FINAL

1)
a) ¿A qué se denomina propiedad de programa? ¿Qué son las propiedades de vida y seguridad? Ejemplifique.
b) Defina fairness. Relacione dicho concepto con las políticas de scheduling.
a) Una propiedad de un programa es un atributo que es verdadero en cada posible historia de ese programa. Toda propiedad interesante puede ser formulada en términos de dos clases de propiedades: Seguridad y vida.
Una propiedad de seguridad asegura que nada malo ocurre durante la ejecución; una propiedad de vida afirma que algo bueno eventualmente ocurre.
Una propiedad de vida asegura que eventualmente ocurre algo bueno con una actividad. (progresa, no hay deadlocks) (Una falla de vida indica que se deja de ejecutar).
En los programas secuenciales, la propiedad de seguridad clave es que el estado final es correcto, y la clave de la propiedad de vida es terminación. Estas propiedades son igualmente importantes para programas concurrentes.

Ejemplos de propiedad de seguridad:

- Exclusión mutua: A lo sumo un proceso está en su SC.
- Ausencia de Deadlock (Livelock): si 2 o más procesos tratan de entrar a sus SC, al menos uno tendrá éxito.
- 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.

Ejemplos de propiedad de vida:

- Eventual Entrada: un proceso que intenta entrar a su SC tiene posibilidades de hacerlo (eventualmente lo hará)..
- Bloqueos temporarios: una operación es demorada porque otra está ejecutando código crítico, aceptables por un cierto período.
- Espera: puede darse por un evento, mensaje o condición que ya produjo otro proceso.
- Input: un proceso espera entrada desde otro proceso/device.
- Contención de CPU: el procesador está ocupado por otros procesos.

Fuente: Pag 40 del libro y teoria 2(52) y 3(8) de la tarde.

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

Las acciones atómicas pueden ser ejecutadas en paralelo solo si no interfieren, la ejecución paralela puede ser modelizada por ejecución serial, interleaved.

Una política de scheduling de bajo nivel, tal como la política de alocación de procesador

en un sistema operativo, concierne a la performance y utilización del hardware. Esto es importante, pero igualmente importante son los atributos globales de las políticas de scheduling y sus efecto sobre la terminación y otras propiedades de vida de los programas concurrentes.

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

Cuando un programa contiene acciones atómicas condicionales, necesitamos hacer suposiciones más fuertes para garantizar que los procesos progresarán. Esto es porque una acción atómica condicional, aún si es elegible, es demorada hasta que la guarda es true.

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

En síntesis, si < **await** B --> S > es elegible y B se vuelve true y permanece true, entonces la acción atómica eventualmente se ejecuta. Round-robin y timeslicing son políticas débilmente fair si todo proceso tiene chance de ejecutar. Esto es porque cualquier proceso demorado eventualmente verá que su condición de demora es true.

Sin embargo, esto no es suficiente para asegurar que cualquier sentencia **await** elegible eventualmente se ejecuta. Esto es porque la guarda podría cambiar el valor (de false a true y nuevamente a false) mientras un proceso está demorado. En este caso, necesitamos una política de scheduling más fuerte.

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.
Una guarda es true con infinita frecuencia si es true un número infinito de veces en cada historia de ejecución de un programa (non-terminating). Para ser fuertemente fair, una política no puede considerar seleccionar sólo una acción cuando la guarda es false; debe seleccionar alguna vez la acción cuando la guarda es true.

Fuente: Pag 42 del libro y teoria 2(56).

2)
a) ¿Cuáles son y describa las propiedades que debe cumplir un protocolo de E/S a una sección crítica?
b) ¿Cuáles son los defectos que presenta la sincronización por busy waiting? Diferencie esta situación respecto de los semáforos.
c) Explique la semántica de la instrucción de grano grueso AWAIT y su relación con instrucciones tipo Test & Set o Fetch & Add.
a) Las propiedades que debe cumplir un protocolo de E/S a una sección crítica son 4:

- Exclusión mutua: A lo sumo un proceso está en su SC.
- Ausencia de Deadlock (Livelock): si 2 o más procesos tratan de entrar a sus SC, al menos uno tendrá éxito.
- 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.
- Eventual Entrada: un proceso que intenta entrar a su SC tiene posibilidades de hacerlo (eventualmente lo hará).

Fuente: Pag 45 del libro y teoria 3 (7).

b) Los protocolos de sincronización que usan solo busy waiting pueden ser difíciles de diseñar, entender y probar su corrección. La mayoría de estos protocolos son bastante complejos, no hay clara separación entre las variables usadas para sincronización y las usadas para computar resultados. Una consecuencia de estos atributos es que se debe ser muy cuidadoso para asegurar que los procesos se sincronizan correctamente.
Otra deficiencia es que es ineficiente cuando los procesos son implementados por multiprogramación. Un procesador que está ejecutando un proceso “spinning” podría ser empleado más productivamente por otro proceso. Esto también ocurre en un multiprocesador pues usualmente hay más procesos que procesadores.
Dado que la sincronización es fundamental en los programas concurrentes, es deseable tener herramientas especiales que ayuden en el diseño de protocolos de sincronización correctos y que puedan ser usadas para bloquear procesos que deben ser demorados. Los *Semáforos* son una de las primeras de tales herramientas y una de las más importantes. Hacen fácil proteger SC y pueden usarse de manera disciplinada para implementar sincronización por condición.
Los semáforos pueden ser implementados de más de una manera. En particular, pueden implementarse con busy waiting, pero también interactuando con un proceso scheduler para obtener sincronización sin busy waiting.
El concepto de semáforo es motivado por una de las maneras en que el tráfico de trenes es sincronizado para evitar colisiones: es una flag de señalización que indica si la pista está desocupada o hay otro tren. Los semáforos en los programas concurrentes proveen un mecanismo de señalización básico y se usan para implementar exclusión mutua y sincronización por condición.

Usando variables de tipo semáforo se distingue claramente entre los tipos de variables para sincronizar y para computar resultados.

Fuente: Pag 73 del libro y teoria 4(2).

c) Si una expresión o asignación no satisface ASV con frecuencia es necesario ejecutarla atómicamente.
En general, es necesario ejecutar secuencias de sentencias como una única acción atómica.

<await (B) S;> se utiliza para especificar sincronización.

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.

Sentencia con alto poder expresivo, pero el costo de implementación de la forma general de await (exclusión mutua
y sincronización por condición) es alto.

• Await general: await (s>0) s=s-1;
• Await para exclusión mutua: x = x + 1; y = y + 1

• Ejemplo await para sincronización por condición: await (count > 0)

Si B satisface ASV, puede implementarse como busy waiting o spin loop
do (not B) skip od
(while (not B);)

Solución de "grano fino": Spin Locks

Objetivo: hacer "atómico" el await de grano grueso.

Idea: usar instrucciones como **Test & Set (TS)**, **Fetch & Add (FA)** o Compare & Swap, disponibles en la mayoría de los procesadores.

¿Como funciona Test & Set?

```
bool TS (bool ok);
{  < bool inicial = ok;
   ok = true;
    return inicial; >
}
```

bool lock = false;
process SC [i=1..n]
{ while (true)
{await (not lock) lock= true;
sección crítica;
lock = false;
	bool lock=false;
	process SC[i=1 to n]
	{ while (true)
	{ while (TS(lock)) skip ;
	sección crítica;
	lock = false;

sección no crítica; } }	sección no crítica; } }

Solución tipo **“spin locks”**: los procesos se quedan iterando (spinning) mientras esperan que se limpie lock.

Cumple las 4 propiedades si el scheduling es fuertemente fair.
Una política débilmente fair es aceptable (rara vez todos los procesos están simultáneamente tratando de entrar a su SC).

… (leer completo de la teoría 3 hoja 14)

Fuente: Pag 73 del libro y teoria 3(14 hasta el final de la teoría).

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

ALOCACION DE RECURSOS
La alocación de recursos es el problema de decidir cuándo se le puede dar a un proceso acceso a un recurso. En programas concurrentes, un recurso es cualquier cosa por la que un proceso podría ser demorado esperando adquirirla. Esto incluye entrada a una SC, acceso a una BD, un slot en un buffer limitado, una región de memoria, el uso de una impresora, etc. Ya hemos examinado varios problemas de alocación de recursos específicos. En la mayoría, se empleó la política de alocación posible más simple: si algún proceso está esperando y el recurso está disponible, se lo aloca. Por ejemplo, la solución al problema de la SC aseguraba que se le daba permiso para entrar a *algún* proceso que estaba esperando; no intentaba controlar a cuál proceso se le daba permiso si había una elección. De manera similar, la solución al problema del buffer limitado no intentaba controlar cuál productor o cuál consumidor eran el próximo en acceder al buffer. La política de alocación más compleja que consideramos fue en el problema de lectores/escritores. Sin embargo, nuestra atención estuvo en darle preferencia a clases de procesos, no a procesos individuales.
Esta sección muestra cómo implementar políticas de alocación de recursos generales y en particular muestra cómo controlar explícitamente cuál proceso toma un recurso cuando hay más de uno esperando. Primero describimos el patrón de solución general. Luego implementamos una política de alocación específica (shortest job next). La solución emplea la técnica de passing the baton. También introduce el concepto de semáforos privados, lo cual provee la base para resolver otros problemas de alocación de recursos.

Definición del problema y Patrón de solución general

En cualquier problema de alocación de recursos, los procesos compiten por el uso de unidades de un recurso compartido. Un proceso pide una o más unidades ejecutando la operación *request*, la cual con frecuencia es implementada por un procedure. Los parámetros a *request* indican cuantas unidades se requieren, identifican alguna característica especial tal como el tamaño de un bloque de memoria, y dan la identidad del proceso que pide. Cada unidad del recurso compartido está libre o en uso. Un pedido puede ser satisfecho cuando todas las unidades del recurso compartido están libres. Por lo tanto *request* se demora hasta que esta condición es true, luego retorna el número requerido de unidades. Después de usar los recursos alocados, un proceso los retorna al pool de libres ejecutando la operación *release*. Los parámetros a *release* indican la identidad de las unidades que son retornadas. Ignorando la representación de las unidades del recurso, las operaciones *request* y *release* tienen la siguiente forma general:
request(parámetros): á **await** request puede ser satisfecho ® tomar unidades ñ

release(parámetros): á retornar unidades ñ

Las operaciones necesitan ser atómicas dado que ambas necesitan acceder a la representación de las unidades del recurso. Siempre que esta representación use variables diferentes de otras variables del programa, las operaciones aparecerán como atómicas con respecto a otras acciones y por lo tanto pueden ejecutar concurrentemente con otras acciones. Este patrón de solución general puede ser implementado usando la técnica de passing the baton. En particular, *request* tiene la forma de F2 de modo que es implementada por un fragmento similar a (4.7):

(4.9) *request*(parámetros): **P**(e)
if request no puede ser satisfecho ® DELAY **fi**
toma unidades
SIGNAL

Similarmente, *release* la forma de F1 de modo que es implementada por un fragmento de programa similar a (4.6):

(4.10) *release*(parámetros): **P**(e)
retorna unidades
SIGNAL

Como antes, *e* es un semáforo que controla la entrada a las operaciones, y SIGNAL es un fragmento de código como (4.8); SIGNAL o despierta un proceso demorado (si algún pedido demorado puede ser satisfecho) o ejecuta **V**(e). El código de DELAY es un fragmento similar al mostrado en (4.7): registra que hay un request demorado, ejecuta **V**(e), luego se demora en un semáforo de condición. Los detalles exactos de cómo es implementado SIGNAL para un problema específico depende de cómo son las condiciones de demora diferentes y cómo son representadas. En cualquier evento, el código DELAY necesita salvar los parámetros que describen un pedido demorado de modo que puedan ser examinados en SIGNAL. Además, se necesita que haya un semáforo de condición por cada condición de demora diferente.

La próxima sección desarrolla una solución a un problema de alocación de recursos específica. La solución ilustra cómo resolver tal problema.

Alocación Shortest-Job-Next

Es una política de alocación usada para distintas clases de recursos. Asumimos que los recursos compartidos tienen una única unidad (luego consideraremos el caso general). Esta política se define como sigue:

(4.11) **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.

Por ejemplo, la política SJN puede ser usada para alocación de procesador (en la cual *time* es el tiempo de ejecución), para spooling de archivos a una impresora (*time* sería el tiempo de impresión), o para servicio de file trasfer (ftp) remoto (*time* sería el tiempo estimado de transferencia). La política SJN es atractiva pues minimiza el tiempo promedio de ejecución. Sin embargo, es inherentemente unfair: un proceso puede ser demorado para siempre si hay una corriente continua de requests especificando tiempos de uso menores. (Tal unfairness es extremadamente improbable en la práctica a menos que el recurso esté totalmente sobrecargado. Si interesa unfairness, la política SJN puede ser levemente modificada de modo que un proceso que ha estado demorado un largo tiempo tenga preferencia; esta técnica es llamada “aging”).

Si un proceso hace un pedido y el recurso está libre, el pedido puede ser satisfecho inmediatamente pues no hay otros pedidos pendientes. Así, el aspecto SJN de la política de alocación se pone en juego solo si más de un proceso tiene un pedido pendiente. Dado que hay un solo recurso, es suficiente usar una variable booleana para registrar si el recurso está disponible. Sea *free* una variable booleana que es true cuando el recurso está disponible y false cuando está en uso. Para implementar la política SJN, los request pendientes necesitan ser recordados y ordenados. Sea *P* un conjunto de pares (*time*, *id*), ordenado por los valores del campo *time*. Si dos pares tienen el mismo valor para *time*, están en P en el orden en el que fueron insertados. Con esta especificación, el siguiente predicado debe ser un invariante global:

SJN: *P* es un conjunto ordenado Ù *free* Þ (*P* = Æ)

En síntesis, *P* está ordenado, y, si el recurso está libre, *P* es el conjunto vacío. Inicialmente, *free* es true y *P* es vacío, de modo que SJN es trivialmente true.

Ignorando la política SJN por el momento, un request puede ser satisfecho exactamente cuando el recurso está disponible. Esto resulta en la solución coarse-grained:

var free := true

request(time,id): á **await** free ® free := false ñ

release( ): á free := true ñ

Sin embargo, con la política SJN un proceso que ejecuta *request* necesita demorarse hasta que el recurso esté libre y el pedido del proceso es el próximo en ser atendido de acuerdo a la política SJN. A partir del segundo conjuntor de SJN, si *free* es true en el momento en que un proceso ejecuta *request*, el conjunto P está vacío. Por lo tanto la condición de demora es suficiente para determinar si un pedido puede ser satisfecho inmediatamente. El parámetro *time* incide solo si un request debe ser demorado, es decir, si *free* es false. Basado en estas observaciones, podemos implementar *request* como se mostró en (4.9):

request(time, id): **P**(e)
if not free ® DELAY **fi**
free := false
SIGNAL

Y podemos implementar *release* como mostraba (4.10):

release( ): **P**(e)
free := true
SIGNAL

En *request*, asumimos que las operaciones **P** sobre el semáforo de entrada *e* se completan en el orden en el cual fueron intentados; es decir, **P**(e) es FCFS. Si esto no es así, los pedidos no necesariamente serán servidos en orden SJN.
Lo que resta es implementar el aspecto SJN de la política de alocación. Esto involucra usar el conjunto P y semáforos para implementar DELAY y SIGNAL. Cuando un request no puede ser satisfecho, necesita ser salvado para ser examinado más tarde cuando el recurso es liberado. Así, en DELAY un proceso necesita:

- insertar sus parámetros en P
- liberar el control de la SC protegiendo *request* y *release* ejecutando **V**(e), luego demorarse en un semáforo

Cuando el recurso es liberado, si el conjunto P no está vacío, el recurso necesita ser alocado a exactamente un proceso de acuerdo con la política SJN. En este caso, insertar los parámetros en P corresponde a incrementar un contador cj en (4.7). En ambos casos, un proceso indica que está a punto de demorarse e indica qué condición está esperando.
En anteriores ejemplos (como la solución a lectores/escritores) había solo unas pocas condiciones de demora distintas, y por lo tanto se necesitaban pocos semáforos de condición. Aquí, sin embargo, cada proceso tiene una condición de demora distinta, dependiendo de su posición en P: el primer proceso en P necesita ser despertado antes del segundo, y así siguiendo. Asumimos que hay *n* procesos que usan el recurso. Sea *b*[1:n] un arreglo de semáforos, donde cada entry es inicialmente 0. También asumimos que los valores de *id* son únicos y están en el rango de 1 a *n*. Entonces el proceso *id* se demora sobre el semáforo *b*[id]. Aumentando *request* y *release* con los usos de P y *b* como especificamos, tenemos la siguiente

solución al problema de alocación SJN:

```
var free := true, e : sem := 1, b[1:n] : sem := ( [n] 0 )
var P : set of (int, int) := Æ
{ SJN: P es un conjunto ordenado Ù free Þ ( P = Æ ) }
request(time, id):  P(e)          {SJN}
                        if not free ® insert(time,id) en P; V(e); P(b[id]) fi
                        free := false    {SJN}
                        V(e)     # optimizado pues free es false en este punto
release( ): P(e)          {SJN}
                free := true
                if P ¹ Æ ® remover el primer par (time,id) de P; V(b[id])
                 P = Æ ® {SJN}  V(e)
                fi
```

En esta solución, el insert en *request* se asume que pone el par en el lugar apropiado en P para mantener el primer conjunto de SJN. Luego, SJN es invariante fuera de *request* y *release*; es decir, SJN es true luego de cada **P**(e) y antes de cada **V**(e). La primera sentencia guardada en el código de señalización en *release* despierta exactamente un proceso si hay un pedido pendiente, y por lo tanto P no está vacío. El "baton" es pasado a ese proceso, el cual setea *free* a false. Esto asegura que el segundo conjuntor en SJN es true si P no es vacío. Dado que hay un único recurso, no se pueden satisfacer requests posteriores, de modo que el código signal en *request* es simplemente **V**(e).

Los semáforos b[id] son ejemplos de lo que se llaman semáforos privados.

(4.12) **Semáforo Privado**. El semáforo *s* es llamado semáforo privado si exactamente un proceso ejecuta operaciones **P** sobre *s*.

Los semáforos privados son útiles en cualquier situación en la cual es necesario ser capaz de señalar procesos individuales. Para algunos problemas de alocación, sin embargo, puede haber menos condiciones de demora diferentes que procesos que compiten por un recurso. En ese caso, puede ser más eficiente usar un semáforo para cada condición diferente que usar semáforos privados para cada proceso. Por ejemplo, si los bloques de memoria son alocados solo en unos pocos tamaños (y no interesa en qué orden los bloques son alocados a los procesos que compiten por el mismo tamaño) entonces sería suficiente tener un semáforo de demora para cada tamaño diferente.

Podemos generalizar rápidamente la solución a recursos con más de una unidad. En este caso, cada unidad estaría libre o alocado, y *request* y *release* tendrían un parámetro, *amount*, para indicar cuántas unidades requiere o devuelve un proceso. Podríamos modificar la solución como sigue:

- Reemplazar *free* por un entero *avail* que registra el número de unidades disponibles
- En *request*, testear si *amount* unidades están libres, es decir, si *amount* £ *avail*. Si es así, alocarlas; si no, registrar cuántas unidades se requieren antes de demorarse
- En *release*, incrementar *avail* por *amount*, luego determinar si el proceso demorado hace más tiempo que tiene el mínimo valor para *time* puede ser satisfecho. Si es así,

despertarlo; si no, ejecutar **V**(e).

La otra modificación es que ahora podría ser posible satisfacer más de un request pendiente cuando las unidades son liberadas. Por ejemplo, podría haber dos procesos demorados que requieren juntos menos unidades que las que fueron liberadas. En este caso, el que es despertado primero necesita hacer signal al segundo después de tomar las unidades que requiere. En resumen, el protocolo de signal al final de *request* necesita ser el mismo que el del final de *release*.

IMPLEMENTACION

Dado que las operaciones sobre semáforos son casos especiales de sentencias **await**, podemos implementarlas usando busy waiting y la técnica del capítulo 3. Sin embargo, la única razón por la que uno querría hacerlo así es para ser capaz de escribir programas usando semáforos más que spin locks y flags de bajo nivel. En consecuencia, mostraremos cómo agregar semáforos al kernel descripto en el capítulo 3. Esto involucra aumentar el kernel con descriptores de semáforo y tres primitivas adicionales: *create_sem*, *P*, y *V*.

Un descriptor de semáforo contiene el valor de un semáforo; es inicializado invocando *create_sem*. Las primitivas *P* y *V* implementan las operaciones **P** y **V**. Asumimos aquí que todos los semáforos son generales; por lo tanto la operación **V** nunca bloquea. En esta sección, describimos cómo estas componentes son incluidas en un kernel monoprocesador y cómo cambiar el kernel resultante para soportar múltiples procesadores.

Recordemos que en el kernel monoprocesador, un proceso a la vez estaba ejecutando, y todos los otros estaban listos para ejecutar. Como antes, el índice del descriptor del proceso ejecutante es almacenado en la variable *executing*, y los descriptores para todos los procesos listos son almacenados en la ready list. Cuando se agregan semáforos, hay un tercer estado de proceso posible: bloqueado. En particular, un proceso está bloqueado si está esperando completar una operación **P**. Para seguir la pista de los procesos bloqueados, cada descriptor de semáforo contiene una lista enlazada de los descriptores de procesos bloqueados en ese semáforo. En un monoprocesador, exactamente un proceso está ejecutando, y su descriptor no está en ninguna lista; todo otro descriptor de proceso o está en la ready list o en la lista de bloqueados de algún semáforo.

Para cada declaración de semáforo en un programa concurrente, se genera un llamado a la primitiva *create_sem*; el valor inicial del semáforo se pasa como argumento. La primitiva *create_sem* encuentra un descriptor de semáforo vacío, inicializa el valor y la lista de bloqueados, y retorna un “nombre” para el descriptor. Este nombre típicamente es la dirección del descriptor o un índice en una tabla que contiene la dirección.

Después de que un semáforo es creado, es usado para invocar las primitivas **P** y **V**. Ambas tienen un único argumento que es el nombre de un descriptor de semáforo. La primitiva P chequea el valor en el descriptor. Si el valor es positivo, es decrementado; en otro caso, el descriptor del proceso ejecutante se inserta en la lista de bloqueados del semáforo. Similarmente, V chequea la lista de bloqueados del descriptor del semáforo. Si está vacía, el valor del semáforo es imcrementado; en otro caso un descriptor de proceso es removido de la lista de bloqueados e insertado en la ready list, Es común para cada lista de bloqueados implementarla como una cola FIFO pues asegura que las operaciones sobre semáforos son

fair.
Las outlines de estas primitivas son las siguientes (se agregan a las rutinas en el kernel monoprocesador ya visto. Nuevamente, el procedure *dispatcher* es llamado al final de cada primitiva; sus acciones son las mismas que antes):

```
procedure create_sem(valor inicial) returns name
  tomar un descriptor de semáforo vacío
  inicializar el descriptor
  setear name con el nombre (índice) del descriptor
  call dispatcher( )
end
procedure P(name)
  encontrar el descriptor de semáforo de name
  if valor > 0 ® valor := valor - 1
   valor = 0 ® insertar descriptor de executing al final de la lista de bloqueados
                executing := 0          # indica que executing ahora está bloqueado
  fi
  call dispatcher( )
end
procedure V(name)
  encontrar el descriptor de semáforo de name
  if lista de bloqueados vacía ® valor := valor + 1
   lista de bloqueados no vacía ®
        remover el descriptor de proceso del principio de la lista de bloqueados
        insertar el descriptor al final de la ready list
  fi
  call dispatcher( )
end
```

Por simplicidad, la implementación de las primitivas de semáforo no reusan descriptores de semáforos. Esto sería suficiente si todos los semáforos son globales a los procesos, pero en general esto no sucede. Así usualmente es necesario reusar descriptores de semáforos como los descriptores de proceso. Una aproximación es que el kernel provea una primitiva adicional *destroy_sem*; sería invocada por un proceso cuando ya no necesita un semáforo. Una alternativa es registrar en el descriptor de cada proceso los nombres de todos los semáforos que ese proceso creó. Luego, los semáforos que creó podrían ser destruidos por el kernel cuando el proceso invoca la primitiva *quit*. Con esto, es imperativo que un semáforo no sea usado luego de ser destruido.
Podemos extender la implementación monoprocesador a una para un multiprocesador de la misma manera que en el capítulo 3. Nuevamente, el requerimiento crítico es bloquear las estructuras de datos compartidas, pero solo por el mínimo tiempo requerido. Por lo tanto, debería haber un lock separado para cada descriptor de semáforo. Este descriptor es lockeado en P y V justo antes de ser accedido; el lock es liberado tan pronto como el descriptor ya no se necesita. Como en el primer kernel multiprocesador, los locks son adquiridos y liberados por

una solución busy-waiting al problema de la SC.

¿Minimiza el tiempo promedio de espera?¿Es fair? Si no lo es, plantee una alternativa que lo sea.

Minimiza el tiempo promedio de espera pero no es fair, ya que pueden llegar procesos de mayor duración que van quedando relegados por el scheduler a la hora de la elección del siguiente proceso, y viene siempre un proceso nuevo más corto que le gana la posición y lo deja relegado haciéndole inanición.

Para evitar la inanición se plantea una alternativa fair que es igual a SJN agregando la técnica de Aging. Esta técnica toma en cuenta los tiempos de cada proceso en espera para considerar si un proceso que lleva más tiempo en el CPU pero su trabajo es más largo es elegido sobre otro proceso de más corta duración que acaba de entrar en la disputa por la CPU. Suele contar los ciclos de espera.

Wiki: La técnica de aging es el proceso de aumentar gradualmente la prioridad de un proceso, en función de su tiempo de espera. El aging se puede utilizar para reducir la inanición de procesos de baja prioridad. El aging se utiliza para garantizar que los procesos en el menor nivel de las colas finalmente completen su ejecución.

4) En qué consiste la técnica de "passing the baton"? ¿Cuál es su utilidad?

b) Aplique este concepto a la resolución del problema de lectores y escritores.

c) ¿Qué relación encuentra con la técnica de "passing the condition"?

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

a) La técnica de Passing the Baton

Esta técnica se llama *passing the baton* por la manera en que los semáforos son señalizados. Cuando un proceso está ejecutando dentro de una región crítica, podemos pensar que mantiene el baton que significa permiso para ejecutar. Cuando ese proceso alcanza un fragmento SIGNAL, pasa el baton a otro proceso. Si algún proceso está esperando una condición que ahora es verdadera, el baton es paso a tal proceso, el cual a su turno ejecuta la región crítica y pasa el baton a otro proceso. Cuando ningún proceso está esperando una condición que es true, el baton es pasado al próximo proceso que trata de entrar a su región crítica por primera vez.

Esta técnica es lo suficientemente poderosa para implementar cualquier sentencia **await**.

b) Solución a lectores y escritores

Podemos usar la técnica de passing the baton para implementar la solución coarse-grained al problema de lectores/escritores. En esa solución, hay dos guardas distintas, de modo que necesitamos dos semáforos de condición y contadores asociados. Sea que el semáforo *r* representa la condición de demora del lector *nw* = 0, y *w* representa la condición de demora del escritor *nr* = 0 AND *nw* = 0. Sean *dr* y *dw* los contadores asociados. Finalmente, sea ***e*** el semáforo de entrada.

La siguiente solución:

var nr := 0, nw := 0 { RW: (nr = 0 ∨ nw = 0) ∧ nw ≤ 1 }

```
var e : sem := 1, r : sem := 0, w : sem := 0 {SPLIT: 0 ≤ (e + r + w) ≤ 1 }
var dr := 0, dw := 0 {COUNTERS: dr ≥ 0 ∧ dw ≥ 0 }

Reader[i:1..m] :: do true →
        P(e)
        if nw > 0 →
                dr := dr + 1;
                V(e);
                P(r)
        fi
        nr := nr + 1
        SIGNAL 1
        lee la BD
        P(e)
        nr := nr - 1
        SIGNAL 2
od

Writer[j:1..n] :: do true →
        P(e)
        if nr > 0 or nw > 0 →
                dw := dw + 1;
                V(e);
                P(w)
        fi
        nw := nw + 1
        SIGNAL 3
        escribe la BD
        P(e)
        nw := nw - 1
        SIGNAL 4
od

Aquí SIGNALi es una abreviación de:
        if nw = 0 and dr > 0 →
                dr := dr - 1;
                V(r)
        (nr = 0 and nw = 0 and dw > 0) →
                dw := dw - 1; V(w)
        (nw > 0 or dr = 0 and (nr > 0 or nw > 0 od dw = 0) →
                V(e)
        fi
```

Aquí, SIGNALi asegura que nw es 0 cuando el semáforo *r* es señalizado y asegura que tanto nr como nw son 0 cuando el semáforo *w* es señalizado. El semáforo *e* es señalizado solo cuando no hay lectores o escritores demorados que podrían seguir.
En la solución anterior (y en general) las precondiciones de los fragmentos SIGNAL permiten que varias de las guardas sean simplificadas o eliminadas. En los procesos lectores, tanto nr > 0 como nw = 0 son true antes de SIGNAL1. Por lo tanto ese fragmento signal se simplifica a:
if dr > 0 → dr := dr -1; V(r) dr = 0 → V(e) fi

Antes de SIGNAL2 en los lectores, tanto nw como dr son 0. En los escritores, nr = 0 y nw > 0 antes de SIGNAL3, y nr = 0 y nw = 0 antes de SIGNAL4. Usando estos hechos para simplificar los protocolos de señalización obtenemos la siguiente solución final:

```
var nr := 0, nw := 0 { RW: ( nr = 0 ∨ nw = 0 ) ∧ nw ≤ 1 }
var e : sem := 1, r : sem := 0, w : sem := 0 {SPLIT: 0 ≤ (e + r + w) ≤ 1 }
var dr := 0, dw := 0 {COUNTERS: dr ≥ 0 ∧ dw ≥ 0 }

Reader[i:1..m] :: do true →
        P(e)
        if nw > 0 →
                dr := dr + 1;
                V(e);
                P(r)
        fi
        nr := nr + 1
        if dr > 0 →
                dr := dr - 1;
                V(r) nr = 0 →
                        V(e)
        fi
        lee la BD
        P(e)
        nr := nr - 1
        if nr = 0 and dw > 0 →
                dw := dw - 1;
                V(w)
        nr > 0 or dw = 0 →
                V(e)
        fi
od

Writer[j:1..n] :: do true →
        P(e)
        if nr > 0 or nw > 0 →
```

```
            dw := dw + 1;
            V(e);
            P(w)
        fi
        nw := nw + 1
        V(e)
        escribe la BD
        P(e)
        nw := nw - 1
        if dr > 0 →
            dr := dr - 1; V(r)
        dw > 0 →
            dw := dw - 1;
            V(w)
        dr = 0 and dw = 0 →
            V(e)
        fi
od
```

En esta solución, la última sentencia **if** en los escritores es no determinística. Si hay lectores y escritores demorados, cualquiera podría ser señalizado cuando un escritor termina su exit protocol. Además, cuando finaliza un escritor, si hay más de un lector demorado y uno es despertado, los otros son despertados en forma de "cascada". El primer lector incrementa *nr*, luego despierta al segundo, el cual incrementa *nr* y despierta al tercero, etc. **El baton se va pasando de un lector demorado a otro hasta que todos son despertados.**

c) Relación con Passing The Condition:

Passing the condition es una técnica utilizada en monitores donde un proceso pasa una condición implícitamente a un proceso que está demorado en espera de esa condición sin que los demás procesos se perciban de ello.

En la técnica passing the baton se consigue despertar a un proceso de un determinado tipo si se comprueba que la condición por la que estaba esperando es verdadera. De alguna manera, es como si se ocultara a los otros procesos que el baton a cambiado de dueño. Con "passing the condition" pasa algo similar ya que se consigue despertar a un proceso en particular pasándole la condición que se ha vuelto verdadera al proceso que se va a despertar. Al no hacer esta condición visible, sólo el proceso al que va a despertarse puede verla.

d)Un semáforo fair: Passing the Condition

A veces es necesario asegurar 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. Esto se hace pasando una condición directamente a un proceso despertado en lugar de hacerla globalmente visible. Ilustramos la técnica derivando un monitor que implementa un semáforo fair: un semáforo en el cual los procesos son capaces de completar las operaciones P en orden FCFS.

Recordemos la implementación del monitor Semaphore ya vista. Allí, puede parecer que los procesos pueden completar P en el orden en el cual llamaron a P. Las colas de las variables condición son FIFO en ausencia de sentencias wait con prioridad. En consecuencia, los procesos demorados son despertados en el orden en que fueron demorados. Sin embargo, dado que signal es no preemptivo, antes de que un proceso despertado tome chance de ejecutar, algún otro proceso podría llamar a P, encontrar que s > 0, y completar P antes que el proceso despertado. En resumen, los llamados desde afuera del monitor pueden “robar” condiciones señalizadas.

Para evitar que las condiciones sean robadas, el proceso de señalización necesita pasar la condición directamente al proceso despertado más que hacerla globalmente visible. 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 nVnPpending, 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).

Esta técnica de pasar una condición directamente a un proceso que espera puede ser aplicada a muchos otros problemas. Sin embargo esta técnica debe ser usada con cuidado, pues las sentencias wait no están en loops que rechequean la condición. En lugar de esto, la tarea de establecer la condición y asegurar que se mantendrá verdadera está en el señalador. Además, las
sentencias signal no pueden ya ser reemplazadas por signal_all.

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

Algoritmos broadcast
Permiten alcanzar una información global en una arquitectura distribuida.
Sirven para toma de decisiones descentralizadas.

Broadcast es la manera de enviar un mensaje abierto para la recepción de varios procesos a la vez. En programación concurrente usamos SIGNAL ALL que hace un signal a todos los procesos dormidos a la vez.
En ambientes distribuidos tienden a ser más eficientes los mecanismos de PM. En estos ambientes se necesita un mecanismo de comunicación/sincronización como pasaje de mensajes. (Pasaje de Mensajes Asincrónicos (PMA) - Pasaje de Mensajes Sincrónico (PMS) - Llamado a Procedimientos Remotos (RPC) - Rendezvous).
Las dificultades que puede tener el hacer un broadcast en ambientes distribuidos con pasaje de mensajes sincrónicos a diferencia del asincrónico es que en el primero para enviar un mensaje se necesita tener al receptor del otro lado preparado para recibir ese mensaje. Entonces si el proceso que debe hacer el broadcast del mensaje tiene que esperar a que todos los receptores del otro lado estén listos puede llegar a quedar trabado hasta que por fin todos se alisten. De esta manera puede pasar mucho tiempo esperando. El asincrónico tiene la ventaja de que los canales de comunicación pueden almacenar varios mensajes y no se necesita que el receptor esté del otro lado listo para recibir el mensaje, sino que puede leerlo más tarde.
fuente: nuestra brillantez

6)
a) ¿Por qué el problema de los filósofos es de exclusión mutua selectiva? Si en lugar de 5 filósofos fueran 3, ¿el problema seguiría siendo de exclusión mutua selectiva? ¿Por qué?
b) ¿El problema de los filósofos resuelto en forma centralizada y sin posiciones fijas es de exclusión mutua selectiva? ¿Por qué?
c) ¿Si en el problema de los lectores-escritores se acepta sólo 1 escritor o 1 lector en la BD, tenemos un problema de exclusión mutua selectiva? ¿Por qué?
d) ¿Que significa que un problema sea de "exclusión mutua selectiva"?
e) ¿El problema de los lectores-escritores es de exclusión mutua selectiva?¿Por qué?
f) De los problemas de los baños planteados en teoría, ¿Cuál podría ser de exclusión mutua selectiva?

a) El problema de los filósofos es de exclusión mutua selectiva porque cada proceso "filósofo" compite con los filósofos que tiene a sus lados por los recursos de los "cubiertos o palillos" y no compite con todos los procesos a la vez (solo un subconjunto de ellos).
Si en lugar de 5 filósofos fueran 3 ya no sería de exclusión mutua selectiva porque cada uno de los filósofos tendría que competir con los otros dos que hacen al total de filósofos y ya no serían un subconjunto de ellos.
b) Si el problema de los filósofos fuera resuelto de forma centralizada y sin posiciones fijas dejaría de ser de exclusión mutua selectiva porque los filósofos estarían compitiendo no con un subconjunto de filósofos (procesos) sino con cualquier filósofo que necesite de los cubiertos. Hay 5 filósofos, 5 cubiertos. Entonces como mucho siempre pueden estar comiendo dos a la vez. Todos los que están queriendo comer compiten entre si por los 5 cubiertos.
c) Si el problema de los lectores y escritores aceptara en la bd un solo lector o un solo escritor dejaría de ser también de exclusión mutua selectiva porque cada proceso estaría compitiendo con todos los procesos y ya no contra un subconjunto de ellos (en el caso de lectores y escritores normal, los lectores al poder entrar concurrentemente solo compiten contra los escritores. Hay competencia entre tipos de procesos).

d) En los problemas de exclusión mutua selectiva, cada proceso compite por sus recursos no con todos los demás procesos sino con un subconjunto de ellos. *Dos casos típicos de dicha competencia se producen cuando los procesos compiten por los recursos según su tipo de proceso o por su proximidad. Ejemplos de Exclusión mutua selectiva son: El problema de los filósofos y el problema de los lectores y escritores. (de la web)*

e) Si porque los lectores al poder entrar concurrentemente solo compiten contra los escritores.
f)Un baño único para varones o mujeres (excluyente) sin límite de usuarios.

7) Analice que tipo de mecanismos de pasaje de mensajes son más adecuados para resolver problemas de tipo Cliente/Servidor, Pares que interactúan, Filtros, y Productores y Consumidores. Justifique.
Problemas tipo:

- Cliente/Servidor:
- Pares que interactúan:
- Filtros:
- Productores y Consumidores:
  (Capítulos 7 y 8 de Andrews)

8) Describa la solución utilizando la criba de Eratóstenes al problema de hallar los primos entre 2 y n. ¿Cómo termina el algoritmo?¿Qué modificaría para que no termine de esa manera?

Conceptualmente podemos resolver el problema con las siguientes tareas:

-Comenzando con el primer número (2), recorremos la lista y borramos los múltiplos de ese número. Si n es impar: 2 3 5 7 . . . n

-Pasamos al próximo número (3) y borramos sus múltiplos.

-Siguiendo hasta que todo número fue considerado, los que quedan son todos los primos entre 2 y n.

La criba captura primos y deja caer múltiplos de los primos.
Esta solucion se puede paralelizar a traves de un Pipe de procesos filtro: cada uno recibe un stream de números de su predecesor y envía un stream a su sucesor. El primer número que recibe es el próximo primo, y pasa los no múltiplos.

Codigo:

```
Process Criba[1]
{ int p = 2;

for [i = 3 to n by 2] Criba[2] ! i;

}

Process Criba[i = 2 to L]
{ int p, proximo;

Criba[i-1] ? P;
```

```
do Criba[i-1] ? Proximo -->
•                                   if ((proximo MOD p) <> 0 )-->  Criba[i+1] ! proximo; fi
od

}
```

El número total de procesos Criba (L) debe ser lo suficientemente grande para garantizar que se generan todos los primos hasta n.

Excepto Criba[1], los procesos terminan bloqueados esperando un mensaje de su predecesor. Cuando el programa para, los valores de p en los procesos son los primos. Puede modificarse con centinelas.

Por el libro: La criba de Eratóstenes es un algoritmo clásico para determinar cuáles números en un rango son primos. Supongamos que queremos generar todos los primos entre 2 y n. Primero, escribimos una lista con todos los números:
2 3 4 5 6 7 ... n

Comenzando con el primer número no tachado en la lista, 2, recorremos la lista y borramos los múltiplos de ese número. Si n es impar, obtenemos la lista:
2 3 5 7 ... n

En este momento, los números borrados no son primos; los números que quedan todavía son candidatos a ser primos. Pasamos al próximo número, 3, y repetimos el anterior proceso borrando los múltiplos de 3. Si seguimos este proceso hasta que todo número fue considerado, los números que quedan en la lista final serán todos los primos entre 2 y n.
El siguiente programa secuencial implementa el algoritmo.

```
var num[2:n] : ([n-1] 0)
var p := 2, i : int
{ I: p es primo ∧ ∀j: 2 ≤ j ≤ (p-1)2: num[j] = 0 si y solo si j es primo) }
do p * p ≤ n →
fa i := 2*p to n by p → num[i] := 1 af # "tacha" los múltiplos de p
p := p + 1
do num[p] = 1 → p := p + 1 od
# busca el próximo número no tachado
od
```

La lista es representada por un arreglo, num[2:n]. Las entradas en num son inicialmente 0, para indicar que los números de 2 a n no están tachados; un número i es tachado seteando num[i] en 1. El loop externo itera sobre los primos p, los cuales son elementos de num que permanecen en 0. El cuerpo de ese loop primero tacha los múltiplos de p y luego avanza p al próximo primo. Como indicamos, el invariante para el loop externo dice que si num[j] es 0 y j

está entre 2 y (p-1)2 entonces j
es primo. El loop termina cuando p2 es mayor que n, es decir, cuando p es mayor que sqrt(n). El límite superior para j en el invariante y la condición de terminación del loop se desprenden del hecho de que si un número i no es primo, tiene un factor primo menor que sqrt(i).

Consideremos ahora cómo paralelizar este algoritmo. Una posibilidad es asignar un proceso diferente a cada posible valor de p y que cada uno, en paralelo, tache los múltiplos de p. Sin embargo, esto tiene dos problemas. Primero, dado que asumimos que los procesos se pueden comunicar solo intercambiando mensajes, deberíamos darle a cada proceso una copia privada de num, y tendríamos que usar otro proceso para combinar los resultados. Segundo, tendríamos que usar más procesos que primos (aún si num pudiera ser compartido). En el algoritmo secuencial, el final de cada iteración del loop externo avanza p al siguiente número no tachado. En ese punto, p se
sabe que es primo. Sin embargo, si los procesos están ejecutando en paralelo, tenemos que asegurar que hay uno asignado a cada posible primo. Rápidamente podemos sacar todos los números pares distintos de 2, pero no es posible sacar números impares sin saber cuáles son primos.

Podemos evitar ambos problemas paralelizando la criba de Eratóstenes de manera distinta. En particular, podemos emplear un pipeline de procesos filtro. Cada filtro en esencia ejecuta el cuerpo del loop externo del algoritmo secuencial. En particular, cada filtro del pipeline recibe un stream de números de su predecesor y envía un stream de números a su sucesor. El primer número que recibe un filtro es el próximo primo más grande; le pasa a su sucesor todos los números que no son múltiplos del primero.

El siguiente es el algoritmo pipeline para la generación de números primos:

```
# Por cada canal, el primer número es primo y todos los otros números
# no son múltiplo de ningún primo menor que el primer número
Sieve[1]:: var p := 2, i : int
        # pasa los número impares a Sieve[2]
        fa i := 3 to n by 2  Sieve[2] ! i af
Sieve[i:2..L]: var p : int, next : int
            Sieve[i-1] ? p        # p es primo
            do true
                    # recibe el próximo candidato
                    Sieve[i-1] ? next
                    # pasa next si no es múltiplo de p
                    if next mod p  0  Sieve[i+1] ! next fi
            od
```

El primer proceso, Sieve[1], envía todos los números impares desde 3 a n a Sieve[2]. Cada uno de los otros procesos recibe un stream de números de su predecesor. El primer número p que recibe el proceso Sieve[i] es el i-ésimo primo. Cada Sieve[i] subsecuentemente pasa todos los otros números que recibe que no son múltiplos de su primo p. El número total L de procesos Sieve debe ser lo suficientemente grande para garantizar que todos los primos hasta n son generados. Por ejemplo, hay 25 primos menores que 100; el porcentaje decrece para valores

crecientes de n.
El programa anterior termina en deadlock. Podemos fácilmente modificarlo para que termine normalmente usando centinelas, como en la red de filtros merge.

9)Sea el problema en el cual N procesos poseen inicialmente cada uno un valor V, y el objetivo es que todos conozcan cuál es el máximo y cuál es el mínimo de todos los valores.
a) Plantee conceptualmente posibles soluciones con las siguientes arquitecturas de red: centralizada, simétrica (o totalmente conectada) y anillo circular (NO IMPLEMENTE).
b) Analice las soluciones desde el punto de vista del número de mensajes y la perfomance global del sistema.

a) y b)
Centralizada:
La arquitectura centralizada es apta para una solución en que todos envían su dato local V al procesador central, éste ordena los N datos y reenvía la información del mayor y menor a todos los procesos 2(N-1) mensajes.
Si p[0] dispone de una primitiva broadcast se reduce a N mensajes.

Con esta arquitectura la solucion seria que uno de los procesos sea el que reciba los numeros del resto de los procesos, luego este calcula el maximo y el minimo, y por ultimo, lo envia a cada uno de los procesos para que todos conozcan estos resultados. El algoritmo tendria un numero de mensajes de 2(n-1). Pero si usamos la tecnica de broadcast seria n, ya que todos le envian al proceso central y este hace un solo envio general.

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

Anillo circular:
Un tercer modo de organizar la solución es tener un anillo donde P[i] recibe mensajes de P[i-1] y envía mensajes a P[i+1]. P[n-1] tiene como sucesor a P[0].
Esquema de 2 etapas. En la primera cada proceso recibe dos valores y los compara con su valor local, trasmitiendo un máximo local y un mínimo local a su sucesor. En la segunda etapa todos deben recibir la circulación del máximo y el mínimo global.

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

La granularidad de una aplicación está dada por la relación entre el cómputo y la comunicación.
Relación y adaptación a la arquitectura.
Grano fino y grano grueso.
Para una dada aplicación, significa optimizar la relación entre el número de procesadores y el tamaño de memoria total.

Clasificación por la granularidad de los procesadores

- De grano grueso (coarse-grained): pocos procesadores muy poderosos.
- De grano fino (fine-grained): gran número de procesadores menos potentes.
- De grano medio (medium-grained).

Aplicaciones adecuadas para una u otra clase:

- si tienen concurrencia limitada pueden usar eficientemente pocos procesadores ⇒ convienen máquinas de grano grueso
- las máquinas de grano fino son más efectivas en costo para aplicaciones con alta concurrencia ⇒ Es importante el matching entre la arquitectura y la aplicación...

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

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

P3::
x:= x*8 + x*2 + 1;
```

a) El valor de x al terminar el programa es 9.
b) El valor de x al terminar el programa es 6.
c) El valor de x al terminar el programa es 11.
d) Y siempre termina con alguno de los siguientes valores: 10 o 6 o 2 o 0.

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.

CSP (Communicating Sequential Processes, Hoare 1978) fue uno de los desarrollos fundamentales en Programación Concurrente. Muchos lenguajes reales (OCCAM, ADA, SR) se basan en CSP.

Las ideas básicas introducidas por Hoare fueron PMS y comunicación guardada: PM con waiting selectivo.

Canal: link directo entre dos procesos en lugar de mailbox global. Son half-duplex y nominados. Limitaciones de ? y ! ya que son bloqueantes. Hay problema si un proceso quiere comunicarse con otros (quizás por ports) sin conocer el orden en que los otros quieren hacerlo con él.

Por ejemplo, el proceso Copiar podría extenderse para hacer buffering de k caracteres: si hay más de 1 pero menos de k caracteres en el buffer, Copiar podría recibir otro carácter o sacar 1.

Las operaciones de comunicación (? y !) pueden ser guardadas, es decir hacer un AWAIT hasta que una condición sea verdadera.

El do e if de CSP usan los comandos guardados de Dijkstra (B --> S)

If y Do:

Las sentencias de comunicación guardadas aparecen en if y do.
if B1; comunicación1 S1;
• B2; comunicación2 S2;
fi

Ejecución:
Primero, se evalúan las guardas.

- Si todas las guardas fallan, el if termina sin efecto.
- Si al menos una guarda tiene éxito, se elige una de ellas (no determinísticamente).
- Si algunas guardas se bloquean, se espera hasta que alguna de ellas tenga éxito.

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

Tercero, se ejecuta la sentencia Si.

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

13) Describa brevemente los mecanismos de comunicación y de sincronización provistos por MPI, Linda y Java.

Java: Java provee para la programación concurrente el uso de threads. Estos son procesos livianos que mantienen un contador de programa, una pila de ejecución y un working set de manera privada.
Los Procesos “corren a través de los threads” y necesitan acceder a objetos compartidos que poseen datos u otros métodos. Para acceder a estos, los métodos de cada objeto están declarados con la palabra synchronized. Los procesos compiten por entrar al objeto que tiene un lock. Si no logra entrar al objeto cada objeto tiene una única cola de demora en el lock. El método wait libera el lock de un objeto y demora al thread.
JAVA permite la sincronización por condición con los métodos wait, notify y notifyAll, similares al “wait” y “signal” que vimos en monitores.
Si bien no existen las variables "condition" los "lock" implícitos en cada objeto cumplen esta función.
El método notify despierta al thread que está al frente en la cola de demora, si hay.
La semántica que tienen es de **SC (Signal and Continue).**
Si un método sincronizado en un objeto contiene un llamado a otro método en otro objeto, el lock del primer objeto es retenido mientras se ejecuta el llamado.

Las arquitecturas distribuidas han potenciado las soluciones basadas en PVM o MPI, que son básicamente bibliotecas de comunicaciones. Un esquema anterior (y original) es el de LINDA.

Los programas MPI usan un estilo SPMD. C/ proceso ejecuta una copia del mismo programa, y puede tomar distintas acciones de acuerdo a su “identidad”. Las instancias interactúan llamando a funciones MPI, que soportan comunicación proceso-a-proceso y grupales.

MPI_Init inicializa la biblioteca MPI y obtiene una copia de los comandos pasados al programa.
La variable MPI_COMM_WORLD es inicializada con el conjunto de procesos que se arrancan.
MPI_COMM_Size determina el número de procesos arrancados (2)
MPI_COMM_Rank determina la id del proceso (0 a size-1)
MPI_Finalize Termina este proceso y libera la bilioteca MPI.

MPI_Send envía un mensaje a otro proceso. Los argumentos son: el buffer que contiene el mensaje, el nro de elementos a enviar, el tipo de datos del mensaje, la identidad del proceso destino, un tag del usuario para identificar el tipo de mensaje y la variable MPI_COMM_WORLD.

MPI_Recv recibe un mensaje de otro proceso. Los argumentos son: el buffer en el cual poner el

mensaje, el número de elementos del mensaje, el tipo de datos del mensaje, la identidad del proceso que envía o “no importa”= MPI_any_source, un tag del usuario para identificar el tipo de mensaje, el grupo de comunicación y el status de retorno.

MPI

- MPI define una librería estándar para pasaje de mensajes que puede ser empleada desde C o Fortran (y potencialmente desde otros lenguajes).
- El estándar MPI define la sintaxis y la semántica de más de 125 rutinas.
- Hay implementaciones de MPI de la mayoría de los proveedores de hardware.
- Modelo SPMD.
- Todas las rutinas, tipos de datos y constantes en MPI tienen el prefijo “MPI_”. El código de retorno para operaciones terminadas exitosamente es MPI_SUCCESS.
- Básicamente con 6 rutinas podemos escribir programas paralelos basados en pasaje de mensajes: MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank, MPI_Send y MPI_Recv.

Diferentes protocolos para Send.

• Send bloqueantes con buffering (Bsend).
• Send bloqueantes sin buffering (Ssend).
• Send no bloqueantes (Isend).
Diferentes protocolos para Recv.
• Recv bloqueantes (Recv).
• Recv no bloqueantes (Irecv).

LINDA

LINDA aproximación distintiva al procesamiento concurrente que combina aspectos de MC y PMA.
No es un lenguaje de programación, sino un conjunto de 6 primitivas que operan sobre una Memoria Compartida donde hay “tuplas nombradas” (tagged tuples) que pueden ser pasivas (datos) o activas (tareas).
El núcleo de LINDA es el espacio de tuplas compartido (TS) 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 MC, 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.

EVAL provee el medio para incorporar concurrencia en un programa Linda

MPI

14) Explique sintéticamente los 7 paradigmas de interacción entre procesos en programación distribuida planteados en teoría. Ejemplifique.

Paradigmas para la interacción entre procesos

3 esquemas básicos de interacción entre procesos: productor/consumidor, cliente/servidor e interacción entre pares.
Estos esquemas básicos se pueden combinar de muchas maneras, dando lugar a otros paradigmas o modelos de interacción entre procesos.

Paradigma 1: master / worker
Implementación distribuida del modelo Bag of Task.

Paradigma 2: algoritmos heartbeat
Los procesos periódicamente deben intercambiar información con mecanismos tipo send/receive.

Paradigma 3: algoritmos pipeline
La información recorre una serie de procesos utilizando alguna forma de receive/send.

Paradigma 4: probes (send) y echoes(receive)
La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información.

Paradigma 5: algoritmos broadcast
Permiten alcanzar una información global en una arquitectura distribuida. Sirven para toma de decisiones descentralizadas.

Paradigma 6: token passing
En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos. También permite la toma de decisiones distribuidas.

Paradigma 7: servidores replicados

Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos.

15)

a) ¿Cuál es el objetivo de la programación paralela?
b) Mencione las tres técnicas fundamentales de la computación científica. Ejemplifique.
c) Defina las métricas de speedup y eficiencia. ¿Cuál es el significado de cada una de ellas (que miden)? Ejemplifique.
d) En que consiste la "ley de Amdahl"?
e) Suponga que el tiempo de ejecución de un algoritmo secuencial es de 1000 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?

a) El objetivo de la programación paralela es ejecutar una tarea en el menor tiempo utilizando una arquitectura multiprocesador. Distribuyendo la tarea en partes a cada procesador.
b) Las 3 técnicas suponemos que son programación multihilo, computo distribuido y cómputo paralelo. Ejemplos..
c) El procesamiento paralelo lleva a los conceptos de speedup y eficiencia.

Speedup ⇒ S = Ts / Tp
Eficiencia ⇒ E = S / p

Speedup, lo que tarda el procesador más rápido en ejecución secuencial dividido lo que tarda el procesador más lento en ejecución paralela.
La eficiencia es el Speedup dividido la cantidad de procesadores. (entre 0 y 1)
d) La ley de Amdahl dice que "para un dado problema existe un máximo speedup alcanzable independiente del número de procesadores".
e) Si tenemos 1000 unidades de tiempo y 800 son paralelizables un procesador debe ejecutar las 200 unidades secuenciales y esta nos va a marcar el límite para dividir las 800 en 200 entonces con 5 procesadores llegamos a un speedup de 5 y una eficiencia de 100%. Un speedup de 5 es la mejora.

16) Suponga que quiere ordenar n números enteros utilizando mensajes con el siguiente algoritmos (odd/even exchange sort). Hay n procesos P[1:n], con n par. Cada proceso ejecuta una serie de rondas. En las rondas "impares", los procesos con número impar P[impar] intercambian valores con P[impar+1] si los números están desordenados. En las rondas "pares", los procesos con número par P[par] intercambian valores con P[par+1] si los números están desordenados (P[1] y P[n] no hacen nada en las rondas "pares").
a) Determine cuántas rondas deben ejecutarse en el peor caso para ordenar n números.
N

b) ¿Considere que para este caso es más adecuado utilizar mensajes sincrónicos o asincrónicos? Justifique.

Mejor PMS porque asincrónico es innecesario mantener las colas de PMA

c) Escriba un algoritmo paralelo para odenar un arreglo a[1:n] en forma ascendente. ¿Cuántos mensajes se utilizan?

```
P[i:1..n]:: var valor:int; valor = a[i];
var aux: int;
ronda=1;
while (not cantidadDeRondas == n){

        if      (isImpar(ronda) and  and (i<>n))) ; P[i+1]! nombreCanal(valor) →
                        P[i+1]?nombreCanal(aux);
                        if (aux < valor) valor = aux;

        *       (isPar(ronda) and (i<>1) and (i<>n)) ; P[i-1]? nombreCanal(aux) →
                        P[i-1]!nombreCanal(valor)
                        if (aux > valor) valor = aux;

        fi
        a[i] = valor;
        ronda ++;
}
```

d) ¿Cómo modificaría el algoritmo del punto c) para que termine tan rápido como el arreglo esté ordenado (por ej, podría estar ordenado inicialmente)? ¿Esto agrega overhead de mensajes? ¿Cuanto?

Con un proceso coordinador al que los procesos le avisan si hicieron cambios en la ronda después de probar si debian intercambiar valores. Entonces este coordinador les avisa si deben continuar o no. Esto agrega overhead de mensajes 2*k mensajes entonces.

e) Modifique la respuesta dada en c) para usar k procesos (asuma que n es múltiplo de k).

n = 10, k = 5 entonces cada proceso k toma dos valores del arreglo

17) Dado el siguiente programa concurrente con memoria compartida:

x = 4; y = 2; z = 3;

co

x = y – z // z = z * 2 // y = z + 4

oc

a) ¿Cuáles de las asignaciones dentro de la sentencia co cumplen con la propiedad de

“A lo sumo una vez”?
Justifique claramente.
b) Indique los resultados posibles de la ejecución. Justifique.
Nota 1: las instrucciones NO SON atómicas.
Nota 2: no es necesario que liste TODOS los resultados.

18)
a) Analice conceptualmente la resolución de problemas con memoria compartida y memoria distribuida. Compare en términos de facilidad de programación.
b) Analice conceptualmente los modelos de mensajes sincrónicos y asíncrónicos. Compárelos en términos de concurrencia y facilidad de programación.

a)
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 trasmitir mensajes.

Memoria Compartida

- Variables compartidas
- Semáforos
- Regiones Críticas Condicionales
- Monitores

Memoria distribuida (pasaje de mensajes)

- Mensajes asincrónicos
- Mensajes sincrónicos
- Remote Procedure Call (RPC)
- Rendezvous

b) PMS vs PMA
El PMA hace SEND y RECEIVE
El PMS hace nombreCanal! y nombreCanal?

El PMA el SEND es no bloqueante y acumulativo (en una cola) y el RECEIVE bloquea hasta recibir algo. Se puede evitar usando isEmpty.
El PMS el envio y el recibo se debe coordinar, utilizando comunicación guardada. Entonces el envio y recibo son bloqueantes
PMS es menos concurrente que PMA dado que se bloquea y debe esperar al proceso del otro lado para coordinar la comunicación, y no puede seguir ejecutando otras tareas a diferencia de PMA que el que envia deposita en una cola, y sigue ejecutando otras tareas sin importar si el que recibe ya recibió su mensaje.
En cuanto a la programación el PMS se tiene que considerar la coordinación de comunicación entre los procesos para que ambos estén disponibles al momento de hacerlo. Esto en PMA es más fácil y no se tiene que tener tan en cuenta.
Para PMS si un proceso debe recibir datos de por ejemplo 10 procesos ese primer proceso tiene que declarar 10 canales con cada uno de los procesos. Y si quisiera responderles con un envio necesita declarar otros 10 canales. Porque en PMS los canales son de tipo link (1 a 1) Uno para recibir y otro para enviar.
En cambio en PMA se puede manejar muchos menos canales. Si un proceso necesita recibir de 10 procesos un dato solo se necesita un canal. Y si les quiere responder un canal por cada proceso. Entonces
PMA = 11 canales
PMS = 20 canales.
Hacés las cuentas.

19) (Broadcast atómico). Suponga que un proceso productor y n procesos consumidores comparten un buffer unitario. El productor deposita mensajes en el buffer y los consumidores los retiran. Cada mensaje depositado por el productor tiene que ser retirado por los n consumidores antes de que el productor pueda depositar otro mensaje en el buffer.
a) Desarrolle una solución utilizando semáforos.
b) Suponga que el buffer tiene b slots. El productor puede depositar mensaje sólo en slots vacíos y cada mensaje tiene que ser recibido por los n consumidores antes de que el slot pueda ser reusado. Además, cada consumidor debe recibir los mensajes en el orden en que fueron depositados (note que los distintos consumidores pueden recibir los mensajes en distintos momentos siempre que los reciban en orden). Extienda la respuesta dada en a) para resolver este problema más general.

a)

```
consumo = 1, vacio = 1, lleno[1:n[0]] sem
cant: 0 int;
Process Productor {
// produce elemento
        while (true){
                P(vacio);
```

```
                //llena el buffer
                cant := n;
                for (i:=1 ..n){
                        v(lleno[i]);
                }
        }
}

Process Consumidor [c:1..n]{
        while (true){
                p(lleno[c]);
                p(consumo);
                // lee datos
                cant--;
                if (cant == 0){
                        v(vacio);
                }
                v(consumo);
        }
}
```

a)

```
sem consumo [1..b[1]]
sem lleno[1..k][1..b[0]]
sem consumo [1..b[0]]
sem slotvacio : 1

Process Productor {
// produce
while (true){
        p(slotvacio)
        for (i:1..b){
                if (cant[i] == 0){
                        //deposita
                        cant[i] := n;
                        for (j: 1..k)
                                v(lleno[j][i];
                        }
                }
        }
}

Process Consumidor [c:1..k]{
```

```
        x:= 1;
        while (true){
                p(lleno[c][x];
                p(consumo[x]);
                //lee
                cant[x]--;
                if (cant[x] == 0)
                        p(slotvacio);
                v(consumo[x]);
                if (x == b)
                        x:= 1;
                else
                        x++;
        }
}
```

20) 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);
        }
}
```

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

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

21) Sea "cantidad" una variable inicializada en 0 que representa la cantidad de elementos de un buffer, y sean P1 y P2 dos programas que se ejecutan de manera concurrente, donde cada una de las instrucciones que los componen son atómicas.

P1::

```
if (cantidad = 0) then
begin
  cantidad := cantidad +1;
  buffer := elemento_a_agregar;
end;
P2::
if (cantidad > 0 ) then
begin
  elemento_a_sacar := buffer;
  cantidad := cantidad-1;
end;
```

Además existen dos alumnos de concurrente que analizan el programa y opinan lo siguiente:
“Pepe: este programa funciona correctamente ya que las instrucciones son atómicas”
“José: no Pepe estás equivocado, hay por lo menos una secuencia de ejecución en la cual funciona erróneamente” ¿Con cual de los dos alumnos esta de acuerdo? Si está de acuerdo con Pepe justifique su respuesta. Si está de acuerdo con José encuentre una secuencia de ejecución que verifique lo que José opina y escríbala, y modifique la solución para que funciones correctamente (suponga buffer y elemento variables declaradas).

Josecito tiene razón, y Pepe es un boludo. Pepe recursó concurrente como dos años seguidos.
Josecito notó que cuando el P1 hace cantidad := cantidad + 1 y todavía no hace buffer:= elemento_a_agregar el P2 entra en el if (cantidad > 0) y saca un elemento viejo del buffer.
La solución sería limitar el acceso con semaforos. Entonces con un semaforo binario dividido por ejemplo

```
P1::
while (true){
  P(vacio);
  buffer := elemento_a_agregar;
  V(lleno);
}

P2::
while (true){
  P(lleno);
  elemento_a_sacar := buffer;
  V(vacio);
}
```

24)
a) Describa brevemente en que consisten los mecanismos de RPC y Rendezvous. ¿Para que tipo de problemas son más adecuados?
b) ¿Por qué es adecuado proveer sincronización dentro de los módulos RPC? ¿Cómo puede realizarse esta sincronización?
c) ¿Qué elementos de la forma general de Rendezvous no se encuentran en ADA?

RPC (Remote Procedure Call) y Rendezvous :técnicas de comunicación y sincronización entre procesos que suponen un canal bidireccional ideales para programar aplicaciones C/S
RPC y Rendezvous combinan 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).

Diferencias entre RPC y Rendezvous
Difieren en la manera de servir la invocación 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).
Teoria 10

25) <u>Resuelva con monitores el siguiente problema:</u>
Tres clases de procesos comparten el acceso a una lista enlazada: searchers, inserters y deleters.

- **Los searchers sólo examinan la lista, y por lo tanto pueden ejecutar concurrentemente unos con otros.**
- **Los inserters agregan nuevos items al final de la lista; las inserciones deben ser mutuamente exclusivas para evitar insertar dos items casi al mismo tiempo. Sin embargo un insert se puede hacerse en paralelo con uno o más searchers.**
- **Por último, los deleters remueven items de cualquier lugar de la lista. A lo sumo**

un deleter puede acceder a la lista a la vez, y el borrado también debe ser mutuamente exclusivo con searchers e inserciones.

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

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

2. Una red de procesos filtro(como la del Dibujo 1).

3. Odd/even exchange sort. Hay n procesos P[1:n], cada uno ejecuta una serie de rondas. En las rondas "impares", los procesos con un número impar P[impar] intercambian valores con P[impar+1]. En las rondas "pares", los proceso con un número par P[par] intercambian valores con P[par+1] (P[1] y P[n] no hacen nada en las rondas "pares"). En cada caso, si los números están desordenados actualizan su valor con el recibido. Asuma que cada proceso tiene almacenamiento local sólo para dos valores (el próximo valor y el mantenido hasta ese momento)

a) ¿Cuántos procesos son necesarios en 1 y 2? Justifique.
b) ¿Cuántos mensajes envía cada algoritmo para ordenar los valores? Justifique.
c) En cada caso, ¿Cuáles mensajes pueden ser enviados en paralelo (asumiendo que existe el hardware apropiado) y cuáles son enviados secuencialmente? Justifique.
d) ¿Cuál es el tiempo total de ejecución de cada algoritmo? Asuma que cada operación o de envío de mensajes toma una unidad de tiempo. Justifique

Preguntas de finales de la web

1. ¿En qué arquitecturas no es directamente aplicable el modelo de variables compartidas?.

En las arquitecturas distribuidas

2. Explica los problemas que pueden surgir en el anidamiento de llamadas a monitores.

La situación puede explicarse de la forma siguiente: Un proceso P1, entra en un Monitor A para ejecutar un procedimiento, en ese procedimiento se produce una llamada a un procedimiento de otro monitor B. Simultáneamente otro proceso P2 entra a ejecutar un procedimiento del monitor B y desde ese procedimiento se produce una llamada a un procedimiento del monitor A. En esta situación, si los procesos mantienen la exclusión mutua de los monitores, se produce un interbloqueo de procesos.

3. ¿Qué ventajas proporcionan los monitores frente a los semáforos?.

Mayor estructuración del código, ya que los accesos a las variables compartidas se reunen en el código de los monitores.

Mayor seguridad ya que el acceso a las variables del monitor está garantizado, por el propio monitor, que se realiza bajo exclusión mutua

4. ¿En qué consiste el problema de la exclusión mutua selectiva?. Pon un ejemplo de problema de esta clase y justifica la respuesta.

En los problemas de exclusión mutua selectiva, cada proceso compite por sus recursos no con todos los demás procesos sino con un subconjunto de ellos. Dos casos típicos de dicha competencia se producen cuando los procesos compiten por los recursos según su tipo de proceso o por su proximidad. Ejemplos de Exclusión mutua selectiva son: El problema de los filósofos y el problema de los lectores y escritores.

APUNTES IMPORTANTES

Cantidad de Historias sabiendo cantidad de procesos y cantidad de acciones

Un programa concurrente con n procesos, donde c/u ejecuta m acciones atómicas tiene una cantidad de historias posibles dada por **(n*m)! / (m!)n**
Para 3 procesos con 2 acciones, hay 90 interleavings posibles…

Formas de sincronización

Sincronización por exclusión mutua
Asegurar que sólo un proceso tenga acceso a un recurso compartido en un instante de tiempo.

Si el programa tiene secciones críticas que pueden compartir más de un proceso, EM 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.

Ejemplo de los dos mecanismos de sincronización en un problema de utilización de un área de memoria compartida (buffer limitado con productores y consumidores)

Elección de la Granularidad.

Para una dada aplicación, significa optimizar la relación entre el número de procesadores y el tamaño de memoria total.
Puede verse también como la relación entre cómputo y comunicación
Elección entre Grano fino y grano grueso.
Grano fino: mucha comunicación y varios computos pequeños
Grano grueso: mucho computo, poca comunicación.

El problema del deadlock

4 propiedades necesarias y suficientes p/ que exista deadlock:

- **Recursos reusables serialmente:** Los procesos comparten recursos que pueden usar con Exclusión Mutua.
- **Adquisición incremental:** Los procesos mantienen los recursos que poseen mientras esperar adquirir recursos adicionales.
- **No-preemption:** Una vez que son adquiridos por un proceso, los recursos no pueden quitarse de manera forzada sino que sólo son liberados voluntariamente.
- **Espera cíclica:** Existe una cadena circular (ciclo) de procesos t.q. c/u tiene un recurso que su sucesor en el ciclo está esperando adquirir.

Patrones de resolución Concurrentes

1- Paralelismo iterativo
2- Paralelismo recursivo
3- Productores y consumidores (pipelines o workflows)
4- Clientes y servidores
5- Pares que interactúan (interacting peers)