**CUESTIONARIO CLASE 1 Y 2**

1. **Mencione al menos 3 ejemplos donde pueda encontrarse concurrencia.**
2. Videojuego de estrategia. Mientras mantiene actualizado el mapa en pantalla, calcula el movimiento de enemigos, captura órdenes de los usuarios, controla el desarrollo del juego, etc.
3. Reproductor de DVD. Mientras lee el dispositivo, decodifica audio y video, controla la temporización de subtítulos, envía las señales de salida, etc.
4. Procesador de texto. Mientras recibe entras del usuario, controla ortografía, realiza sugerencias, guardados automáticos, etc.
5. **Escriba una definición de concurrencia. Diferencie procesamiento secuencial, concurrente y paralelo.**

La concurrencia es ejecutar dos o más procesos en forma simultánea.

* Procesamiento Secuencial: cuando se tiene un único hilo de ejecución o control.
* Procesamiento Concurrente: cuando se tiene más un hilo de ejecución o control, no implicando la existencia de más de un procesador.
* Procesamiento Paralelo: procesamiento concurrente sobre más de un procesador, ejecutando cada uno un hilo diferente en un momento dado.

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

Deadlock es un 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:

* Exclusión mutua: existencia de al menos un recurso compartido por los procesos al cual solo puede acceder uno en un momento dado.
* Posesión y espera: al menos un proceso Pi ha adquirido un recurso Ri, y lo mantiene mientras espera al menos un recurso Rj que ya ha sido asignado a otro proceso.
* No expropiación: un recurso asignado a un proceso Pi no puede ser expropiado por otro proceso Pj.
* Espera circular: dado un conjunto de n procesos Pi, cada proceso retiene un recurso que espera su sucesor, esperando P1 el recurso retenido por Pn.

1. **Defina inanición. Ejemplifique.**

Inanición significa que 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 la inanición, como así también el livelock que es 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.

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

El no determinismo es la producción de resultados impredecibles: incluso ante los mismos valores y condiciones iniciales, puede obtener distintos resultados.

En un programa concurrente compuesto por un conjunto de 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 del programa aun en corridas con iguales valores de entrada. La historia especifica de una corrida será decidida NO por el programa concurrente, sino por la política de scheduling que aplique el SO.

1. **Defina comunicación. Explique los mecanismos de comunicación que conozca.**

Indica el modo en que se organizan y trasmiten datos entre tareas concurrentes. Esta organización requiere especificar ciertos protocolos que nos permitan controlar el progreso y la corrección.

Hay comunicación:

* Por Memoria Compartida: los procesos intercambian información sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella.
* Por Pasaje de Mensajes: es necesario establecer un canal (lógico o físico) para transmitir información entre procesos. Para que la comunicación sea efectiva los procesos deben saber cuándo tienen mensajes para leer y cuando deben trasmitir mensajes.

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

La sincronización es la interacción entre procesos que permite controlar el orden en que se ejecutan, restringiendo las historias posibles solo a las deseables. Hay dos formas básicas:

- Por Exclusión Mutua: asegura que las sentencias en las secciones críticas de distintos procesos no puedan ejecutarse al mismo tiempo.

- Por Condición: retrasa la ejecución de un proceso hasta que una condición dada se vuelva verdadera.

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

Ambos métodos de sincronización pueden estar presentes en un mismo programa ya que puede haber más de una circunstancia en la que los procesos precisen sincronizarse. Además, tiene distintos fines: delimitar secciones críticas y garantizar que el estado del programa sea adecuado antes de continuar la ejecución de un proceso.

Por ejemplo, puede lograrse la exclusión mutua utilizando sincronización por condición para los protocolos de entrada a las secciones críticas.

1. **¿Qué significa el problema de “interferencia” en programación concurrente? ¿Cómo puede evitarse?**

La interferencia es el resultado de dos o más procesos leyendo y escribiendo variables compartidas en un orden impredecible y, por tanto, con resultados impredecibles. En general, una instrucción atómica en un proceso interfiere con una precondición de una sección crítica o una suposición hecha por otro, si modifica su valor de verdad.

Puede evitarse por medio de:

* Variables disjuntas: validar que el conjunto de escritura de un proceso y el conjunto de referencias de otro sean distintos.
* Asunciones débiles: escribir los procesos considerando los efectos de otros procesos en ejecución, realizando menos asunciones o más generales.
* Invariante global: si todas las asunciones de los procesos pueden escribir de la forma I ^ L, donde I es un invariante global y donde L solo referencia variables locales o globales que solo ese proceso escribe, entonces los procesos están libres de interferencia.
* Sincronización: evitar interferencia mediante exclusión mutua o sincronización por condición.

1. **¿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 indica que una expresión no causara interferencia con otros procesos si tiene como máximo una referencia critica. En el caso de una asignación variable = expresión, es necesario que ningún proceso lea la variable o que en ese caso, no haya referencias críticas en expresión.

Las sentencias de un programa concurrente pueden referenciar a lo sumo una variable compartida, ASV, para estar libres de interferencias. Si así lo hacen, su ejecución parecerá atómica al resto de los procesos.

|  |  |
| --- | --- |
| *No Cumplen ASV:*  int a = 3, j = 6  co  **j = a + j #2 ref. Críticas**  // j = 1 #1 ref. críticas  oc | *Cumplen ASV:*  int a = 3, j = 6  co     a = a + j #1 ref. Crítica, *a* NO es leída  // j = 1 #0 ref. críticas  oc |

1. **Dado el siguiente programa concurrente:**

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

**co**

**x = y - z // z = x \* 2 // y = y - 1**

**oc**

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

x = y – z 🡪 NO CUMPLE. Dos referencias críticas, y x es leída por otros procesos.

z = x \* 2 🡪 NO CUMPLE. Una referencia critica, y z es leída por otros procesos.

Y = y – 1 🡪 CUMPLE. No posee referencias críticas al lado derecho; y puede ser leída por otros procesos.

1. **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.**

Se asigna x, luego z y ultimo y: x=1, z=4, y=3.

Si los procesos ejecutan una instrucción atómica de bajo nivel por vez: x=1, z=4, y=3.

Se asigna x, luego y, ultimo z: x=1, z=4, y=3.

Se asigna y, luego z, ultimo x: x=-1, z=4, y=3.

Se asigna y, luego x, ultimo z: x=0, z=0, y=3.

Se asigna z, luego x, último y: x=0, z=4, y=3.

Se asigna z, luego y, ultimo x: x=-1, z=4, y=3.

Nótese que el valor de y para todas las ejecuciones presentadas se mantiene en 3, dado que su asignación es la única que cumple ASV. Las variables x y z varían en su resultado.

1. **Defina acciones atómicas condicionales e incondicionales. Ejemplifique.**

**FALTA COMPLETAR**

Incondicionales: son las acciones que se utilizan únicamente para exclusión mutua, es decir, aquellas que no llevan una condición booleana.

Condicionales: son acciones que si tienen la condición booleana de espera.

1. **Defina propiedad de seguridad y propiedad de vida.**

Propiedad de seguridad (safety): asegura estados consistentes (nada malo le ocurre a un proceso). Una falla de seguridad indica que algo anda mal.

Propiedad de vida (liveness): hay progreso, no hay deadlocks (eventualmente ocurre algo bueno con una actividad). Una falla de vida indica que las cosas dejan de ejecutar.

1. **¿Qué es una política de scheduling? Relacione con fairness. ¿Qué tipos de fairness conoce?**

Una política de scheduling se encarga de elegir la tarea siguiente que hay que admitir en el sistema y el proceso siguiente que hay que ejecutar.

Fairness: trata de garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los demás.

Tipos de Fairness:

* Fairness Incondicional: una política de scheduling es incondicionalmente fair si toda acción atómica incondicional que es elegible eventualmente es ejecutada.
* Fairness Débil: una política de scheduling es débilmente fair si:
  + Es incondicionalmente fair y si toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve true y permanece hasta que es vista por el proceso que ejecuta la acción atómica condicional.
* Fairness Fuerte: una política de scheduling es fuertemente fair si:
  + Es incondicionalmente fair y si toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en true con infinita frecuencia.

**CUESTIONARIO CLASE 3 Y 4**

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

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.

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

* Si el ingreso de un lector permite el ingreso de otros, aun si hay escritores esperando 🡪 débilmente fair: el ingreso permanente de lectores causaría inanición a los escritores.
* 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.
* 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 llegue 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.

1. **Dado el siguiente programa concurrente, indique cual es la respuesta correcta (justifique claramente).**

**Int a = 1, b = 0;**

**Co < await (b = 1) a = 0 > // while (a = 1) { b = 1; b = 0 } oc**

1. **Siempre termina.**
2. **Nunca termina.**
3. **Puede terminar o no.**

**FALTA COMPLETAR**

1. **¿Qué propiedades que deben garantizarse en la administración de una sección critica 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?**

|  |  |  |
| --- | --- | --- |
| **Propiedad** | **Tipo** | **Estado “malo”** |
| Exclusión mutua | SEGURIDAD | En un momento dado, dos o más procesos se encuentran dentro de una SC. |
| Ausencia de Deadlock (Livelock) | SEGURIDAD | Ninguno de los procesos que intentan acceder a una SC tiene éxito, quedando todos bloqueados. |
| Ausencia de Demora Innecesaria | SEGURIDAD | Un proceso intenta ingresar a su SC y es bloqueado aun siendo el único que lo intenta. |
| Eventual Entrada | VIDA | No corresponde.- |

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

}

}

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

**FALTA COMPLETAR**

1. **Modifique el algoritmo Ticket para el caso en que no se dispone de una instrucción Fetch and Add.**

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

**bool lock = false; # lock compartido**

{TICKET: próximo > 0 ∧ (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i]==

próximo) ∧ (turno[i] >0) ⇒( ∀j : 1 ≤ j ≤ n, j ≠ i: turno[i] ≠ turno[j] ) ) }

process SC[i = 1 to n] {

while (true) {

sección 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] <> próximo) skip;

sección critica;

próximo = próximo + 1;

sección no critica;

}

}

En este código al no contar con FA, se realizó la toma de turnos y el incremento de número 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 critica inicial, la elección del algoritmo de ticket no fue la mejor opción en ese caso.

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

**FALTA COMPLETAR**

1. **(Opcional). Implemente una butterfly Barrier para 8 procesos usando variables compartidas.**

8 procesos = 2³ → 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 log28 = 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;

}

}

}

1. **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:

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 continua la ejecución: Equivale a: < await (valor>0) valor = valor – 1>

V, signal o incremento: incrementa el valor del semáforo, “despertando” a alguno de los procesos demorados si los hubiere. Equivale a: < valor = valor + 1 >

1. **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); #P1**

**// P(s2); P(s1); x = x \* 3; V(s1); #P2**

**// P(s1); x = x – 2; V(s2); V(s1); #P3**

**Oc**

S1 🡪 mutex: puede pasar un solo proceso por vez.

S2 🡪 semáforo de señalización: esperan señalización de un evento y pasa solo uno.

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

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.

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:

Si gana P1: asigna x=1, libera a s1 y termina; finaliza P2 y asigna x = 3. Valor final x=3.

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.

1. **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 filosofo puede comer en cualquier posición siempre que tenga los dos tenedores correspondientes).**

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]);

}

}

Utilizando monitores

monitor AdminTenedores {

int sentados = 0, pos = 1;

bool tenedores[1:5] = ([5] false);

cond hayLugar; # signal cuando sentados<2

procedure request(int &t1, int &t2) {

if (sentados==2) {

wait(hayLugar);

} else {

sentados = sentados + 1;

}

while(tenedores[pos] or tenedores[pos mod 5 +1])

pos = pos mod 5 +1;

t1 = pos;

t2 = pos mod 5 +1;

tenedores[t1] = true;

tenedores[t2] = true;

}

procedure release(int t1, int t2) {

tenedores[t1] = false;

tenedores[t2] = false;

if(isEmpty(hayLugar)) {

sentados = sentados – 1;

} else {

signal(hayLugar);

}

}

}

1. **Describa la técnica de Passing the Baton. ¿Cuál es su utilidad en la resolución de problemas mediante semáforos?**

Passing the baton permite implementar sentencias await de cualquier tipo y ofrece flexibilidad para realizar modificaciones. Sentencias de la forma: F1: < Si > F2: < await (Bi) Sj >

Se implementan con:

* Un mutex e que controla el acceso a sentencias atomicas coarse-grained.
* Un semáforo bj y un contador dj por cada sentencia await condicional, permitiendo demorar procesos hasta que Bj sea verdadero y contando la cantidad de procesos demorados.

Se reemplaza cada < por P(e). En el caso de await condicionales se debe controlar si no se cumple la Bj, se incrementa dj, se libera el mutex y se bloquea el proceso.

A la salida de cada sentencia atómica se intenta pasar el bastón de mando (e, el permiso exclusivo de ejecución) a algún proceso demorado, revisando las guardas y los contadores. Se reemplaza cada > por un bloque de la forma:

if (B1 and d1 > 0) {d1 = d1 - 1; V(b1)} # Pasa el bastón a algún proceso esperando por B1

[] ...

[] (Bn and dn > 0) {dn = dn - 1; V(bn)} # Pasa el bastón a algún proceso esperando por Bn

else V(e); # Libera el bastón, permitiendo que ingrese algún proceso

# que espera acceso a sentencias atómicas coarse-grained

fi

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

Ofrece una forma simple 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.

1. **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.**

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);

}

}

Con monitores

monitor Controlador\_RW {

int nr = 0, nw = 0;

cond okLeer;

cond okEscribir;

procedure pedido\_leer() {

while (nw>0)or(not empty(okEscribir))or(nr==10) wait(okLeer);

nr = nr + 1;

if(nr<10) { signal(okLeer); }

}

procedure libera\_leer() {

nr = nr - 1;

if(nr==0)and(not empty(okEscribir)) { signal(okEscribir); }

elseif(empty(okEscribir) and nr<10) { signal(okLeer); }

}

procedure pedido\_escribir() {

while (nr>0 OR nw>0) wait(okEscribir);

nw = nw + 1;

}

procedure libera\_escribir() {

nw = nw - 1;

if(not empty(okEscribir)) { signal(okEscribir); }

else { signal(okLeer); }

}

}

1. **(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**

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 vacíos y cada mensaje tiene que ser recibido por los n consumidores antes de que el slot pueda ser reusado. Además, cada consumidor debe recibir los mensajes en el orden en que fueron depositados (note que los distintos consumidores pueden recibir los mensajes en distintos momentos siempre que los reciban en orden). Extienda la respuesta dada en (a) para resolver este problema más general.**

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 decrementa 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).

1. **(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]);

}

}

}