el tiempo de ejecución o resolver problemas más grandes o con mayor precisión en el mismo tiempo. Al contrario que en la programación concurrente esta técnica enfatiza la verdadera simultaneidad en el tiempo de la ejecución de las tareas.

28) Mencione 3 técnicas fundamentales de la computación científica. Ejemplifique.

Entre las diferentes aplicaciones de cómputo científicas y modelos computacionales existen tres técnicas fundamentales:

- *Computación de grillas:* (soluciones numéricas a PDE, imágenes). Dividen una región espacial en un conjunto de puntos. Muchas máquinas haciendo diferentes procesos todas con un objetivo en común.

- *Computación de partículas*: modelos que simulan interacciones de partículas individuales como moléculas u objetos estelares. Ejemplos, partículas cargadas que interactúan debido a las fuerzas eléctricas o magnéticas, moléculas que interactúan debido a la unión química.
- *Computación de matrices:* (sistemas de ecuaciones simultáneas, aplicaciones científicas y de ingeniería así como también modelos económicos)

29) En qué consiste la comunicación guardada (introducida por CSP) y cuál es su utilidad?

Un proceso puede tener que realizar una comunicación solo si se da una condición o también se puede dar el caso que se quiere comunicar con uno o más procesos y no sabe el orden en el cual otros procesos podrían querer comunicarse con él. Por esto la comunicación no determinística es soportada en forma elegante extendiendo las sentencias guardadas para incluir sentencias de comunicación.
Una sentencia de comunicación guardada tiene la forma B; C -> S donde B es una expresión booleana opcional, C es una sentencia de comunicación opcional y S es una lista de sentencias. Si B se omite tiene el valor implícito de true. Si C se omite una sentencia de comunicación guardada es simplemente una sentencia guardada.
Juntos B y C forman la guarda. La guarda tiene *éxito* si B es true y ejecutar C no causaría una demora. La guarda *falla* si B es falsa. La guarda se *bloquea* si B es true pero C no puede ser ejecutada sin causar demora.

Por ejemplo podemos programar Copy para implementar un buffer limitado. Por ejemplo, lo siguiente bufferea hasta 10 caracteres:

```
Copy:: var buffer[1:10] : char
       var front := 1, rear := 1, count := 0
       do count < 10; West ? buffer[rear] →
              count := count + 1;
              rear := (rear mod 10) + 1
        [] count > 0; East ! buffer[front] →
              count := count - 1;
              front := (front mod 10) + 1
       od
```

Nótese que la interface a West (el productor) e East (el consumidor) no cambió. La primera tiene éxito si hay lugar en el buffer y West está listo para sacar un carácter; la segunda tiene éxito si el buffer contiene un carácter e East está listo para tomarlo. La sentencia do no termina nunca pues ambas guardas nunca pueden fallar al mismo tiempo (al menos una de las expresiones booleanas en las guardas es siempre true).

30) Explique el concepto de broadcast y sus dificultades de implementación en un ambiente distribuido con mensajes sincrónicos y asincrónicos.

En la mayoría de las LAN, los procesadores comparten un canal de comunicación común tal como un Ethernet o token ring. Tales redes soportan un primitiva especial llamada broadcast, la cual transmite un mensaje de un procesador a todos los otros. Podemos utilizar broadcast para diseminar o reunir información.
Con AMP ejecutar broadcast es equivalente a enviar concurrentemente el mismo mensaje a varios canales. Todas las copias de mensajes se encolan y cada copia puede ser recibida más tarde por uno de los procesos participantes. Necesitamos la respuesta para confirmar la llegada de un mensaje, no se garantiza la llegada de un mensaje ante fallas.
Con SMP por la naturaleza bloqueante de las sentencias de salida un broadcast sincrónico debería bloquear al emisor hasta que los procesos receptores hayan recibido el mensaje. Hay dos maneras de obtener el efecto de broadcast con SMP. Una es usar un proceso separado para simular un canal broadcast; es decir, los clientes enviarían mensajes broadcast y los recibirían de ese proceso.

La segunda manera es usar comunicación guardada. Cuando un proceso quiere tanto enviar como recibir mensajes broadcast puede usar comunicación guardada con dos guardas. Cada guarda tiene un cuantificador para enumerar los otros partners. Una guarda saca el mensaje hacia los otros partners; la otra guarda ingresa un mensaje desde todos los otros partners.

31) Describa el paradigma "Bag of Tasks". Cuál es la principal ventaja del mismo? Ejemplifique.

Se parte del concepto de 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);
}
```

Este enfoque 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.
Por ejemplo: Multiplicación de matrices con Bag of Tasks
Consideraremos la multiplicación de 2 matrices a y b de nxn. El ejemplo requiere n2 productos internos entre filas de a y columnas de b. Cada producto interno es independiente y se puede realizar en paralelo
Suponemos que se dispone de una máquina multiprocesador con PR procesadores físicos, PR < n (PR procesos worker). Tratando de balancear la carga, puede pensarse en n tareas en la bolsa, una por fila y que cada PR haga uso de un número de tareas de modo de distribuir la carga.
Podemos representar la bolsa, simplemente contando filas: INT proxfila = 0;
Un worker saca una tarea de la bolsa ejecutando una acción atómica: < fila = proxfila; proxfila ++ > donde fila es una variable local. Notar que lo podemos implementar con un FETCH and ADD.

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

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

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.

32) En qué consiste la utilización de relojes lógicos para resolver problemas de sincronización distribuida? Ejemplifique.

Las acciones de los procesos en un programa distribuido pueden dividirse en:

-Acciones locales: no afectan a otros procesos

-Acciones de comunicación: si afectan a otros procesos

Evento → ejecución de un send o un receive.

- Si A y B ejecutan acciones locales, no se sabe el orden relativo
- Si A envía un msg a B, el send en A sucede antes que el receive en B
- Si B luego envía msg a C, el send de B debe ocurrir antes del receive en C

Aunque hay un ordenamiento total entre las 4 acciones de comunicación, hay solo orden parcial entre todos los eventos en un programa distribuido: las secuencias no relacionadas de eventos (ej. Comunicación entre distintos conjuntos de procesos) pueden ocurrir antes, después o concurrentemente con otras.

Si hubiera un único reloj central podríamos ordenar completamente los eventos dando a c/u un timestamp único.

Reloj lógico → contador entero incrementado en cada evento. Cada proceso tiene un reloj lógico init 0 y cada mensaje un timestamp.

Reglas de actualización de relojes: sea LC un reloj lógico en el proceso A. Dicho proceso actualiza el reloj de la siguiente manera:

(1) Cuando A envía o broadcast un msg, setea el timestamp del mensaje al valor de LC y luego lo incrementa en 1.

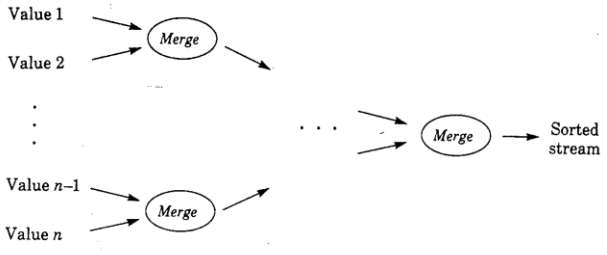
(2) Cuando A recibe un msg con timestamp TS, setea LC al máximo de LC y TS+1 y luego incrementa LC en 1.

Ejemplo: semáforos distribuidos.

33) Suponga los siguientes métodos de ordenación de menor a mayor para n valores (n par y potencia de 2), utilizando pasaje de mensajes:

i. Un pipeline de filtros. El primero hace input de los valores de a uno por vez, mantiene el mínimo y le pasa los otros al siguiente. Cada filtro hace lo mismo: recibe un stream de valores desde el predecesor, mantiene el más chico y pasa a los otros el sucesor.

ii. Una red de procesos filtro



iii. Odd/even Exchange sort. Hay n procesos P[1:n], cada uno ejecuta una serie de rondas. En las rondas impares los procesos con número impar P[impar] intercambian valores con P[impar + 1]. En las rondas pares, los procesos con numero par P[par] intercambian valores con P[par+1] (P[1] y P[n] no hacen nada en las rondas pares). En cada caso, si los números están desordenados actualizan su valor con el recibido.

Asuma que cada proceso tiene almacenamiento local solo para dos valores (el próximo valor y el mantenido hasta ese momento).

a. Cuántos procesos son necesarios en i e ii?

En i cada filtro es un proceso que recibe datos de un canal de entrada y entrega resultados por un canal de salida. Se necesitan n procesos.

Y en ii la información fluye de izq a der. A cada nodo de la izq se le dan 2 valores de entrada, los cuales ordena para formar un stream.

Los siguientes nodos forman streams de 4 valores ordenados y así sucesivamente. El nodo de más a la derecha produce el stream final. La red contiene n-1 procesos y el ancho de la red es log2n.

b. Cuántos mensajes envía cada algoritmo para ordenar valores?

En i se envían n + 1 mensajes.
En ii se envían 2 (n-1).
En iii en el mejor de los casos, los procesos necesitan intercambiar solo un par de valores. Esto ocurrirá si los n/N valores menores están inicialmente en P1, y los n/N mayores en PN. En el peor caso (que ocurrirá si todo valor está inicialmente en el proceso equivocado) los procesos tendrán que intercambiar n/N + 1 valores: n/N para tener cada valor en el proceso correcto y uno más para detectar terminación.

c. En cada caso, cuáles mensajes pueden ser enviados en paralelo (asumiendo que existe el hardware apropiado) y cuáles son enviados secuencialmente?

En el caso de i, pueden ser enviados secuencialmente.
Para ii los mensajes pueden ser enviados en paralelo.
En iii asumimos que ponemos los proceso en una secuencia lineal desde P[1] hasta P[k] (con k procesos) y que cada proceso primero ordena sus n/k valores. Luego podemos ordenar los n elementos usando aplicaciones paralelas repetidas del algoritmo de dos procesos compare-and-exchange.

d. Cuál es el tiempo total de ejecución de cada algoritmo? Asuma que cada operación de comparación o de envío de mensaje toma una unidad de tiempo.

Para i el tiempo de ejecución sería 2n+1 y para ii en cada porción hace n/2 comparaciones y el ancho de la red es log2n entonces el tiempo sería (n/2*log2n)+2n-1.

34) Considere el siguiente código:

```
Process Criba[1] {
        INT p=2;
        for [i = 3
to n by 2] Criba[2]
! i
}
```

```
Process Criba[i = 2 TO L] {
        INT p, proximo;
        Criba[i-1] ? p
        do Criba[i-1] ? proximo →
                if (proximo MOD p) <> 0
        → Criba[i+1] ! proximo; fi
        od
}
```

a. Que hace el programa?

El programa es un algoritmo clásico para determinar cuáles números entre 2 y N son primos.

b. Por que esta solución termina con todos los procesos bloqueados?

Porque todos los procesos quedan esperando un mensaje de su predecesor.

c. Modifique la solución para que los procesos no terminen bloqueados.

Podemos modificarlo para que termine normalmente usando centinelas como en la red de filtros merge. Es decir agregar al final de la lista de número un centinela que es un valor especial que indica que todos los números fueron leídos.

```
Process Criba[1] {
    INT p=2;
    for [i = 3 to n by 2] Criba[2] ! i # pasa impares a Criba[2]
    Criba[2] ! –1;
}
```

```
Process Criba[i = 2 TO L] {
    INT p, proximo;
    boolean seguir = true;
    Criba[i-1] ? p              # p es primo
    do (seguir); Criba[i-1] ? proximo -> # recibe próximo candidato
        if (proximo = -1) {
                seguir = false;
                Criba[i+1] ! -1;
        }
        else if ((proximo MOD p) <> 0)   # si es primo
                Criba[i+1] ! proximo;     # entonces lo pasa
        fi
    od
}
```

35) Defina el problema general de alocación de recursos y su resolución mediante una política SJN. Minimiza el tiempo promedio de espera? Es fair? Si no lo es, plantee una alternativa que lo sea.

El problema de la alocación de recursos consiste en decidir cuando se le puede dar a un proceso acceso a un recurso. Un recurso es cualquier cosa por la que un proceso podría ser demorado esperando adquirirlo. Esto incluye entrada a una SC, acceso a una BD, una región de memoria, uso de una impresora.

La resolución mediante SJN teniendo en cuenta que existe solo una unidad del recurso compartido consiste en ejecutar request(time, id) cuando un proceso requiere el uso del recurso, donde time es un entero que especifica cuánto va a usar el recurso el proceso e id es un entero que identifica al proceso que pide. Cuando un proceso ejecuta request, si el recurso esta libre es inmediatamente alocado al proceso; sino el proceso se demora. Después de usar el recurso, un proceso lo libera ejecutando release(). Cuando el recurso es liberado, se aloca para el proceso demorado (si lo hay) que tiene el mínimo valor de time. Si 2 o más procesos tienen el mismo valor de time, el recurso es alocado al que ha esperado más.
La política SJN minimiza el tiempo promedio de ejecución.
No es fair, es unfair porque un proceso puede ser demorado para siempre si hay un flujo continuo de request especificando tiempos de uso menores.
La política SJN puede ser modificada para que sea fair de modo que un proceso que ha estado demorado un largo tiempo tenga preferencia; esta técnica se llama 'aging'.

36) Resuelva el problema de encontrar la topología de una red utilizando mensajes asincrónicos. Muestre con un ejemplo la evolución de la matriz de adyacencia para una red con al menos 7 nodos y de diámetro al menos 4. Compare conceptualmente con una solución utilizando PMS.

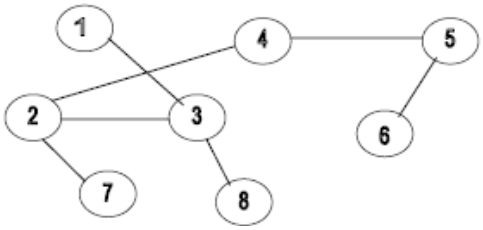
```
chan topologia[1:n](emisor : int; listo : bool; top : [1:n,1:n] bool)

Process Nodo[p:1..n] {
        bool vecinos[1:n];          # inicialmente vecinos[q] true si q es vecino de Nodo[p]
        bool activo[1:n] = vecinos # vecinos aún activos
        bool top[1:n,1:n] = ([n*n]false) # vecinos conocidos
        bool nuevatop[1:n,1:n];
        int r = 0; bool listo = false;
        int emisor; bool qlisto;
        top[p,1..n] = vecinos; # llena la fila para los vecinos
        while (not listo) {
                # envía conocimiento local de la topología a sus vecinos
                for [q = 1 to n st activo[q] ] send topologia[q](p,false,top);
```

```
                # recibe las topologías y hace or con su top
                for [q = 1 to n st activo[q] ] {
                        receive topologia[p](emisor,qlisto,nuevatop);
                        top = top or nuevatop;
                        if (qlisto) activo[emisor] = false;
                }
                if (todas las filas de top tiene 1 entry true) listo=true;
                r := r + 1
        }

        # envía topología a todos sus vecinos aún activos
        for [q = 1 to n st activo[q] ] send topologia[q](p,listo,top);
        # recibe un mensaje de cada uno para limpiar el canal
        for [q=1 to n st activo[q]] receive topologia[p](emisor,d,nuevatop)
}
```

Representamos cada nodo por un proceso Nodo[p:1..n].Dentro de cada proceso, podemos representar los vecinos de un nodo por un vector booleano vecinos[1:n]con el elemento vecinos[q] true en Nodo[p]si q es vecino de p. Estos vectores se asumen inicializados con los valores apropiados. La topología final top puede ser representada por una matriz de adyacencia, con top:[1:n,1:n]true si p y q son nodos vecinos.
Después de r rondas, el nodo p conocerá la topología a distancia r de él. En particular, para cada nodo q dentro de la distancia r de p, los vecinos de q estarán almacenados en la fila q de top. Dado que la red es conectada, cada nodo tiene al menos un vecino. Así, el nodo p ejecutó las suficientes rondas para conocer la topología tan pronto como cada fila de top tiene algún valor true. En ese punto, p necesita ejecutar una última ronda en la cual intercambia la topología con sus vecinos; luego p puede terminar. Esta última ronda es necesaria pues p habrá recibido nueva información en la ronda previa. También evita dejar mensajes no procesados en los canales. Dado que un nodo podría terminar una ronda antes (o después) que un vecino, cada nodo también necesita decirles a sus vecinos cuando termina. Para evitar deadlock, en la última ronda un nodo debería intercambiar mensajes solo con sus vecinos que no terminaron en la ronda previa.



Inicialmente en el nodo 3:

	1	2	3	4	5	6	7	8
1								
2								
3	T	T	T					T
4								
5								
6								
7								
8								

Después de una ronda, en nodo 3:

	1	2	3	4	5	6	7	8
1	T		T					
2		T	T	T			T	
3	T	T	T					T
4								
5								
6								
7								
8			T					T

Después de dos rondas, en nodo 3:

	1	2	3	4	5	6	7	8
1	T		T					
2		T	T	T			T	
3	T	T	T					T
4		T		T	T			
5								
6								
7		T					T	
8			T					T

El proceso del broadcast es un proceso asincrónico, resolverlo con PMS causa que se reduzca la eficiencia.

a. Que mecanismo de pasaje de mensaje es más adecuado para la resolución? Justifique claramente.

El mecanismo de pasaje de mensajes asincrónico es el más adecuado para la resolución junto con algoritmos heartbeat ya que nos permite una interacción entre procesos de manera que cada procesador es modelizado por un proceso y los links de comunicación con canales compartidos. En particular, cada proceso ejecuta una secuencia de iteraciones. En cada iteración, un proceso envía su conocimiento local de la topología a todos sus vecinos, luego recibe la información de ellos y la combina con la suya. La computación termina cuando todos los procesos aprendieron la topología de la red entera. El criterio de terminación no siempre puede ser decidido localmente.

37) Sea el problema en el cual N procesos poseen inicialmente cada uno un valor V, y el objetivo es que todos conozcan cual es el máximo y cuál es el mínimo de todos los valores.

a. Esquematice posibles soluciones con los siguientes arquitecturas de red: centralizada, simétrica (o totalmente conectada) y anillo circular.

En la solución *centralizada* se tiene un único proceso coordinador que recibe la entrada desde un canal, emplea uno de los algoritmos de ordenación y luego escribe el resultado en otro canal.

```
Process P[0] { #Proceso coordinador. v ya está inicializado.
        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] (minimo, maximo);
}
```

```
Process P[i=1 to n-1] { # Proceso cliente. v ya está inicializado.
        INT v; INT minimo, máximo;
        send valores (v);
        receive resultados [i] (minimo, maximo);
}
```

En la solución *simétrica* 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.

```
Chan valores[n] (INT);
Process P[i=0 to n-1] { # Todos los procesos idénticos
        INT v; # asumimos que v fue inicializado
        INT nuevo, minimo = v, máximo=v; # estado inicial,
        FOR [k=0 to n-1 st k <> i ] # envío del dato local
                send valores[k] (v);
        FOR [k=0 to n-1 st k <> i ] { # recibo y proceso los datos remotos
                receive valores[k] (nuevo);
                IF (nuevo < minimo)
                minimo = nuevo;
                IF (nuevo > maximo)
                maximo = nuevo;
        }
}
```

En la solución de *anillo circular* se tiene 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 1ra c/ proceso recibe dos valores y los compara con su valor local, trasmitiendo un máximo local y un mínimo local a su sucesor. En la 2da etapa todos deben recibir la circulación del máximo y el mínimo global.

```
chan valores[n] (INT minimo, INT maximo);
Process P[0] { # Proceso que inicia los intercambios.
        INT v; INT minimo = v, máximo=v;
        # Enviar v a P[1]
        send valores[1] (minimo, maximo);
        # Recibir los valores minimo y maximo globales de P[n-1] y
pasarlos
        receive valores[0] (minimo, maximo);
        send valores[1] (minimo, maximo);
}

Process P[i = 1 to n-1] { # Procesos del anillo.
        INT v; INT minimo, máximo;
        # Recibe los valores minimo y maximo hasta P[i-1]
        receive valores[i] (minimo, maximo);
        IF (v < minimo) minimo = v;
        IF (v > maximo) maximo = v;
        # Enviar el minimo y maximo al proceso i+1
        send valores[i+1 MOD n] (minimo, maximo);
```

```
                    # Esperar el minimo y maximo global
                    receive valores[i] (minimo, maximo);
                    IF (i < n-1) send valores[i+1] (minimo, maximo);
}
```

b. Analice las soluciones desde el punto de vista del número de mensajes y la performance global del sistema.

En la solución *centralizada* se utilizan 2(n-1) mensajes, n para recibir todos los valores y n más para enviar el resultado. Tiene distintos patrones de comunicación que llevan a distinta performance. Los msgs al coordinador se envían casi al mismo tiempo. Sólo el 1er receive del coordinador demora mucho.

En la solución *simétrica* se usan n(n – 1) mensajes con SPMD, n con broadcast. Se pueden ir ejecutando procesos en paralelo. A medida que obtienen sus dos valores de entrada los procedimientos pueden comenzar a ejecutar. 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.

En la solución *anillo circular* tiene un número lineal de mensajes 2(n-1). Todos los procesos son productores y consumidores. El último tiene que esperar a que todos los otros (uno por vez) reciban un msg, hacer poco cómputo, y enviar su resultado. Los msg circulan 2 veces completas por el anillo por lo tanto es una solución inherentemente lineal y lenta para este problema, pero puede funcionar si cada proceso tiene mucho cómputo.

38) **Resuelva con semáforos el problema del oso, las abejas y el tarro de miel: "Hay n abejas y un oso hambriento que comparten un tarro de miel. El tarro inicialmente está vacío, y tiene una capacidad de H porciones de miel. El oso duerme hasta que el tarro está lleno, luego come toda la miel y se vuelve a dormir". Cada abeja repetidamente produce una porción de miel y la pone en el tarro; el que llena el tarro despierta el oso.**

```
sem lleno=0;
sem vacio[1..N]=1;
sem abejaActual=1;
sem abejasDormidas=0;
type tarroMiel[H];
int cantPorciones=0;
process productor[i:1..N]{
        while(true){
                <Produce miel>;
                P(vacio[i]);
                P(abejaActual)
                cantPorciones++;
                tarroMiel[cantPorciones]=miel;
                if(cantPorciones==H){
                        V(abejaActual);
                        for(i=1;i<=N-1;x++)
                                V(abejasDormidas[i]);
                        V(lleno);
                }else{
                        V(abejaActual);
                        P(abejasDormidas[i]);}
        }
}
```

```
process consumidor{
        int x,y;
        while(true){
                P(lleno);
                for(x=1;x<=H;x++)
                        tarroMiel[x]=miel;
                        <Consume miel>
                cantPorciones=0;
                for(y=1;y<=N;y++)
                        V(vacio[y]);
        }
}
```

39) Resuelva con monitores el siguiente problema:
Tres clases de procesos comparten el acceso a una lista enlazada: searchers, inserters, deleters. Los searchers sólo examinan la lista, y por lo tanto pueden ejecutar concurrentemente unos con otros. Los inserters agregan nuevos ítems al final de la lista; las inserciones deben ser mutuamente exclusivas para evitar insertar dos ítems casi al mismo tiempo. Sin embargo, un insert puede hacerse en paralelo con uno o más searches. Por último, los deleters remueven ítems de cualquier lugar de la lista. A lo sumo un deleter puede acceder la lista a la vez, y el borrado también debe ser mutuamente exclusivo con searches e inserciones.

```
monitor lista
 cantD,cantS,cantI:integer:=0;
 deleters,inserters,searchers:cond;

 procedure accesoSearcher() {
  while (cantD>0)
    wait(searchers);
  cantS:=cantS+1;
 }

 procedure accesoInserter() {
  while ((cantI>0) or (cantD>0)
    wait(inserters);
  cantI:=cantI+1;
 }

 procedure accesoDeleter() {
     while ((cantI>0) or (cantD>0) or
(cantS>0))
    wait(deleters);
  cantD:=cantD+1;
 }

 procedure liberarSearcher() {
  cantS:=cantS-1;
     if((cantS=0) and (cantI=0))
                signal(deleters);
 }

 procedure liberarInserter() {
  cantI:=cantI-1;
     signal(deleters);
     signal(inserters);
 }

 procedure liberarDeleter() {
  cantD:=cantD-1;
     signal_all(searchers);
     signal(deleters);
     signal(inserters);
 }

process Searchers[i=1..S] {
 lista.accesoSearcher();
 <Realiza búsqueda en la lista>
 lista.liberarSearcher();
}

process Inserters[j=1..I] {
 lista.accesoInserter();
 <Inserta en la lista>
 lista.liberarInserter();
}

process Deleters[k=1..D] {
 lista.accesoDeleter();
 <Borra en la lista>
 lista.liberarDeleter();
```

40) Defina el concepto de granularidad. ¿Qué relación existe entre la granularidad de programas y de procesos?

Cuando el número de procesadores crece, normalmente la cantidad de procesamiento en c/u disminuye y las comunicaciones aumentan. Esta relación se conoce como granularidad.
Puede definirse la granularidad de una aplicación o una maquina paralela como la relación entre la cantidad mínima o promedio de operaciones aritmético-lógicas con respecto a la cantidad mínima o promedio de datos que se comunican.
La relación computo/comunicación impacta en la complejidad de los procesadores: a medida que son más independientes y realizan más operaciones A-L entre comunicaciones, también deben ser más complejos.
Si la granularidad del algoritmo es diferente a la de la arquitectura, normalmente se tendrá perdida de rendimiento.

41) Utilice la técnica "passing the condition" para implementar un semáforo fair usando monitores.

```
Monitor SemaforoFIFO
    s:integer:=0;
    pos:cond;
procedure Psem(){
    if(s=0)
      wait(pos);
    else
      s=s-1;
}
Procedure Vsem(){
    If(empty(pos))
      s=s+1;
    Else
      signal(pos);
}
```

42) Defina el concepto de no determinismo. Ejemplifique

Los programas concurrentes son No Determinísticos: no se puede determinar que para los mismos datos de entrada se ejecute la misma secuencia de instrucciones, tampoco se puede determinar si dará la misma salida.
Sólo se puede asegurar un Orden Parcial, esto quiere decir que si tenemos dos procesos concurrentes "**p**" y "**q**" y estos fueron divididos en tres instrucciones cada uno, podemos asegurar que la instrucción $\mathbf{p}_i$ se ejecutará antes que la instrucción $\mathbf{p}_j$ si es que **i** es menor a **j**, y lo mismo para el proceso **q**. Ejemplo x=0 //P y x=x+1 //Q. Escenario 1: P→Q → x=1, Escenario 2: Q→P → x=0.

43) En qué consiste la sincronización barrier? Mencione alguna de las soluciones posibles usando variables compartidas

Consiste en un punto de demora al final de cada iteración que hace de barrera y a la cual deben llegar todos los procesos antes de permitirles continuar.
Ejemplo:

```
Int cantidad=0;
Process Worker[i=1 to n]{
        While(true){
            #Código para implementar la tarea i;
            <cantidad=cantidad+1> #Puede implementarse con un FA
            <await(cantidad==n);> #Puede implementarse como while(cantidad!=n) skip;
        }
}
```

Otra solución consiste en distribuir la variable cantidad usando n variables (arreglo arribo[1..n]). El await pasaría a ser <await(arribo[1]+…+arribo[n] == n);>

44) Que se entiende por arquitectura de grano grueso? Es más adecuada para programas con mucha o poca comunicación?

Cuando hablamos de arquitectura de grano grueso decimos, que se tratan de pocos procesadores muy poderosos (que realizan mucho cálculo). Dicha arquitectura es más adecuada para programas con poca comunicación.

45) Que significa el problema de "interferencia" en programación concurrente? Como puede evitarse?

Se llama interferencia cuando el interleaving de la ejecución de un programa concurrente, hace que las acciones de un proceso afecten a los resultados que genere otro proceso del programa concurrente.
Se puede evitar utilizando mecanismos de sincronización, (exclusión mutua o por condición) y también asegurando propiedades como la de ASV (a lo sumo una vez).

46) Dado el siguiente programa concurrente con variables compartidas:

```
x=4;
y=2;
z=3;
co
        x=y*z // z=z*2 // y=y+2x
oc
```

a) Cuáles de las asignaciones dentro del co cumple la propiedad de ASV. Justifique

Las asignaciones (1) "x = y * z" y (3) "y = y +2x" no cumplen ASV. La asignación (2) "z = z*2" sí cumple con la propiedad. Esto es así porque en (1), la expresión de asignación tiene una referencia crítica ("y" es modificada en otro proceso), y a la vez "x" es leída por otro proceso. En (3) ocurre algo similar, "x" es referencia crítica, e "y" está siendo leída en (1). En cambio en (2), la expresión de asignación no tiene referencias críticas ("z" no es modificada en otro proceso) por lo tanto, la variable asignada "z" puede ser leída por otro proceso, y de hecho, es leída por (1). Por ende, "z = z * 2" cumple ASV.

b) Indique los resultados posibles de ejecución.

Secuencial

```
x=6;
z=6;
y=14;
```

Como ninguna de las instrucciones son atómicas podríamos decir que las tres asignaciones cuentan con varias acciones menores, entonces podría suceder que mientras se realiza la asignación (3), la variable x en ese punto tiene el valor 4 mientras la asignación (1) está tomando el valor de y=2 y no el valor de y=10, que es cuando termino de hacer la asignación (3) y suponiendo que (2) si se ejecuto previa a estas 2 últimas asignaciones, los valores serían los siguientes, para el caso (b) la asignación (1) tomo los primeros valores de z e y:

a) x=12;
z=6;
y=10;

b) x=6;
z=6;
y=10;

47) Dado los siguientes procesos

P1:

```
chan canal(double)
process Genera{
        int fila,col;
        double sum;
        for[fila=1 to 10000]
                for[col=1 to 10000]
                        send canal(a(fila,col));
        send canal(EOS);
}

process Acumula{
        double valor,sumT;
        sumT=0;
        receive canal(valor);
        while(valor<>EOS){
                sumT=sumT+valor;
                receive canal(valor);
        }
        printf(sumT);
}
```

P2:

```
chan canal(double)
process Genera{
        int fila,col;
        double sum;
        for[fila=1 to 10000]{
                sum=0;
                for[col=1 to 10000]
                        sum=sum+a(fila,col);
                send canal(sum);
        }
        send canal(EOS);
}

process Acumula{
        double valor,sumT;
        sumT=0;
        receive canal(valor);
        while(valor<>EOS){
                sumT=sumT+valor;
                receive canal(valor);
        }
        printf(sumT);
}
```

a) Que hacen los programas?

El primer programa acumula la suma de todas las componentes de una matriz de 10000x10000 y luego imprime el resultado.
El segundo programa realiza la sumatoria de todas las componentes de cada columna de la matriz y luego las acumula e imprime el total.

b) Analice desde el punto de vista del numero de mensajes

La cantidad de mensajes del primer programa es de (n*n)+1. Siendo n la cantidad de filas y columnas de la matriz, es decir, n=10000.
Por lo tanto la cantidad de mensajes enviados serían 10000*10000+1, este último mensaje es un centinela para avisar que se han terminado de procesar todos los componentes de la matriz.
Para el segundo programa la cantidad de mensajes es de n+1. Ya que realiza la sumatoria de todos los elementos de cada columna y recién envía esa suma por cada fila.
Por lo tanto la cantidad de mensajes enviados serían 10000+1, nuevamente tenemos un mensaje más para el valor final de la secuencia de mensajes.

c) Analice desde el punto de vista de la granularidad de los procesos

Para el primer programa la cantidad de mensajes enviados es considerable, por lo tanto sería más adecuado pensar en una arquitectura donde haya más procesadores con menos potencia en la ejecución de los programas.
Para el segundo programa la cantidad de mensajes es menor y tiene un poco mas de procesamiento al tener que realizar la sumatoria de los elementos de las columnas. Por lo tanto podríamos usar menos procesadores más potentes.

d) Cuál de los programas le parece más adecuado para ejecutar sobre una arquitectura de grano grueso de tipo clúster de PCs? Justifique

El segundo programa sería más adecuado para este tipo de arquitectura, por la cantidad de procesamiento, y la reducción de envío de mensajes con respecto al primer programa.

48) Dado el siguiente bloque de código indique para cada inciso que valor queda en aux, o si el código queda bloqueado. Justifique su respuesta.

```
aux= -1;
...
if(A==0);P2?(aux)->aux=aux+2;
[](A==1);P3?(aux)->aux=aux+5;
[](B==0);P3?(aux)->aux=aux+7;
end if;
...
```

a) Si el valor de A=1 y B=2 antes del if, y solo P2 envía el valor 6.

El único valor de las guardas que coincide es el de A==1 pero queda bloqueado porque la guarda espera un mensaje del proceso P3 y no de P2.

b) Si el valor de A=0 y B=2 antes del if, y solo P2 envía el valor 8.

Nuevamente el único valor de las guardas que coincide es el de A==0 esta vez como recibe de P2 efectivamente no queda bloqueado y el valor de aux es 10.

c) Si el valor de A=2 y B=0 antes del if, y solo P3 envía el valor de 6.

La única guarda que tiene éxito es B==0 y dicha guarda recibe de P3 por lo tanto no queda bloqueada y el valor de aux es 13.

d) Si el valor de A=2 y B=1 antes del if, y solo P3 envía el valor 9.

El if termina sin efecto ya que ninguna guarda cumple con los valores de A y B.

e) Si el valor de A=1 y B=0 antes del if, y solo P3 envía el valor 14.

En este caso al menos una guarda tiene éxito por lo tanto se elige una de ellas NO DETERMINISTICAMENTE. En el caso que sea B==0, ésta recibe de P3 y el valor aux es 21, caso contrario si entra por la guarda con A==1, ésta recibe de P3 y el valor de aux es 19.

f) Si el valor de A=0 y B=0 antes del if, y P3 envía el valor a 9 y P2 el valor 5.

En este caso al menos una guarda tiene éxito por lo tanto se elige una de ellas NO DETERMINISTICAMENTE. En el caso que sea B==0, ésta recibe de P3 y el valor aux es 16, caso contrario si entra por la guarda con A==0, ésta recibe de P2 y el valor de aux es 7.

49) En los protocolos de acceso a sección crítica vistos en clase, cada proceso ejecuta el mismo algoritmo. Una manera alternativa de resolver el problema es usando un proceso coordinador. En este caso, cuando cada proceso SC[i] quiere entrar a su sección crítica le avisa al coordinador, y espera a que éste le de permiso. Al terminar de ejecutar su sección crítica, el proceso SC[i] le avisa al coordinador. Desarrolle protocolos para los procesos SC[i] y el coordinador usando sólo variables compartidas.

```
int arribo[1:n]=([n]0), continuar[1:n]=([n]0);
process worker[i=1..N]{
        while(true){
                #Codigo para implementar la tarea i;
                arribo[i]=1;
                while(continuar[i]!=1)
                        skip;
                continuar[i]=0;
        }
}
process coordinador{
        while(true){
                for[i=1 to n]{
                        while(arribo[i]!=1)
                                skip;
                        arribo[i]=0;
                }
                for[i=1 to n]
                        continuar[i]=1;}
}
```

50) Dado el siguiente programa concurrente indique cuales valores de K son posibles al finalizar, y describa una secuencia de instrucciones para obtener dicho resultado:

```
process P1{
        fa i=1 to k -> n=n+1 af
}
process P2{
        fa i=1 to k -> n=n+1 af
}
```

i)2K -> VALOR POSIBLE (Si ejecuta todo el ForAll P1 y luego todo el ForAll P2)
ii)2K+2 -> VALOR IMPOSIBLE
iii) K -> VALOR POSIBLE (Si ejecutan al mismo tiempo P1 y P2)
iv) 2 -> VALOR POSIBLE (Si P1 carga la variable, P2 hace ForAll de k-1, luego P1 realiza el incremento de n y luego sucede lo mismo para P2)

51) Dado el siguiente programa concurrente, indique cual es la respuesta correcta (justifique claramente):

```
int a=1,b=0;
co
        <await(b=1)a=0> // while(a=1){b=1; b=0;}
Oc
```

Desde el punto de vista conceptual con una política fuertemente fair, el programa eventualmente termina, porque b se vuelve infinitamente 1. Sin embargo con una política débilmente fair, el programa puede no terminar, porque b es también infinitamente 0. Desafortunadamente, es imposible idear un procesador con una política de scheduling que sea práctica y fuertemente fair. Por ejemplo, RR y el time slicing son prácticos pero no fuertemente fair, porque, en general, los procesos ejecutan en orden impredecible.
Un scheduler multiprocesador que ejecute los procesos en paralelo también es práctico, pero no es fuertemente fair. Esto se debe a que el segundo proceso siempre puede examinar b cuando es 0.

52) Dada la siguiente solución con monitores al problema de alocacion de un recurso con múltiples unidades, transforme la misma solución en una solución utilizando PMA

```
monitor alocador_recurso
        int disponible=MAXUNIDADES;
        set unidades=valores iniciales;
        cond libre;
        procedure adquirir(int d){
                if(disponible==0)
                        wait(libre);
                else
                        disponible=disponible-1;
                remove(unidades,id)
        }
        procedure liberar(int id){
                insert(unidades, id);
                if(empty(libre))
                        disponible=disponible+1;
                else
                        signal(libre);
}
```

Solución con PMA:

```
type op=enum(adquirir,liberar);
chan request(int idCli, op oper, int idUnidad);
chan respuesta[n](int idUnidad);
process alocador{
        int disponible=MAXUNIDADES;
        set unidades=valor inicial disponible;
        queue pendientes;
        while(true){
                receive request(idCli,oper,idUnidad);
                if(oper == adquirir){
                        if(disponible>0){
                                disponible=disponible-1;
                                remove(unidades,idUnidad);
                                send respuesta[idCli](idUnidad);
                        }else
                                insert(pendientes,idCli);
                }else{
                        if(empty(pendientes)){
                                disponible=disponible+1;
                                insert(unidades,idUnidad);
                        }else{
                                remove(pendientes,idCli);
                                send respuesta[idCli](idUnidad);
                        }
                }
        }
}
```

53) Implemente una solución al problema de EM distribuida entre N procesos utilizando un algoritmo de tipo Token Passing con PMA

```
chan token[n](),enter[n](),go[n](),exit[n]();
process helper[i=1..N]{
        while(true){
                receive token[i]();
                if(!(empty(enter[i]))){
                        receive enter[i]();
                        send go[i]();
                        receive exit[i]();
                }
                send token[i+1 MOD n]();
        }
}
process user[i:1..N]{
        while(true){
                send enter[i]();
                receive go[i]();
                SECCION CRÍTICA;
                send exit[i]();
                SECCION NO CRITICA;}
}
```

54) El siguiente código intenta resolver el siguiente problema: en una casa de pastas se realiza la venta de las mismas para comprar los clientes deben respetar el orden de llegada; además se pueden atender 5 personas a la vez.

```
monitor casaDePastas
        esperar:cond;
        cant:integer:=0;

        procedure quieroComprar(){
                if(cant<5)then
                        cant:=cant+1;
                else
                        wait(esperar);
        }
        procedure listoCompra(){
                cant:=cant-1;
                signal(esperar);
        }

process cliente[i:1..N]{
        casaDePastas.quieroComprar();
        delay(x); #Compra las pastas
        casaDePastas.listoCompra();
}
```

El siguiente código no es correcto porque al hacer el signal no se garantiza que el siguiente que está esperando va a poder ingresar efectivamente al monitor. Deberíamos usar Passing the condition.
La solución es:

```
monitor casaDePastas
        esperar:cond;
        cant:integer:=0;

        procedure quieroComprar(){
                if(cant<5)then
                        cant:=cant+1;
                else
                        wait(esperar);
        }
        procedure listoCompra(){
                if(empty(esperar))
                        cant:=cant-1;
                else
                        signal(esperar);
        }

process cliente[i:1..N]{
        casaDePastas.quieroComprar();
        delay(x); #Compra las pastas
        casaDePastas.listoCompra();
}
```

55) Dados los siguientes dos segmentos de código, indicar para cada uno de los items si son equivalentes o no. Justificar cada caso (de ser necesario dar ejemplos).

Segmento 1:

```
...
int cant=1000;
DO (cant<-10); datos?(cant)-> Sentencias1;
[] (cant>10); datos?(cant)-> Sentencias2;
[] (INCOGNITA); datos?(cant)-> Sentencias3;
END DO;
...
```

Segmento 2:

```
...
int cant=1000;
while(true){
IF (cant<-10); datos?(cant)-> Sentencias1;
[] (cant>10); datos?(cant)-> Sentencias2;
[] (INCOGNITA); datos?(cant)-> Sentencias3;
END IF;
}
...
```

a) **INCOGNITA equivale a (cant = 0)** -> Para este caso no son equivalentes ya que si cant ingresa por la guarda cant>10, puede ser que si llegara a valer un numero entre -10 y 0 o entre 0 y 10, no quedaría contemplado ese intervalo por las guardas por lo tanto el DO terminaría ante la falla de todas las guardas mientras el if no tendría efecto pero seguiría ejecutando debido al loop infinito en el que se encuentra.

b) **INCOGNITA equivale a (cant > -100)** -> para este caso si son equivalentes ya que cant>-100 cubre todo el intervalo posible de números, por lo tanto ambos se van a ejecutar, ante cualquier número que tome cant.

c) **INCOGNITA equivale a ((cant>0)or(cant<0))** -> para este caso no son equivalentes ya que si cant adquiere el valor 0 no está contemplado por ninguna guarda

d) **INCOGNITA equivale a ((cant>-10)and(cant<10))** -> para este caso tampoco son equivalentes ya que si cant adquiere el valor 10 o -10 no está contemplado dentro del intervalo con las guardas.

e) **INCOGNITA equivale a ((cant>=-10)and(cant<=10))** -> para este caso si son equivalentes ya que están contemplado cualquier valor que pueda llegar a tomar cant, siempre tendría una guarda para ejecutar.

56) Suponga que la solución a un problema es paralelizada sobre p procesadores de dos maneras diferentes. En un caso, el speedup (S) está regido por la función S=p/3 y en el otro por la función S=p-3. Cuál de las dos soluciones de comportara mas eficientemente al crecer la cantidad de procesadores? justifique claramente.

Suponiendo el uso de 5 procesadores:

Solución 1

S=5/3=1,66

Solución 2

S=5-3=2

Ahora, incrementamos la cantidad de procesadores suponemos 100 procesadores:

Solución 1

S=100/3=33,33

Solución 2

S=100-3=97

Podemos decir, que a medida que p tiende a infinito, para la solución 1 siempre el Speedup será la tercera parte en cambio para la solución 2 el valor "-3" se vuelve despreciable. Por lo tanto la solución 2 es la que se comporta más eficientemente al crecer la cantidad de procesadores.

57) Suponga que el tiempo de ejecución de un algoritmo secuencial es de 8000 unidades de tiempo, de las cuales solo el 90% corresponde a código paralelizable. Cuál es el límite en la mejora que puede obtenerse paralelizando el algoritmo? Justifique.

Si yo tengo 8000 unidades de tiempo de las cuales el 90% de éstas pueden ser paralelizables entonces:

La mejora de la paralelizacion de un algoritmo se mide con el Speedup.

El tiempo total del algoritmo lo podemos dividir en:

T = Tsec + Tpar
T = 800 + 7200

Ahora suponiendo que el tiempo paralelizable (Tpar) que es 7200 podemos reducirlo a 1 si pudiéramos utilizar 7200 procesadores, y el tiempo secuencial (Tsec) que es 800 no puede disminuir de ese número porque se debe ejecutar solo por un procesador, por lo tanto:

Tmejor = 800 + 1 = 801

Y ahora calculamos el límite de mejora que podemos llegar a obtener:

S = T / Tmejor (paralelo)
S = 8000 / 801 -> Aproximadamente 10, por lo tanto, ese es el límite de mejora alcanzable.

58) Sea la siguiente solución propuesta al problema de alocacion SJN:

```
monitor SJN
        bool libre=true;
        cond turno;
        procedure request(int tiempo){
                if(not libre)
                        wait(turno,tiempo);
                libre=false;
        }
        procedure release(){
                libre=true;
                signal(turno);
        }
```

a) Funciona correctamente con disciplina de señalización Signal and Continue?

NO para la disciplina Signal and Continue no funciona, ya que un proceso al ser despertado de la cola de wait va a pelear por el acceso al monitor, en la cola de inicio con el resto de los procesos que recién llegan, si llegan continuamente procesos con tiempo menores al del proceso recién despertado no podrá ejecutar, ya que SJN da prioridad a los procesos con menor tiempo.

b) Funciona correctamente con disciplina de señalización Signal and Wait?

SI para esta disciplina funciona correctamente, porque el próximo proceso que es despertado de la cola wait, tiene el acceso garantizado al monitor, es decir, es el próximo en ejecutar en el monitor.

59) Mencione al menos 4 problemas en los cuales Ud. Entiende que es conveniente el uso de técnicas de programación paralela.

- Procesamiento de imágenes
- Simulación de circulación oceánica.
- Multiplicación de matrices
- Generación de números primos

60) Algunos algoritmos:

Tie-Breaker (costoso y complejo de diseñar)

```
Int ultimo[1:n]=([n]0), in[n]=([n]0);
Process SC[i:1..N]{
        While(true){
           For[j=1 to N]{
              Ultimo[j]=i; In[i]=j;
              For[k=1 to N st k<>i]
                 While((in[k]>=in[i])and(ultimo[j]==i)) skip;
           }
           SECCION CRÍTICA;
           In[i]=0;
           SECCION NO CRÍTICA;
        }
}
```

Ticket (necesita FA y próximo y turno son ilimitados)

```
Int turno[1:n]=([n]0);
Int próximo=1, numero=1;
Process SC[i:1..N]{
        While(true){
            Turno[i]=FA(numero,1); #Fetch & Add
            While(turno[i]<>proximo) skip;
            SECCION CRÍTICA;
            Próximo++;
            SECCION NO CRÍTICA;
        }
}
```

Bakery (Es más complejo, pero es fair y no necesita instrucciones especiales)

```
Int turno[1:n]=([n]0);
Process SC[i:1..N]{
        While(true){
            Turno[i]=max(turno[n])+1;
            For[j=1 to N st j<>i]
                While((turno[j]<>0)and(turno[i]>turno[j])) skip;
            SECCION CRÍTICA;
            Turno[i]=0;
            SECCION NO CRÍTICA;
        }
}
```

Lectores y Escritores con Monitores (Broadcast Signal)

```
Monitor BD
        Lector,escritor:cond;
        Nw,nr:integer:=0;
Procedure leer(){
        While(nw>0) wait(lector);
        Nr:=nr+1;
}
Procedure sale_lector(){
        Nr:=nr-1;
        If(nr=0) signal(escritor);
}
Procedure leer(){
        While(nw>0 or nr>0) wait(escritor);
        Nw:=nw+1;
}
Procedure sale_lector(){
        Nw:=nw-1;
        Signal_all(lector);
        signal(escritor);
}
```

Sleeping Barber (Monitores & Rendezvous)

```
Monitor peluquería
        Peluquero_disponible,cliente_esperando,corte_terminado,corte_pelo:cond;
        Silla_libre,puerta_abierta,peluquero_libre:integer:=0;
Procedure corte_de_pelo(){
        While(peluquero_libre=0) wait(peluquero_disponible);
        Peluquero_libre:=peluquero_libre-1;
        Silla_libre:=silla_libre+1;
        Signal(cliente_esperando);
        While(puerta_abierta=0) wait(corte_pelo);
        Puerta_abierta:=puerta_abierta-1;
        Signal(corte_terminado);
}
Procedure llega_cliente(){
        Peluquero_libre:=peluquero_libre+1;
        Signal(peluquero_disponible);
        While(silla_libre=0) wait(cliente_esperando);
        Silla_libre:=silla_libre-1
}
Procedure corte_terminado(){
        Puerta_abierta:=puerta_abierta+1;
        Signal(corte_pelo);
        While(puerta_abierta>0) wait(corte_terminado);
}

Process peluquero{
        While(true)
                Peluquería.Corte_de_pelo();
}

Process cliente[i:1..C]{
        Peluquería.llega_cliente();
        #Se corta el pelo
        Peluquería.corte_terminado();
}
```

PMA - Cliente / Servidor – File Servers / Continuidad Conversacional

```
Process cliente[i:1..C]{
        Int idFS
        Send abrir("prueba.doc");
        Receive rta_abrir[i](idFS);
        Send acceder[idFS](Leer,argumentos); #Por ejemplo, use la operación Leer
        Receive rta_acceder[i](resultados);
        …
}
```

```
Process FS[i:1..N]{
        String nombre;
        Int idCli;
        Oper op;
        Bool seguir=false;
        While(true){
                Receive abrir(idCli, nombre)
                #Si se puede abrir el archivo
                Send rta_abrir[idCli](i);
                Seguir=true;
                While(seguir){
                        Receive acceder[i](op,argumentos);
                        If(op==Leer)
                                #Realiza la lectura del archivo
                        Else{
                                If(op==Escribir)
                                        #Realiza la escritura del archivo
                                Else{
                                        #Realiza el cierre del archivo
                                        Seguir=false;
                                }
                        }
                        Send rta_acceder[idCli](resultados);
                }
        }
}
```