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. Permite a distintos objetos actuar al mismo tiempo.

Ejemplos de donde se puede encontrar concurrencia:

1. Funcionamiento interno de un Router.
2. Sistema de Gestión de Bases de Datos (DBMS).
3. Sistemas financieros.

Objetivos de los sistemas concurrentes:

1. Ajustar el modelo de arquitectura de hardware y software al problema del mundo real a resolver.
2. Incrementar la performance, mejorando los tiempos de respuesta de los sistemas de procesamiento de datos, a través de un enfoque diferente de la arquitectura física y lógica de las soluciones.

El **paralelismo** se refiere a la existencia de múltiples procesadores ejecutando un algoritmo en forma coordinada y cooperante. Si sólo existe un procesador gestionando multiprogramación, se puede decir que existe pseudo-paralelismo. Al mismo tiempo se requiere que el algoritmo admita una descomposición en múltiples procesos ejecutables en diferentes procesadores (concurrencia).
Hay varios tipos diferentes de paralelismo: nivel de bit, nivel de instrucción, de datos y de tarea (proceso).

El **procesamiento distribuido** se define como la forma en que es posible conectar un conjunto de elementos heterogéneos de procesamiento (por ejemplo PCs) en cierto tipo de red de comunicaciones, para que cooperen entre sí, logrando que una sola tarea de procesamiento de datos pueda ser procesada o ejecutada entre varios elementos de la red.

2) **Defina programa concurrente, programa paralelo y programa distribuido**

Programa concurrente: Es un programa en el que intervienen dos o más procesos secuenciales que cooperan para realizar una tarea. Cada proceso es un programa secuencial que ejecuta una secuencia de sentencias. Los procesos cooperan por comunicación: se comunican usando variables compartidas o pasaje de mensajes.

Concurrencia "**Interleaved**":
- Procesamiento lógicamente simultáneo.
- Ejecución intercalada en un único procesador.
- Pseudo-paralelismo.

Concurrencia "**Simultánea**":
- Procesamiento físicamente simultáneo.
- Requiere un sistema multiprocesador o multicore.
- Paralelismo "full".

Programa paralelo: Es un programa concurrente el cual ejecuta sus procesos simultáneamente sobre múltiples procesadores, sincronizándolos y comunicándolos generalmente a través de memoria compartida.

Programa distribuido: Es un programa que se ejecuta en un sistema distribuido, es decir, en un sistema donde múltiples computadoras se comunican a través de una red e interactúan entre sí para obtener un fin común. Al no haber memoria compartida en los sistemas distribuidos, toda la comunicación se debe hacer mediante envío y recepción de mensajes.

3) Diferencie procesamiento secuencial, concurrente y paralelo

Procesamiento Secuencial

En el **procesamiento secuencial** las acciones se ejecutan una tras otra en un sentido estricto, esto es, una acción posterior no se inicia hasta que se haya completado la anterior.

Cuando se ejecuta un programa secuencial, hay un único thread de control: el contador de programa comienza en la primera acción atómica del proceso y se mueve a través del proceso a medida que las acciones atómicas son ejecutadas.

Procesamiento Concurrente

En el **procesamiento concurrente** las acciones se ejecutan de manera simultánea.

La ejecución de un programa concurrente resulta en múltiples thread de control, uno por cada proceso concurrente.

Ventajas

- No depende de la arquitectura o del nº de elementos de proceso.
- Menos necesidad de compartir el trabajo.
- Menos necesidad de resoluciones de errores.
- Permite resolver problemas que no sería posible con un procesamiento secuencial.

Procesamiento Paralelo

En el **procesamiento paralelo** las acciones se ejecutan de manera **físicamente** simultánea a través de la multiprogramación, en varios procesadores a la vez.

Ventajas

- Reducción del tiempo para completar el trabajo.
- Reducción del esfuerzo individual.

Inconvenientes

- Necesidad de compartir el trabajo.
- Necesidad de compartir los recursos.
- Necesidad de esperas en puntos clave.
- Necesidad de comunicación.
- Necesidad de resolución de posibles errores.

4)

a) **¿Cuál es el objetivo de la programación paralela?**

La programación paralela tiene como objetivo escribir programas concurrentes que resuelvan un problema en menos tiempo que su correspondiente secuencial, reduciendo así el tiempo de ejecución; o resolver problemas más grandes o con mayor precisión en el mismo tiempo.

b) **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.
- Predicción meteorológica.

c) **Defina las métricas de speedup y eficiencia. ¿Cuál es el significado de cada una de ellas (qué miden)? ¿Cuáles son los rangos de valores en cada caso? Ejemplifique**

La métrica de **speedup** mide el costo de ejecución de un algoritmo, es decir, 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. Entonces **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 rango está 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 métrica de **eficiencia** 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 rango está entre 0 y 1. Si ***E*** alcanza el valor 1, entonces el speedup es "perfecto".

d) **¿En qué consiste la "ley de Amdahl"?**

La ley de Amdahl indica el incremento de prestaciones en un sistema como consecuencia de la mejora de una o varias partes del mismo. Esta mejora representada como una mejora del rendimiento, va a depender tanto de la calidad de las mejoras efectuadas como del tiempo que éstas utilicen. De forma abreviada podemos decir que este incremento de prestaciones dará la medida de cómo un ordenador rinde en relación con un rendimiento previo después de efectuar en él una o varias mejoras. El incremento de prestaciones global está limitado intrínsecamente por las operaciones que no están afectadas por esta mejora.

e) **Suponga que el tiempo de ejecución de un algoritmo secuencial es de 10.000 unidades de tiempo, de las cuales el 80% corresponden a código paralelizable. ¿Cuál es el límite en la mejora que puede obtenerse paralelizando el algoritmo?**

Si el tiempo de ejecución de un algoritmo secuencial es de 10.000 unidades y el porcentaje de unidades paralelizables es del 80%, quiere decir que 2.000 unidades (el 20%) deben ejecutarse secuencialmente, por lo tanto el tiempo mínimo esperable es de 2.000 para un procesador. Por esta razón nos conviene utilizar una cantidad de procesadores tal que los mantenga a todos ocupados esa cantidad de tiempo, es decir en este caso conviene tener **5 procesadores** (10.000 / 2.000) porque si tuviéramos más, un procesador estaría trabajando 2.000 unidades de tiempo y los demás terminarían de ejecutar las instrucciones antes y tendrían que esperarlo.

f) **Suponga que el tiempo de ejecución de un algoritmo secuencial es de 8.000 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 el tiempo de ejecución de un algoritmo secuencial es de 8.000 unidades y el porcentaje de unidades paralelizables es del 90%, quiere decir que 800 unidades (el 10%) deben ejecutarse secuencialmente por lo tanto el tiempo mínimo esperable es de 800 para un procesador. Por esta razón nos conviene utilizar una cantidad de procesadores tal que los mantenga a todos ocupados esa cantidad de tiempo, es decir en este caso conviene tener **10 procesadores** (8.000 / 800) porque si tuviéramos más, un procesador estaría trabajando 800 unidades de tiempo y los demás terminarían de ejecutar las instrucciones antes y tendrían que esperarlo.

g) **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 se comportará más eficientemente al crecer la cantidad de procesadores? justifique claramente.**

Claramente se observa que a partir de p = 5, el speedup $S_2 = p - 3$ siempre será mayor que $S_1 = p/3$, por lo tanto lograremos una mayor eficiencia de la solución al problema a medida que p >= 5, observemos los siguientes casos:

i. **p = 4** □ para S_1 □ S_1 = **4**/3 = 1,33 con $\mathbf{E_1}$ = 1,33/4 = **0,332** y para S_2 □ S_2 = **4** - 3 = 1 con $\mathbf{E_2}$ = 1/4 = **0,25**

ii. **p = 5** □ para S_1 □ S_1 = **5**/3 = 1,66 con $\mathbf{E_1}$ = 1,66/5 = **0,332** y para S_2 □ S_2 = **5** - 3 = 2 con $\mathbf{E_2}$ = 2/5 = **0,4**

iii. **p = 6** □ para S_1 □ S_1 = **6**/3 = 2 con $\mathbf{E_1}$ = 2/6 = **0,333** y para S_2 □ S_2 = **6** - 3 = 3 con $\mathbf{E_2}$ = 3/6 = **0,5**

Como se puede apreciar, con p = 4, S_1 es más eficiente; pero con p = 5 y p = 6, es decir p >= 5, S_2 es más eficiente.

Podemos decir entonces, que a medida que p tiende a infinito, para S_1 el speedup siempre será la tercera parte, en cambio para S_2 el valor "-3" se vuelve despreciable. Por lo tanto **S_2 es la que se comporta más eficientemente al crecer la cantidad de procesadores**.

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

Las tres grandes clases de aplicaciones concurrentes que se pueden encontrar son:

Sistemas Multithreaded: Manejan simultáneamente tareas independientes, asignando los procesadores de acuerdo a alguna política (por ejemplo, por tiempos).

Ejemplos de estos sistemas son:

- ✔ Sistemas de ventanas en PCs o WS.
- ✔ Sistemas Operativos time-sharing y multiprocesador.
- ✔ Sistemas de tiempo real.

Sistemas de Cómputo Distribuido: Es una colección de computadores separados físicamente y conectados entre sí por una red de comunicaciones distribuida; sobre las que se ejecutan procesos que se comunican esencialmente por mensajes.
Cada componente del sistema distribuido puede hacer a su vez multithreading.

Ejemplos de estos sistemas son:

- ✔ Servidores de archivos en una red (acceso a datos remotos).
- ✔ Sistemas de Bases de datos distribuidas (bancos, reservas de vuelos).
- ✔ Servidores WEB distribuidos (acceso a datos remotos).
- ✔ Arquitecturas cliente-servidor.

Sistemas de Cómputo Paralelo: Son sistemas que buscan resolver un problema en el menor tiempo (o un problema más grande en el mismo tiempo), usando una arquitectura multiprocesador en la que se pueda distribuir la tarea global en tareas que puedan ejecutarse en distintos procesadores.

Algunos ejemplos:

- ✔ Cálculo científico para modelar y simular sistemas naturales.
- ✔ Procesamiento gráfico y de imágenes, efectos especiales, procesamiento de video, realidad virtual.
- ✔ Problemas combinatorios y de optimización lineal o no lineal. Modelos econométricos.

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

Puede definirse la granularidad de una aplicación o una máquina 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.

Los niveles de granularidad existentes son:

- Paralelismo independiente
  - Aplicaciones o trabajos separados.
  - Sin sincronización entre los procesos.

- Grano grueso
    - Sincronización entre procesos a gran nivel, con poca frecuencia.
    - Se trata como procesos concurrentes que ejecutan en un monoprocesador con multiprogramación.

- Grano medio
    - Necesidades de comunicación e interacción mayores.
    - Los hilos interactúan entre sí y se coordinan con frecuencia.

- Grano fino
    - Aplicaciones altamente paralelas.
    - Área muy especializada y fragmentada, con muchas propuestas diferentes.

En un modelo de granularidad gruesa, el programa se divide en varias partes que precisan poca comunicación entre sí. Los sistemas paralelos formados por procesadores potentes y débilmente interconectados parecen más adecuados para aplicaciones de granularidad gruesa.
Con este tipo de paralelismo existe una sincronización entre procesos pero a un nivel muy bajo.

Por el contrario, en un modelo de granularidad fina, el programa se divide en varias partes que precisan mucha comunicación entre sí y que se coordinan con mucha frecuencia. Aquí, los sistemas paralelos formados por varios procesadores no tan potentes pero fuertemente interconectados parecen ser los más adecuados.

7) **¿Qué 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 trata de una arquitectura de pocos procesadores pero muy potentes (que realizan mucho cálculo). Dicha arquitectura es más adecuada para programas con poca comunicación y mucho cómputo.

8) **¿Qué se entiende por arquitectura de grano fino? ¿Es más adecuada para programas con mucha o poca comunicación?**

Cuando hablamos de arquitectura de grano fino decimos que se trata de una arquitectura de muchos procesadores pero poco potentes (que realizan poco cálculo). Dicha arquitectura es más adecuada para programas con mucha comunicación y poco cómputo.

9) **Describa el concepto de deadlock y qué condiciones deben darse para que ocurra. Ejemplifique**

Deadlock es el bloqueo permanente de un conjunto de procesos o hilos de ejecución en un sistema concurrente que compiten por recursos del sistema o bien se comunican entre ellos.

Las cuatro propiedades necesarias y suficientes para que exista deadlock son:

Recursos reusables serialmente: Existencia de al menos un recurso compartido por los procesos, al cual sólo puede acceder uno simultáneamente.

Adquisición incremental: Los procesos mantienen los recursos que poseen mientras esperan adquirir recursos adicionales.

No-preemption: Una vez que son adquiridos por un proceso, los recursos no pueden quitarse de manera forzada sino que solo son liberados voluntariamente.

Espera cíclica: Existe una cadena circular (ciclo) de procesos tal que cada uno tiene un recurso que su sucesor en el ciclo está esperando adquirir.

En la vida real, un ejemplo puede ser el de dos niños que intentan jugar al arco y flecha, uno toma el arco, el otro la flecha. Ninguno puede jugar hasta que alguno libere lo que tomó.

10) **Defina inanición. Ejemplifique.**

La **inanición** se produce cuando algún proceso dentro del sistema no puede satisfacer sus necesidades de recursos debido a que otros procesos los están utilizando continuamente. Este escenario se presenta mucho en algoritmos de asignación por importancia o de asignación por tamaño. Como en un sistema operativo las tareas nunca dejan de llegar puede haber trabajos muy poco importantes o muy grandes a los cuales no les sea permitido hacer uso de los recursos.

Un ejemplo podría ser el de un proceso de baja importancia que está esperando en memoria para usar el procesador, pero debido a que hay otros trabajos de mayor importancia éste está en espera. Durante un tiempo los trabajos que estaban usando el procesador con anterioridad se retiraron, pero llegaron otros trabajos también de mucha importancia y el primer programa sigue esperando. Esto puede continuar indefinidamente y el proceso puede que nunca llegue a ejecutarse.

11) ¿Qué entiende por no determinismo? ¿Cómo se aplica este concepto a la ejecución concurrente? Ejemplifique

No determinismo es un término que se refiere a situaciones donde un mismo conjunto de datos de entrada puede producir diferentes datos de salida, según el orden de ejecución de los procesos. Esto significa que dos ejecuciones del mismo programa no necesariamente son idénticas.

Justamente los programas concurrentes pueden ser no-determinísticos al variar el orden de ejecución de sus tareas o procesos, en una ejecución de otra.

Ejemplo:

```
process imprime10 {
        for [i = 1 to 10]
                write(i);
}
```

```
process imprime1 [i = 1 to 10] {
        write(i);
}
```

Al ejecutarse concurrentemente una vez el proceso imprime10 y los 10 procesos imprime1 los resultados en pantalla se intercalarán. Si se vuelve a ejecutar una segunda vez, puede darse que los resultados se intercalen de una manera distinta, por lo tanto estaríamos frente a una situación de no determinismo.

12) ¿En qué consiste la comunicación guardada (introducida por CSP) y cuál es su utilidad? Describa cómo es la ejecución de sentencias de alternativa e iteración que contienen comunicaciones guardadas.

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
```

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 y 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).

Sentencias que contienen comunicaciones guardadas (if – de alternativa | do – de iteración):

```
if   B1; comunicación1 □ S1;
     B2; comunicación2 □ S2;
fi
```

```
do   B1; comunicación1 □ S1;
     B2; comunicación2 □ S2;
od
```

a. Primero, se evalúan las expresiones booleanas:
   i. Si todas las guardas fallan, el if termina sin efecto.
   ii. Si al menos una guarda tiene éxito, se elige una de ellas (no determinísticamente).
   iii. Si algunas guardas se bloquean, se espera hasta que alguna de ellas tenga éxito.

b. Segundo, luego de elegir una guarda exitosa, se ejecuta la sentencia de comunicación en la guarda elegida.

c. Tercero, se ejecuta la sentencia Si.

La ejecución del **do** es similar, solo que se repite hasta que todas las guardas fallen.

13) Comunicación 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” cuándo tienen mensajes para leer y cuándo deben trasmitir mensajes.

14)

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

La sincronización hace referencia a la posesión de información de los procesos en un sistema concurrente, para poder coordinar sus actividades y obtener un orden de ejecución correcto, evitando así estados inesperados.

Los procesos se **SINCRONIZAN** por exclusión mutua y por condición.

- **Exclusión mutua:** Asegura que solo un proceso tenga acceso a un recurso compartido (sección critica) en un instante de tiempo.
- **Sincronización por condición:** Permite bloquear la ejecución de un proceso hasta que se cumpla una determinada condición.

Ambas formas de sincronización causan que los procesos se demoren y por lo tanto restringen el conjunto de acciones atómicas que son elegibles para su ejecución.

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

Si, un programa puede utilizar los dos mecanismos simultáneamente, por ejemplo la comunicación entre un proceso productor y uno consumidor es implementado, con frecuencia, usando un buffer compartido. El que envía escribe en un buffer; el que recibe lee del buffer. Se usa exclusión mutua para asegurar que ambos no accedan al buffer al mismo tiempo; y se usa sincronización por condición para asegurar que un mensaje no sea recibido antes de que haya sido enviado y que un mensaje no sea sobrescrito antes de ser recibido.

15)

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

El método de sincronización barrier consiste en un punto de demora al final de cada iteración, a la cual deben llegar todos los procesos antes de permitirles continuar. Es útil para poner un punto de encuentro entre los procesos de un programa concurrente a la espera de una determinada condición que le permita proseguir su curso normal. Dicha condición puede ser para mantener la integridad de los datos o bien para realizar una nueva iteración de los procesos.

Soluciones posibles usando variables compartidas:

1. Contador Compartido.
2. Flags y Coordinadores.
3. Arboles.
4. Barreras Simétricas.
5. Butterfly Barrier.

Por 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;
    }
}
```

b) **¿Qué es una barrera simétrica?**

Una barrera simétrica para *n* procesos se construye a partir de pares de barreras simples para dos procesos. Consiste en que cada proceso tiene un flag que setea cuando arriba a la barrera. Luego espera a que el otro proceso setee su flag y finalmente limpia el flag del otro. Por ejemplo:

```
W[i]::  < await (arribo[i] == 0); >
          arribo[i] = 1;
        < await (arribo[j] == 1); >
          arribo[j] = 0;
```

```
W[j]:: < await (arribo[j] == 0); >
          arribo[j] = 1;
       < await (arribo[i] == 1); >
          arribo[i] = 0;
```

c) **Describa “combining tree barrier” y “butterfly barrier”. Marque ventajas y desventajas en cada caso.**

Combining tree barrier: Los procesos se organizan en forma de árbol y cumplen diferentes roles. Básicamente, los procesos envían el aviso de llegada a la barrera hacia arriba en el árbol y la señal de continuar, cuando todos arribaron, es enviada de arriba hacia abajo. Esto hace que cada proceso deba combinar los resultados de sus hijos y luego se los pase a su padre.

Ventajas:

- Su implementación es sencilla.

Desventajas:

- Los procesos no son simétricos ya que cada uno cumple diferentes roles, por lo que los nodos centrales realizaran más trabajo que los nodos hoja y raíz.

Butterfly barrier: Surge debido a la forma del patrón de interconexión al combinar las barreras para dos procesos, para construir una barrera n procesos. Una butterfly barrier tiene $\log_2 n$ etapas. Cada proceso sincroniza con un proceso distinto en cada etapa. En particular, en la etapa **s** un proceso sincroniza con un proceso a distancia 2^{s-1}. Cuando cada proceso pasó a través de $\log_2 n$ etapas, todos los proceso deben haber arribado a la barrera y por lo tanto todos pueden seguir. Esta técnica sirve si **n** es potencia de 2, cuando no lo es, es más eficiente usar dissemination barrier.

Ventajas:

- La implementación de sus procesos es simétrica ya que todos realizan la misma tarea en cada etapa, que es la de sincronizar con un par a distancia 2^{s-1}.

Desventajas:

- Su implementación es más compleja que la del combining tree barrier.
- Si **n** no es potencia de 2, esta implementación no es eficiente.

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

La resolución de problemas con memoria compartida requiere el uso de exclusión mutua o sincronización por condición para evitar interferencias entre los procesos, mientras que con memoria distribuida la información no es compartida, si no que cada procesador tiene su propia memoria local y para poder compartir información se requiere del intercambio de mensajes, evitando así problemas de inconsistencia. Por lo tanto, resulta más fácil programar con memoria distribuida porque el programador puede olvidarse de la exclusión mutua y muchas veces de la necesidad de sincronización básica entre los procesos, la cual es provista por los mecanismos de pasajes de mensajes.

También 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 cuándo tienen mensajes para leer y cuándo 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.

De todas maneras, la facilidad de programación con uno u otro modelo estará dada por el problema particular a resolver.

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

Con **pasaje de mensajes asincrónico** (PMA) 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** (PMS) 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 PMA. Como contrapartida los casos de falla y recuperación de errores son más fáciles de manejar. Con PMS 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.

En términos de concurrencia, PMA la maximiza con respecto a PMS ya que su *send* es no bloqueante, por lo que un proceso puede seguir realizando tareas después de enviar un mensaje, aunque este no haya sido recibido. Además PMS tiene más riesgo de deadlock que PMA por la semántica de la sentencia *send*, e implica muchas veces implementar procesos asimétrico o utilizar comunicación guardada. Por lo que desde el punto de vista de facilidad de programación es mejor PMA, aunque la sincronización deba ser implementada por el programador en muchos casos. Igualmente la elección de un tipo u otro con respecto a la facilidad dependerá del tipo de problema que se requiera resolver ya que mucho problemas pueden por ejemplo ser resueltos con ambos mecanismos pero uno de ellos se adapta mejor haciendo más fácil la resolución.

18)

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

1- **Paralelismo iterativo**

En el paralelismo iterativo un programa consta de un conjunto de procesos (posiblemente idénticos), cada uno de los cuales itera sobre un subconjunto de los datos de entrada y es, a su vez, independiente del resto.
Este paradigma es adecuado para resolver un único problema (por ejemplo un sistema de ecuaciones), donde cada proceso puede trabajar independientemente y sincronizarse por memoria compartida o pasaje de mensajes.

2- **Paralelismo recursivo**

En el paralelismo recursivo el problema general (programa) puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos. La principal diferencia con el paralelismo iterativo, es que en el paralelismo recursivo se crean subprocesos donde cada uno resuelve problemas más pequeños que su padre.
Ejemplos clásicos son por ejemplo el sorting by merging o el cálculo de raíces en funciones continuas.

3- Productores y consumidores (pipelines o workflows)

En el esquema productor-consumidor se tienen uno o más procesos que generan datos y uno o más procesos que hacen uso de ellos para obtener algún propósito final. Es habitual que estos procesos se organicen en pipes a través de los cuáles fluye la información, así cada proceso en el pipe es un filtro que consume la salida de su proceso predecesor y produce una salida para el proceso siguiente.
Un ejemplo son las llamadas "tuberías" de UNIX, por ejemplo: less archivo | grep hola | grep -v chau | less

4- **Clientes y servidores**

Cliente-servidor es el esquema dominante en las aplicaciones de procesamiento distribuido. Este esquema es adecuado para los casos en los que se necesite proveer un servicio. En tales casos, habrá uno o más procesos que provean el servicio (servidores) y uno o más procesos que utilicen dichos servicios (clientes). Unos y otros pueden ejecutarse en procesadores diferentes
Un ejemplo son los servidores de archivos, de contenido web, de mail.

5- Pares que interactúan (interacting peers)

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 y completar el objetivo.
El esquema permite mayor grado de asincronismo que C/S.
Un ejemplo es Boinc, un software que se utiliza para procesar datos científicos en millones de computadoras del mundo, interconectadas a través de internet.

b) **¿Qué relación encuentra entre el paralelismo recursivo y la estrategia de "dividir y conquistar"? ¿Cómo aplicaría este concepto a un problema de ordenación de un arreglo (por ejemplo usando un algoritmo de tipo "quicksort" o uno de tipo "sorting by merging").?**

La relación entre el paralelismo recursivo y la estrategia de "dividir y conquistar" es que la recursión, en sí misma, se basa en dividir un problema grande en varios problemas más pequeños hasta obtener uno de solución trivial, que es exactamente lo que plantea la estrategia de "dividir y conquistar". Adicionalmente, al dividir un problema en subproblemas (que son a su vez independientes entre sí) cada uno de estos pueden ser resueltos en paralelo por agentes diferentes; aquí se ve el paralelismo.

El problema de ordenar un arreglo se resolvería de forma recursiva, por ejemplo con sorting by merging, dividiendo el arreglo en dos mitades, cada proceso se encargaría de ordenar su mitad, (para ello volvería a invocar a dos subprocesos, de la misma naturaleza que el llamante, que recibirían dos mitades de la porción asignada del arreglo, el caso base es que el arreglo recibido tenga dimensión de 1 en el caso que se retornará el mismo sin realizar ninguna otra acción), luego habría que unificar los resultados parciales, hasta obtener el arreglo completo y ordenado. Para esto usaríamos el comando "CO" ya que cada subproceso debe retornar un resultado para poder avanzar con una instancia de nivel superior.

c) **Analice qué tipos de mecanismos de pasaje de mensajes son más adecuados para resolver problemas del tipo "cliente-servidor", "pares que interactúan", "filtro" y "productor- consumidor". Justificar.**

Tanto **PMA como PMS** son lo suficientemente poderosos para programar las cuatro clases de procesos: cliente/servidor, pares que interactúan, filtros y productor-consumidor. Dado que la información fluye a través de canales en una dirección, **ambos son ideales para programar pares que interactúan, filtros y productor-consumidor**. Dependiendo del tipo de problema particular dependerá si es más adecuado PMA que provee buffering implícito y mayor grado de concurrencia o PMS que provee mayor sincronización.

Sin embargo, el flujo de información en dos direcciones entre clientes y servidores tiene que ser programada con dos intercambios de mensajes explícitos usando dos canales diferentes. Más aún, cada cliente necesita un canal de reply diferente; esto lleva a un gran número de canales. Por esto, para **cliente-servidor** es más adecuado el uso de **RPC** o **Rendezvous,** ya que el problema requiere de un flujo de comunicación bidireccional: el cliente solicita un servicio y el servidor responde a la solicitud, es decir que no solo el cliente le envía información para llevar a cabo la operación sino que el servidor le debe devolver los resultados de dicha ejecución.

19) **Analice en qué aspectos puede influir la arquitectura de soporte en la ejecución de aplicaciones concurrentes / paralelas / distribuidas. Ejemplifique.**

Si bien cualquier arquitectura puede soportar cualquier tipo de aplicación concurrente, cabe mencionar que las aplicaciones paralelas conservarán su ganancia si el hardware es paralelo. Aunque pueda ejecutarse en hardware no paralelo, la mejora será visible si el paralelismo existe también en la arquitectura de la máquina. Por ejemplo, si divido el trabajo de procesamiento en dos procesos iguales y el hardware no es paralelo, el efecto será solamente a nivel del planificador de la CPU, entonces no se reducirá el tiempo de ejecución a la mitad, lo que sería previsible o deseable.

En el caso de las aplicaciones distribuidas, es importante considerar la arquitectura o topología de la red, puesto que en este tipo de aplicaciones es la comunicación tanto o más importante que el procesamiento en sí, en particular, si la aplicación depende en demasía de los mensajes, es deseable que las unidades de procesamiento estén físicamente cerca o, de no ser posible, que el sistema de interconexión sea muy rápido para no ralentizar la velocidad. También es importante destacar la forma de interconexión puesto que si los procesadores están conectados en serie y un mensaje del primer procesador debe atravesar todos los procesadores intermedios para llegar al procesador destino, la eficiencia en términos de velocidad será bastante baja si el procesamiento grueso depende de realizar esta actividad para cada mensaje. Por ejemplo, si tengo máquinas conectadas en red (distribuidas) y el tipo de conexión es estrella, debo notar que un mensaje de la máquina central a cualquiera o viceversa tarda una unidad de tiempo, mientras que desde una máquina periférica a otra periférica tardará dos unidades de tiempo, por tanto debo ser cuidadoso en la distribución del trabajo y tener en cuenta la topología de la red para poder estimar su desempeño.

Otro aspecto importante en las arquitecturas es la granularidad. Las de grano fino son más efectivas en costo para aplicaciones con alta concurrencia (comunicación). Si tienen concurrencia limitada pueden usar eficientemente pocos procesadores, siendo conveniente máquinas de grano grueso.

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

En la arquitectura **MIMD** cada procesador tiene su propio flujo de instrucciones y de datos, por lo tanto cada uno 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.

Por el contrario, la arquitectura **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, es decir, ejecuta la misma instrucción en todos los procesadores.

21) **¿Qué significa el problema de "interferencia" en programación concurrente? ¿Cómo puede evitarse?**

Interferencia: Un proceso toma una acción que invalida las suposiciones hechas por otro proceso. La utilización de recursos comunes a diferentes procesos puede dar lugar a actualizaciones incorrectas en el estado de estos recursos.

Para evitar el problema debe asegurarse que mientras uno de los procesos utiliza un recurso compartido, el resto no pueda acceder al mismo. Esto se logra utilizando mecanismos de sincronización (exclusión mutua o sincronización por condición). Otra manera de evitarlo es asegurando ciertas propiedades como por ejemplo la de "A lo sumo una vez".

22) ¿En qué consiste la propiedad de "A lo sumo una vez" y qué efecto tiene sobre las sentencias de un programa concurrente? De ejemplos de sentencias que cumplan y de sentencias que no cumplan con ASV.

Una sentencia de asignación **x = e** satisface la propiedad de *A lo sumo una vez* si:
(1) **e** contiene a lo sumo una referencia crítica y **x** no es referenciada por otro proceso, o
(2) **e** no contiene referencias críticas, en cuyo caso **x** puede ser leída por otro proceso

Una expresión **e** que no está en una sentencia de asignación satisface la propiedad de *A lo sumo una vez* si no contiene más de una referencia crítica.

Referencia crítica en una expresión ⇒ referencia a una variable que es modificada por otro proceso
Si una sentencia de asignación cumple la propiedad ASV, entonces su ejecución parece atómica, pues la variable compartida será leída o escrita sólo una vez.

Ejemplos:

```
int x = 0, y = 0;
co
        x = x + 1 // y = y + 1
oc;                          No hay referencias críticas en ningún proceso

int x = 0, y = 0;
co
        x = y + 1 // y = y + 1
oc;                          El 1er proceso tiene 1 referencia crítica.
                             El 2do ninguna. Los dos cumplen con ASV.

int x = 0, y = 0;
co
        x = y + 1 // y = x + 1
oc;                          Ninguna asignación satisface ASV
```

Si una expresión o asignación no satisface ASV con frecuencia es necesario ejecutarla atómicamente.

23) **Dado el siguiente programa concurrente:**

```
x = 3; y = 5; z = 2;
co
      x = y - z // z = x * 3 // y = y + 3
        p1          p2              p3
oc
```

a) **Cuáles de las asignaciones dentro de la sentencia co cumplen con ASV?. Justifique claramente.**

x = y - z no cumple con ASV ya que posee 2 referencias críticas (“y” y “z” son modificadas en otros procesos).

z = x * 3 no cumple con ASV ya que, aunque posee una referencia critica (“x” es modificada por otro proceso), “z” es utilizada por otro proceso.

y = y + 3 cumple con ASV ya que no posee referencias críticas (“y” no es modificado por ningún proceso)

b) **Indique los resultados posibles de la ejecución**

1) p1, p2, p3 □ **x = 3, z = 9, y = 8**
2) p1, p3, p2 □ **x = 3, z = 9, y = 8**
3) p2, p1, p3 □ **x = -4, z = 9, y = 8**
4) p2, p3, p1 □ **x =- 1, z = 9, y = 8**

Nota 1: las instrucciones NO SON atómicas.

Nota 2: no es necesario que liste TODOS los resultados pero si los que sean representativos de las diferentes situaciones que pueden darse.

24) **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**

x = y * z no cumple con ASV ya que posee 2 referencias críticas (“y” y “z” son modificadas en otros procesos).

z = z * 2 sí cumple con ASV ya que no posee referencias críticas (“z” no es modificado por ningún proceso)

y = y + 2x no cumple con ASV ya que, aunque posee una referencia critica (“x” es modificada por otro proceso), “y” es utilizada por otro proceso.

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;

25) **Dado el siguiente programa concurrente con memoria compartida, y suponiendo que todas las variables están inicializadas en 0 al empezar el programa, y que las instrucciones no son atómicas. Para cada una de las opciones indique verdadero o falso. En caso de ser verdadero indique el camino de ejecución para llegar a ese valor y de ser falso justifique claramente su respuesta.**

P1::	**P2::**	**P3::**
if (x = 0) then	**if (x > 0) then**	**x := x * 8 + x * 2 + 1;**
y := 4 * x + 2;	**x := x + 1;**	
x := y + 2 + x;		

a) **El valor de x al terminar el programa es 9.**

Verdadero

- Se ejecuta **"if (x = 0) then**"en P1
- Se ejecuta P2 (el valor de **x** queda en 0)
- Se ejecuta P3 (el valor de **x** queda en 1)
- Se ejecuta **"y := 4 * x + 2;"** en P1 (el valor de **y** queda en 6)
- Se ejecuta **"x := y + 2 + x;"** en P1 (el valor de **x** queda en 9).

b) **El valor de x al terminar el programa es 6.**

Verdadero

- Se ejecuta **"if (x = 0) then**"en P1
- Se ejecuta "**y := 4 * x + 2;**" en P1 (el valor de **y** queda en 2)
- Se ejecuta P3 (el valor de **x** queda en 1)
- Se ejecuta P2 (el valor de **x** queda en 2)
- Se ejecuta "**x := y + 2 + x;"** en P1 (el valor de **x** queda en 6).

c) **El valor de x al terminar el programa es 11.**

Falso. La única forma de que el valor de x sea 11 es que al ejecutar P3, x valga 1. Esto no puede darse porque P2 no puede alterar x, porque x es igual a 0 y si P1 modificara el valor de x este seria 4.

d) **Y siempre termina con alguno de los siguientes valores: 10 o 6 o 2 o 0.**

Verdadero.

y = 0:

- Se ejecuta P3
- Se ejecuta P2
- Se ejecuta P1 (el valor de **y** queda en 0).

y = 2:

Siempre que la asignación de **y** se realice antes de que los demás procesos alteren x, el valor final de **y** será 2. Por ejemplo:

- Se ejecuta P1 (el valor de **y** queda en 2)
- Se ejecuta P3
- Se ejecuta P2.

y = 6:

- Se ejecuta **“if (x = 0) then**“en P1
- Se ejecuta P2
- Se ejecuta P3
- Se ejecuta **“y := 4 * x + 2;”** en P1 (el valor de **y** queda en 6)
- Se ejecuta **“x := y + 2 + x;”** en P1

y = 10:

- Se ejecuta **“if (x = 0) then**“en P1
- Se ejecuta P3
- Se ejecuta P2
- Se ejecuta **“y := 4 * x + 2;”** en P1 (el valor de **y** queda en 10)
- Se ejecuta **“x := y + 2 + x;”** en P1

26) Defina la instrucción Test & Set. ¿Cuál es su utilidad para la sincronización?

La instrucción Test & Set es una instrucción empleada para testear y escribir en un lugar en memoria de manera atómica (sin interrupción). Toma dos argumentos booleanos: un *lock* compartido y un código de condición local cc. Como una acción atómica, TS setea *cc* al valor de *lock*, luego setea *lock* a true:

TS(lock,cc): ⟨cc := lock; lock := true ⟩

La instrucción Test & Set se puede utilizar para especificar sincronización a través de la utilización de una variable de bloqueo. Si ambos procesos están tratando de entrar a su sección crítica, solo uno puede tener éxito en ser el primero en setear la variable de bloqueo en true; por lo tanto, solo uno terminará su protocolo de entrada. Cuando se usa una variable de bloqueo de este modo, se lo llama spin lock pues los procesos “dan vueltas” (spin) mientras esperan que se libere el bloqueo.

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

Especificamos sincronización por medio de la sentencia **await:**

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

- **<e>** indica que la expresión **e** debe ser evaluada atómicamente.
- La expresión booleana **B** especifica una condición de demora.
- **S** es una secuencia de sentencias que se garantiza que termina.
- Se garantiza que **B** es **true** cuando comienza la ejecución de **S**.
- Ningún estado interno de **S** es visible para los otros procesos.

Por ejemplo:

< **await** s > 0 ⟶ s := s – 1 >

Se demora hasta que **s** es positiva, y luego la decrementa. El valor de **s** se garantiza que es positivo antes de ser decrementado.

La relación entre la instrucción de grano grueso AWAIT y las instrucciones “Test & Set” o “Fetch & Add” es que estas últimas están presentes en casi todas las máquinas (especialmente multiprocesadores), las cuales pueden usarse para implementar las acciones atómicas condicionales del AWAIT.

Por ejemplo, supongamos el siguiente programa:

```
var lock := false
P1 ::  do true → < await not lock → lock := true >      # protocolo de entrada
                 sección crítica
                 lock := false                          # protocolo de salida
                 sección no crítica
       od

P2 ::  do true → < await not lock → lock := true >      # protocolo de entrada
                 sección crítica
                 lock := false                          # protocolo de salida
                 sección no crítica
       od
```

Usando **Test & Set** podemos implementar la solución coarse-grained anterior:

```
var lock := false
P1 ::  var cc : bool
       do true →TS(lock, cc)                    # protocolo de entrada
                 do cc → TS(lock, cc) od
                 sección crítica
                 lock := false                  # protocolo de salida
                 sección no crítica
       od

P2 ::  var cc : bool
       do true →TS(lock, cc)                    # protocolo de entrada
                 do cc → TS(lock, cc) od
                 sección crítica
                 lock := false                  # protocolo de salida
                 sección no crítica
       od
```

Reemplazamos las acciones atómicas condicionales en la solución coarse-grained por loops que no terminan hasta que **lock** es false, y por lo tanto TS setea **cc** a falso. Si ambos procesos están tratando de entrar a su SC, solo uno puede tener éxito en ser el primero en setear **lock** en true; por lo tanto, solo uno terminará su protocolo de entrada. Cuando se usa una variable de lockeo de este modo, se lo llama spin lock pues los procesos “dan vueltas” (spin) mientras esperan que se libere el lock.

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

Una **propiedad** de un **programa concurrente** es un atributo que es verdadero en cualquiera de las historias de ejecución del mismo.

Toda 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 <u>propiedad de seguridad</u>:

1. **Ausencia de demora innecesaria:** Si un proceso está tratando de entrar a su sección crítica y los otros están ejecutando sus secciones NO críticas o terminaron, el primero no está impedido de entrar a su sección crítica.

2. **Ausencia de interferencia (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 dos o más procesos tratan de entrar a su sección crítica, al menos uno tendrá éxito.

Ejemplos de <u>propiedad de vida</u>:

1. **Terminación:** Asegura que un programa eventualmente terminará.

2. **Eventual Entrada.** Un proceso que está intentando entrar a su sección crítica eventualmente lo hará.

29) ¿Qué es una acción atómica? Diferencie acciones atómicas condicionales de incondicionales

En programación, una acción *atómica* es una que sucede efectivamente de una sola vez. Una acción atómica no puede detenerse en el medio: o sucede completamente, o no sucede en absoluto. Ningún efecto colateral de una acción atómica es visible hasta que la acción está completa.

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.

30) Describa fairness. Relacione dicho concepto con las políticas de scheduling. ¿Por qué las propiedades de vida dependen de la política de scheduling? ¿Cómo aplicaría el concepto de fairness al acceso a una base de datos compartida por n procesos concurrentes?

Fairness (Justicia) hace referencia a que procesos similares deben recibir servicios similares. Dar a un proceso mucho más tiempo de CPU que a otro proceso equivalente no es justo. Por supuesto, diferentes categorías de procesos pueden recibir un trato muy diferente.

La política de scheduling tiene como objetivo, entre otros, el de la ecuanimidad (fairness), es decir, que cada proceso reciba una dosis “justa" de CPU, permitiéndole avanzar.

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, asumiendo que su condición se vuelve true y permanece true hasta que es vista por el proceso que ejecuta la acción atómica condicional.

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.

Las propiedades de vida dependen de la política de scheduling ya que éste se encarga de repartir el tiempo disponible de un microprocesador entre todos los procesos que están disponibles para su ejecución. Así en algunos casos solo se garantizará la eventual entrada si la política del scheduling es fuertemente fair.

El concepto de fairness, en una base de datos, se reflejaría si todos los procesos, dependiendo de su prioridad, eventualmente puedan acceder a la información que precisen de ella.

31) **Describa las propiedades que debe cumplir un protocolo de entrada y salida a una sección crítica.**
¿Cuáles de ellas son propiedades de seguridad y cuáles de vida?
¿En el caso de las propiedades de seguridad, cuál es en cada caso el estado "malo" que se debe evitar?

Las propiedades que debe cumplir un protocolo de entrada y salida a una sección crítica son:

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

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

Ausencia de Demora Innecesaria. Si un proceso está tratando de entrar a su sección crítica y los otros están ejecutando sus secciones NO críticas o terminaron, el primero no está impedido de entrar a su sección crítica.

Eventual Entrada. Un proceso que está intentando entrar a su sección crítica eventualmente lo hará.

La **exclusión mutua,** la **ausencia de deadlock** y la **ausencia de demora innecesaria** son propiedades de seguridad. La **eventual entrada** es una propiedad de vida.

En el caso de las propiedades de seguridad, el estado "malo" en cada caso es:

Exclusión mutua: Dos o más procesos no se excluyen en el acceso a la sección crítica.

Ausencia de Deadlock (Livelock): Dos o más procesos se encuentra en estado de deadlock.

Ausencia de Demora Innecesaria: Un proceso pretende entrar en su sección crítica, y no lo consigue aunque no haya otros procesos en competencia.

32)

a) Explique la semántica de un semáforo.

Un semáforo es una instancia de un tipo de datos abstracto: tiene una representación que es manipulada solo por dos operaciones especiales: **P** y **V**. La operación **V** señala la ocurrencia de un evento; la operación **P** se usa para demorar un proceso hasta que ocurra un evento.

Las definiciones de **P** y **V** son:

P(s): ⟨ **await** s > 0 → s := s - 1 ⟩
V(s): ⟨ s := s + 1 ⟩

Ambas operaciones son acciones atómicas. También son las *únicas* operaciones sobre semáforos, es decir: el valor no puede ser examinado directamente.

Los semáforos así definidos se llaman *semáforos generales*: el valor de *s* puede ser cualquier entero no negativo. Un *semáforo binario* es un semáforo cuyo valor es solo 0 o 1. En particular, un semáforo binario *b* satisface un invariante global más fuerte:

BSEM: $0 \leq b \leq 1$

Así, las operaciones sobre un semáforo binario tienen las siguientes definiciones:

P(b): ⟨ **await** b > 0 ⟶ b := b - 1 ⟩
V(b): ⟨ **await** b < 1 ⟶ b := b + 1 ⟩

b) Indique los posibles valores finales de x en el siguiente programa (justifique claramente su respuesta):

```
int x = 3; sem s1 = 1, s2 = 0;
co
        P(s1); x = x * x ; V(s1);  {tarea 1}
    // P(s2); P(s1); x = x * 3; V(s1): {tarea 2}
    // P(s1); x = x - 2; V(s2); V(s1); {tarea 3}
oc
```

La tarea 2 no puede ejecutarse primera por que el S2 se inicializó en 0. Tampoco pueden ejecutarse las tarea concurrentemente en ningún caso, debido a la existencia de los semáforos. Hay solo 2 semáforos, hay un solo momento en que están los 2 habilitados al mismo tiempo (luego de la ejecución de la tarea 3). En este último caso luego es demandado el S1 por las 2 tareas restantes a ejecutar (1 y 2).
Entonces hay 3 posibilidades de ejecución:

Tarea 1, Tarea 3, Tarea 2: X= (9-2)*3=21
Tarea 3, Tarea 1, Tarea 2: X= ((3-2)*1)*3=3
Tarea 3, Tarea 2, Tarea 1: X= ((3-2)*3)*3=9

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

Los semáforos presentan como inconveniente que son mecanismos de muy bajo nivel, lo que lleva a cometer errores fácilmente que pueden ser difíciles de detectar. Entre sus problemas podemos mencionar:

- **Falta de estructura:** No son estructurados, esto condiciona su utilización a una adecuada estructuración por parte del programador; de forma que al recaer esta labor en el programador existen grandes posibilidades de error. Así, la omisión o la inadecuada ubicación de las operaciones **wait()** y **signal()** provoca efectos no deseables sobre el sistema, como la falta de exclusión o el interbloqueo.
- **Falta de semántica:** Esto es, por un lado, los semáforos utilizan el mismo conjunto de operaciones tanto para conseguir exclusión mutua como sincronización; y, por otro lado, los semáforos no están asociados al recurso o al objetivo que persiguen. Esto conlleva una gran dificultad a la hora de identificar el propósito de los semáforos en el seguimiento de un programa.
- **Dispersión en el código:** El acceso en exclusión mutua a variables compartidas y la sincronización entre procesos y hebras está disperso por el código del programa. Esto conlleva a un costoso mantenimiento, ya que cualquier modificación supone una revisión de todo el programa para localizar las regiones afectadas.
- **Operaciones no restringidas:** Al no estar definidas las operaciones que se pueden realizar sobre los recursos compartidos, una vez accedida la variable compartida, el proceso o hebra puede realizar cualquier operación con ésta. Esto es, no existe un control sobre las operaciones que se pueden realizar sobre las variables compartidas.
- **Uso de variables globales:** El uso de variables globales no permite realizar un diseño modular adecuado de las aplicaciones concurrentes. Así, la reutilización de código y otras propiedades propias del diseño modular no pueden ser empleadas.

34) **Defina monitores. Diferencia monitores y semáforos.**

Un monitor es un conjunto de procedimientos que proporciona el acceso con exclusión mutua a un recurso o conjunto de recursos compartidos por un grupo de procesos. Los procedimientos van encapsulados dentro de un módulo que tiene la propiedad especial de que sólo un proceso puede estar activo cada vez para ejecutar un procedimiento del monitor.

Una variable que se utilice como mecanismo de sincronización en un monitor se conoce como variable de condición. A cada causa diferente por la que un proceso deba esperar se asocia una variable de condición. Sobre ellas sólo se puede actuar con dos procedimientos: **espera** y **señal**. En este caso, cuando un proceso ejecuta una operación de *espera* se suspende y se coloca en una cola asociada a dicha variable de condición. La diferencia con el semáforo radica en que ahora la ejecución de esta operación siempre suspende el proceso que la emite. La suspensión del proceso hace que se libere la posesión del monitor, lo que permite que entre otro proceso. Cuando un proceso ejecuta una operación de *señal* se libera un proceso suspendido en la cola de la variable de condición utilizada. Si no hay ningún proceso suspendido en la cola de la variable de condición invocada la operación *señal* no tiene ningún efecto.

El control de los recursos esta centralizado en el monitor, lo que hace que sea más fácil su mantenimiento, a diferencia de los semáforos que su código queda distribuido en varias partes del programa. A su vez, el monitor provee una mayor protección a las variables de control respecto de los semáforos.

35) Definir exclusión mutua y la diferencia para obtenerla con semáforos y monitores.

La exclusión mutua consiste en que un solo proceso excluye temporalmente a todos los demás para usar un recurso compartido de forma que garantice la integridad del sistema.

La ventaja para la exclusión mutua que presenta un monitor frente a los semáforos u otro mecanismo es que ésta está implícita: la única acción que debe realizar el programador del proceso que usa un recurso es invocar una entrada del monitor. Si el monitor se ha codificado correctamente no puede ser utilizado incorrectamente por un programa de aplicación que desee usar el recurso. Cosa que no ocurre con los semáforos, donde el programador debe proporcionar la correcta secuencia de operaciones espera y señal para no bloquear al sistema.

36)

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) **¿Por qué el problema de los filósofos es de exclusión mutua selectiva?**

El problema de los filósofos es de exclusión mutua selectiva ya que se basa en que 5 filósofos tratan de comer usando dos tenedores, el de su izquierda y el de su derecha; cuando sólo hay 5 tenedores en total, por lo que se tienen procesos (un filósofo) que compiten con un subconjunto de procesos (los filósofos que lo rodean) por recursos (el tenedor a la izquierda y a la derecha de cada filósofo).

e) **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 dejaría de ser de EMS ya que cada filósofo competiría con todos los demás filósofos por los mismos recursos.

f) **¿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 todos los procesos compiten entre sí.

g) **De los problemas de los baños planteados en la teoría, ¿cuáles podría ser de exclusión mutua selectiva? ¿Por qué?**

De los problemas de los baños planteados en la teoría los que podrían ser de exclusión mutua selectiva son:

***Opción 1*:** Un baño único para varones o mujeres (excluyente) **sin límite de usuarios**. Exclusión mutua selectiva porque tanto las mujeres o los hombres pueden seguir entrando después que uno de su clase obtuvo el acceso.

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

La sincronización por busy waiting es ineficiente cuando los procesos son implementados por multiprogramación ya que se mantendrá ocupado un procesador realizando un spinning cuando este podría ser usado por otro proceso. También tiene el problema de que al estar chequeando por una variable compartida que es modificada por otros procesos muchas veces ese valor se encuentra invalidado en la cache lo que requiere que se acceda a memoria; este problema es muy notorio en implementaciones de acceso a SC.
Busy waiting y semáforos son herramientas que sirven para sincronización pero son distintas ya que semáforos evita los problemas que tiene busy waiting. Al bloquearse en un semáforo un proceso no consume tiempo de procesamiento hasta que no tenga posibilidad de ejecutarse, en cuyo caso, es puesto en la cola de listos para poder usar el procesador.

38)

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.**

La técnica consiste en implementar sincronización por condición arbitraria para proveer un orden en la ejecución de los procesos. Consiste en que mientras un proceso está dentro de una sección crítica, mantiene el "baton" (que significa permiso para ejecutar) y 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 sección crítica por primera vez.
Podemos usar semáforos binarios divididos para implementar tanto la exclusión mutua como sincronización por condición. Sea **e** un semáforo binario cuyo valor inicial es 1 que se usa para controlar la entrada a sentencias atómicas y asociamos un semáforo b_j y un contador d_j 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 $<S_i>$: **P**(e)

S_i;
SIGNAL

Para representar <await (B_j) S_i >: **P**(e)
if (not B_j) { d_j := d_j + 1; **V**(e); **P**(b_j); }
S_i;
SIGNAL

En ambos fragmentos, SIGNAL es la siguiente sentencia:

SIGNAL: **if** (B_1 **and** d_1 > 0) { d_1 := d_1 - 1; **V**(b_1) }
[] ...
[] (B_n **and** d_n > 0) { d_n := d_n - 1; **V**(b_n)}
[] **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 entrada "*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 <S_i> como <await (B_j) S_j >.

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);
    }
}
```

b) **¿Cuál es su utilidad en la resolución de problemas mediante semáforos?**

Esta técnica emplea split binary semaphores para proveer exclusión mutua y controlar cuál proceso demorado es el próximo en seguir. Puede usarse para implementar sentencias **await** arbitrarias y así implementar sincronización por condición arbitraria. La técnica también puede usarse para controlar precisamente el orden en el cual los procesos demorados son despertados.

c) **¿En qué consiste la técnica de Passing the Condition y cuál es su utilidad en la resolución de problemas con monitores?**

Esta técnica consiste en evitar que las condiciones sean robadas, para esto, el proceso de señalización necesita pasar la condición directamente al proceso despertado más que hacerla globalmente visible, así los procesos son capaces de completar las operaciones **P** en orden FCFS. Podemos implementar esto como sigue. Primero, cambiamos el cuerpo de *V* a una sentencia alternativa. Si algún proceso está demorado sobre *pos* cuando *V* es llamado, despertamos uno pero *no* incrementamos *s*; en otro caso incrementamos *s*. Segundo, reemplazar el loop de demora en *P* por una sentencia alternativa que chequea la condición *s* > 0. Si *s* es positivo, el proceso decrementa *s* y sigue; en otro caso, el proceso se demora sobre *pos*. Cuando un proceso demorado sobre *pos* es despertado, solo retorna de *P*; no decrementa *s* en este caso para compensar el hecho de que *s* no fue incrementado antes de ejecutar **signal** en la operación *V*. Incorporando estos cambios tenemos el siguiente monitor:

```
monitor Semaphore        # Invariante SEM: s ≥ 0
 var s := 0
 var pos : cond    #  señalizada en V cuando pos no está vacía
 procedure P( )
  if s > 0 → s := s - 1
  • s = 0 → wait(pos)
  fi
 end
 procedure V( )
  if empty(pos) → s := s + 1
```

```
    • not empty(pos) → signal(pos)
    fi
  end
end
```

Aquí la condición asociada con pos cambió para reflejar la implementación diferente. Además, s ya no es la diferencia entre el número de operaciones V y P completadas. Ahora el valor de s es nV-nP-pending, donde nV es el número de operaciones V completadas, nP es el número de operaciones P completadas, y pending es el número de procesos que han sido despertados por la ejecución de signal pero aún no terminaron de ejecutar P. (Esencialmente, pending es una variable auxiliar implícita que es incrementada antes de signal y decrementada después de wait).

d) **¿Qué relación encuentra entre las técnicas "Passing the Baton" y "Passing the Condition"? Indique un ejemplo concreto donde se muestre la relación.**

Tanto la técnica "Passing the Baton" como "Passing the Condition" son utilizadas para asegurar el orden en el que los procesos toman el control del sistema, para poder ejecutar sus acciones, ya sea utilizando semáforos (con la técnica Passing the Baton), donde los procesos demorados son despertados en un orden especifico, o utilizando monitores (con la técnica Passing the Condition), asegurando que un proceso despertado por un signal toma precedencia sobre otros procesos que llaman a un procedure del monitor antes de que el proceso despertado tenga chance de ejecutar.
También en ambas técnicas, es el proceso que deja de utilizar el sistema el encargado de "avisarle" al siguiente proceso (en caso que hubiera) que puede comenzar o seguir con su ejecución.

Un ejemplo donde se muestre la relación puede ser el ejercicio presentado en el punto dos de esta entrega.

Utilizando Passing the Baton para implementar la solución: Hay una guarda, de modo que necesitamos un semáforo de condición y un contador asociado. Sea "*c*" el semáforo que representa la condición de demora del cliente *nc* = 5. Sea "*dc*" el contador asociado. Finalmente, sea "*e*" el semáforo de entrada:

```
var nc := 0;
var e:sem:= 1, c:sem:= 0;   {SPLIT: 0 ≤ (e + c) ≤ 1 }
var dc := 0;  {COUNTERS: dc ≥ 0 }

Cliente [i:1..N] :: do true then
                    P(e)
                    if (nc = 5) then
                                dc := dc + 1;
                                V(e);
                                P(c);
                    fi
                    nc := nc + 1
                    V(e);
                    delay(x) //compra
                    P(e)
                    nc := nc - 1
                    if (dc > 0) then
                                dc := dc - 1;
                                V(c);
                    • (dc = 0) then
                                V(e);
                    fi
          od
```

En el caso de Passing the Condition se hace pasando una condición directamente a un proceso despertado en lugar de hacerla globalmente visible. La resolución es la misma que se encuentra desarrollada en la respuesta del ejercicio dos de esta entrega;

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

```
Procedure comprar
     Begin
      if (cant == 5) then
            WAIT(esperar);
       else
            cant := cant + 1;
End;

   Procedure listoCompra
    Begin
            if ( empty(esperar))
                 cant := cant - 1;
            else
                SIGNAL(esperar);
    End;
End;

Cliente[i:1..N]
   casaDePastas.comprar();
   delay (x) //compra las pastas
   casaDePastas.listoCompra;
```

39) **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);
}
```

40) **Sea la siguiente solución propuesta al problema de alocación 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);
        }
}
```

Alocación Shortest-Job-Next (SJN). Varios procesos compiten por el uso de un único recurso compartido. Un proceso requiere el uso del recurso ejecutando *request*(*time*,*id*), 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 está libre, es inmediatamente alocado al proceso; si no, el proceso se demora. Después de usar el recurso, un proceso lo libera ejecutando *release*( ). Cuando el recurso es liberado, es alocado al proceso demorado (si lo hay) que tiene el mínimo valor de *time*. Si dos o más procesos tienen el mismo valor de *time*, el recurso es alocado al que ha esperado más.

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

NO. La técnica Signal and Continue implica que el proceso que realiza el signal, despierta a un proceso dormido, y continúa su ejecución en el monitor. Ese proceso que se ha despertado compite por el acceso al monitor con otros procesos que quieran ingresar desde afuera del monitor. Por lo tanto este ejercicio no funciona si el monitor utiliza esta política ya que no se garantiza que el proceso que es despertado y tiene el trabajo de menor tiempo sea el que se ejecute dentro del monitor (esto puede ocurrir ya que cuando un proceso se va cambia la variable ocupado a false y otro proceso de afuera podría ganar el acceso al monitor, sin importar la longitud de su trabajo).

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

SI. La técnica signal and wait implica que el proceso que es despertado es quien continúa ejecutando en el monitor. Por lo tanto como los procesos han sido encolados utilizando su tiempo de ejecución, el primer proceso que es despertado es el de menor tiempo.

41) 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 cuándo 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 sección crítica, acceso a una BD, una región de memoria, uso de una impresora, etc.

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 dos 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 utilizando la técnica de “aging”, que consiste en darle preferencia a un proceso que ha estado demorado un largo tiempo.

42) 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)

43) 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 PMA existen dos maneras de obtener el efecto de broadcast:

- Utilizar un único canal a partir del cual todos los procesos pueden recibir el dato, por lo que el emisor deberá enviar tantos mensajes como procesos receptores existan.
- Enviar concurrentemente el mismo mensaje a varios canales, es decir, a cada uno de los procesos receptores.

El primer caso es mejor porque no hay demoras innecesarias, ya que ni bien exista un mensaje en el canal este puede ser tomado por un receptor. En cambio, con la segunda opción un proceso que está listo para recibir el mensaje debe esperar a que el emisor le deposite el mensaje en su canal.

Con PMS 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:

- Usar un proceso separado para simular un canal broadcast; es decir, los clientes enviarían mensajes broadcast y los recibirían de ese proceso.
- 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.

44)

a) 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.
  Un ejemplo de servidores replicados es la resolución descentralizada al problema de los filósofos donde cada proceso mozo interactúa con otros para obtener los tenedores pero esto es transparente a los procesos filósofos.
- **Paradigma de algoritmos Heartbeat:** Cada proceso ejecuta una secuencia de iteraciones. En cada iteración un proceso envía su conocimiento local a todos sus vecinos, luego recibe la información de ellos y la combina con la suya. Por lo tanto en estos algoritmos las acciones de cada nodo primero se expanden enviando información, luego se contrae incorporando nueva información. Ejemplos de problemas que se pueden resolver son computación de grillas (labeling de imágenes) o autómatas celulares (el juego de la vida).
- **Paradigma algoritmos pipeline**: La información recorre una serie de procesos utilizando alguna forma de receive/send.
  Ejemplos son las redes de filtros o tratamiento de imágenes.
- **Paradigma de *probes* y *echoes*:** Se basa en el envío de un mensaje (“probe”) enviado por un nodo a su sucesor y la espera posterior del “echo” que es el mensaje de respuesta. Dado que los procesos ejecutan concurrentemente, los “probes” se envían en paralelo a todos los sucesores. La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información.

Un ejemplo podría ser recuperar la topología activa de una red móvil o hacer broadcast desde un nodo cuando no se alcanzan o ven todos los destinatarios.

- **Paradigma de Broadcast:** Consiste en transmitir un mensaje desde un emisor a muchos receptores. Sirven para toma de decisiones descentralizadas y para resolver problemas de sincronización distribuida.
  Un ejemplo es la sincronización de relojes en un sistema distribuido de tiempo real.
- **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.
  Un ejemplo podría ser el de determinar la terminación de un proceso en una arquitectura distribuida cuando no puede decidirse localmente.
- **Paradigma Manager/Workers**: Representa una implementación distribuida del modelo de Bags of Tasks, que consiste en un proceso controlador de datos y/o procesos y múltiples procesadores que acceden a él para poder obtener datos y/o tareas para ejecutarlos en forma distribuida. Se trata de un esquema Cliente/Servidor.

b) **A su criterio, cuál de ellos resulta más adecuado para resolver problemas de broadcast entre procesadores, cuando no se conoce la topología de la red. Justifique claramente su respuesta.**

El más adecuado es el paradigma de ***probes (send) y echoes(receive)*** ya que permite hacer un broadcast desde un nodo cuando no se "alcanzan" o "ven" directamente todos los destinatarios.

45)

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

RPC (Remote Procedure Call) y **Rendezvous** son técnicas de comunicación y sincronización entre procesos que suponen un canal bidireccional (el llamador se demora hasta que la operación llamada haya sido ejecutada y se devuelven los resultados). Por esto son ideales para programar aplicaciones Cliente-Servidor.

RPC (Remote Procedure Call) es un protocolo que permite a un programa de ordenador ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambos. Algunos de sus objetivos son:

- Proporcionar un middelware que simplifique el desarrollo de aplicaciones distribuidas
- Evitar que programador tenga que interactuar directamente con el interfaz de Sockets
- Abstraer (ocultar) los detalles relativos a la red
- El Servidor ofrece procedimientos que el cliente llama como si fueran procedimientos locales
- Se busca ofrecer un entorno de programación lo mas similar posible a un entorno no distribuido.

El concepto de RPC se puede simplificar del siguiente modo: El proceso llamador envía un mensaje de llamada y espera por la respuesta. En el lado del servidor un proceso permanece dormido a la espera de mensajes de llamada. Cuando llega una llamada, el proceso servidor extrae los parámetros del procedimiento, calcula los resultados y los devuelve en un mensaje de respuesta.

Otros modelos son igualmente posibles; por ejemplo, el llamador puede continuar su ejecución mientras espera una respuesta, o el servidor puede despachar una tarea separada para cada llamada que reciba de modo que quede libre para recibir otros mensajes.

Rendezvous es una primitiva de sincronización asimétrica que permite a dos procesos concurrentes, el solicitante y el llamado, intercambiar datos de forma coordinada. El proceso que *solicita* el rendezvous debe esperar en el punto de reencuentro hasta que el proceso llamado llegue allí. Igualmente el proceso llamado puede llegar al rendezvous antes que el solicitante y debe esperar que él *llegue al punto de encuentro* para poder continuar procesando. La imagen de esperar en el punto de encuentro corresponde a colocar un proceso en espera inactiva hasta que la cita se cumpla. Durante el rendezvous los procesos pueden intercambiar datos. 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.

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. Por esto 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 de tipo 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, podríamos usar Rendezvous o usar PM.

c) **¿Qué elementos de la forma general de rendezvous no se encuentran en el lenguaje ADA?**

ADA no provee la expresión de Scheduling "e_i" la cual se usa para alterar el orden de servicio de invocaciones por default en las guardas.

46) 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).

47) Describa sintéticamente las características de sincronización y comunicación de Linda, OCCAM, Java, Ada y MPI

Linda:

En si Linda no es un es un lenguaje de programación, sino un conjunto de primitivas que operan sobre una memoria compartida donde hay “tuplas nombradas” (taggedt uples) que pueden ser pasivas (datos) o activas (tareas).
El núcleo de Linda es el espacio de tuplas compartido (TS = Tuple Space) que puede verse como un único canal de comunicaciones compartido, pero en el que no existe orden:

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

Si bien hablamos de memoria compartida, el espacio de tuplas puede estar físicamente distribuida en una arquitectura multiprocesador (más complejo) Linda puede usarse para almacenar estructuras de datos distribuidas, y distintos procesos pueden acceder concurrentemente a diferentes elementos de las estructuras.

OCCAM:
OCCAM es un lenguaje simple con una sintaxis rígida. Los procesos y los caminos de comunicación entre ellos son estáticos (cantidades fijas definidas en compilación). 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*** *protocolo*. 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 declarados son globales a los procesos, pero cada canal debe tener exactamente un emisor y un receptor.

Java:
Permite la sincronización por condición mediante los métodos **wait**, **notify** y **notifyall**. Los métodos son definidos en la superclase Object y siempre deben ser ejecutados en porciones de código “*synchronized*”.
Posee exclusión mutua implícita, cada objeto tiene un **lock** y para ganar acceso a este se le hace un bloque protegido por la palabra clave *sinchronized*; para obtenerlo debe esperar a tener acceso exclusivo, que retendrá hasta salir del bloque.
Soporta el uso de RPC (Remote Procedure Call) en programas distribuidos mediante la invocación de métodos remotos (RMI).
Una aplicación que usa RMI tiene 3 componentes:

- Una interface que declara los Readers para métodos remotos.
- Una clase Server que implementa la interface.
- Uno o más clientes que llaman a los métodos remotos.

El Server y los clientes pueden residir en máquinas diferentes.

Ada:
Rendezvous es la técnica de comunicación y sincronización entre procesos, supone un canal bidireccional ideales para programar aplicaciones C/S.
Combina una interfaz “tipo monitor” con operaciones exportadas a través de llamadas externas (*CALL*) con mensajes sincrónicos (demoran al llamador hasta que la operación llamada se termine de ejecutar y se devuelvan los resultados).
Desde el punto de vista de la concurrencia, un programa Ada tiene tasks (tareas) que pueden ejecutar independientemente y contienen primitivas de sincronización.
Los puntos de invocación (entrada) a una tarea se denominan *entrys* y están especificados en la parte visible (*header* de la tarea). Una tarea puede decidir si acepta la comunicación con otro proceso, mediante la primitiva *accept*.
Se puede declarar un *typetask*, y luego crear instancias de procesos (tareas) identificado con dicho tipo (arreglo, puntero, instancia simple).

MPI ("Message Passing Interface", Interfaz de Paso de Mensajes):
Es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores.
Su principal característica es que no precisa de memoria compartida, por lo que es muy importante en la programación de sistemas distribuidos.
Los elementos principales que intervienen en el paso de mensajes son el proceso que envía, el que recibe y el mensaje.
Dependiendo de si el proceso que envía el mensaje espera a que el mensaje sea recibido, se puede hablar de paso de mensajes síncrono o asíncrono. Generalmente empleando este sistema, el proceso que envía mensajes sólo se bloquea o para, cuando finaliza su ejecución, o si el “buzón” al que envía está lleno.

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

Las acciones de comunicación, tanto *send* como *receive*, afectan la ejecución de otros procesos por lo tanto son eventos relevantes dentro de un programa distribuido y por su semántica debe existir un orden entre las acciones de comunicación. Básicamente cuando se envía un mensaje y luego este es recibido existe un orden entre estos eventos ya que el *send* se ejecuta antes que el *receive* por lo que puede imponerse un orden entre los eventos de todos los procesos. Sea un ejemplo el caso en que procesos solicitan acceso a un recurso, muchos de ellos podrían realizar la solicitud casi al mismo tiempo y el servidor podría recibir las solicitudes desordenadas por lo que no sabría quién fue el primero en solicitar el acceso y no tendría forma de responder a los pedidos en orden. Para solucionar este problema se podrían usar los relojes lógicos.
Para establecer un orden entre eventos es que se utilizan los relojes lógicos que se asocian mediante un timestamps a cada evento. Entonces, un reloj lógico es un contador que es incrementado cuando ocurre un evento dentro de un proceso ya que el reloj es local a cada proceso y se actualiza con la información distribuida del tiempo que va obteniendo en la recepción de mensajes. Este reloj lógico (rl) será actualizado de la siguiente forma dependiendo del evento que suceda:
Cuando el proceso realiza un *send*, setea el timestamp del mensaje al valor actual de rl y luego lo incrementa en 1. Cuando el proceso realiza un *receive* con un timestamp (ts), setea rl como max (rl, ts+1) y luego incrementa rl.
Con los relojes lógicos se puede imponer un orden parcial ya que podría haber dos mensajes con el mismo timestamp pero se puede obtener un ordenamiento total si existe una forma de identificar unívocamente a un proceso de forma tal que si ocurre un empate entre los timestamp primero ocurre el que proviene de un proceso con menor identificador.

Un ejemplo puede ser la implementación de semáforos distribuidos.