12/9/2020

Programación Concurrente 2020

Cuestionario guía - Clases Teóricas 1 y 2

1- Mencione al menos 3 ejemplos donde pueda encontrarse concurrencia

1. Procesador de textos. Mientras recibe entradas del usuario, controla ortografía, realiza sugerencias, guardados automáticos.
2. Usuarios accediendo y reservando pasajes.
3. Navegador accediendo a una web.

2- Escriba una definición de concurrencia. Diferencie procesamiento secuencial, concurrente y paralelo.

- **Concurrencia** es la capacidad de ejecutar múltiples actividades en paralelo o simultáneamente. Permite a distintos objetos actuar a mismo tiempo.

1. **Procesamiento secuencial:** Cuando se tiene un único hilo de ejecución o control. Un único flujo de control que ejecuta
   una instrucción y cuando esta finaliza ejecuta la siguiente.

2. **Procesamiento concurrente:** Cuando se tiene más de un hilo de ejecución o control, no implicando la existencia de más de un procesador. (+ hilos)

   Procesamiento Concurrente sin paralelismo de hardware → Una sola máquina que dedica parte del tiempo a cada componente del trabajo. Aprovecho de los tiempos muertos, por ejemplo, mientras no puedo seguir fabricando una parte del objeto porque hay que, dejarla reposar 5 minutos, aprovecho esos 5 minutos para continuar con la fabricación de otra parte del objeto.

   Pero agregan dificultades como:
   - Distribución de carga de trabajo.
   - Necesidad de compartir recursos.
   - Necesidad de esperarse en puntos clave.
   - Necesidad de comunicarse.
   - Necesidad de recuperar el “estado” de cada proceso al retomarlo.

   Procesamiento Concurrente con paralelismo de hardware → Seria como en el ejemplo de procesamiento paralelo parecido.

   Procesamiento Paralelo → Se asocia con la ejecución concurrente en múltiples procesadores con el objetivo principal de reducir el tiempo de ejecución.
   Una parte dentro de lo que es la concurrencia, agarra uno de esos programas y lo ejecuta sobre una arquitectura que tiene más de una unidad de procesamiento, ya es paralela la ejecución, en general objetivo del paralelismo reducir el tiempo final de ejecución tratando de mantener un buen rendimiento usando la arquitectura completa.

   - Ventajas
     - Menor tiempo para completar el trabajo.
     - Menos esfuerzo individual.
     - Paralelismo de Hardware
   - Dificultades
     - Distribución de las cargas de trabajo, para hacer una distribución correcta del trabajo. Si tengo menos máquinas que partes del objeto a fabricar, ¿cómo sé cuántos módulos le doy a cada máquina ?
     - Necesidad de compartir recursos evitando conflictos.

Necesidad de comunicación.
Tratamiento de fallas.
A qué máquina le asigno el ensamblado final de las partes.

Procesamiento paralelo: Procesamiento concurrente sobre más de un procesador, ejecutando cada uno un hilo diferente en un momento dado. (más recursos, procesadores)

3- Describa el concepto de *deadlock* y qué condiciones deben darse para que ocurra.

Deadlock: Bloqueo permanente de un conjunto de procesos producto de la espera por recursos de uso exclusivo, cada proceso esperando algún recurso sin liberar a los que posee bajo control. Puede producirse entre dos procesos o entre varios, en forma circular.
Las condiciones necesarias para que se produzca (condiciones de *Coffman*) son:

Puede producirse entre dos procesos o entre varios de forma circular.
4 propiedades necesarias y suficientes para 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 esperan adquirir recursos adicionales.
- **No expropiación:** un recurso asignado a un proceso no puede ser expropiado por otro proceso, sino que deben ser liberados voluntariamente.
- **Espera cíclica:** existe una cadena circular de procesos tal que cada uno tiene un recurso que su sucesor, en el ciclo, está esperando adquirir.

4- Defina inanición. Ejemplifique.

Un proceso no puede finalizar su ejecución ya que requiere de un recurso que nunca le es asignado.
El *deadlock* es un caso especial de inanición, como así también el *livelock*: una condición similar, pero en la que el estado de los procesos continúa cambiando, cada uno en respuesta a los cambios del otro, sin retomar el procesamiento.
Ejemplo de Inanición: cualquier esquema de atención por prioridades. Una técnica común para evitarlo es el esquema de envejecimiento.

(**NO LOGRA ACCEDER A LOS RECURSOS COMPARTIDOS**)

5- ¿Qué entiende por *no determinismo*? ¿Cómo se aplica este concepto a la ejecución concurrente?

- El **no determinismo** es la producción de resultados impredecibles, incluso para los mismos valores y condiciones iniciales.
En un programa concurrente compuesto por *n* procesos, en un momento dado habrá *n* operaciones atómicas elegibles (asumiendo que todos están en estado *ready*). Así, pueden obtenerse distintas **historias** de la ejecución de un programa incluso en corridas con los mismos valores de entradas. La historia específica de una corrida/ejecución será decidida no por el programa concurrente, sino por la política de scheduling del SO. Por este motivo es necesario utilizar mecanismos de sincronización si se desea obtener un resultado idéntico (para los mismos valores y condiciones de entrada) en cada corrida.

El no determinismo implica que, a partir de varias ejecuciones de un programa con la misma entrada, puedan generarse diferentes salidas, haciendo que dependiendo de la forma en que se vayan ejecutando los procesos, se puedan obtener resultados diferentes.

P1::	P2::	P3::

If (x = 0) then
y:= 4*2;
x:= y + 2;
	If (x > 0) then
	x:= x + 1;
		x:= (x*3) + (x*2) + 1;

6- Defina comunicación. Explique los mecanismos de comunicación que conozca.

- La comunicación entre procesos concurrentes específica el modo en que se organizan y transmiten datos entre tareas concurrentes. Los procesos se comunican por **memoria compartida** y por **pasaje de mensajes**.

Comunicación por 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 ésta.
- La solución más elemental es la utilización de una variable de control tipo "semáforo" que habilite o no el acceso de un proceso a la memoria compartida (exclusión mutua).

Comunicación por 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 que leer y cuándo deben transmitir mensajes.

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

- La sincronización es la posesión de información (por parte de un proceso) acerca de otro proceso para coordinar actividades.
Una **acción atómica** hace una transformación de estado indivisible.
Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más acciones atómicas.
La **historia** de un programa concurrente está dada por el intercalado de acciones atómicas ejecutadas por procesos individuales.

El objetivo de la sincronización es restringir las historias de un programa concurrente sólo a las permitidas.

Los mecanismos de sincronización son **por exclusión mutua** y **por condición**.

Sincronización por exclusión mutua:
- Asegura que sólo un proceso tenga acceso a un recurso compartido en un instante de tiempo.
- Si el programa tiene secciones críticas que puede compartir más de un proceso, esta sincronización 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.
- En el caso de los productores, no pueden producir hasta que la cola tenga espacio, y en el caso de los consumidores, deben esperar a que haya al menos un elemento en la cola para ser consumido.
Por ejemplo, para extraer un elemento de la cola la misma debe tener al menos un elemento.

b) ¿En un programa concurrente pueden estar presentes más de un mecanismo de sincronización? En caso afirmativo, ejemplifique

Ambos mecanismos de sincronización se pueden ver reflejados durante la comunicación por memoria compartida. Tener un buffer limitado, al cual todos los procesos acceden:

- ***Procesos consumidores y productores . Los productores generan elementos y la ponen en una cola.***
- Sincronización por condición: Los productores no pueden poner datos si la cola está llena, y los consumidores que consumen de la cola, no pueden consumir elementos si la cola está vacía.
- Sincronización por exclusión mutua. Si hay más de un productor y un solo espacio para colocar el elemento, entonces ese sitio debe ser bloqueado por alguno de los consumidores antes de ir a depositar el dato.

8- ¿Qué significa el problema de "interferencia" en programación concurrente? ¿Cómo puede evitarse?

- **La interferencia se da cuando un proceso toma una acción que invalida alguna suposición hecha por otro proceso**, ya que las acciones de los procesos de los programas concurrentes pueden ser intercaladas. De esta forma las asignaciones de un proceso a variables compartidas pueden afectar el comportamiento o invalidar un supuesto de otro proceso.
La misma puede evitarse utilizando los mecanismos de sincronización (por condición y/o exclusión mutua) para restringir el número de historias (intercalado/interleaving de sentencias) a las permitidas.

La interferencia ocurre cuando un proceso toma una acción que invalida las suposiciones hechas por otro proceso. Por ejemplo, un proceso va a utilizar un dato, entendiendo un determinado estado del mismo. Sin embargo, un proceso modifica el estado de ese dato, haciendo que la suposición que hizo el primer proceso sea inválida. La interferencia puede evitarse con la ejecución de acciones atómicas. Cada proceso debe ejecutar acciones atómicas, donde los estados intermedios no sean visibles para los otros procesos. Para esto puede ayudarnos la sincronización por exclusión mutua.(y/o por condicion).

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

La propiedad de 'A lo sumo una vez' (ASV) permite asegurar que, una sentencia que cumpla esta propiedad se ejecuta como si fuera atómica a pesar de que posee referencias críticas. Puede haber a lo sumo una variable compartida, y puede ser referenciada a lo sumo una vez. La ejecución parecerá atómica.

referencia crítica: referencia a una variable que es modificada por otro proceso.

Una sentencia **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, o
- **e** no contiene referencias críticas, en cuyo caso **x** puede ser leída por otro proceso.

Una expresión **e** satisface ASV si no contiene más de una referencia crítica. Las sentencias de un programa concurrente pueden referenciar una variable compartida a lo sumo una vez para evitar interferencias. De esta manera su ejecución parecerá atómica a los otros procesos.

Ejemplos:

```
int x=0, j=0;
co
        x = 1 + j; // j = 1;
oc;
```

-Cumple ASV ya que x tiene una sola referencia critica, j no tiene ninguna, y el resultado final de j es siempre 1, el de x puede ser 1 o 2.

```
int x=0, j=0;
co
        x = 1 + j; // j = 1 + x;
oc;
```

- No cumple ASV ya que ambos procesos poseen referencias críticas, y los resultados finales pueden ser x = 1 j= 2, o x = 2 j = 1.

10- Dado el siguiente programa concurrente:

```
x = 2; y = 4; z = 3;
co
  x = y - z //  z = x * 2 // y = y - 1
oc
```

a) ¿Cuáles de las asignaciones dentro de la sentencia *co* cumplen con ASV?. Justifique claramente.
b) Indique los resultados posibles de la ejecución

Nota 1: las instrucciones NO SON atómicas.

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

- Cumple ASV la asignacion y= y-1, ya que no posee referencia crítica y puede ser leído por otro proceso. Su resultado final siempre será 3.
- No cumple ASV la asignación z= x*2, ya que posee una única referencia crítica, pero es leído por otro proceso. Su resultado puede ser z = 4, 2, 0.
- No cumple ASV la asignación x = y-z, ya que posee 2 referencias críticas y además es leída en otro proceso. Sus posibles resultados son x= 1, 0, -1.

11- Defina acciones atómicas condicionales e incondicionales. Ejemplifique.

- Una **acción atómica** realiza una transformación de estado indivisible, o sea que mientras está siendo ejecutada ningún estado intermedio debe ser visible para los otros procesos. Una **acción atómica de grano fino** es implementada directamente sobre el hardware que ejecuta el programa concurrente.
En los programas secuenciales, las asignaciones aparecen como atómicas ya que ningún estado intermedio es visible al programa. Esto no sucede en los programas concurrentes, ya que una asignación con frecuencia es implementada por varias instrucciones de máquina de grano fino.
Una **acción atómica incondicional** no contiene una condición de demora B, por lo que puede ejecutarse inmediatamente.
Una **acción atómica condicional** es una sentencia AWAIT con una guarda B. Estas son condiciones de demora que hasta no ser true no puede ejecutarse la acción. Si B es false, sólo puede volverse true como resultado de acciones tomadas por otros procesos.

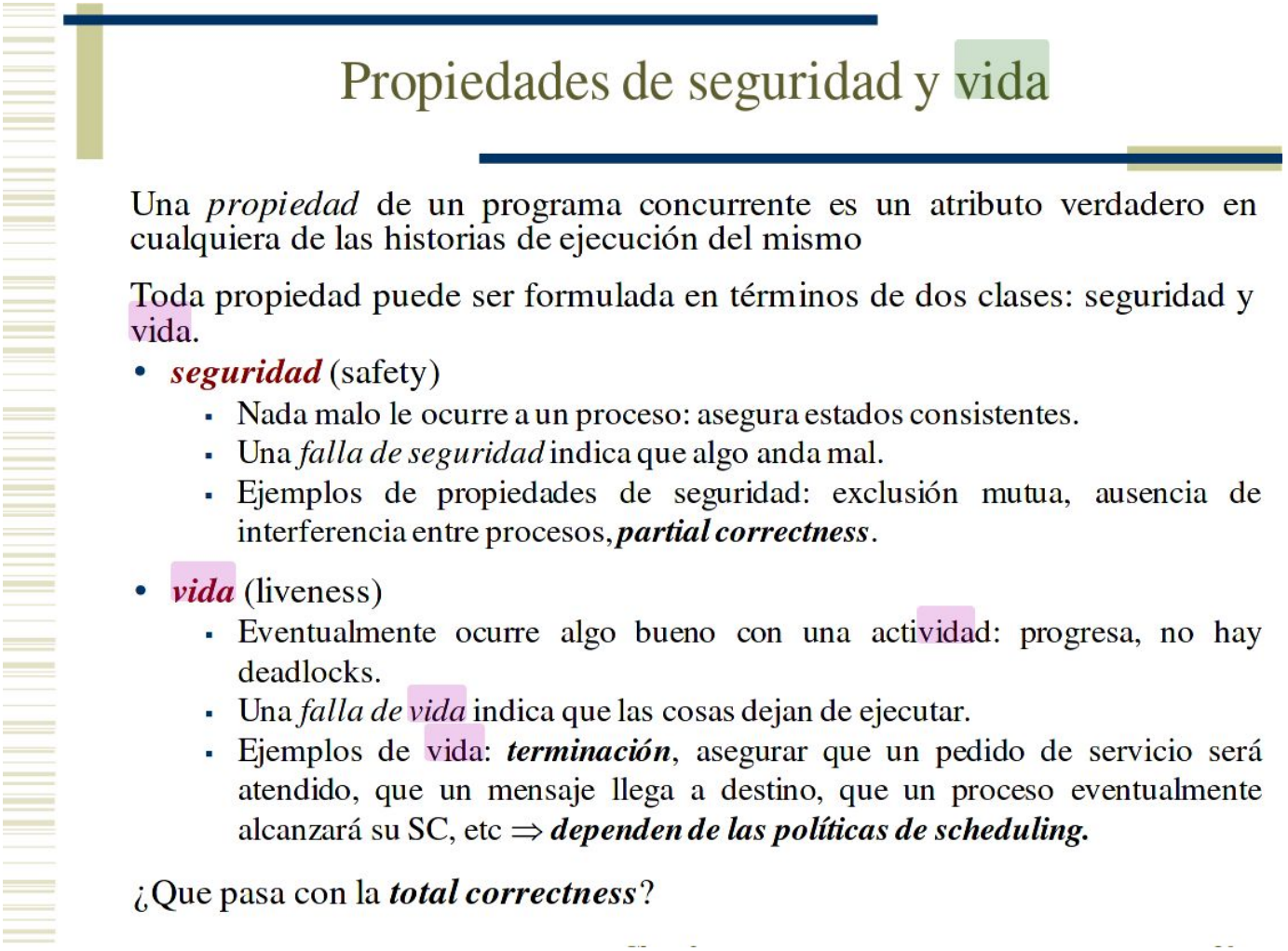
Las acciones atómicas condicionales, son aquellas acciones en los que hay una condición booleana para poder ejecutar la instrucción. Por ejemplo: *<await (s>0) s=s-1>* es atómica condicional, debido a que para poder ejecutar la sentencia, se debe esperar a que ‘s’ sea mayor a 0, y la instrucción que podría es atómica.

Las acciones atómicas incondicionales, son aquellas acciones en los que no hay una condición booleana para su ejecución. Por ejemplo: <x=x+1; y=y+1>. Mientras se ejecutan esas dos sentencias dentro del await, ningún otro proceso va a poder acceder a modificar los valores de x e y (exclusión mutua), por lo que se ejecuta de forma atómica.

12- Defina propiedad de seguridad y propiedad de vida.

- Una propiedad de un programa es un atributo que es verdadero en cualquiera de sus posibles historias (intercalado de acciones atómicas), por lo tanto en todas sus ejecuciones.
Como un programa concurrente está formado por múltiples procesos, una **historia** es cada *interleaving* de acciones atómicas que puede darse en una ejecución del programa.

Una **propiedad de seguridad** establece que el programa nunca entra en estado malo (Asegura estados consistentes). **Ejemplos** de esta propiedad la exclusión mutua, o sea que no haya más de un proceso en la misma sección crítica al mismo tiempo y ausencia de interferencias entre procesos (Partial correctness).
Una **propiedad de vida** establece que algo bueno eventualmente ocurrirá en la ejecución del programa (Progresa, no hay deadlocks). **Ejemplos** de estas propiedades son eventual entrada a la sección crítica, terminación del programa o que un mensaje llegue a destino. Estas propiedades dependen de la política de scheduling.



13- ¿Qué es una política de scheduling? Relacione con fairness. ¿Qué tipos de fairness conoces?

- El **fairness** es una característica de las políticas de scheduling, que trata de garantizar que los procesos tengan chance de avanzar sin importar lo que hagan los demás.
Como el avance está dado por la elección de la próxima sentencia atómica a ejecutar, el concepto se

relaciona con las *políticas de scheduling,* que son las que deciden cual es la próxima sentencia atómica a ejecutarse.

El fairness puede ser: incondicionalmente fair, débilmente fair, fuertemente fair.
Incondicionalmente fair: si toda acción atómica incondicional que es elegible eventualmente es ejecutada. Ej: round robin
Débilmente fair: si es incondicionalmente fair, y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve true y permanece así hasta que es vista por el proceso que ejecuta la acción atómica condicional.

- No es suficiente para asegurar que cualquier sentencia await elegible eventualmente se ejecuta: la guarda podría cambiar el valor (de false a true y nuevamente a false) mientras un proceso está demorado. Ej: round robin

Fuertemente fair: si es incondicionalmente fair, y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en true con infinita frecuencia.

Programación Concurrente 2020

Cuestionario guía - Clases Teóricas 3 y 4

1- ¿Por qué las propiedades de vida dependen de la política de scheduling? ¿Cómo aplicaría el concepto de fairness al acceso a una base de datos compartida por n procesos concurrentes?

Vida (liveness)
- Eventualmente ocurre algo bueno con una actividad: progresa, no hay deadlocks.
- Una falla de vida indica que las cosas dejan de ejecutar.
- Ejemplos de vida: terminación, asegurar que un pedido de servicio sera atendido, que un mensaje llega a destino, que un proceso eventualmente alcanzará su SC, etc ⇒ dependen de las políticas de scheduling

Débilmente fair: si es incondicionalmente fair, y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve true y permanece así hasta que es vista por el proceso que ejecuta la acción atómica condicional.

- No es suficiente para asegurar que cualquier sentencia await elegible eventualmente se ejecuta: la guarda podría cambiar el valor (de false a true y nuevamente a false) mientras un proceso está demorado. Ej: round robin

Weak Fairness. A scheduling policy is weakly fair if (1) it is unconditionally fair, and (2) every conditional atomic action that is eligible is executed eventually, assuming that its condition becomes true and then remains true until it is seen by the process executing the conditional atomic action.

In short, if ⟨ `await (B) S;` ⟩ is eligible and `B` becomes true, then `B` remains true at least until after the conditional atomic action has been executed. Round-robin and time slicing are weakly fair scheduling policies if every process gets a chance to execute. This is because any delayed process will eventually see that its delay condition is true.

Weak fairness is not, however, sufficient to ensure that any eligible `await` statement eventually executes. This is because the condition might change value—from false to true and back to false—while a process is delayed. In this case, we need a stronger scheduling policy.

Fuertemente fair: si es incondicionalmente fair, y toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en true con infinita frecuencia.

Strong Fairness. A scheduling policy is strongly fair if (1) it is unconditionally fair, and (2) every conditional atomic action that is eligible is executed eventually, assuming that its condition is infinitely often true.

A condition is infinitely often true if it is true an infinite number of times in every execution history of a (nonterminating) program. To be strongly fair, a scheduling policy cannot happen only to select an action when the condition is false; it must sometime select the action when the condition is true.

Que un proceso logre un estado bueno depende de que la política de scheduling le permita ejecutarse cuando las condiciones sean las adecuadas. Por ejemplo, una política *incondicionalmente fair* no aseguraría la *terminación* de un programa con acciones atómicas condicionales para todas sus historias posibles.

Página N° 18 → Libro Andrew

Se aplica el concepto de fairness al scheduling que se haga de la base de datos como recurso compartido:

1. Si el ingreso de un lector permite el ingreso de otros, aún si hay escritores esperando → débilmente fair: el ingreso permanente de lectores causaría inanición a los escritores.
2. Si la llegada de un escritor impide el acceso de nuevos procesos y es atendida en primer lugar al liberarse la BD → débilmente fair: el ingreso constante de escritores causaría inanición a los lectores.
3. Si los procesos son atendidos en orden FIFO con acceso concurrente de los lectores → fair: todos serán atendidos.

La última opción, aunque es fuertemente fair, produce un delay innecesario en los lectores que lleguen tras un escritor que espera. Sin embargo, dada la naturaleza del problema, de evitar dicho delay se caería en un esquema de prioridad con riesgo de inanición a los escritores.

2- Dado el siguiente programa concurrente, indique cuál es la respuesta correcta (justifique claramente)

```
int a = 1, b = 0;
    co
            ⟨await (b = 1) a = 0 ⟩
            // while (a = 1)
                    { b = 1; b = 0; }
    oc
```

a) Siempre termina
b) Nunca termina
c) Puede terminar o no

- La respuesta correcta es la c, ya que existen ambos casos. Podría terminar cuando se ejecuta la primer

asignación del while, el procesador se asigna a la ejecución de la sentencia await, la cual termina correctamente dejando la variable a = 0, permitiendo que el while corte el bucle. La otra posibilidad es que se ejecute completo el while siempre, impidiendo la ejecución de la sentencia await, haciendo que el programa no termine nunca ya que nunca sucedería que a=0

Va a depender de la política de scheduling, si es fuertemente fair sabemos que eventualmente va a terminar, en cambio, si es débilmente fair puede que termine o no, dependiendo de las sentencias que se ejecuten, teniendo posibles diferentes historias.

3- ¿Qué propiedades deben garantizarse en la administración de una sección crítica en procesos concurrentes?¿Cuáles de ellas son propiedades de seguridad y cuáles de vida?
En el caso de las propiedades de seguridad, ¿cuál es en cada caso el estado "malo" que se debe evitar?

- ***seguridad (safety)***
  - Nada malo le ocurre a un proceso: asegura estados consistentes.
  - Una falla de seguridad indica que algo anda mal.
  - Ejemplos de propiedades de seguridad: *exclusión mutua*, ausencia de interferencia entre procesos, partial correctness.
- ***vida (liveness)***
  - Eventualmente ocurre algo bueno con una actividad: progresa, *no hay deadlocks*.
  - Una falla de vida indica que las cosas dejan de ejecutar.
  - Ejemplos de vida: terminación, asegurar que un pedido de servicio será atendido, que un mensaje llega a destino, que un proceso eventualmente alcanzará su SC, etc ⇒ dependen de las políticas de scheduling.

Las propiedades que deben garantizarse en la administración de la sección crítica son:

- **Exclusión Mutua:** propiedad de seguridad. Establece que sólo un proceso debe estar en su sección crítica en un momento dado. El caso malo es que más de un proceso se encuentren en la misma sección crítica.

"El estado malo en este caso sería uno en el cual las acciones en las regiones críticas en distintos procesos fueran ambas elegibles para su ejecución." **Página N° 3 → Libro Andrew**

- **Ausencia de Deadlock:** propiedad de seguridad. Establece que si dos o más procesos quieren entrar en su sección crítica, al menos uno tendrá éxito. El caso malo es que ninguno de los procesos que intentan acceder a su sección crítica lo haga, quedando todos bloqueados.
- **Ausencia de demora innecesaria:** propiedad de seguridad. Establece que si un proceso trata de entrar a su sección crítica, y el resto está en su sección no crítica o terminaron, aquel no está impedido de entrar. El caso malo es que un proceso intenta acceder a su sección crítica y es bloqueado aún siendo el único que intenta acceder.
- **Eventual entrada:** propiedad de vida. Establece que un proceso que intenta acceder a su sección crítica tiene posibilidades de hacerlo, es decir eventualmente lo hará.

4- Resuelva el problema de acceso a sección crítica para N procesos usando un proceso coordinador. En este caso, cuando un proceso SC[i] quiere entrar a su sección crítica le avisa al coordinador, y espera a que éste le otorgue permiso. Al terminar de ejecutar su sección crítica, el proceso SC[i] le avisa al coordinador.

Desarrolle una solución de grano fino usando únicamente variables compartidas (ni semáforos ni monitores).

```
int permiso[1:N] = ([N] 0), aviso[1:N] = ([N] 0);

process SC[i = 1 to N] {
   SNC;
   permiso[i] = 1; # Protocolo
   while (aviso[i]==0) skip; # de entrada
   SC;
   aviso[i] = 0; # Protocolo de salida
   SNC;
}

process Coordinador {
   int i = 1;
   while (true) {
      while (permiso[i]==0) i = i mod N +1;
      permiso[i] = 0;
       aviso[i] = 1;
      while (aviso[i]==1) skip;
   }
}
```

5- ¿Qué mejoras introducen los algoritmos Tie-breaker, Ticket o Bakery en relación a las soluciones de tipo spin-locks?

SPIN-LOCKS: La solución coarse-grained vista emplea dos variables. Para generalizar la solución a n procesos, deberíamos usar n variables. Pero, hay solo dos estados que nos interesan: algún proceso está en su SC o ningún proceso lo está. Una variable es suficiente para distinguir entre estos dos estados, independiente del número de procesos.

- El problema de los algoritmos **spin-locks** es que no mantienen un orden de los procesos demorados, pudiendo alguno de ellos no entrar nunca **si** el **scheduling no** es **fuertemente fair**.
- En el algoritmo **Tie-Breaker** se decrementa prioridad al último en ingresar al protocolo de entrada. Requiere scheduling débilmente fair y no usa instrucciones especiales (es complejo y costoso en tiempo). Una variable por cada proceso para indicar que el proceso comenzó a ejecutar su protocolo de entrada a la SC, y una variable adicional para romper empates, indicando qué proceso fue el último en comenzar dicha entrada.

Algoritmo Tie-Breaker (2 procesos): protocolo de SC que requiere scheduling sólo débilmente fair y no usa instrucciones especiales ⇒ 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.

```
bool in1 = false, in2 = false;
int ultimo = 1;

process SC1 {
   while (true) {      ultimo = 1; in1 = true;
                       ⟨await (not in2 or ultimo==2);⟩
                       sección crítica;
                       in1 = false;
                       sección no crítica;
   }
}

process SC2 {
   while (true) {      ultimo = 2; in2 = true;
                       ⟨await (not in1 or ultimo==1);⟩
                       sección crítica;
                       in2 = false;
                       sección no crítica;
   }
}
```

- En el algoritmo **Ticket** se reparten números y se espera que sea el turno (potencial problema → valores de turno ilimitados). Los procesos toman un número mayor que el de cualquier otro que espera ser atendido; luego espera que todos los que tienen números más chicos hayan sido atendidos.

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );

{ TICKET: proximo > 0 ^ (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i]== proximo) ^
   (turno[i] >0) ⇒ (∀j: 1 ≤ j ≤ n, j ≠ i: turno[i] ≠ turno[j] ) )}

process SC [i: 1..n]
{ while (true)
   {  < turno[i] = numero;  numero = numero +1; >
      < await turno[i] == proximo; >
      sección crítica;
      < proximo = proximo + 1; >
      sección no crítica;
   }
}
```

- En el algoritmo **Bakery** cada proceso que intenta entrar recorre los números de los demás y se auto asigna uno mayor. Luego espera que sea el menor de los que esperan (esta solución no es implementable directamente). Los procesos se chequean entre ellos y no contra un global. Este algoritmo es más complejo, pero es fair y no requiere instrucciones especiales.

```
int turno[1:n] = ([n] 0);

{BAKERY: (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i] > 0) ∧ ( ∀j : 1 ≤ j ≤ n, j ≠ i:
turno[j] = 0 ∨ turno[i] < turno[j] ) ) }

process SC[i = 1 to n]
{  while (true)
       { turno[i] = 1;  //indica que comenzó el protocolo de entrada
         turno[i] = max(turno[1:n]) + 1;
         for [j = 1 to n st j != i]       //espera su turno
              while (turno[j] != 0)  and (  (turno[i],i) > (turno[j],j) ) →  skip;
         sección crítica
         turno[i] = 0;
         sección no crítica
       }
}
```

6- Modifique el algoritmo Ticket para el caso en que no se dispone de una instrucción Fetch and Add

```
int numero = 1, proximo = 1, turno[1:n] = ([n] 0);
bool lock = false;            # lock compartido
{TICKET: proximo > 0 ⋀ (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i]==
proximo) ⋀ (turno[i] >0) ⇒( ∀j : 1 ≤ j ≤ n, j ≠ i: turno[i] ≠ turno[j] ) ) }

process SC[i = 1 to n] {
  while (true) {
    seccion no critica;
    while (lock) skip;   # Protocolo
    while (TS(lock))     # de
      while (lock) skip; # Entrada (Test-and-TS)
    turno[i] = numero; numero = numero + 1; #FA(numero, 1)
    lock = false;        # Protocolo de Salida
    while (turno[i] <> proximo) skip;
    seccion critica;
    proximo = proximo + 1;
    seccion no critica;
  }
}
```

En este código al no contar con FA, se realizó la toma de turnos y el incremento de *numero* en una sección protegida por protocolos de entrada/salida utilizando TS. De no contarse con TS se podría implementar los protocolos con otro método -por ejemplo *Tie breaker*-, pero en ese caso quizá hubiese convenido utilizar dicho método para proteger la sección crítica inicial; la elección del algoritmo de ticket no fue la mejor opción en ese caso.

7- Analice las soluciones para las barreras de sincronización desde el punto de vista de la complejidad de la programación y de la performance.

8- (OPCIONAL). Implemente una butterfly barrier para 8 procesos usando variables compartidas.

8 procesos = 2^3 → puede implementarse *butterfly barrier*.
En una etapa *s* cada proceso sincroniza con uno a distancia *2^(s-1)*.

```
int arribo[1:n] = ([n] 0);

process Worker[i = 1 to n] {
int j, resto, distancia;
  while (true) {
    # codigo de tarea i
    # barrera en log2 8 = 3 etapas:
    for [s = 1 to 3] {
      #calcula la distancia y si debe sumarla o restarla
      distancia = 2^(s-1);
```

```
        #(0 < i mod 2^s <= distancia) → suma distancia, sino resta
        resto = i mod 2^s;
        if (resto=0)or(resto>distancia) then distancia = -distancia;
        j = i + distancia;
        while (arribo[i] != 0) skip;
        arribo[i] = 1;
        while (arribo[j] != 1) skip;
        arribo[j] = 0;
      }
    }
  }
```

9- a) Explique la semántica de un semáforo.

Un semáforo es utilizado para implementar exclusión mutua y sincronización por condición. Se relaciona a un valor entero y dos operaciones atómicas para manipularlo:

1. **P, wait o decremento:** Si el valor del semáforo es positivo, lo decrementa; sino demora al proceso hasta que se satisfaga dicha condición y luego decrementa el valor y continúa la ejecución. Equivale a:
   `<await (valor>0) valor = valor - 1>`
2. **V, signal o incremento:** Incrementa el valor del semáforo, "despertando" a alguno de los procesos demorados si los hubiere. Equivale a:
   `<valor = valor + 1>`

b) Indique los posibles valores finales de *x* en el siguiente programa (**justifique claramente su respuesta**): int x = 4; sem s1 = 1, s2 = 0;

```
co P(s1); x = x * x ; V(s1);
   // P(s2); P(s1); x = x * 3; V(s1):
   // P(s1); x = x - 2; V(s2); V(s1);
oc
```

Opcion 1

P1 y P3 comienzan esperando a s1. Por ser un mutex, sólo puede continuar uno de ellos y no será interrumpido por el otro hasta liberar a s1.

1. Si comienza P1: Asigna x=9, luego incrementa s1 permitiendo que continúe P3. P3 asigna x=7 y señala s2. Esto habilita a P2 que estaba esperando. Si P2 continúa, intentará obtener s1 con lo cual se vuelve a bloquear volviendo el control a P3. En cualquier caso, P3 libera a s1 y termina. P2 es despertado, asigna x = 21 y termina. Valor final **x=21**.
2. Si comienza P3: Asigna x = 1 y señala a s2. Esto habilita a P2 que estaba esperando. Si P2 continúa, intentará obtener s1 con lo cual se vuelve a bloquear volviendo el control a P3. Cuando P3 libera a s1, P1 y P2 pueden competir por él:
   1 Si gana P1: asigna x=1, libera a s1 y termina; finaliza P2 y asigna x = 3. Valor final x=3.

2 Si gana P2: asigna x=3, libera a s1 y termina; finaliza P1 y asigna x = 9. Valor final x=9.

P2 nunca puede comenzar la historia ya que espera un semáforo de señalización que sólo P3 señala. Cualquier historia en la que P2 esté antes de P3 es inválida.
En todas las historias los semáforos terminan con los mismos valores con los que están inicializados.

Opcion 2

- Los posibles valores de x son 42, 12, 36. Al utilizar semáforos (correctamente) no se presenta problemas de consistencia en los valores.
El resultado final 42 se da de la ejecución de los procesos de la siguiente manera: 1er co, 3er co, 2^{o} co; el resultado 12 se da por la siguiente ejecución: 3er co, 1er co, 2^{o} co; y el número 36 se obtiene de la ejecución de 3er co, 2^{o} co, 1er co.

10- Desarrolle utilizando semáforos una solución centralizada al problema de los filósofos, con un administrador único de los tenedores, y posiciones libres para los filósofos (es decir, cada filósofo puede comer en cualquier posición siempre que tenga los dos tenedores correspondientes).

a) Utilizando semáforos

```
int posicion[1:5];
int pedidos[1:5], libre = 1;
sem mutexCola = 1, mutexSentados = 1, pedido = 0, permiso[1:5] = ([5] 0);
bool tenedores[1:5] = ([5] false);     # todos libres inicialmente
bool administradorEspera = false;
int sentados = 0;

process Filosofo[i = 1 to 5] {
  while(true) {
    P(mutexCola);                                    # Request
    pedidos[libre] = i;
    libre = libre mod 5 +1;
    V(mutexCola);
    V(pedido);
    P(permiso[i]);

    ComerEn(posicion[i]);

    tenedores[posicion[i]] = false;                  # Release
    tenedores[posicion[i] mod 5 +1] = false;
    P(mutexSentados);
    sentados = sentados - 1;
    if(administradorEspera) {
      administradorEspera = false;
      V(YaComio);
    } else {
      V(mutexSentados);
    }

    Pensar();
  }
}

process Administrador {
  int fil, ocupado = 1, pos = 1;
  while(true) {
    P(pedido);
    fil = pedidos[ocupado];
    ocupado = ocupado mod 5 +1;
    P(mutexSentados);
    if(sentados==2) {
      administradorEspera = true;
      V(mutexSentados);
      P(YaComio);
    }
    sentados = sentados + 1;
    V(mutexSentados);
    while(tenedores[pos] or tenedores[pos mod 5 +1])
      pos = pos mod 5 +1;
    posicion[fil] = pos;
    tenedores[pos] = true;
    tenedores[pos mod 5 +1] = true;
    V(permiso[fil]);
  }
}
```

11- Describa la técnica de *Passing the Baton*.

- Passing the Baton es una técnica general para implementar sentencias AWAIT. Cuando un proceso está dentro de la SC, éste mantiene el baton (o token) que equivale al permiso para ejecutar.
Cuando el proceso llega a un SIGNAL, pasa el baton (control) a otro proceso. Si no hay procesos esperando el baton (o sea entrar a la SC), entonces se libera para que lo tome el próximo que trata de entrar.

Passing the baton proveer exclusión y controlar cuál proceso demorado es el próximo en seguir. Puede usarse para implementar sentencias await arbitrarias y así implementar sincronización por condición arbitraria. La técnica también puede usarse para controlar precisamente el orden en el cual los procesos demorados son despertados.
El funcionamiento es el siguiente: cuando un proceso está ejecutando su sección crítica, puede pensarse que posee la "posta" o "baton", esto significa que tiene permiso de ejecución. Cuando este proceso termina, pasa la posta a otro. Si alguno está esperando por una condición que es ahora verdadera, la posta se le otorga al mismo. Este a su vez, ejecuta su SC y la pasa a otro. Cuando no hay procesos esperando por la condición
verdadera, la posta se pasa al próximo proceso que trate de entrar en su SC por primera vez.

Passing the batton se utiliza en semáforos, mientras que passing the condition con monitores.

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

- Presenta una forma sencilla y sistemática de implementar cualquier tipo de sincronización entre procesos, con o sin condiciones, utilizando semáforos para una solución de grano fino. Los programas resultantes son fáciles de modificar y extender.

12- Modifique las soluciones de Lectores-Escritores con semáforos de modo de no permitir más de 10 lectores simultáneos en la BD y además que no se admita el ingreso a más lectores cuando hay escritores esperando.

a) Resuelva con semáforos

```
# contadores de procesos trabajando y demorados por cada tipo
int nr = 0, nw = 0, dr = 0, dw = 0;
# mutex para sección crítica y semáforos de señalización
sem e = 1, r = 0, w = 0;

process Lector [i = 1 to M] {
  while(true) {
    P(e);
    if (nw>0 or dw>0 or nr==10) {dr = dr+1; V(e); P(r);}
    nr = nr + 1;
    if (dr>0 and dw==0 and nr<10) {dr = dr – 1; V(r); }
      else V(e);
    leerBD();
    P(e);
    nr = nr – 1;
    if (nr==0 and dw>0) {dw = dw – 1; V(w); }
      elseif (dw==0 and dr>0 and nr<10) {dr = dr – 1; V(r);}
      else V(e);
  }
}

process Escritor [j = 1 to N] {
  while(true) {
    P(e);
    if (nr>0 or nw>0) {dw = dw+1; V(e); P(w); }
    nw = nw + 1;
    V(e);
    escribirBD();
    P(e);
    nw = nw – 1;
    if (dw>0) {dw = dw - 1; V(w);}
      elseif (dr>0) {dr = dr - 1; V(r);}
      else V(e);
  }
}
```

13- (OPCIONAL) 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 usando semáforos

a) Desarrolle una solución usando semáforos

```
sem okLeer = 0, okEscribir = 1, barrera = 0, mutex = 1;
int cont = 0;
int BD;              # datos compartidos

process Productor {
  while(true) {
    P(okEscribir);
    BD = CalcularValor();
    for[c = 1 to N] V(okLeer);
  }
}

process Consumidor[i = 1 to N] {
  while(true){
    P(okLeer);
    Procesar(BD);

    #esperan a haber leido todos
    P(mutex);
    cont = cont + 1;
    if(cont == N)
      for[c = 1 to N] V(barrera);
    V(mutex);
    P(barrera);

    # una vez que leyeron todos, uno avisa al productor (preciso esperar que
    # todos salgan de la barrera para evitar race conditions)
    P(mutex);
    cont = cont - 1;
    if(cont==0) V(okEscribir);
    V(mutex);
  }
}
```

b) Suponga que el buffer tiene *b* slots. El productor puede depositar mensajes sólo en slots vacios 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.

```
sem okLeer[1:N] = ([N] 0), okEscribir = b, mutex[1:b] = ([b] 1);
int cont[1:b] = ([b] 0);
int BD[1:b];              # datos compartidos

process Productor {
  int libre = 1;
  while(true) {
    P(okEscribir);
    cont[libre] = 0;
    BD[libre] = CalcularValor();
    for[c = 1 to N] V(okLeer[c]);
    libre = libre mod b +1;
  }
}

process Consumidor[i = 1 to N] {
  int consumir = 1;
  while(true){
    P(okLeer[i]);
    Procesar(BD[consumir]);
```

```
    P(mutex[consumir]);
    cont[consumir] = cont[consumir] + 1;
    if(cont[consumir] == N) V(okEscribir);
    V(mutex[consumir]);
    consumir = consumir mod N +1;
  }
}
```

En la resolución del punto *a* utilizamos una barrera para sincronizar a los consumidores. El contador era incrementado antes de la barrera y decrementado luego para evitar *race conditions*. Al tener que extender la resolución, notamos que el uso de semáforos privados facilitaría la tarea, logrando el mismo efecto. Se pudieron utilizar también en el punto *a* simplificando el código (aunque con mayor uso de memoria por el arreglo de semáforos).

14- (OPCIONAL) Implemente una butterfly barrier de *n* procesos usando semáforos (siendo *n* potencia de 2)

```
sem arribo[1:n] = ([n] 0);
process Worker[i = 1 to n] {
  int j, resto, distancia;
  while (true) {
    ...                                                  # codigo de tarea i
    # barrera
    for [s = 1 to log2(n)] { # calcula la distancia y si debe sumarla o restarla

      distancia = 2^(s-1);
      # (0 < i mod 2^s <= distancia) → suma distancia, sino resta
      resto = i mod 2^s;
      if (resto=0)or(resto>distancia) distancia = -distancia;
      j = i + distancia;
      V(arribo[i]);
      P(arribo[j]);
    }
  }
}
```

5/10/2020
Programación Concurrente 2020

Cuestionario guía - Clases Teóricas 5 y 6

1- Describa el funcionamiento de los monitores como herramienta de sincronización.

Los monitores son módulos de programa con más estructura, y que pueden ser implementados tan eficientemente como los semáforos.
Es un mecanismo de abstracción de datos:

- Encapsulan las representaciones de recursos.
- Brindan un conjunto de operaciones que son los únicos medios para manipular esos recursos.

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

- La única forma de comunicar datos entre monitores o entre un proceso y un monitor es por medio de invocaciones al procedimiento del monitor del cual se quieren obtener (o enviar) los datos.

```
monitor NombreMonitor {
  declaraciones de variables permanentes;
  código de inicialización

  procedure op1 (par. formales1)
    {  cuerpo de op1
    }
  .....
  procedure opn (par. formalesn)
    {  cuerpo de opn
    }
}
```

cola → FIFO : First in First out | LIFO - pilas

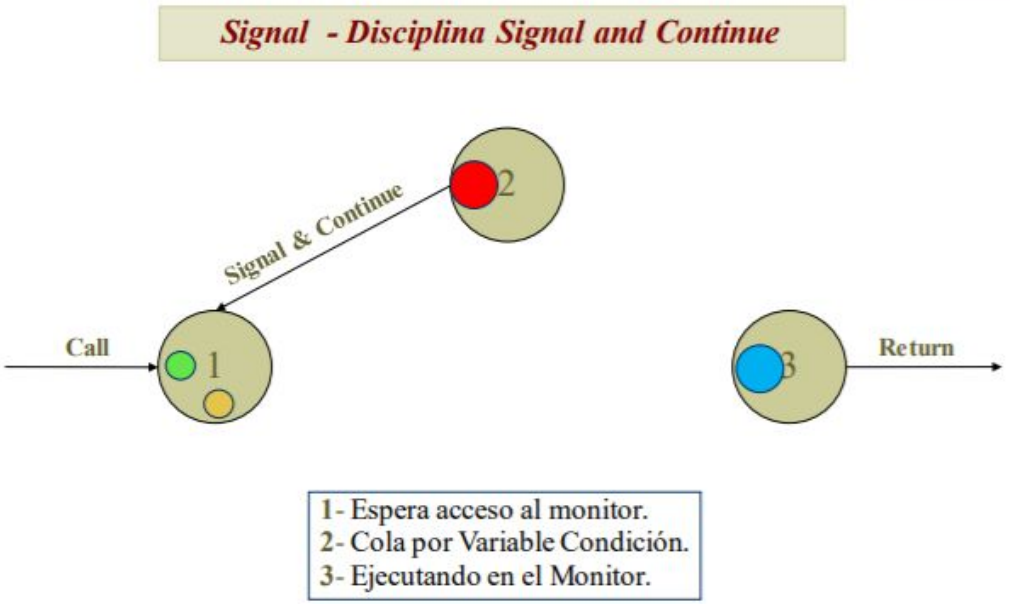
La exclusión mutua es implícita asegurando que los procedures en el mismo monitor no se ejecutan concurrentemente.
La sincronización por condición es explícita con variables condición → cond cv;
El valor asociado a cv es una cola de procesos demorados, no visible directamente al programador. Las operaciones sobre las variables condición son:

- ***wait(cv)*** → el proceso se demora al final de la cola de cv y deja el acceso exclusivo al monitor.
- ***signal(cv)*** → despierta al proceso que está al frente de la cola (si hay alguno) y lo saca de ella. El proceso despertado recién podrá ejecutar cuando readquiera el acceso exclusivo al monitor.
- ***signal_all(cv)*** → despierta todos los procesos demorados en cv, quedando vacía la cola asociada a cv.

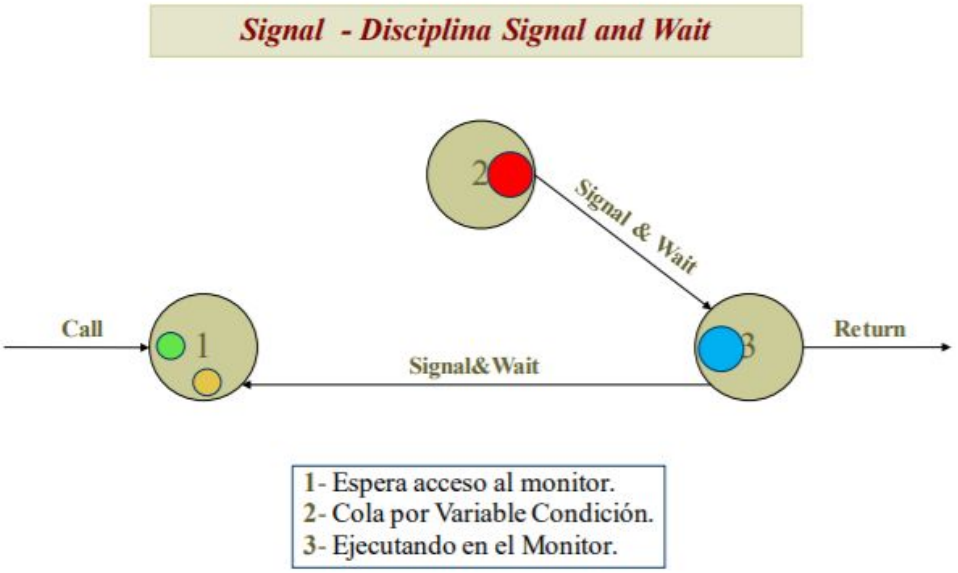
Las disciplinas de señalización son: el signal and continue que es el usado en la cátedra y el signal and wait.

2- ¿Qué diferencias existen entre las disciplinas de señalización "Signal and wait" y "Signal and continue"?

Signal and continue: el proceso que ejecuta SIGNAL retiene el control exclusivo del monitor y puede seguir ejecutando, mientras el proceso despertado pasa a competir por acceder nuevamente al monitor y continuar su ejecución en la instrucción siguiente al wait (Unix, Java, Pthreads).



Signal and wait: el proceso que hace SIGNAL pasa a competir por acceder nuevamente al monitor, mientras que el despertado pasa a ejecutar dentro del monitor en la instrucción siguiente al wait.



3- ¿En qué consiste la técnica de *Passing the Condition* y cuál es su utilidad en la resolución de problemas con monitores?¿Qué relación encuentra entre *passing the condition* y *passing the baton*?

- Passing the condition trata de señalar (signal) una variable condición que tenga procesos esperando para ejecutar pero sin volver verdadera la condición asociada; o cambiando a verdadera la condición asociada, pero sin señalar (signal) la variable condición, si no hay procesos esperando para ejecutar.
De esta manera se asegura que el siguiente proceso en ejecutar su SC es el despertado de la lista, y no otro que gane el acceso al monitor, ya que la condición será falsa y pasará a la cola de espera.

Passing the baton permite que un proceso pase a otro demorado el control (mutex *e*) y que éste sea el único que pueda ejecutar secciones exclusivas; *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 por *passing the baton* se ejecutará antes que el resto de los que competían por el mutex; el proceso despertado por *passing the condition* se ejecutará antes que el resto de los que requieran la condición (tal vez tomen control del monitor, pero se bloquearán).
Aunque *passing the baton* puede utilizarse para cualquier tipo de sincronización, *passing the condition* 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).

passing the baton. Esta técnica emplea semaforos binarios divididos para proveer exclusión y controlar cuál proceso demorado es el próximo en seguir. Puede usarse para implementar sentencias await arbitrarias y así implementar sincronización por condición arbitraria. La técnica también puede usarse para controlar precisamente el orden en el cual los procesos demorados son despertados.

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.

- Ambas técnicas pasan el control a un proceso demorado que será despertado para continuar su ejecución de la SC. La diferencia está en que passing the baton pasa el control a otro proceso para que sea el único que pueda ejecutar SC; mientras que passing the condition le avisa a un proceso que están dadas las condiciones para que pueda seguir ejecutando y este sea el único que pueda continuar de todos los que esperan dicha condición.

Semáforos binarios divididos, 0 o 1, si la suma de los semáforos está entre 0 y 1. Los puedo ver como un único semáforo binario. Van a servir para implem excl mutua, me asegura que lo q esta entre un sem y otro se ejecuta con em.

4- Desarrolle utilizando monitores una solución centralizada al problema de los filósofos, con un administrador único de los tenedores, y posiciones libres para los filósofos (es decir, cada filósofo puede comer en cualquier posición siempre que tenga los dos tenedores correspondientes).

5- 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?
NO funciona correctamente. Cuando un proceso realiza un *release*, libera el recurso y *signal(turno)* pasa al primer proceso de la cola de espera de *turno* a la cola de entrada al monitor, donde competirá por el monitor con los procesos que allí se encuentren. Si antes que logre control del monitor, otro proceso realiza un *request*, encontrará *libre* en true y utilizará el recurso (posiblemente rompiendo la política SJN, si es de duración mayor al despertado); cuando el proceso despertado tome control del monitor volvería a setear *libre* en false, quedando los dos procesos con control sobre el recurso e interfiriendo entre sí.
Para evitar esta situación se debería volver a controlar si el recurso está en uso para volver a dormir al proceso, convirtiendo al *if* del *request* en un *while*. Otra solución, sería emplear *passing the condition*:

b) Funciona correctamente con disciplina de señalización Signal and Wait?
SI :D
SÍ funciona correctamente. El problema del caso anterior no existe ya que el *signal(turno)* en el *release* pasa el control del monitor directamente al primer proceso de la cola de *turno*, si lo hubiere.

EXPLIQUE CLARAMENTE SUS RESPUESTAS

6- Modifique la solución anterior para el caso de no contar con una instrucción wait con prioridad.
cola ordenada por tiempo. se despierta al primero de la cola.

```
monitor SJN {
  bool libre = true;
  cond turno;
  cola colaOrdenadaPorTiempo;
  procedure request(int tiempo) {

        if (not libre){
              //llamo prod para ordenar la cola
              wait(turno);
        }
        libre = false;
  }
  procedure release() {
        libre = true
        signal(turno);
  }
}
```

7- Modifique utilizando monitores las soluciones de Lectores-Escritores de modo de no permitir más de 10 lectores simultáneos en la BD, y además que no se admita el ingreso a más lectores cuando hay escritores esperando.

```
monitor Controlador_RW {
  int nr = 0, nw = 0;
ew = 0;
er = 0
  cond okLeer;
  cond okEscribir;

  procedure pedido_leer() {
    while ((nw>0)or(ew > 0)or(nr==10) {
         er = er + 1; // incremento la cantidad de lectores esperando y luego ZZZ
         wait(okLeer);
         }
         nr = nr + 1;
    if(nr<10) { signal(okLeer); }
  }

  procedure libera_leer() {
    nr = nr - 1;
    if(nr==0)and(ew > 0) {
         ew = ew -1
         signal(okEscribir);
     }
      elseif((ew = 0) and nr<10) and er > 0) {
         er = er -1
         signal(okLeer); }
  }

  procedure pedido_escribir() {
    while (nr>0 OR nw>0) {
        ew = ew + 1;
       wait(okEscribir);
       }
    nw = nw + 1;
  }

  procedure libera_escribir() {
    nw = nw - 1;
    if(ew>0)) {
         ew = ew -1
         signal(okEscribir); }
      else {
          if (er > 0 ){
                signal(okLeer);
                }
         }
  }
}
```

Otra solución al problema de lectores y escritores

```
monitor Controlador_RW
{  int nr = 0, nw = 0, dr = 0, dw = 0;
   cond  ok_leer, ok_escribir

   procedure pedido_leer( )
    { if (nw > 0)
                { dr = dr +1;
                  wait (ok_leer);
                }
      else  nr = nr + 1;
    }

   procedure libera_leer( )
    { nr = nr - 1;
      if (nr == 0 and dw > 0)
                { dw = dw -1;
                  signal (ok_escribir);
                  nw = nw +1;
                }
    }

   procedure pedido_escribir( )
     { if (nr>0 OR nw>0)
                { dw = dw +1;
                  wait (ok_escribir);
                }
       else  nw = nw + 1;
     }

   procedure libera_escribir( )
    { if (dw > 0)
                { dw = dw -1;
                  signal (ok_escribir);
                }
      else { nw = nw -1;
             if (dr > 0)
                  { nr = dr;
                    dr = 0;
                    signal_all (ok_leer);
                  }
           }
     }
}
```

8- Resuelva con monitores el siguiente problema: 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 ítems al final de la lista; las inserciones deben ser mutuamente exclusivas para evitar insertar dos ítems casi al mismo tiempo. Sin embargo, un insert puede hacerse en paralelo con uno o más searches. Por último, los deleters remueven ítems de cualquier lugar de la lista. A lo sumo un deleter puede acceder la lista a la vez, y el borrado también debe ser mutuamente exclusivo con searches e inserciones.

```
monitor Acceso_Lista {

 cond okSearch;
 cond okInsert;
 cond okDelete;
 int nS=0, nI=0, nD=0;

 procedure pedidoSearch() {
        while(nD>0) wait(okSearch);
                nS = nS + 1;
}

 procedure liberaSearch() {
        nS = nS – 1;
        if(nS==0 AND nI==0 AND nD==0)
                signal(okDelete);
}

 procedure pedidoInsert() {
        while(nI==1 or nD>0) wait(okInsert);
        nI = nI + 1;
}

 procedure liberaInsert() {
```

```
        nI = nI – 1;
        # no puede haber mas de un insert en paralelo, asi que nI == 0
        if(nD==0){
                signal(okInsert);
                if(nS==0)
                        signal(okDelete);
        }
}
procedure pedidoDelete() {
        while(nD==1 or nS>0 or nI>0)
                wait(okDelete);
        nD = nD + 1;
}

procedure liberaDelete() {
        nD = nD – 1;
        # no puede haber mas de un delete en paralelo, asi que nD == 0
        signal_all(okSearch);
        if(nI==0) signal(okInsert);
                if(nS==0) signal (okDelete);
}
}
```

OPCION 2

```
monitor Acceso_Lista {
# BAD:     nI>1   OR (nD>0  AND (nS>0  OR  nI>0 ))  OR nD>1
# NOT BAD: nI<=1 AND (nD==0 OR  (nS==0 AND nI==0)) AND nD<=1
  cond okSearch;    # signal cuando nD==0
  cond okInsert;    # signal cuando nD==0 AND nI==0
  cond okDelete;    # signal cuando nS==0 AND nI==0 AND nD==0
  int nS=0, nI=0, nD=0;

  procedure pedidoSearch() {
    while(nD>0) wait(okSearch);
    nS = nS + 1;
  }
  procedure liberaSearch() {
    nS = nS - 1;
    if(nS==0 AND nI==0) signal(okDelete);
  }

  procedure pedidoInsert() {
    while(nI==1 or nD>0) wait(okInsert);
    nI = nI + 1;
  }
  procedure liberaInsert() {
    nI = nI - 1;
    if(nS==0) signal(okDelete);
    signal(okInsert);
  }

  procedure pedidoDelete() {
    while(nD==1 or nS>0 or nI>0) wait(okDelete);
    nD = nD + 1;
  }
  procedure liberaDelete() {
    nD = nD - 1;
    signal_all(okSearch);
    signal(okInsert);
    signal(okDelete);
```

```
  }
}
```

9- El problema del "Puente de una sola vía" (*One-Lane Bridge*): autos que provienen del Norte y del Sur llegan a un puente con una sola vía. Los autos en la misma dirección pueden atravesar el puente al mismo tiempo, pero no puede haber autos en distintas direcciones sobre el puente.

a) Desarrolle una solución al problema, modelizando los autos como procesos y sincronizando con un monitor (no es necesario que la solución sea fair ni dar preferencia a ningún tipo de auto).

b) Modifique la solución para asegurar fairness (Pista: los autos podrían obtener turnos)

10- OPCIONAL: 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 usando monitores

a) Broadcast con buffer unitario

```
monitor BD {
  int dato, cont = N, leido[1:N] = ([N] 1);
  cond okEscribir; # signal cuando leido[] sea todos 1
  cond okLeer;     # signal cuando leido[] sea todos 0

  procedure escribir(int d) {
    if(cont<>N) wait(okEscribir);
    dato = d;
    cont = 0;
    for [c = 1 to N] leido[c] = 0;
    signal_all(okLeer);
  }

  procedure leer(int pid, int &d) {
    if(leido[pid] == 1) wait(okLeer);
    d = dato;
    cont = cont + 1;
    if(cont == N) signal(okEscribir);
  }
}
```

b) Suponga que el buffer tiene *b* slots. El productor puede depositar mensajes sólo en slots vacios 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.

b) Broadcast con buffer de b posiciones

```
monitor BD {
  int dato[1:b], cont[1:b]=([b] N), consumir[1:N]=([N] 1), libre=1;
  cond okEscribir;   # signal cuando cont[libre] = N
  cond okLeer;       # signal cuando hay dato nuevo

  procedure escribir(int d) {
    if(cont[libre]<>N) wait(okEscribir);
    dato[libre] = d;
    cont[libre] = 0;
    signal_all(okLeer);
    libre = libre mod b +1;
  }

  procedure leer(int pid, int &d) {
    if(cont[ consumir[pid] ] == N) wait(okLeer);
    d = dato[ consumir[pid] ];
    cont[ consumir[pid] ] = cont [ consumir[pid] ] + 1;
    if(cont[ consumir[pid] ] == N) signal(okEscribir);
```

```
        consumir[pid] = consumir[pid] mod b +1;
      }
    }
```

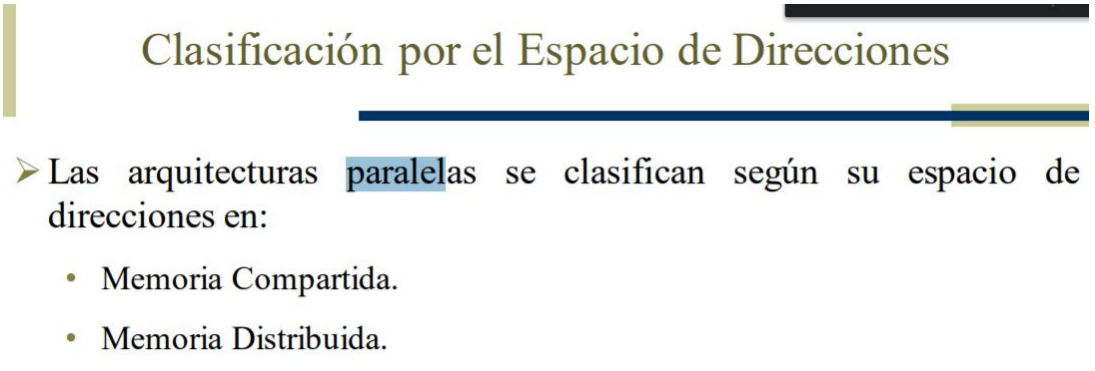
11- Investigue sobre implementaciones de mecanismos del tipo semáforos y/o monitores en lenguajes de programación (Pthreads, Java, otros).

27/10/2020
Programación Concurrente 2020

Cuestionario guía - Clases Teóricas 7 a 11

1- Defina y diferencie programa concurrente, programa distribuido y programa paralelo.

Un programa concurrente consiste en un conjunto de tareas o procesos secuenciales que pueden ejecutarse intercalándose en el tiempo y que cooperan para resolver un problema. Básicamente, es un concepto de software que dependiendo de la arquitectura subyacente da lugar a las definiciones de programación paralela o distribuida.
Un programa concurrente es un programa que tiene más de un hilo de ejecución o flujo de control, es decir, está dividido en tareas que pueden ejecutarse en paralelo o simultáneamente. No depende de un número determinado de procesadores ni de una arquitectura particular.
La programación paralela consiste en un programa concurrente que se ejecuta sobre múltiples procesadores que pueden tener una memoria compartida y que son utilizados para incrementar la performance de un programa concurrente.
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. Los procesos se interconectan por una red de comunicaciones, para interactuar se comunican mediante pasaje de mensajes.



2- Marque al menos 2 similitudes y 2 diferencias entre los pasajes de mensajes sincrónicos y asincrónicos.

link: un emisor, un receptor.
input port: un receptor y muchos emisores.
mailbox: cualquier proceso puede enviar o recibir a través de un canal.

similitudes:existen instrucciones de envío y recepción. la recepción es bloqueante para ambos

diferencias: pma no se bloquea en el envio y pms sí, hasta que el mensaje es recibido. PMA ⇒ Mailbox

PMS ⇒ Link

La diferencia entre PMA y PMS es la primitiva de trasmisión Send. En PMS es bloqueante y la llamaremos (por ahora) sync_send.

• El trasmisor queda esperando que el mensaje sea recibido por el receptor.

• La cola de mensajes asociada con un send sobre un canal se reduce a 1 mensaje ⇒ menos memoria.

• Naturalmente el grado de concurrencia se reduce respecto de la sincronización por PMA (los emisores se bloquean).

Si bien send y sync_send son similares (en algunos casos intercambiables) la semántica es diferente y las posibilidades de deadlock son mayores en comunicación sincrónica.

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

3- Analice qué 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 claramente su respuesta.

Cliente/servidor:PMA ya que el cliente envía y no se bloquea, deposita el mensaje en una cola la cual el servidor en algún momento desencolará y atenderá. Además a la hora de responder al cliente, el servidor podrá responder individualmente sin necesidad de que el cliente reciba (servidor no se bloquea)

Pares que interactúan: PMS. pensar en los modelos de arquitectura (centralizado, simétrico, anillo circular). Cada proceso tiene un dato local que los N procesos deben conocer.

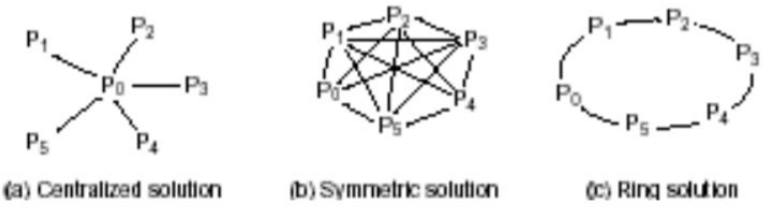
Productores y consumidores PMA porque si el productor funciona a una velocidad diferente a la que funciona el consumidor, con PMS existiría demora. Con PMA el productor encola y sigue produciendo sin esperar a que el consumidor consuma.O tb se podria usar PMS con un buffer intermedio.

Pares (peers) interactuantes.

Intercambio de Valores

Problema: cada proceso tiene un dato local V y los N procesos deben saber cuál es el menor y cuál el mayor de los valores.

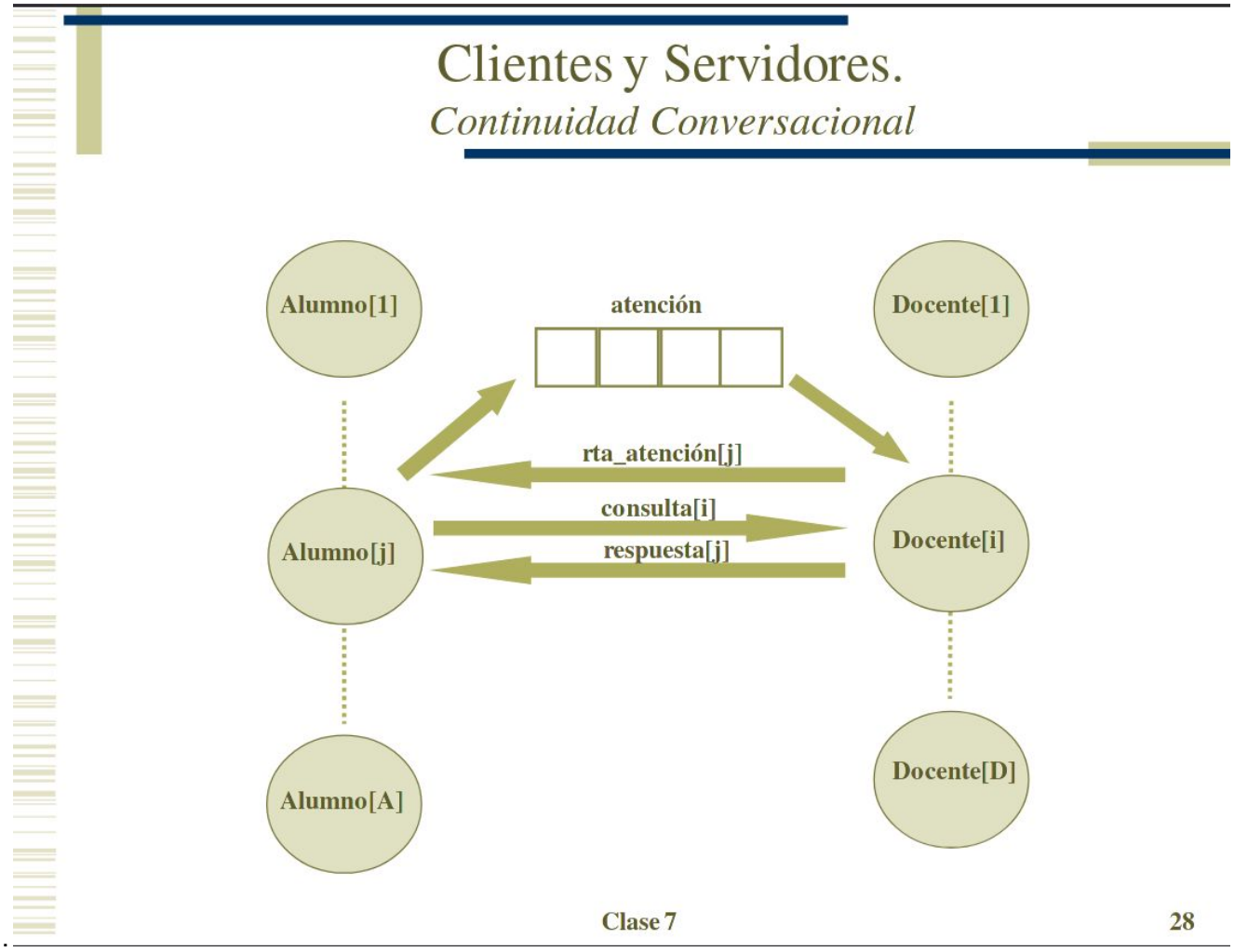
Ejemplo donde los procesadores están conectados por tres modelos de arquitectura: ***centralizado***, ***simétrico*** y en ***anillo circular***.



(a) Centralized solution (b) Symmetric solution (c) Ring solution

4- Defina el concepto de "continuidad conversacional" entre procesos.

Es un modo de interacción entre clientes y servidores, en los cuales contamos con un canal global, donde los clientes realizan las peticiones para ser atendidos y éstos recibirán su respuesta por un canal propio con el servidor.



5- Indique por qué puede considerarse que existe una dualidad entre los mecanismos de monitores y pasaje de mensajes. Ejemplifique

Existe dualidad entre monitores y PM porque cada uno de ellos puede simular al otro. Por ejemplo los simularemos usando procesos servidores y PM, como procesos activos en lugar de como conjuntos pasivos de procedures.

Dualidad entre Monitores y Pasaje de Mensajes

Programas con Monitores	**Programas basados en PM**
Variables permanentes Identificadores de procedures Llamado a procedure Entry del monitor Retorno del procedure Sentencia wait Sentencia signal Cuerpos de los procedure	Variables locales del servidor Canal request y tipos de operación send request( ); receive respuesta receive request( ) send respuesta( ) Salvar pedido pendiente Recuperar/ procesar pedido pendiente Sentencias del “case” de acuerdo a la clase de operación.

- El monitor y el Servidor muestran la dualidad entre monitores y PM: hay una correspondencia directa entre los mecanismos de ambos.
- La eficiencia de monitores o PM depende de la arquitectura física de soporte:
  - Con MC conviene la invocación a procedimientos y la operación sobre variables condición.
  - Con arquitecturas físicamente distribuidas tienden a ser más eficientes los mecanismos de PM.
- Dualidad entre Monitores y Pasaje de Mensajes

Programas con Monitores		***Programas basados en PM***
• Variables permanentes	↔	Variables locales del servidor
• Identificadores de procedures	↔	Canal *request* y tipos de operación
• Llamado a procedure	↔	*send request( ); receive respuesta*
• Entry del monitor	↔	*receive request( )*
• Retorno del procedure	↔	*send respuesta( )*
• Sentencia *wait*	↔	Salvar pedido pendiente
• Sentencia *signal*	↔	Recuperar/ procesar pedido pendiente
• Cuerpos de los procedure	↔	Sentencias del "case" de acuerdo a la clase de operación.

CONSIDERACIONES PARA RESOLVER LOS EJERCICIOS DE PMA:

- Los canales son compartidos por todos los procesos.
- Cada canal es una cola de mensajes, por lo tanto el primer mensaje encolado es el primero en ser atendido.
- Por ser pasaje de mensajes asincrónico el **send** no bloquea al emisor.
- Se puede usar la sentencia **empty** para saber si hay algún mensaje en el canal, pero no se puede consultar por la cantidad de mensajes encolados.
- Se puede utilizar el **if/do** no determinístico donde cada opción es una condición boolena donde se puede preguntar por variables locales y/o por **empty** de canales.

```
if   (cond 1) -> Acciones 1;
(cond 2) -> Acciones 2;                    ….
                (cond N) -> Acciones N;                    end if
```

De todas las opciones cuya condición sea Verdadera elige una en forma no determinística y ejecuta las acciones correspondientes. Si ninguna es verdadera sale del if/do si hacer nada.

- Se debe tratar de evitar hacer **busy waiting** (sólo hacerlo si no hay otra opción).
- En todos los ejercicios el tiempo debe representarse con la función **delay**.

CONSIDERACIONES PARA RESOLVER LOS EJERCICIOS DE PMS:

- Los canales son punto a punto y no deben declararse.
- No se puede usar la sentencia **empty** para saber si hay algún mensaje en un canal.
- Tanto el envío como la recepción de mensajes es bloqueante.
- Sintaxis de las sentencias de envío y recepción:

  Envío: **nombreProcesoReceptor!port (datos a enviar)**
  Recepción: **nombreProcesoEmisor?port (datos a recibir)**

  El port (o etiqueta) puede no ir. Se utiliza para diferenciar los tipos de mensajes que se podrían comunicarse entre dos procesos.

- En la sentencia de comunicación de recepción se puede usar el **comodín** * si el origen es un proceso dentro de un arreglo de procesos. Ejemplo: **Clientes[*]?port(datos).**

- 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**.

```
Monitor Puente
        cond cola;
        int cant= 0;
        bool libre = true
        Procedure entrarPuente (int au)
                if ( not libre ) {
                        cant = cant + 1;
                        wait (cola);}
                else {libre = false;}

        end;
        Procedure salirPuente (int au)
                if(cant > 0){
                cant = cant – 1;
                signal(cola);
                }else
                {libre = true;}
        end;
End Monitor;
```

```
Process Auto [a:1..M]
        Puente. entrarPuente (a);
        "el auto cruza el puente"
        Puente. salirPuente(a);
End Process;
```

```
Process Robot[id:1..R]{
Pagina sitioInfectado;
        while (true){

        //Busca página infectada
        sitioInfectado = buscarSitioWeb()

        //avisa sitio infectado
        AdminAnalizador!infectado(sitioInfectado);
        }
}
```

```
Process Analizador{
        Página pag
        while (true){

                //envia solicitud para esperar sitio
                infectado
                AdminAnalizador!solicitaSitio();

                //recibe sitio infectado
                AdminAnalizador?infectado(pag);

                //realiza pruebas
                analizaPagina(pag);
        }
}
```

```
Process AdminAnalizador{
Cola colaSitios;
Pagina sitio;
        do Robot[*]?infectado(sitio) → push
        (colaSitios, (sitio));
        [] not empty (colaSitios);
        Analizador?solicitaSitio() →
                        pop(colaSitios, (sitio));
                        Analizador!infectado(sitio)
}
```

```
chan reclamo(String), solicitud(int), proximo[3](String)
```

```
Process Portero[id: 1..3]{
  String reclamo
  while(true){
        //espera recibir reclamos
        send solicitud(id);
        receive proximo[id](reclamo)
        if (reclamo != "VACIO") atender(reclamo)
        else delay(600);
  }
}
```

```
Process Persona[id: 1..P]{
        //envia reclamo
        while (true){
                reclam = generarReclamo()
                send reclamo(reclam)
        }
}
```

```
Process Coordinador{
        String reclamo;
        int idP;
        while (true){
           receive solicitud(idP);
           if ( empty.reclamo)
                reclamo = "VACIO";
           else receive reclamo(reclamo);
        send proximo[idP](reclamo)
        }
}
```

6- ¿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

Con frecuencia un proceso se quiere comunicar con más de un proceso, quizás por distintos canales y no sabe el orden en el cual los otros procesos podrían querer comunicarse con él. Es decir, podría querer recibir información desde diferentes canales y no querer quedar bloqueado si en algún canal no hay mensajes. Para poder comunicarse por distintos canales se utilizan sentencias de comunicación guardadas.

Las sentencias de comunicación guardada soportan comunicación no determinística: B; C->S;

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

B y C forman la guarda.

La guarda tiene éxito si B es true y ejecutar C no causa demora.

La guarda falla si B es falsa.

La guarda se bloquea si B es true pero C no puede ejecutarse inmediatamente

Las sentencias de comunicación guardada soportan comunicación no determinística:

$$B; C \rightarrow S;$$

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

Sea la sentencia de alternativa con comunicación guardada de la forma:
If B1; comunicacion1 -> S1
□ B2; comunicacion2 -> S2
Fi
Primero, se evalúan las expresiones booleanas, Bi y la sentencia de comunicación.
•Si todas las guardas fallan (una guarda falla si B es false), el if termina sin efecto.
•Si al menos una guarda tiene éxito, se elige una de ellas (no determinísticamente).
•Si algunas guardas se bloquean (B es true pero la ejecución de la sentencia de comunicación no puede ejecutarse inmediatamente), se espera hasta que alguna de ellas tenga éxito.
Segundo, luego de elegir una guarda exitosa se ejecuta la sentencia de comunicación asociada.
Tercero, y último, se ejecutan las sentencias S relacionadas.
La ejecución de la iteración es similar realizando los pasos anteriores hasta que todas las guardas fallen.

7- Marque similitudes y diferencias entre los mecanismos RPC y Rendezvous. Ejemplifique para la resolución de un problema a su elección.

Rpc crea un proceso que atiende la solicitud y luego, una vez atendida lo elimina.

Rendezvous los procesos que atienden son estáticos.

RPC puede no tener procesos , cuanto que Rendezvous tiene un solo proceso

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)

- El Pasaje de Mensajes se ajusta bien a problemas de filtros y pares que interactúan, ya que se plantea la ***comunicación unidireccional***.
- Para resolver C/S la comunicación bidireccional obliga a especificar 2 tipos de canales (requerimientos y respuestas).
- Además, cada cliente necesita un canal de reply distinto...
- 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)*.

Rendezvous

- **RPC** por si mismo sólo brinda un mecanismo de comunicación intermódulo. Dentro de un módulo es necesario programar la sincronización. Además, a veces son necesarios procesos extra sólo para manipular los datos comunicados por medio de RPC (ej: Merge).
- **Rendezvous** combina ***comunicación*** **y** ***sincronización***:
  - Como con RPC, 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.

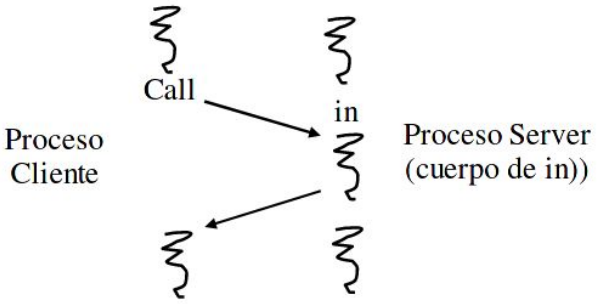
Rendezvous

- La especificación de un módulo contiene declaraciones de los ***headers*** de las operaciones exportadas, pero el cuerpo consta de un único proceso que sirve operaciones.
- Si un módulo exporta ***opname***, el proceso server en el módulo realiza ***rendezvous*** con un llamador de ***opname*** ejecutando una ***sentencia de entrada***:

  in ***opname*** (parámetros formales) → **S; ni**
- Las partes entre ***in*** y ***ni*** se llaman ***operación guardada***.
- Una sentencia de entrada demora al proceso server hasta que haya al menos un llamado pendiente de ***opname***; luego elige el llamado pendiente más viejo, copia los argumentos en los parámetros formales, ejecuta **S** y finalmente retorna los parámetros de resultado al llamador. Luego, ambos procesos pueden continuar.

Rendezvous

- A diferencia de RPC el server es un proceso activo.



- Combinando comunicación guardada con rendezvous:

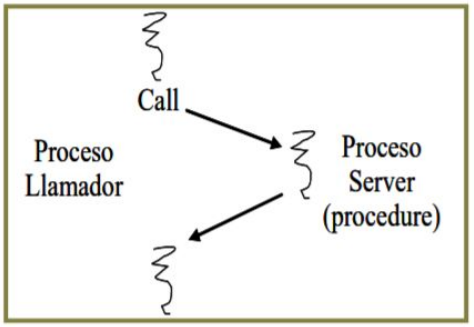
in $\mathbf{op_1}$ (formales$_1$) **and** $\mathbf{B_1}$ **by** $\mathbf{e_1} \rightarrow \mathbf{S_1}$**;**
□ **...**
□ $\mathbf{op_n}$ (formales$_n$) **and** $\mathbf{B_n}$ **by** $\mathbf{e_n} \rightarrow \mathbf{S_n}$**;**
ni

  - Los B_i son ***expresiones de sincronización*** opcionales.
  - Los e_i son ***expresiones de scheduling*** opcionales.

} Pueden referenciar a los parámetros formales.

Remote Procedure Call (RPC)

- La implementación de un llamado intermódulo es distinta que para uno local, ya que los dos módulos pueden estar en distintos espacios: un **nuevo proceso** sirve el llamado, y los argumentos son pasados como mensajes entre el llamador y el proceso server.
- El llamador se demora mientras el proceso servidor ejecuta el cuerpo del procedure que implementa *opname*.
- Cuando el server vuelve de *opname* envía los resultados al llamador y termina. Después de recibir los resultados, el llamador sigue.
- Si el proceso llamador y el procedure están en el mismo espacio de direcciones, es posible evitar crear un nuevo proceso.
- En general, un llamado será remoto ⇒ se debe crear un proceso server o alocarlo de un pool preexistente.



Sincronización en módulos

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).
- Necesitamos que los procesos en un módulo sincronicen (procesos server ejecutando llamados remotos y procesos del módulo). Esto comprende Exclusión Mutua y Sincronización por Condición.
- Existen dos enfoques para proveer sincronización, dependiendo de si los procesos en un módulo ejecutan:
  - ***Con exclusión mutua (un solo proceso por vez).***
  - ***Concurrentemente.***

8- Describa el paradigma "bag of tasks". ¿Cuáles son las principales ventajas del mismo?

El concepto de bag of tasks 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.
Para esto un proceso manager implementará la "bolsa" manejando las tasks, comunicándose con los workers y detectando fin de tareas. Se trata de un esquema C/S.

9- Describa sintéticamente las características de sincronización y comunicación de MPI

MPI (Interfaz de pasaje de mensajes) es una biblioteca para manejar procesos de tipo SPMD. La biblioteca debe ser inicializada y finalizada explícitamente, provee primitivas de envío sincrónico y asincrónico, recepción de mensajes sincrónico y de tipo polling o no bloqueante, sea desde un proceso particular o desde cualquiera dentro de un grupo. Los grupos de procesos se pueden crear y manejar dinámicamente, pudiendo utilizar primitivas de sincronización en barrera, comunicación broadcast, distribuir elementos de un array entre procesos o esperar mensajes desde los procesos del grupo y reunirlos en un array, realizar alguna operación entre los valores recibidos de otros procesos -sumar, buscar el máximo o mínimo...-, entre otras operaciones.

10- Describa sintéticamente las características de sincronización y comunicación de Java y ADA.

1. **Java** provee concurrencia mediante el uso de *threads* que comparten la zona de datos, por lo que el mecanismo que utilizan para comunicarse es memoria compartida. Como método de sincronización java permite utilizar la palabra clave *synchronized* sobre un método lo que brinda sincronización y exclusión mutua. También permite la sincronización por condición con los métodos *wait*, *notify* y *notifyAll* sobre un único lock por objeto, similares al "wait" y "signal" de los monitores, que deben usarse siempre dentro de un bloque synchronized. La semántica que utiliza el *notify* es de *signal and continue*. Importando paquetes estándar puede

utilizarse pasaje de mensajes (java.net soporta TCP o UDP) y RPC (java.rmi soporte una versión orientada a objetos, *Remote Method Invocation*).

2. **Ada** utiliza *Rendezvous* como mecanismo de sincronización y comunicación. En Ada un proceso realiza pedidos haciendo un *call* de un entry definido por otro proceso, este llamado bloquea al proceso llamador hasta que termina la ejecución del pedido. Para servir los llamados a sus entrys una *task* utiliza la sentencia *accept* que demora a la tarea hasta que haya una invocación para el entry asociado a esa sentencia. Cuando recibe una invocación copia los parámetros y ejecuta la lista de sentencias correspondiente; al terminar copia los parámetros de salida y tanto el llamador como el invocador pueden continuar su ejecución. Además, provee tres clases de sentencias select:

- Wait selectivo:

```
Select [when B1 =>] Sentencia accept; sentencias
Or…
Or [when Bn =>] Sentencia accept; sentencias
[Or delay tiempo; sentencias | Or terminate | Else sentencias]
End select
```

- Entry call condicional:

```
Select entry call; sentencias
Else sentencias
End select
```

  Sólo se selecciona el entry call si puede ser ejecutado inmediatamente sino se selecciona el *else*.

- Entry call timed:

```
Select entry call; sentencias
Or delay tiempo; sentencias
End select
```

En este caso, el entry call se selecciona si puede ser ejecutado antes de que expire el intervalo *delay*.

11- Considere el problema de lectores/escritores. Desarrolle un proceso servidor para implementar el acceso a la base de datos, y muestre las interfaces de los lectores y escritores con el servidor. Los procesos deben interactuar:

a) con mensajes asincrónicos.

```
chan requestLectura(int idL), requestEscritura(int idL),
     releaseLectura, releaseEscritura,
     permisoLectura[1:L], permisoEscritura[1:E];

Process BD {
  int qL=0, qE=0, id;
  while(true) {
    if(qE==0 AND !empty(requestLectura)) →
        receive requestLectura(id);
        send permisoLectura[id];
        qL=qL+1;
    [] if(qL==0 AND qE==0 AND !empty(requestEscritura)) →
        receive requestEscritura(id);
        send permisoEscritura[id];
        qE=qE+1;
    [] if(!Empty(releaseLectura)) →
        receive releaseLectura;
        qL=qL-1;
    [] if(!Empty(releaseEscritura)) →
        receive releaseEscritura;
        qE=qE-1;
    fi
  }
}

process Lector[i=1 to L] {
  …
  send requestLectura(i);
  receive permisoLectura[i];
  # lee la base de datos
  send releaseLectura;
  …
```

```
}

process Escritor[i=1 to E] {
  …
  send requestEscritura(i);
  receive permisoEscritura[i];
  # escribe la base de datos
  send releaseEscritura;
  …
}
```

b) con mensajes sincrónicos

```
  # utilizando CSP:

Process BD {
  int qL=0, qE=0, id;
  while(true) {
    if(qE==0; Lector[*]?request) → qL=qL+1;
    [] if(qL==0 AND qE==0; Escritor[*]?request) → qE=qE+1;
    [] if(Lector[*]?release) → qL=qL-1;
    [] if(Escritor[*]?release) → qE=qE-1;
    fi
  }
}

process Lector[i=1 to L] {
  …
  BD!request;
  # lee la base de datos
  BD!release;
  …
}

process Escritor[i=1 to E] {
  …
  BD!request;
  # escribe la base de datos
  BD!release;
  …
}
```

c) con RPC o Rendezvous

```
module BD # Utilizando RPC
  op requestLectura;
  op requestEscritura;
  op releaseLectura;
  op releaseEscritura;
body
  int qL=0, qE=0;

  # sincronización por semáforos (passing the batton)
  sem okLeer = 0, okEscribir = 0, e=1;
  int dL=0, dE=0;

  # implementacion
  proc requestLectura {
    P(e);
    if(qE>0) {
      dL=dL+1;
      V(e);
      P(okLeer);
    }
    qL=qL+1;
```

```
    if(dL>0) {
      dL=dL-1;
      V(okLeer);
    } else V(e);
  }

  proc requestEscritura {
    P(e);
    if(qL>0 or qE>0) {
      dE=dE+1;
      V(e);
      P(okEscribir);
    }
    qE=qE+1;
    V(e);
  }

  proc releaseLectura {
    P(e);
    qL=qL-1;
    if(qL==0 AND dE>0) {
      dE=dE-1;
      V(okEscribir);
    } else V(e);
  }

  proc releaseEscritura{
    P(e);
    qE=qE-1;
    if(dE>0) {
      dE=dE-1;
      V(okEscribir);
    } else if(dL>0) {
      dL=dL-1;
      V(okLeer);
    } else V(e);
  }
end BD;

process Lector[i=1 to L] {
  …
  call BD.requestLectura();
  # lee la base de datos
  call BD.releaseLectura();
  …
}

process Escritor[i=1 to E] {
  …
  call BD.requestEscritura();
  # escribe la base de datos
  call BD.releaseEscritura();
  …
}
```

d) con Rendezvous

(arriba)

12- Modifique la solución con mensajes sincrónicos de la Criba de Eratóstenes para encontrar los números primos detallada en teoría de modo que los procesos no terminen en deadlock.

```
process Worker[i = 1] {
 int p = 2;
 for [t = 3 to N by 2]
    worker[2]!t;
 print p;       # Imprime el primo i-ésimo
```

```
  worker[2]!0;   # Envía la marca EOS al siguiente
}

process Worker[i = 2 to L] {
  int p, t=1;
  worker[i-1]?p; # Recibe el primo i-ésimo
  while(t!=0) {
    worker[i-1]?t;
    if(i<L)and(t mod p != 0) worker[i+1]!t;
  }
  print p;
  if(i<L) worker[i+1]!0;
}
```

El valor de *L* debería ser suficientemente grande para poder detectar todos los números primos entre 2 y *N*. De implementarse en un lenguaje que permita la creación dinámica de procesos podrían crearse a demanda -la primera vez que un worker tenga un dato que pase su filtro, crea al worker siguiente y se lo pasa-.

13- Suponga que N procesos poseen inicialmente cada uno un valor. Se debe calcular la suma de todos los valores y al finalizar la computación todos deben conocer dicha suma.

a) Analice (desde el punto de vista del número de mensajes y la performance global) las soluciones posibles con memoria distribuida para arquitecturas en estrella (centralizada), anillo circular, totalmente conectada, árbol y grilla bidimensional.

b) Escriba las soluciones para las arquitecturas mencionadas.

14- Explique brevemente los 7 paradigmas de interacción entre procesos en programación distribuida vistos en teoría.

En cada caso ejemplifique, indique qué tipo de comunicación por mensajes es más conveniente y qué arquitectura de hardware se ajusta mejor. Justifique sus respuestas.

1. **Servidores replicados:** Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos. Su uso tiene el propósito de incrementar la accesibilidad de datos o servicios donde cada servidor descentralizado interactúa con los demás para darles la sensación a los clientes de que existe un único servidor. Un ejemplo de servidores replicados es la resolución descentralizada al problema de los filósofos donde cada proceso mozo interactúa con otros para obtener los tenedores pero esto es transparente a los procesos filósofos.

- 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
  - Modelo ***centralizado*****:** los Filósofo se comunican con ***UN*** proceso Mozo que decide el acceso o no a los recursos.
  - Modelo ***distribuido*****:** supone **5 procesos Mozo,** cada uno manejando un tenedor. Un Filósofo puede comunicarse con **2** Mozos (izquierdo y derecho), solicitando y devolviendo el recurso. Los Mozos **NO se comunican entre ellos**.
  - Modelo ***descentralizada*****:** cada Filósofo ve **un único** Mozo. Los Mozos se comunican entre ellos (cada uno con sus **2** vecinos) para decidir el manejo del recurso asociado a "su" Filósofo.

2. **Algoritmos Heartbeat:** Procesos que intercambian información haciendo *send* de sus datos y luego un *receive* desde sus vecinos. Es un esquema "divide & conquer" se distribuye la carga (datos) entre los workers; cada uno es responsable de actualizar una parte. Útil para paralelizar soluciones iterativas (cada worker resuelve un paso, pero precisa los resultados de sus vecinos para el paso siguiente, así que comunica los resultados y espera los de ellos).
   Los nuevos valores dependen de los mantenidos por los workers o sus vecinos inmediatos. Cada "paso" debiera significar un progreso hacia la solución. Ejemplos de problemas que se pueden resolver son computación de grillas (labeling de imágenes) o autómatas celulares (el juego de la vida).

- Excesivo intercambio de mensajes ⇒ los procesos cercanos al "centro" conocen la topología más pronto y no aprenden nada nuevo en los intercambios.

3. **Algoritmos Pipeline:** Una serie de procesos conectados en pipe/tubería, por los que fluyen los datos mediante *send/receive* (donde la salida de un proceso es la entrada del siguiente). Cada uno es un productor/consumidor -filtro-. Puede variar la forma de interconexión y pueden manejarse distintos datos o funciones en cada proceso. Ejemplos son las redes de filtros o tratamiento de imágenes.

- 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* (W_1 en el INPUT, W_n en el OUTPUT).
- Un segundo esquema es el pipeline *circular*, donde W_n se conecta con W_1. 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 W_n y W_1.
- Ejemplo: multiplicación de matrices en bloques.

4. **Algoritmos probe (send)/echo (receive):** Procesos que trabajan en una topología de grafo. Un proceso puede diseminar/juntar información haciendo *sends* en paralelo a sus vecinos y esperando que estos lo propaguen, y eventualmente le envíen una respuesta. El recorrido de la sonda es análogo a un DFS del grafo, en paralelo por las distintas ramas. Si no hay información de la topología completa los *sends* se hacen a todos los vecinos, pero puede mejorarse si se cuenta con esa información calculando antes el *árbol mínimo de expansión*. En el primer caso se puede aprovechar el eco para construir una representación de la topología. Además, es posible debido a la presencia de ciclos, que un nodo reciba el probe desde más de un vecino; en dicho caso podría sólo responder al primero con la respuesta precisa, y enviar ecos nulos -sin datos de respuesta- al resto.
   : se basa en el envío de un mensaje ("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).

   - 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).

5. **Algoritmos Broadcast:** Permiten diseminar información en una arquitectura distribuida permitiendo tomar decisiones descentralizadas. Puede utilizar una primitiva *broadcast* disponible en algunas

arquitecturas de red -token ring, ethernet-, que encola los mensajes en todos los canales de un arreglo. Como esta operación NO es atómica, distintos procesos podrían recibir mensajes de broadcast con distinto origen en distinto orden.
La idea del algoritmo es transmitir un mensaje de un procesador a todos los otros. Un ejemplo típico es la sincronización de relojes en un Sistema Distribuido de Tiempo Real.

6. **Token Passing:** 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. En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos. Permite realizar exclusión mutua distribuida y la toma de decisiones distribuidas. Un ejemplo podría ser el de determinar la terminación de un proceso en una arquitectura distribuida cuando no puede decidirse localmente..

   - Un paradigma de interacción muy usado se basa 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***.
   - Ejemplos de recolección de información de estado son los algoritmos de detección de terminación en computación distribuida.
   - Aunque el problema de la SC se da principalmente en programas de MC, puede encontrarse en programas distribuidos cuando hay algún recurso compartido que puede usar un único proceso a la vez. Generalmente es una componente de un problema más grande, tal como asegurar consistencia en un sistema de BD.
   - Soluciones posibles: Monitor activo que da permiso de acceso (ej: locks en archivos), semáforos distribuidos (usando broadcast, con gran intercambio de mensajes), o *token ring* (descentralizado y fair).

7. **Manager/workers:** Un proceso controla los datos a procesar o las tareas a realizar, mientras otros procesos se comunican con él para que les asigne un trabajo. Implementación distribuida del modelo de bag of tasks que consiste en un proceso controlador de tareas y múltiples procesos workers que acceden a él para poder obtener una tarea para ejecutarlas en forma distribuida. Por ejemplo: se utiliza para resolver multiplicación de matrices ralas.

- 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 (ejemplo en LINDA manejando un espacio compartido de tuplas).

- La mayor virtud de este enfoque es la escalabilidad y la facilidad para equilibrar la carga de trabajo de los workers.

- Analizaremos la implementación de este paradigma con mensajes en lugar de MC. Para esto un proceso *manager* implementará la "bolsa" manejando las tasks, comunicándose con los workers y detectando fin de tareas. **Se trata de un esquema C/S.**

- Ejemplo: multiplicación de matrices ralas.

En problemas de broadcast sin conocimientos sobre la topología, como solución general sería conveniente *Probe & Echo*, ya que permite el envío de mensajes a todos los nodos sin importar la arquitectura de red -podría obviarse el *echo* si no se requieren respuestas, dependiendo del problema-.
Los algoritmos de broadcast podrían utilizarse pero sólo si la primitiva broadcast es soportada para la red en cuestión, siendo preferible en esos casos.
Manager/workers y servidores replicados no aplican, están orientados a problemas diferentes. Token passing serviría para topologías de anillo pero no sería la mejor opción para grafos generales.
Heartbeat y Pipeline son más comunes en programas paralelos. Con pipeline los datos fluyen en una dirección y podría no ser posible hacer broadcast. Con heartbeat podrían compartirse datos con los vecinos y luego de varias iteraciones sería de conocimiento global, pero sin datos de la topología no se sabría cuántas iteraciones son necesarias, y obviamente la convergencia sería más lenta. Además el tipo de problemas en los que se aplican estos algoritmos raramente precisarán broadcasts.

___

En una arquitectura de memoria distribuida, tiene que usarse alguna forma de MP, explícita o implícita. En este caso, si el flujo de información es unidireccional (por ejemplo, en redes de filtros –pipeline-) entonces el mecanismo más eficiente es AMP. También AMP es el más eficiente y el más fácil de usar para peers interactuantes (por ejemplo en algoritmos heartbeat, probe/echo, servers replicados, token passing).
Broadcast es más difícil de usar con SMP. Sin embargo, para interacción cliente/servidor, la comunicación bidireccional obliga a especificar dos tipos de canales (requerimientos y respuestas). RPC y Rendezvous son técnicas de comunicación y sincronización entre procesos que suponen un canal bidireccional. Por esto son ideales para programar aplicaciones C/S (por ejemplo manager/workers).

15- Analice la solución al "juego de la vida", y modificarla para que cada worker maneje un *strip* o bloque de células.

dice jose que está en el libro. :P

16- Investigue el problema de detección de terminación en programación distribuida. Plantee soluciones para el caso de procesos conectados en anillo y en grafos.

Aún cuando ningún proceso se encuentre trabajando en un momento dado, puede haber mensajes "en tránsito" que los despierten luego. Por esto y porque normalmente ningún proceso puede ver el estado global (salvo quizá en casos en los que un coordinador controle el avance de workers y les indique cuándo terminar), la detección de terminación es compleja en un ambiente distribuido.

EN PROCESOS CONECTADOS EN ANILLO podría utilizarse el paradigma de *token passing*, incluyendo un token de control que circule por los procesos cuando se encuentren inactivos. Así, si el token da una vuelta completa y regresa a su emisor habiendo este permanecido inactivo todo el tiempo, se habrá detectado la terminación. Sino, se reiniciará el control de terminación.
Inicialmente y cada vez que reciba un mensaje de datos, un proceso se considerará ACTIVO. Al terminar el procesamiento e ingresar a una sentencia *receive* se vuelve inactivo. Si P[1] comienza el control, al volverse inactivo por primera vez enviará el token y fijará su estado como INACTIVO. Cuando el próximo proceso en el anillo lo reciba, se deberá a que él mismo se encuentra en una sentencia *receive* (estará inactivo), y reenviará el token. Si el token regresa a P[1] estando este INACTIVO desde que lo envió, habrá detectado la terminación, la anunciará al resto y finalizará.

```
enum tTipo = (DATOS, TOKEN, EOP);   # EOP = End Of Processing
Chan c[1:N](tTipo tipo, tDatos msg);

Process P[i = 1] {
  enum tEstado = (INICIAL, ACTIVO, INACTIVO);
  tTipo tpo; tDatos msg;
  tEstado estado = INICIAL;
  bool seguir = TRUE;
  while(seguir) {
    receive c[1](tpo, msg);
    if(tpo == DATOS) {
      Procesar(msg);
      send c[2](DATOS, msg);
      if(estado==INICIAL) { # tras procesar datos la 1° vez se vuelve inact.
        estado = INACTIVO;
        send c[2](TOKEN, Null);
      } else estado = ACTIVO;
    } else if(tpo == TOKEN) {
      if(estado == INACTIVO) seguir = false
      else {
        estado = INACTIVO;
        send c[2](TOKEN, Null);
      }
    }
  }
  send c[2](EOP, Null);   # avisa terminación
  receive c[1](tpo, msg); # recibe msj de P_N, fin total del programa
}

Process P[i = 2 to N] {
  tTipo tpo = DATOS; tDatos msg;
  while(tpo<>EOP) {
    receive c[i](tpo, msg);
    if(tpo == DATOS) Procesar(msg);
    send c[i mod N + 1](tpo, msg); # propaga datos, tokens y terminación
  }
}
```

EN PROCESOS CONECTADOS EN GRAFO el esquema anterior se complica ya que un proceso podría recibir mensajes desde más de una fuente. Si una de ellas comenzara el algoritmo de detección de terminación y le enviara un token, el proceso lo distribuiría siguiendo el algoritmo, pero aún así podría luego recibir un mensaje de datos desde alguna de las otras fuentes, reactivándose. Así, la vuelta del token al emisor original no implica que todos los procesos estén inactivos.
Podría adaptarse la técnica si se impone un ciclo C -un "anillo"- que involucre a todos los nodos del grafo G, quizá incluyendo a alguno más de una vez para armar su recorrido (por lo que C_n, la longitud del ciclo, podría ser mayor a N, la cantidad de procesos). Sin embargo, como el emisor del token podría aparecer en más de una oportunidad en el ciclo, el regreso del token sigue sin indicar la terminación. Esto se soluciona utilizando al token como un contador de procesos inactivos (se inicia en cero, y cada proceso lo incrementa al reenviarlo al próximo en C), y así la llegada del token con valor C_n a un proceso inactivo -su emisor original- permitiría detectar la terminación; si en cambio al llegar el token con valor C_n el proceso se encuentra activo, reiniciaría su control de terminación. El algoritmo podría ser iniciado por cualquier proceso al volverse inactivo por primera vez, enviando un token en 0.
Al detectar la terminación se podría avisar a todos mediante broadcast, o continuar anunciando por C: cada proceso reenvía el aviso y termina, excepto aquellos que se repiten luego en C. El último proceso del ciclo terminará directamente sin reenviar -no habrá nadie a quien entregarle el mensaje-.

```
enum tTipo = (DATOS, TOKEN);
```

```
union tMsg {
  tDatos data; # datos de mensajes comunes
  int valor;   # valor del contador del token
}
chan c[1:N](tTipo tipo, tMsg msg);

Process P[i = 1] {
  enum tEstado = (INICIAL, ACTIVO, INACTIVO);
  tTipo tpo; tMsg msg; int indiceEntrada;
  tEstado estado = INICIAL;
  bool seguir = TRUE;
  while(seguir) {
    # P[i] tiene varias aristas de entrada en G puede recibir de cualquiera
    recibirDesdeAlgunaEntrada(tpo, msg, indiceEntrada);

    if(tpo == DATOS) {
      Procesar(msg.data);
      # P[i] tiene POUTi aristas de salida en G puede precisar enviar
      # su salida a todas o algunas de ellas según el problema
      EnviarACadaSalidaNecesaria(tpo, msg);
      if(estado==INICIAL) { # tras procesar datos la 1º vez se vuelve inact.
        estado = INACTIVO;
        send c[SiguienteEnC(i, 0)](TOKEN, 0); # al prox de C (segundo nodo de C)
      } else estado = ACTIVO;
    } else if(tpo == TOKEN) {
      if(estado == INACTIVO) {
        if(msg.valor>=Nc) # detecta terminación
          seguir = SeRepiteLuegoEnC(i, indiceEntrada);
        msg.valor=msg.valor+1;
      } else {
        estado = INACTIVO;
        msg.valor=0; # reinicia el algoritmo de detección
        # en esta implementación el token dará mínimo 3 vueltas a C:
        # 1º volviendo inactivos a todos los procesos: cada uno lo reinicia
        # 2º controlando que permanecen inactivos: cada uno lo incrementa (<Nc)
        # 3º avisando terminación a todos: cada uno lo incrementa (>=Nc)
      }
      # el último de C que detecta terminación no reenvía (todos terminaron)
      if(msg.valor<2*Nc)
        send c[SiguienteEnC(i, indiceEntrada)](TOKEN, msg.valor);
    }
  }
}

Process P[i = 2 to N] {
  enum tEstado = (ACTIVO, INACTIVO);
  tTipo tpo; tMsg msg; int indiceEntrada;
  tEstado estado = ACTIVO;
  bool seguir = TRUE;
  while(seguir) {
    recibirDesdeAlgunaEntrada(tpo, msg, indiceEntrada);

    if(tpo == DATOS) {
      Procesar(msg.data);
      EnviarACadaSalidaNecesaria(tpo, msg);
      estado = ACTIVO;
    } else if(tpo == TOKEN) {
      if(estado == INACTIVO) {
        if(msg.valor>=Nc) #detecta terminación
          seguir = SeRepiteLuegoEnC(i, indiceEntrada);
        msg.valor=msg.valor+1;
      } else {
        estado = INACTIVO;
        msg.valor=0; # reinicia el algoritmo de detección
      }
      if(msg.valor<2*Nc)
        send c[SiguienteEnC(i, indiceEntrada)](TOKEN, msg.valor);
    }
```

```
    }
  }
```

17- ¿En qué consiste la utilización de relojes lógicos para resolver problemas de sincronización distribuida? Ejemplifique para el caso de implementación de semáforos distribuidos.

En un entorno distribuido, el orden de recepción de los mensajes no necesariamente es el mismo que el de emisión. Influye por ejemplo el tiempo que demora un mensaje enviado por un nodo en llegar a los otros, que no será ni uniforme ni constante, dependiendo de la distancia física a la que se encuentren entre otros factores. En particular, mensajes enviados por *broadcast* por dos procesos no serán recibidos en el mismo orden por los otros -incluso podrían ser enviados en exactamente el mismo momento-. En estos casos es necesario encontrar alguna característica que permita diferenciar a uno de otro mensaje y darle prioridad a uno de ellos.
En equipos monoprocesador de MC necesariamente los eventos ocurren en secuencia y podrían ordenarse por el valor del *timer* del equipo en el momento del evento. En un entorno distribuido los *timers* de los distintos equipos no podrían ser perfectamente sincronizados y esto no sería viable. En su lugar pueden usarse *relojes lógicos* y *marcas de tiempo*: cada proceso mantiene un valor entero que incluye en cada mensaje enviado a modo de *timestamp*, permitiendo un ordenamiento de los eventos. Debe ser actualizado para asegurar que si un evento ocurre antes que otro relacionado, su timestamp sea menor.
Para esto, cada proceso mantiene un valor *clock* inicializado en cero. Este es incluido en cada mensaje que se envía, y luego es incrementado. El recibir un mensaje, se lo actualiza con el máximo entre su valor y el timestamp del mensaje más uno, y luego se incrementa. Esto permite un orden parcial entre eventos relacionados. Si además cada proceso tiene un ID único, puede obtenerse un orden total utilizándolo para desempates.
La ordenación impuesta por este esquema no tiene por qué coincidir con el orden temporal real del envío de los mensajes, pero permite asegurar que, recibidos todos los mensajes, todos los nodos que participan los interpreten en un mismo orden. Si un proceso P2 envía un mensaje con marca de tiempo *x* y otro proceso P1 envía luego otro mensaje, antes de recibir el de P2, también con marca de tiempo *x*, todos los procesos ordenarán el mensaje de P1 antes que el de P2, aunque no es este el orden en la realidad.

Pueden utilizarse por ejemplo para implementar semáforos distribuidos, necesarios en este caso ya que si más de un proceso realiza una operación P(), es preciso un consenso entre todos sobre cuál de ellos podrá continuar. El uso de relojes lógicos permite que la decisión se tome de manera distribuida sin tener que recurrir a un servidor/coordinador -esto equivaldría a un monitor activo-.
Los procesos deberían hacer broadcast para realizar una operación sobre el semáforo indicando el valor de su *clock*.
Si cada proceso que comparte el semáforo responde autorizando una operación, una vez que se hayan recibido confirmaciones de todos los otros procesos se puede concretarla actualizando un entero que represente el valor del semáforo. Las confirmaciones deberían hacerse también por broadcast para que todos los procesos puedan saber qué operaciones se llevan a cabo y en qué orden sobre el semáforo, manteniendo cada uno el valor asociado.
Cada proceso llevaría en una cola (ordenada por *timestamp*) los mensajes que se reciban sobre el semáforo, indicando remitente, operación y *timestamp.* Además debería controlar para cada proceso, cuál fue el último timestamp recibido. Si un mensaje m_i de la cola tiene un timestamp t_i, y se recibió de todos los procesos una confirmación con timestamp superior, se dice que m_i está completamente reconocido -todos los procesos lo han recibido y confirmado-, y puede procesarse. Cualquier mensaje anterior a m_i también lo estará ya que su timestamp será menor (forman el "*prefijo estable*" de la cola).
Tras recibir una confirmación y actualizar el prefijo se procesan los mensajes de la cola. Por cada V, se incrementa el semáforo y elimina el mensaje; luego, por cada P, si el semáforo es positivo se lo decrementa y elimina el mensaje. Si se elimina una P() del propio proceso, entonces puede continuar.
Cada proceso debería llevar a cabo su tarea correspondiente y, a la vez, responder con confirmaciones a operaciones de otros procesos. Por esto convendría utilizar, para cada proceso, un auxiliar que se encargue exclusivamente del manejo del semáforo. Si el proceso debe realizar alguna operación, se la solicitaría a su auxiliar. En el caso de P() debería esperar luego que el auxiliar lo autorice a continuar.

18- Suponga n^2 procesos organizados en forma de grilla cuadrada. Cada proceso puede comunicarse solo con los vecinos izquierdo, derecho, de arriba y de abajo (los procesos de las esquinas tienen solo 2 vecinos, y los otros en los bordes de la grilla tienen 3 vecinos). Cada proceso tiene inicialmente un valor local v.

a) Escriba un algoritmo heartbeat que calcule el máximo y el mínimo de los n^2 valores. Al terminar el programa, cada proceso debe conocer ambos valores. (Nota: no es necesario que el algoritmo esté optimizado).

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

c) Puede realizar alguna mejora para reducir el número de mensajes?

d) Modifique la solución de a) para el caso en que los procesos pueden comunicarse también con sus vecinos en las diagonales.

19- ¿Qué relación encuentra entre el paralelismo recursivo y la estrategia de "dividir y conquistar"? ¿Cómo aplicaría este concepto a un problema de ordenación de un arreglo?

En el paralelismo recursivo el problema se llama a sí mismo una o más veces. En cada nivel de la recursión, se aplica la estrategia de dividir y conquistar, que consta de tres pasos.
- Se divide el problema en varios subproblemas similares al problema original pero de menor tamaño (cada subproblema es asignado a un procesador distinto).
- Si los tamaños de los subproblemas son suficientemente pequeños, entonces se los resuelven de manera directa. Sino, se continúa la recursión sobre los subproblemas.
- Se combinan las soluciones para crear la solución al problema original. Para ordenar un arreglo con el concepto mencionado podemos utilizar un algoritmo sorting by merging, se seguirán los siguientes pasos:
1. Se asigna a un procesador el proceso de ordenar el arreglo con un algoritmo sorting by merging.
2. Si la longitud del arreglo es 0 ó 1, entonces ya está ordenada. En otro caso:
3. Se divide el arreglo desordenado en dos sub-arreglos de aproximadamente la mitad del tamaño. Asignando cada uno de esos sub-arreglos a un procesador distinto.
4. Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla. Mezclar las dos sublistas en una sola lista ordenada.

20- a) Cómo puede influir la topología de conexión de los procesadores en el diseño de aplicaciones concurrentes/paralelas/distribuidas? Ejemplifique.

b) Qué relación existe entre la granularidad de la arquitectura y la de las aplicaciones?

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

El objetivo principal es reducir el tiempo de ejecución, o resolver problemas mas grandes o con mayor precisión en el mismo tiempo, usando una arquitectura multiprocesador en la que se pueda distribuir la tarea global en tareas que puedan ejecutarse en distintos procesadores.

b) ¿Cuál es el significado de las métricas de speedup y eficiencia? ¿Cuáles son los rangos de valores en cada caso?

Ambas son métricas asociadas al procesamiento paralelo. El speedup es una medida de la mejora de rendimiento (performance) de una aplicación al aumentar la cantidad de procesadores (comparado con el rendimiento al utilizar un solo procesador). El mismo se calcula: Speedup⇒ S = Ts/ Tp S es el cociente entre el tiempo de ejecución secuencial del algoritmo secuencial conocido más rápido (Ts) y el tiempo de ejecución paralelo del algoritmo elegido (Tp). El rango de valores de S va desde 0 a p, siendo p el número de procesadores. La eficiencia es una medida relativa que permite la comparación de desempeño en diferentes entornos de computación paralela, mide la fracción de tiempo en que los procesadores son útiles para el cómputo. La misma se calcula: Eficiencia⇒ E = S/P Es el cociente entre speedup y número de procesadores. El valor está entre 0 y

1, dependiendo de la efectividad en el uso de los procesadores. Cuando es 1 corresponde al speedup perfecto.

c) ¿En qué consiste la "ley de Amdahl"?

La ley de Amadhi dice que para un problema dado existe un máximo speedup alcanzable independiente del número de procesadores. Esto significa que es el algoritmo el que decide la mejora de velocidad dependiendo de la cantidad de código no paralelizable y no del número de procesadores, llegando finalmente a un momento que no se puede paralelizar más el algoritmo.

d) Suponga que la solución a un problema es paralelizada sobre *p* procesadores de dos maneras diferentes. En un caso, el speedup (S) está regido por la función S=p-1 y en el otro por la función S=p/2. ¿Cuál de las dos soluciones se comportará más eficientemente al crecer la cantidad de procesadores? Justifique claramente.

P=8 Suponemos homogeneidad

Alg1= S = p - 1 -> S = 7 Eficiencia -> S/p -> 7/8 = 0.874 - 87% eficiencia

Alg2= S = p/2 -> S = 4 Eficiencia -> S/p -> 4/8 = 0.5 - 50% eficiencia

e) Suponga que el tiempo de ejecución de un algoritmo secuencial es de 10000 unidades de tiempo, de las cuales sólo el 90% corresponde a código paralelizable. ¿Cuál es el límite en la mejora que puede obtenerse paralelizando el algoritmo? Justifique.

22- a) Analizando el código de multiplicación de matrices en paralelo planteado en la teoría, y suponiendo que N=256 y P=8, indique cuántas asignaciones, cuántas sumas y cuántos productos realiza cada proceso. ¿Cuál sería la cantidad para cada operación en la solución secuencial realizada por un único proceso?

```
process worker [ w = 1 to P]
  { int primera = (w-1)*(n/P) + 1;
    int ultima = primera + (n/P) - 1;
    for [i = primera to ultima]
      { for [j = 1 to n]
          { c[i,j] = 0;
            for [k = 1 to n]
              c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
          }
      }
  }
```

b) Si los procesadores P1 a P7 son iguales, y sus tiempos de asignación son 1, de suma 2 y de producto 3, y si el procesador P8 es 4 veces más lento, ¿cuánto tarda el proceso total

concurrente? ¿Cuál es el valor del speedup? ¿Cómo podría modificar el código para mejorar el speedup?

---

1. **Defina programa distribuido y programa paralelo**
   Programa distribuido: teoria 7 pag 4

   Programa paralelo:

   El paralelismo se asocia con la ejecución concurrente en múltiples procesadores que pueden tener memoria compartida, y generalmente con el objetivo de incrementar performance, velocidad o precisión en los resultados que los secuenciales. Ejecuta cada uno un hilo diferente en un momento dado.
   El procesamiento distribuido es un “caso” de concurrencia con múltiples procesadores y sin memoria compartida, en el cual los procesos se comunican por pasaje de mensajes, RPC o rendezvous. Supone una arquitectura de memoria distribuida aunque esto no es limitante.

2. **¿En qué consiste la comunicación guardada y cuál es su utilidad?**

   Con frecuencia un proceso se quiere comunicar con más de un proceso, quizás por distintos canales y no sabe el orden en el cual los otros procesos podrían querer comunicarse con él. Es decir, podría querer recibir información desde diferentes canales y no querer quedar bloqueado si en algún canal no hay mensajes. Para poder comunicarse por distintos canales se utilizan sentencias de comunicación guardadas.
   Las sentencias de comunicación guardada soportan comunicación no determinística: B; C->S;
   B es una expresión booleana que puede omitirse y se asume true.
   B y C forman la guarda.
   La guarda tiene éxito si B es true y ejecutar C no causa demora.
   La guarda falla si B es falsa.
   La guarda se bloquea si B es true pero C no puede ejecutarse inmediatamente.

3. **Indique al menos dos similitudes y dos diferencias entre RPC y Rendezvous**

Rpc crea un proceso que atiende la solicitud y luego, una vez atendida lo elimina.

Rendezvous los procesos que atienden son estáticos.

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)

La **sincronización** es la posesión de información acerca de otro proceso para coordinar actividades. Los procesos se sincronizan:

- Por *exclusión mutua*.
- Por *condición*.

- **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, 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.

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

1. **Marque la o las respuestas correctas:**
   Sincronizar significa:

- Ejecutar paso a paso las instrucciones de los procesos. (prog secuencial)
- Hacer que un proceso espere por una condición para continuar (sincronizacion por condicion)
- Que solo pueden utilizarse variables compartidas por 2 procesos.
- Combinar acciones atómicas de grano fino en acciones compuestas para obtener exclusión mutua.
- Intercambiar información. (COMUNICACIÓN)
- Demorar a un proceso cuando otro está en la sección crítica
- Que no se puede asegurar cuál será el resultado de un programa concurrente (no determinismo)
- Que la ejecución depende del tiempo de procesamiento de las instrucciones.
- Restringir las interacciones entre procesos.

2. **Marque la o las respuestas correctas:**
   ¿Cuáles de las siguientes afirmaciones son correctas?
   - La interacción cliente/servidor es bidireccional.
   - En una solución típica de tipo cliente/servidor, el cliente espera a que su pedido sea procesado por el servidor para poder continuar.
   - En el paralelismo recursivo, sólo pueden realizarse n invocaciones.
   - Un pipeline logra buena performance cuando las etapas consumen tiempos distintos.
   - Las soluciones de paralelismo iterativo solo dividen el procesamiento tomando en cuenta las tareas a realizar.
   - Los esquemas de peers son sincrónicos. (PMA)

3. **Marque cuales de las siguientes afirmaciones son correctas:**
   - La ausencia de deadlock es una propiedad de seguridad. **pag.3 andrew**
   - Un programa concurrente bien sincronizado evita el no determinismo.
   - Un programa concurrente con memoria compartida y variables disjuntas puede producir interferencia. (NO POR QUE SON VARIABLES DISJUNTAS)
   - En un programa concurrente sólo es posible tener un mecanismo de sincronización.
   - La ausencia de demora innecesaria es una propiedad de vida.

- Las soluciones de tipo spin-locks son más eficientes que la utilización de semáforos.

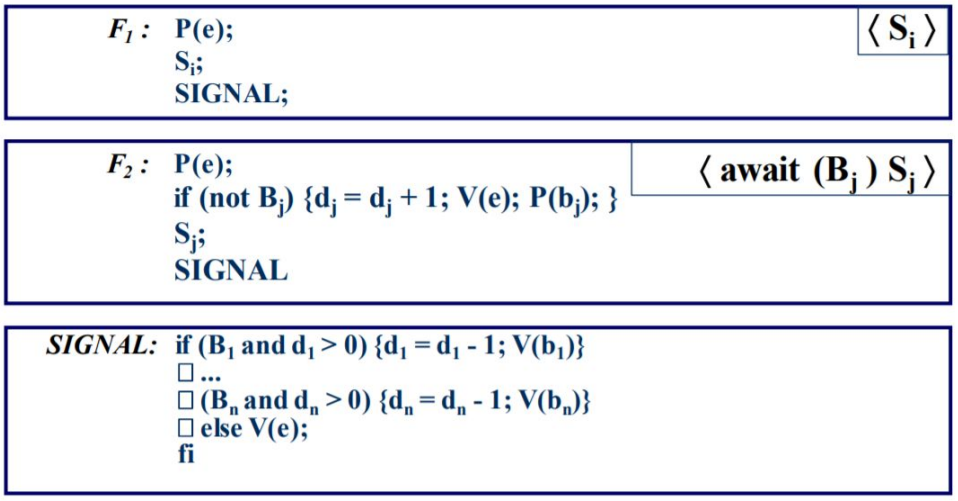
4.

a) Describa la técnica de Passing the Baton.

- **Passing the Baton** es una técnica general para implementar sentencias AWAIT. Cuando un proceso está dentro de la SC, éste mantiene el baton (o token) que equivale al permiso para ejecutar. Cuando el proceso llega a un SIGNAL, pasa el baton (control) a otro proceso. Si no hay procesos esperando el baton (o sea entrar a la SC), entonces se libera para que lo tome el próximo que trata de entrar.

Passing the baton proveer exclusión y controlar cuál proceso demorado es el próximo en seguir. Puede usarse para implementar sentencias await arbitrarias y así implementar sincronización por condición arbitraria. La técnica también puede usarse para controlar precisamente el orden en el cual los procesos demorados son despertados.
El funcionamiento es el siguiente: cuando un proceso está ejecutando su sección crítica, puede pensarse que posee la "posta" o "baton", esto significa que tiene permiso de ejecución. Cuando este proceso termina, pasa la posta a otro. Si alguno está esperando por una condición que es ahora verdadera, la posta se le otorga al mismo. Este a su vez, ejecuta su SC y la pasa a otro. Cuando no hay procesos esperando por la condición
verdadera, la posta se pasa al próximo proceso que trate de entrar en su SC por primera vez.

Passing the batton se utiliza en semáforos, mientras que passing the condition con monitores.



```
int nr = 0, nw = 0;

process Lector [i = 1 to M]
{ while(true)
    { …
      ⟨ await (nw == 0) nr = nr + 1; ⟩
      lee la BD;
      ⟨ nr = nr − 1; ⟩
    }
}


process Escritor [j = 1 to N]
{ while(true)
    { …
      ⟨ await (nr==0 and nw==0) nw=nw+1; ⟩
      escribe la BD;
      ⟨ nw = nw – 1; ⟩
    }
}
```

```
int nr = 0, nw = 0, dr = 0, dw = 0;
sem e = 1, r = 0, w = 0;

process Lector [i = 1 to M]
{ while(true) {
        P(e);
        if (nw > 0) {dr = dr+1; V(e); P(r); }
        nr = nr + 1;
        SIGNAL1 ;
        lee la BD;
        P(e); nr = nr – 1; SIGNAL2 ;
    }
}
process Escritor [j = 1 to N]
{ while(true) {
        P(e);
        if (nr > 0 or nw > 0) {dw = dw+1; V(e); P(w); }
        nw = nw + 1;
        SIGNAL3 ;
        escribe la BD;
        P(e); nw = nw – 1; SIGNAL4 ;
    }
}
```

```
int nr = 0, nw = 0, dr = 0, dw = 0;
sem e = 1, r = 0, w = 0;

process Lector [i = 1 to M]
{ while(true)
    {   P(e);
        if (nw > 0) {dr = dr+1; V(e); P(r); }
        nr = nr + 1;
        SIGNAL1 ;
        lee la BD;
        P(e); nr = nr – 1; SIGNAL2 ;
    }
}
process Escritor [j = 1 to N]
{ while(true)
    {   P(e);
        if (nr > 0 or nw > 0) {dw = dw+1; V(e); P(w); }
        nw = nw + 1;
        SIGNAL3 ;
        escribe la BD;
        P(e); nw = nw – 1; SIGNAL4 ;
    }
}
```

El rol de los **SIGNAL$_i$** es el de señalizar ***exactamente*** a uno de los semáforos ⇒ los procesos se van pasando el ***baton***.

SIGNAL$_i$ es una abreviación de:

```
if (nw == 0 and dr > 0)
      {dr = dr - 1; V(r);}
elsif (nr == 0 and nw == 0 and dw > 0)
      {dw = dw - 1; V(w);}
else  V(e);
```

Algunos de los SIGNAL se pueden simplificar.

```
int nr = 0, nw = 0, dr = 0, dw = 0;          sem e = 1, r = 0, w = 0:

process Lector [i = 1 to M]
{  while(true)
     {  P(e);
        if (nw > 0) {dr = dr+1; V(e); P(r); }
        nr = nr + 1;
        if (dr > 0) {dr = dr – 1; V(r); }
        else V(e);
        lee la BD;
        P(e);
        nr = nr – 1;
        if (nr == 0 and dw > 0)
              {dw = dw – 1; V(w); }
        else V(e);
     }
}

process Escritor [j = 1 to N]
{ while(true)
   {  P(e);
      if (nr > 0 or  nw > 0)
              {dw=dw+1; V(e); P(w);}
      nw = nw + 1;
      V(e);
      escribe la BD;
      P(e);
      nw = nw – 1;
      if (dr > 0) {dr = dr - 1; V(r); }
      elseif (dw > 0) {dw = dw - 1; V(w); }
      else V(e);
   }
}
```

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

- Presenta una forma sencilla y sistemática de implementar cualquier tipo de sincronización entre procesos, con o sin condiciones, utilizando semáforos para una solución de grano fino. Los programas resultantes son fáciles de modificar y extender.

5. **¿Qué significa que un problema sea de "exclusión mutua selectiva" (EMS)? El problema de lectores/escritores en su definición tradicional es de EMS? ¿Por qué? Si también los lectores necesitarán acceso exclusivo el problema es de EMS? ¿Por qué?**

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. Por ejemplo los problemas de lectores/escritores y filósofos.

Por ejemplo , sean dos clases de procesos A y B, primero llega un proceso de clase A, gana el acceso al recurso, luego llega un proceso de clase B, pero es excluído y debe esperar que liberen el recurso, posteriormente llega otro proceso de clase A, el cual no es impedido de entrar ya que el acceso fue ganado por otro de su misma clase; así seguirían bloqueados todos los procesos de clase B que puedan llegar, hasta que todos los procesos de clase A hayan liberado el recurso en cuestión.

NOTA: el problema de los lectores y escritores es:
Tenemos una única BD donde pueden haber múltiples lectores al mismo tiempo pero solo un único escritor.

Si, el problema de los lectores y escritores es de EMS, esto se ve en el caso de los lectores que compiten en conjunto contra los escritores donde pueden acceder un número ilimitado de lectores simultáneamente y no se permiten escritores mientras haya al menos un lector.

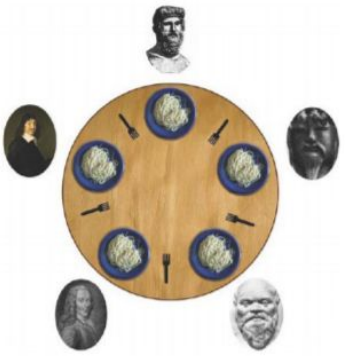
Si los lectores necesitan acceso exclusivo el problema dejaría de ser EMS y pasaría a ser de EM, ya que se deja de seleccionar a una clase de proceso en específico y pasa a verse a nivel de proceso individual. Ahora solo podría estar un único proceso accediendo a la DB.

Problemas básicos y técnicas

Problema de los filósofos: *exclusión mutua selectiva*

- Problema de exclusión mutua entre procesos que compiten por el acceso a conjuntos superpuestos de variables compartidas.
- ***Problema de los filósofos:***

```
process Filosofo [i = 0 to 4]
{ while (true)
    { adquiere tenedores;
      come;
      libera tenedores;
      piensa;
    }
}
```



- ***Cada tenedor es una SC***: puede ser tomado por un único filósofo a la vez ⇒ pueden representarse los tenedores por un arreglo de semáforos.
- Levantar un tenedor ⇒ ***P*** Bajar un tenedor ⇒ ***V***
- Cada filósofo necesita el tenedor izquierdo y el derecho.
- ¿Qué efecto puede darse si todos los filósofos hacen ***exactamente*** lo mismo?.

Problema de los filósofos: *exclusión mutua selectiva*

```
sem  tenedores [5] = {1,1,1,1,1};

process Filososfos[i = 0..3]
{ while(true)
    { P(tenedor[i]);  P(tenedor[i+1]);
      comer;
      V(tenedor[i]);  V(tenedor[i+1]);
    }
}

process Filososfos[4]
{ while(true)
    { P(tenedor[0]);  P(tenedor[4]);
      comer;
      V(tenedor[0]);  V(tenedor[4]);
    }
}
```

---

1. **Que diferencia/s existe/n entre las disciplinas de señalización Signal and wait y Signal and continue?**

- ***Signal and Continued***: el proceso que hace el *signal* continua usando el monitor, y el proceso despertado pasa a competir por acceder nuevamente al monitor para continuar con su ejecución (en la instrucción que lógicamente le sigue al *wait*).

- ***Signal and Wait***: el proceso que hace el *signal* pasa a competir por acceder nuevamente al monitor, mientras que el proceso despertado pasa a ejecutar dentro del monitor a partir de instrucción que lógicamente le sigue al *wait*.

2. Con disciplina Signal and continue, a que es equivalente funcionalmente la primitiva signal_all en términos de la primitiva signal? (Escriba código)

• signal_all(cv) → despierta todos los procesos demorados en cv, quedando vacía la cola asociada a cv.

Equivalente a: **do not empty(cv) → signal(cv) od**

• signal(cv) → despierta al proceso que está al frente de la cola (si hay alguno) y lo saca de ella. El proceso despertado recién podrá ejecutar cuando readquiera el acceso exclusivo al monitor

Siempre que los wait que nombran una variable condición estén en loops que rechequean la condición, signal_all puede usarse en lugar de signal. Dado que los procesos despertados ejecutan con exclusión mutua y rechequean sus condiciones de demora, los que encuentran que la condición ya no es true simplemente vuelve a dormirse. Por ejemplo, signal_all podría ser usada en lugar de signal en el monitor del buffer. Sin embargo, en este caso es más eficiente usar signal pues a lo sumo un proceso despertado podría seguir; los otros deberían volver a dormir.

3. a) Describa la técnica Passing the Condition

Passing the condition trata de señalar (signal) una variable condición que tenga procesos esperando para ejecutar pero sin volver verdadera la condición asociada; o cambiando a verdadera la condición asociada, pero sin señalar (signal) la variable condición, si no hay procesos esperando para ejecutar. De esta manera se asegura que el siguiente proceso en ejecutar su SC es el despertado de la lista, y no otro que gane el acceso al monitor, ya que la condición será falsa y pasará a la cola de espera.

- Ambas técnicas pasan el control a un proceso demorado que será despertado para continuar su ejecución de la SC. La diferencia está en que passing the baton pasa el control a otro proceso para que sea el único que pueda ejecutar SC; mientras que passing the condition le avisa a un proceso que están dadas las condiciones para que pueda seguir ejecutando y este sea el único que pueda continuar de todos los que esperan dicha condición.

b) Cuál es su utilidad en la resolución de problemas mediante monitores?

c) Ejemplifique

SC - Defectos busy waiting

Defectos de la sincronización por ***busy waiting***

- Protocolos "***busy-waiting***": complejos y sin clara separación entre variables de sincronización y las usadas para computar resultados.
- Es difícil diseñar para probar corrección. Incluso la verificación es compleja cuando se incrementa el número de procesos.
- Es una técnica ineficiente si se la utiliza en multiprogramación. Un procesador ejecutando un proceso ***spinning*** puede ser usado de manera más productiva por otro proceso.