## PARTE A: Definiciones y preguntas conceptuales que debieran ser contestadas brevemente. (15 puntos)

**1) Defina programa: Concurrente, Paralelo, Distribuido.**

Un **programa concurrente** especifica dos o más “programas secuenciales” que pueden ejecutarse concurrentemente en el tiempo como tareas o procesos. Tiene múltiples *threads* o flujos de control (como los procesos paralelos).

Un ***proceso*** o tarea es un elemento concurrente abstracto que puede ejecutarse simultáneamente con otros procesos o tareas, si el hardware lo permite (por ejemplo los TASKs de ADA).

Un programa concurrente puede tener ***N procesos*** habilitados para ejecutarse concurrentemente y un sistema concurrente puede disponer de ***M procesadores*** cada uno de los cuales puede ejecutar uno o más procesos.

Un **programa paralelo** es un programa concurrente escrito para resolver un problema en menos tiempo que el secuencial. El objetivo principal es reducir el tiempo de ejecución, o resolver problemas más grandes o con más precisión en el mismo tiempo. Ejecución concurrente en múltiples procesadores.

Un **programa distribuido** es un programa concurrente en el cual los procesos se comunican y sincronizan por PM, RPC o Rendezvous.

La **programación distribuida** es un caso especial de la programación concurrente en la que se cuenta con varios procesadores pero no se posee una memoria compartida y la comunicación está dada por el pasaje de mensajes.

**2) Defina comunicación entre procesos. Describa los mecanismos que conozca.**

La comunicación entre procesos concurrentes indica el modo en que se organiza y transmiten datos entre tareas concurrentes. Esta organización requiere especificar ***protocolos*** para controlar el progreso y la corrección. Los procesos se **COMUNICAN** a través de:

* Memoria compartida: Los procesos intercambian información sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella. Lógicamente no pueden operar simultáneamente sobre la memoria compartida, lo que obliga a **bloquear y liberar** el acceso a la memoria. La solución más elemental es 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 cuando deben transmitir mensajes.

**3) Defina sincronización entre procesos. Describa los mecanismos que conozca.**

Es la interacción entre procesos que permite  
- Controlar el orden en que se ejecutan y   
- Coordinar actividades

Restringiendo las historias posibles sólo a las deseables o permitidas.

Es la posesión de información acerca de otro proceso para coordinar actividades. En la mayoría de los sistemas el tiempo absoluto no es importante. Con frecuencia los sistemas son actualizados con componentes más rápidos. La corrección del programa concurrente no debe depender del tiempo absoluto.

Puede haber distintos órdenes (**interleavings**) en que se ejecutan las instrucciones de los diferentes procesos: los programas deben ser correctos para todos ellos.

Mecanismos de sincronización de procesos:

* Sincronización por exclusión mutua: asegura que solo un proceso tenga acceso a un recurso compartido en un instante de tiempo. Si el proceso tiene **secciones críticas** que pueden compartir más de un recurso, la exclusión mutua evita que dos o más procesos puedan encontrarse en la misma sección crítica al mismo tiempo.
* Sincronización por condición: permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada.

**4) ¿Qué significa el problema de la interferencia? Cómo puede evitarse?**

Una interferencia se da cuando un proceso toma una acción que invalida las suposiciones hechas por otro proceso. Se puede dar por ejemplo en programas concurrentes con memoria compartida. Para evitarse el problema de la interferencia, los procesos se deben sincronizar.

**5) ¿En qué consiste la propiedad de “a lo sumo una vez”? ¿En qué afecta a un programa?**

Una sentencia de asignación **x = e** satisface la propiedad de “a lo sumo una vez” si:

* **e** contiene a lo sumo una referencia crítica y **x** no es referenciada por otro proceso, ó
* **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.

Si una sentencia de asignación cumple la propiedad de “a lo sumo una vez”, entonces su ejecución parece atómica, pues la variable compartida será leída o escrita sólo una vez.

Una **referencia crítica** en una expresión es una referencia a una variable que es modificada por otro proceso. Asumamos que toda referencia crítica es a una variable simple leída y escrita atómicamente.

Ejemplos

|  |  |  |
| --- | --- | --- |
| *int x=0, y=0;*  *co x=x+1 //*  *y=y+1 oc;* | No hay referencias críticas en ningún proceso. En todas las historias x = 1 e y = 1 | Cumple ASV |
| *int x = 0, y = 0;*  *co x=y+1 //*  *y=y+1 oc;* | El primer proceso tiene una referencia crítica, y x no es referenciada por otro proceso.  El segundo ninguna. Siempre y = 1 y x = 1 o 2 | Cumple ASV |
| *int x = 0, y = 0;*  *co x=y+1 //*  *y=x+1 oc;* | La primer sentencia tiene una referencia crítica y x es referenciada por otro proceso.  Posibles resultados: x= 1 e y =2 / x = 2 e y = 1.  Nunca debería ocurrir x = 1 e y = 1 **→ ERROR** | Ninguna asignación cumple ASV |

**6) ¿Por qué las propiedades de vida dependen de las políticas de scheduling?**

**¿Cuándo una política de scheduling es fuertemente fair?**

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

La propiedad de **vida** (*liveness*) indica que ocurre algo bueno con una actividad: progresa, no hay *deadlocks*. Una *falla de vida* indica que las cosas se dejan de ejecutar.

Esto depende de las políticas de scheduling ya que éstas apuntan a garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los demás.

Una acción atómica en un proceso es **elegible** si es la próxima acción atómica en el proceso que será ejecutada. Si hay varios procesos → hay varias acciones atómicas elegibles.

Una **política de scheduling** determina cuál será la próxima en ejecutarse.

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

**Fairness Débil**. Una política de scheduling es débilmente fair si:

1. Es incondicionalmente fair y
2. 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.

**Fairness Fuerte**: Una política de scheduling es fuertemente fairsi:

1. Es incondicionalmente fair y
2. Toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en ***true*** con infinita frecuencia.

**7) Describa la técnica de passing the condition. Ejemplifique.**

**Passing the condition** permite que un proceso avise a otro que están dadas las condiciones para que continúe y que éste sea el único que pueda continuar la ejecución de los que requieran dicha condición. El proceso despertado se ejecutará antes que el resto de los que requieran la condición (tal vez tomen control del monitor, pero se bloquearán). Ésta técnica sólo puede aplicarse cuando los procesos que ejecutan *wait* y *signal* realizan acciones complementarias entre sí (por ejemplo, uno decrementa un contador y el otro lo incrementa, o uno setea una variable a verdadero y el otro a falso).

Es una técnica general para implementar sentencias *await* arbitrarias y para decidir cuál de los procesos es el próximo en seguir ejecutando (orden en el cual despertarlos) pasandole una condición.

Ejemplo

###### **Monitor MáquinaADN {**

**VAR**

boolean libre = true;

cond [1..100] personas\_esperando;

queue cola;

**BEGIN**

**procedure entrar(int p) {**

if (libre){

libre = false;

} else {

cola.push(p);

**wait(personas\_esperando[p]);**

}

}

**procedure salir() {**

if empty(cola) {

libre = true;

} else {

i = cola.pop();

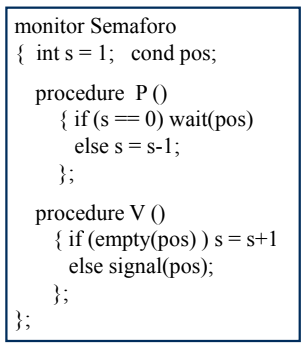
**signal(personas\_esperando[i]);**

}

}

}

Otro ejemplo:



**8) ¿En qué consiste la técnica barrier? Mencione al menos 2 soluciones posibles usando variables compartidas**

La técnica barrier consiste en un punto de encuentro (sincronización) en donde todos los procesos se demoran hasta que todos hayan llegado y luego de eso puedan continuar su ejecución.

* Solución 1: **Contador compartido**. n procesos necesitan encontrarse en una barrera. Cada proceso incrementa una variable compartida “*cantidad”* al llegar. Cuando *cantidad* es n los procesos pueden pasar.
* Solución 2: **Flags y coordinadores**. Puede distribuirse Cantidad usando n variables. Es decir, tener un arreglo de 1 a n. Está solución reintroduce contención de memoria y es ineficiente.

**9) Qué se entiende por arquitectura de grano grueso? ¿Es más adecuado para programas con mucha o poca comunicación?**

La **granularidad** de una aplicación está dada por la relación entre el cómputo y la comunicación.

En una arquitectura de grano grueso se tienen pocos procesadores con mucha capacidad de procesamiento por lo que son más adecuados para programas en los cuales se tienen pocos procesos que realizan mucho procesamiento y requieren de poca comunicación.

Si las aplicaciones tienen concurrencia limitada pueden usar eficientemente pocos procesadores, por lo que convienen máquinas de grano grueso.

En una arquitectura de grano fino se tiene más comunicación, más procesadores, y el cómputo está más distribuido.

**10) ¿Qué significa que un problema sea de exclusión mutua selectiva? ¿El problema de los filósofos tradicional, lo es? Y si hubiera 3?**

Un problema es de **exclusión mutua selectiva** cuando cada proceso compite por el acceso a los recursos no con todos los demás procesos, sino con un subconjunto de ellos.

El problema de los filósofos es de exclusión mutua selectiva ya que para comer un filósofo no compite con todos los demás sino que solo compite con sus adyacentes. En el caso de que fueran 3 filósofos, no sería de exclusión mutua selectiva ya que un proceso compite con todos los demás procesos y no solo con un subconjunto de ellos, dado que el resto de los procesos son sus adyacentes.

**11) ¿En qué consiste la comunicación guardada y cual es su utilidad? Describa cómo son las sentencias alternativas e iterativas con comunicación guardada.**

Generalmente los procesos se quieren comunicar con más de un proceso y no saben el orden en el cual los otros procesos podrían querer comunicarse con él. Las sentencias de comunicación guardada soportan comunicación no determinística.

***Guarda****: (condición booleana); sentencia de recepción → sentencia a realizar*

##### **Alternativa**

* **Simple**

*if B; R → S*

**B** es una expresión booleana, que puede omitirse, y se asume true.

**R** es la sentencia de recepción.

**S** es una instrucción que puede sersimple o compuesta, la cual no se ejecuta si B no es verdadera.

* **Múltiple**

*if B1; R → S1*

*\* B2; R → S2*

*.....*

*\* Bn; R → Sn*

*fi*

Las guardas se evalúan en algún orden arbitrario. La elección se realiza de forma no determinística. Si ninguna guarda es verdadera, el if *no tiene efecto*.

##### **Iteración**

* **Múltiple:**

*do B1 → S1*

*\* B2 → S2*

*.....*

*\* Bn → Sn*

*od*

Las guardas se evalúan y ejecutan hasta que todas las guardas sean falsas. La elección se realiza de forma *no determinística* si más de una guarda es verdadera.

* Sintaxis de la Comunicación guardada:

***Guarda****: (condición booleana); sentencia de recepción → sentencia a realizar*

Si no se especifica la condición booleana se considera **verdadera** (la condición booleana sólo puede hacer referencia a **variables locales al proceso**).

Cada guarda tiene tres posibles estados:

* **Elegible:** la condición booleana es verdadera y la sentencia de comunicación se puede resolver inmediatamente.
* **No elegible**: la condición booleana es falsa.
* **Bloqueada**: la condición booleana es verdadera y la sentencia de comunicación no se puede resolver inmediatamente.

Sólo se puede usar dentro de un **if** o un **do** guardado:

El **IF** funciona de la siguiente manera: de todas las guardas elegibles se selecciona una en forma no determinística, se realiza la sentencia de comunicación correspondiente, y luego las acciones asociadas a esa guarda.

Si todas las guardas tienen el estado de no elegibles, se sale sin hacer nada.

Si no hay ninguna guarda elegible, pero algunas están en estado bloqueado, se queda esperando en el if hasta que alguna se vuelva elegible.

El **DO** funciona de la siguiente manera: sigue iterando de la misma manera que el if hasta que todas las guardas sean no elegibles.

**12) ¿Cuál es el objetivo de la programación paralela? Mencione al menos 4 problemas en los cuáles Ud. entiende que es conveniente el uso de técnicas de programación paralela.**

El objetivo de la programación paralela es reducir el tiempo de ejecución, o resolver problemas más grandes o con más precisión en el mismo tiempo.

Problemas en los que conviene usar técnicas de programación paralela:

- Procesamiento de imágenes.  
- Multiplicación de matrices.  
- Simulación de circulación oceánica.  
- Generación de números primos.  
- Ataques por fuerza bruta.  
- Uso de técnicas criptográficas.

## PARTE B: Interpretación de código (10 puntos)

**13) Dado el siguiente fragmento de programa concurrente con variables compartidas**

int s = 6; t = 4;

co

<s = s + t;> //

<t = s \* t;>

oc

**a) ¿Cuáles son los posibles valores finales de s y t?**

s= 6, t=4

**s=** s+t → **10**

**t=** s\*t → **40**

s= 6, t=4

**t=** s\*t → **24**

**s=** s+t → **30**

**b) Si se eliminan los corchetes angulares y las asignaciones se implementan con 3 acciones atómicas (leer, sumar o multiplicar y escribir en una variable), cuáles son los valores posibles de s y t? (Nota, no es necesario que liste TODOS los resultados, pero si todos los casos que resulten significativos)**

int s = 6; t = 4;

co

**1)** s = s + t;

**2)** t = s \* t;

oc

**1)**

**1.1 load PosMemS, acumulador**

**1.2 add PosMemT, acumulador**

**1.3 store PosMemS, acumulador**

**2)**

**2.1 load PosMemS, acumulador**

**2.2 mult PosMemT, acumulador**

**2.3 store PosMemT, acumulador**

**Posibles interleavings (órdenes):**

**1er interleaving:**

**1.1** Acumulador = PosMemS = 6

**2.1** Acumulador = PosMemS = 6

**1.2** Acumulador = 6 + 4 = 10

**2.2** Acumulador = 10 \* 4 = 40

**1.3** PosMemS = **40**

**2.3** PosMemT = **40**

**2do interleaving:**

**2.1** Acumulador = PosMemS = 6

**2.2** Acumulador = 6 \* 4 = 24

**1.1** Acumulador = PosMemS = 6

**2.3** PosMemT = **6**

**1.2** Acumulador = 6 + 4 = 10

**1.3** PosMemS = **10**

**3er interleaving:**

**1.1** Acumulador = PosMemS = 6

**2.1** Acumulador = PosMemS = 6

**2.2** Acumulador = 6 \* 4 = 24

**1.2** Acumulador = 24 + 4 = 28

**2.3** PosMemT = **28**

**1.3** PosMemS = **28**

**14) Suponga los siguientes programas concurrentes. Asuma que a un es arreglo bidimensional, EOS es un valor especial que indica el fin de la secuencia de mensajes, y que los procesos son iniciados desde el programa principal.**

|  |  |
| --- | --- |
| **P1** | **P2** |
| chan canal (double)  **process Genera** {  int fila, col; double sum;  for [fila= 1 to 10000]  for [col=1 to 10000]  send canal (a(fila,col));  send canal(EOS);  }  **process Acumula** {  double valor, sumtT;  sumtT = 0;  receive canal(valor);  while valor <> EOS {  sumT = sumtT + valor;  receive canal(valor);  }  printf(sumT);  } | chan canal (double)  **process Genera** {  int fila, col; double sum;  for [fila= 1 to 10000]{  sum = 0;  for [col=1 to 10000]  sum = sum +a(fila,col);  send canal(sum);  }  send canal(EOS);  }  **process Acumula** {  double valor, sumtT;  sumtT = 0;  receive canal(valor);  while valor <> EOS {  sumT = sumtT + valor;  receive canal(valor);  }  printf(sumT);  } |

**a) ¿Qué hacen los programas?**

**P1**. El proceso ***Genera*** recorre todas las filas, por cada fila, recorre todas sus columnas y le envía a través de un canal llamado “canal” al proceso ***Acumula*** el valor correspondiente al arreglo bidimensional en esa posición (una fila - *primer for-* y una columna -*segundo for*-), el proceso ***Acumula*** recibe dichos valores a través de “canal”, y va totalizando por fila, mientras no se haya terminado de recorrer el total de filas. Luego imprime en pantalla el total, es decir, la sumatoria de todas las filas.

**P2**. El proceso ***Genera*** recorre todas las filas, por cada fila, inicia un contador en 0. Recorre todas sus columnas y le envía a través de “canal” al proceso ***Acumula*** el valor total de la fila. Luego ***Acumula*** va recibiendo los valores de cada fila, y los va totalizando en su variable “*sum”*. Una vez que finaliza, imprime en pantalla el total, es decir, la sumatoria de todas las filas.

**b) Analice desde el punto de vista del número de mensajes.**

El Programa 1 va a transmitir un número mayor de mensajes, debido a que el proceso Genera no totaliza por fila, sino que le envía al proceso Acumula cada valor del arreglo bidimensional a través de ***send***, es decir que hace 10000x10000 sends. Y el proceso Acumula recibe esa misma cantidad de ***receive***.

En cambio, el Programa 2, va a transmitir un número menor de mensajes, debido a que el proceso Genera totaliza por fila y una vez totalizada la fila, le envía este valor al proceso Acumula, por lo que la cantidad de mensajes enviados y recibidos disminuye significativamente. (10000 send y 10000 receive)

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

Desde el punto de vista de la granularidad, el programa 1 apunta a una solución de grano fino debido a la cantidad de mensajes que necesita transmitir. El programa 2, de lo contrario, apunta a una solución de grano grueso, debido a que transmite una menor cantidad de mensajes, pero realiza un mayor procesamiento.

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

Una arquitectura de grano grueso consta de pocos procesadores, con alta capacidad de procesamiento, pero menor comunicación. Por ende, me parece más adecuado el proceso 2, ya que a pesar de que realiza mucho procesamiento, utiliza menor comunicación que el proceso 1.

**15) Dado el siguiente bloque de código, indique para cada uno de los ítems si son equivalentes o no. Justificar cada caso.**

|  |
| --- |
| **Segmento 1** |
| int cant= 1000;  while (true) {  if (cant > 15);datos?(cant)→ Sentencias1  \* (cant < 5);datos?(cant)→ Sentencias2  \* (INCOGNITA); datos?(cant) → Sentencias3  END IF  } |
| **Segmento 2** |
| int cant= 1000;  DO (cant > 15);datos?(cant) → Sentencias1  \* (cant < 5);datos?(cant) → Sentencias2  \* (INCOGNITA); datos?(cant) → Sentencias3  END DO |

En el IF, se selecciona alguna guarda que sea elegible, en forma no determinística, y se sale.

En este caso el IF está dentro de un while true, por lo que nunca terminará su ejecución.

En el DO, se seleccionan las guardas elegibles en forma no determinística, hasta que TODAS sean falsas, y luego se sale.

La guarda cant > 15 cubre todos los posibles valores desde el 16 hasta el infinito.

La guarda cant < 5 cubre todos los posibles valores desde el menos infinito hasta el 4.

Falta cubrir los valores desde el 5 hasta el 15.

**a) INCÓGNITA equivale a (cant=5) or (cant=15)**

No son equivalentes, ya que en el caso de valores desde 6 a 14, el primer segmento queda en un loop infinito y el segundo segmento termina.

**b) INCÓGNITA equivale a (cant>0)**

Son equivalentes para cualquier valor que tome la variable cant, ya que quedan cubiertos todos los posibles valores que pueden llegar.

**c) INCÓGNITA equivale a ( (cant>= 2) and (cant <= 20) )**

Son equivalentes para cualquier valor que tome la variable cant, ya que quedan cubiertos todos los posibles valores que pueden llegar.

**d) INCÓGNITA equivale a ( (cant>5) and (cant <= 15) )**

No son equivalentes, ya que en el caso que cant vale 5 el primer segmento queda en un loop infinito y el segundo segmento termina.

Puesto que las 3 guardas fallan, entonces el if termina sin efecto pero vuelve a iterar con el while true.

**e) INCÓGNITA equivale a ( (cant>5) and (cant < 15) )**

No son equivalentes, ya que cuando cant vale 5 o 15, el primer segmento queda en un loop infinito y el segundo segmento termina.

## PARTE C: Temas para desarrollar (12 puntos)

**16) Defina el problema de la sección crítica. Compare los algoritmos para resolver estos problemas (spin locks, tie breaker, ticket, bakery) marcando ventajas y desventajas de cada uno.**

La sección crítica está precedida por un protocolo de entrada y seguido de un protocolo de salida. Cada sección crítica es una secuencia de sentencias que acceden a algún objeto compartido. Cada sección no crítica es otra secuencia de sentencias. Asumimos que un proceso que entra en su sección crítica eventualmente sale; así, un proceso solo puede finalizar fuera de su sección crítica.

Propiedades a cumplir:

* **Exclusión mutua**. A lo sumo un proceso está en su SC. (*propiedad de seguridad*)
* **Ausencia de Deadlock:** Si 2 o más procesos tratan de entrar a sus SC, al menos uno tendrá éxito. (*propiedad de seguridad*)
* **Ausencia de Demora Innecesaria:** Si un proceso trata de entrar a su SC y los otros están en sus SNC o terminaron, el primero no está impedido de entrar a su SC. (*propiedad de seguridad*)
* **Eventual Entrada:** Un proceso que intenta entrar a su SC tiene posibilidades de hacerlo (eventualmente lo hará). (*propiedad de vida*)

**Spin Locks:**los procesos se quedan iterando (*spinning*) mientras esperan que se limpie ***lock****.* Cumple las 4 propiedades si el *scheduling es* ***fuertemente fair***.

Características

* Baja performance en multiprocesadores si varios procesos compiten por el acceso.
* Se necesita una variable compartida (“lock”), y su acceso continuo es muy costoso (“*memory contention*”).
* Puede producirse un alto *overhead* por caché inválida.
* Está técnica no controla el orden de los procesos demorados, es posible que alguno no entre nunca si el scheduling no es *fuertemente fair*

**Tie-breaker:** Protocolo de SC que requiere scheduling sólo *débilmente fair* y no usa instrucciones especiales, pero es más complejo.

Usa una variable por cada proceso para indicar que el proceso comenzó a ejecutar su protocolo de entrada a la sección crítica, y una variable adicional para romper empates, indicando qué proceso fue el último en comenzar dicha entrada ⇒ esta última variable es compartida y de acceso protegido.

Demora (quita prioridad) al último en comenzar su *entry protocol*.

Tie-Breaker para muchos procesos es *costoso en tiempo y complejo.*

**Ticket:** Se reparten números y se espera turno. Los procesos toman un número mayor que el de cualquier otro que espera ser atendido; luego esperan hasta que todos los procesos con número más chico hayan sido atendidos.

Problema: los valores de las variables próximo y turno son ilimitados. En la práctica, podrían resetearse a un valor chico (por ejemplo, 1).

**Bakery**: Cada proceso que trata de ingresar recorre los números de los demás y se auto asigna uno mayor. Luego espera a que su número sea el menor de los que esperan. ***Los procesos se chequean entre ellos y no contra un global***.

Este algoritmo es más complejo, pero es *fair* y no requiere instrucciones especiales. No requiere un contador global **próximo** que se “entrega” a cada proceso al llegar a la SC. Esta solución de grano grueso no es implementable directamente.

**17) a) Describa brevemente para qué sirven los algoritmos de RPC y Rendevouz. ¿Para qué tipos de problemas son más adecuados?**

RPC y Rendezvous son notaciones o mecanismos de programación que combinan aspectos de monitores y PMS y suponen un ***canal bidireccional*** de comunicación entre procesos y son ideales para programar aplicaciones Cliente/Servidor, donde la comunicación entre ellos debe ser sincrónica, el cliente solicita un servicio y el servidor le responde con lo solicitado, ya que el cliente no debe realizar ninguna otra tarea hasta no obtener una respuesta del servidor.

RPC y Rendezvous demoran al llamador hasta que la operación llamada se termine de ejecutar y se devuelvan los resultados.

**RPC:** Los programas se descomponen en **módulos** (con procesos y *procedures*), que pueden residir en espacios de direcciones distintos.

* Los procesos de un módulo pueden compartir variables y llamar a *procedures* de ese módulo.
* Un proceso en un módulo puede comunicarse con procesos de otro módulo sólo invocando procedimientos exportados por éste.
* Los módulos tienen especificación e implementación de *procedures.*

**RENDEZVOUS:** combina **comunicación** y **sincronización**. Un proceso cliente *invoca* una operación por medio de un *call*, pero esta operación es servida por un proceso existente en lugar de por uno nuevo. Un proceso servidor usa una *sentencia de entrada* para esperar por un *call* y actuar. Las operaciones se atienden una por vez más que concurrentemente.

**Diferencias entre RPC y Rendezvous**

Difieren en la manera en la cual se sirven las invocaciones de operaciones.

* Un enfoque es declarar un *procedure* para cada operación y crear un nuevo proceso (al menos conceptualmente) para manejar cada llamado (*RPC* porque el llamador y el cuerpo del *procedure* pueden estar en distintas máquinas). Para el cliente, durante la ejecución del servicio, es como si tuviera en su sitio el proceso remoto que lo sirve (Ej: *JAVA*).
* El segundo enfoque es hacer *rendezvous* con un proceso existente. Un *rendezvous* es servido por una *sentencia de Entrada* (o ***accept***) que espera una invocación, la procesa y devuelve los resultados (Ej:*Ada*)

**b) ¿Por qué es necesario proveer de mecanismos de sincronización en módulos de RPC? ¿Cómo puede realizarse esta sincronización?**

Por sí mismo, RPC es solo un mecanismo de comunicación. Aunque un proceso llamador y su server sincronizan, el único rol del server es actuar en nombre del llamador (como si éste estuviera ejecutando el llamado → la sincronización entre ambos es implícita).

Se necesita que los procesos en un módulo sincronicen → esto comprende *Exclusión Mutua* y *Sincronización por Condición*.

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

Ada no provee la posibilidad de asociar sentencias de scheduling y de poder usar los parámetros formales de la operación tanto en las sentencias de sincronización como en las sentencias de scheduling.

**18) Explique sintéticamente los 7 paradigmas de interacción entre procesos en la programación distribuida. En cada caso, ejemplifique, indique que tipo de comunicación por mensajes es mas conveniente. Justificar.**

***Servidores Replicados***: Un server puede ser replicado cuando hay múltiples instancias de un recurso: cada server maneja una instancia. La replicación también puede usarse para darle a los clientes la sensación de un único recurso cuando en realidad hay varios.

Ejemplo: problema de los filósofos.

***Algoritmos Heartbeat***: útil para soluciones iterativas que se quieren paralelizar. Usando un esquema “*divide & conquer*” se distribuye la carga (datos) entre los *workers*; cada uno es responsable de actualizar una parte. Los nuevos valores dependen de los mantenidos por los workers o sus vecinos inmediatos. Cada “paso” debiera significar un progreso hacia la solución. Su uso más importante es paralelizar soluciones iterativas. Ejemplos: computación de grillas (labeling de imágenes) o autómatas celulares (el juego de la vida).

***Algoritmos Pipeline***: Un pipeline es un arreglo lineal de procesos “filtro” que reciben datos de un puerto (canal) de entrada y entregan resultados por un canal de salida. Estos procesos (“*workers*”) pueden estar en procesadores que operan en paralelo, en un primer esquema a lazo abierto. Un segundo esquema es el pipeline circular, donde *Wn* se conecta con *W1*. Estos esquemas sirven en procesos iterativos o bien donde la aplicación no se resuelve en una pasada por el pipe. En un tercer esquema posible (cerrado), existe un proceso coordinador que maneja la “realimentación” entre *Wn* y *W1*. Ejemplos: redes de filtros o tratamiento de imágenes.

***Probes & Echos*:** Arboles y grafos son utilizados en muchas aplicaciones distribuidas como búsquedas en la WEB, BD, sistemas expertos y juegos. Las arquitecturas distribuidas se pueden asimilar a los nodos de grafos y árboles, con canales de comunicación que los vinculan. DFS es uno de los paradigmas secuenciales clásicos para visitar todos los nodos en un árbol o grafo. Este paradigma es el análogo concurrente de DFS.

*Prueba-eco* se basa en el envío de un mensajes (“*probe*”) de un nodo al sucesor, y la espera posterior del mensaje de respuesta (“*echo*”). Los *probes* se envían en paralelo a todos los sucesores. Los algoritmos de prueba-eco son particularmente interesantes cuando se trata de recorrer redes donde no hay (o no se conoce) un número fijo de nodos activos (ejemplo: redes móviles)

***Algoritmos Broadcast***: En la mayoría de las LAN cada procesador se conecta directamente con los otros. Estas redes normalmente soportan la primitiva broadcast. Los mensajes *broadcast* de un proceso se encolan en los canales en el orden de envío, pero broadcast no es atómico y los mensajes enviados por procesos A y B podrían ser recibidos por otros en distinto orden. Se puede usar broadcast para diseminar información o para resolver problemas de sincronización distribuida. Ejemplo: semáforos distribuidos, la base es un *ordenamiento total de eventos de comunicación* mediante el uso de *relojes lógicos*.

***Token Passing:*** basado en un tipo especial de mensaje (“*token*”) que puede usarse para otorgar un permiso (control) o recoger información global de la arquitectura distribuida. Un ejemplo del primer tipo de algoritmos es el caso de tener que controlar *exclusión mutua distribuida*.

Un token es una clase especial de mensaje que puede ser usado o para dar permiso para tomar una acción o para reunir información de estado global. El envío de información global y/o la toma de determinadas decisiones se basan en el uso de tokens enviados entre los procesos. La topología no necesariamente debe ser de anillo ni los tokens globales.

***Manager/workers***: El concepto de *bag of tasks* usando variables compartidas supone que un conjunto de *workers* comparten una “bolsa” con tareas independientes. Los *workers* sacan una tarea de la bolsa, la ejecutan, y posiblemente crean nuevas tareas que ponen en la bolsa. La mayor virtud de este enfoque es la escalabilidad y la facilidad para equilibrar la carga de trabajo de los *workers*.

**19) a) Definir las métricas de speedup y eficiencia. ¿Qué miden? ¿Cuál es el rango de valores de cada una?**

**Speedup (S):** *S* es el cociente entre el tiempo de ejecución serial del algoritmo serial conocido más rápido (*Ts*) y el tiempo de ejecución paralelo del algoritmo elegido (*Tp*)

S = Ts/ Tp.

Rango de valores: en general entre 0 y Sóptimo.

Mide cuanto más rápido es el algoritmo paralelo con respecto al algoritmo secuencial, es decir cuánto se gana por usar más procesadores.

**Eficiencia (E):** Cociente entre speedup y Speedup Óptimo. E = S / p, p es el n° de procesadores.

Mide la fracción de tiempo en que los procesadores son útiles para el cómputo.

Rango de valores: entre 0 y 1, dependiendo de la efectividad de uso de los procesadores. Cuando es 1 corresponde al speedup perfecto.

**b) ¿Qué se entiende por escalabilidad de un sistema paralelo?**

La escalabilidad de un sistema paralelo apunta a la medida de usar eficientemente un número creciente de procesadores.

**c) Suponga que se tiene dos soluciones paralelizables por p procesadores S=p-4 y la otra S=p/2. ¿Cuál de las 2 se comportará más eficientemente cuando incremente el número de procesadores?**

S1 = **p-4**

S2 = **p/2**

Fórmula de eficiencia: E= S/p

**Si p = 1**

S1 = **(1-4)** / 1 = -3

S2 = **(1/2)** / 1 = 0,5

**Si p = 2**

S1 = **(2-4)** / 2 = -1

S2 = **(2/2)** / 2 = 0.5

**Si p = 3**

S1= **(3 - 4)** / 3 = - 0.33

S2 = **(3 / 2)** / 3 = 0,5

**Si p = 4**

S1= **(4 - 4)** / 4 = 0

S2 = **(4 / 2)** / 4 = 0,5

Tenemos

* E = (p-4) / p
* E = (p / 2) / p

Cuanto más grande es P, mayor será el valor en E= (p- 4 ) / p. En cambio E= (p/2)/p mantiene su valor constante.

Entonces, podemos ver que la Solución 1 se comportará de manera más eficiente cuando incrementa el número de procesadores.

**d) Suponga que el tiempo de ejecución de un algoritmo secuencial es de 1000 unidades de tiempo, de las cuales sólo el 90% es paralelizable. ¿Cuál es el límite en la mejora que se puede obtener paralelizando el algoritmo? Justifique.**

1000 x 90% = 900 esto es tiempo paralelizable, el resto es secuencial. osea 100. Recordar la ley de Amdahl que dice que el máximo valor paralelizable se calcula en base a la cantidad de **código no paralelizable** en este caso 100 (la parte paralelizable, osea 900, no se tiene en cuenta), no en base a los números de procesadores.

Por lo tanto el límite de mejora va a ser:

**1000 (total de unidades) /100 (código no paralelizable) = 10 limite de mejora**

Con 10 procesadores se alcanza el límite de la mejora.

## PARTE D: Ejercicio a resolver (6 puntos)

**20) Sea la siguiente solución al problema del producto de matrices de NxN con P procesos en paralelo con variables compartidas.**

process Worker [w= 1 to P] {

int first = (w-1) \* n/P +1;

int last = first + n/P - 1;

for [i = first to last] {

for [j = 1 to n] {

c[i,j] = 0, 0;

for [k = 1 to n]

c[i,j] = c[i,j] + a[i,k] \* b[k,j];

}

}

}

**a) Suponga n=256 y cada procesador capaz de ejecutar un proceso.**

**Cuántas asignaciones, sumas y productos se hacen secuencialmente (caso en que P=1 )? Cuántas se realizan en cada procesador en la solución paralela con P=8?**

**Si P = 1**

n = 256

process Worker [w= 1 to P] {

int first = (w-1) \* n/P +1;

int last = first + n/P - 1;

for [i = first to last] { 0 a 255 = 256 veces

for [j = 1 to n] { 1 a 256 = 256 veces

c[i,j] = 0, 0; 2562 ASIGNACIONES

for [k = 1 to n] 1 a 256 = 256 veces

c[i,j] = c[i,j] + a[i,k] \* b[k,j]; 2563 PRODUCTOS, 2563 SUMAS, 2563 ASIGNACIONES

}

}

}

2563  **\*1+** 2563 **\* 2 +** 2563 **\*3=100663296 →tiempo secuencial**

**Si P=8**

n = 256

256/8 procesadores = 32.

Cada procesador trabaja sobre 32 filas.

process Worker [w= 1 to P] {

int first = (w-1) \* n/P +1;

int last = first + n/P - 1;

for [i = first to last] {

for [j = 1 to n] { 1 a 256 = 256 veces

c[i,j] = 0, 0; **32 X** 256 ASIGNACIONES

for [k = 1 to n] 1 a 256 = 256 veces

c[i,j] = c[i,j] + a[i,k] \* b[k,j]; **32 X** 2562 PRODUCTOS, **32 X** 2562 SUMAS, **32 X** 2562 ASIGNACIONES

}

}

}

**b) Si P1= P2=...=P7 y los tiempos de asignación son 1, de suma 2 y de producto 3, y si P4 es 3 veces más lento, cuánto tarda el proceso concurrente? Cuál es el valor del speedup? Cómo modificaría el código para lograr un mejor speedup?**

**En el caso de 8 procesadores:**

32 x 2562 + 32 x 256 = 2.105.344 x 1

32 x 2562 productos = 2.097.152 x 3

32 x 2562 sumas = 2.097.152 x 2

**Cuánto tarda el proceso concurrente?**

2.097.152 x 2 suma + 2.097.152 X 3 producto + 2.105.344 x 1 asignación = **12.591.104** x 3, ya que P4 es 3 veces más lento = **37.773.312** → Esto es TP, es lo que más podría tardar.

**Cuál es el valor del speedup?**

**SpeedUp = Ts/Tp**

Ts = tiempo secuencial

Tp = tiempo paralelo

SpeedUp = 12.591.104 / 37.773.312 yo diria que 100663296/37.773.312=2.66

**Cómo modificaría el código para lograr un mejor speedup? Este se deja en blancooo jaja.. Este es el quilombooo**

**WARNING**

solución si hubieran 8 procesadores,(el concepto se entiende, no se pongan puntillosos)

si a p4 se le asignan 21 filas menos y se reparten entre los otros 7.

p4→11

p1 a p7 →35

***p1 a p7***

35 x 2562 + 35 x 256 asignaciones =2302720x 1

35 x 2562 productos = 2293760 x 3

35 x 2562 sumas = 2293760 x 2

Sumamos los 3 …. =13771520

***p4***

11 x 2562 + 11 x 256 asignaciones =723712 x 1 \*3

11 x 2562 productos = 720896 x 3 \*3

11 x 2562 sumas = 720896 x 2 \*3

Sumamos los 3 …. =4328192 \*3=12984576

speedup 100663296/13771520=7.31 lo cual es bastante cercano a 8

Speedup = TS / TP

Su rango de valores varía entre [1; p) siendo p la cantidad de procesadores empleados en paralelo.

La eficiencia es una medida que permite determinar cuan bien un programa paralelo utiliza procesadores extra.

Eficiencia = Speedup / p = TS / (p \* TP)

Si la eficiencia es 1 entonces el speedup es perfecto.

en este caso 7.31/8=0.91 que es mucho mejor que 2.66/8=0.33