Monitores

Los monitores corresponden a la segunda herramienta de sincronización que veremos en el curso. Se usan para garantizar la exclusión mutua en secciones críticas y para esperar la ocurrencia de un evento sin tener que recurrir a *busy-waiting*. Los monitores que veremos a continuación corresponden al estilo Brinch Hansen (su inventor). Los monitores de Java usan este mismo estilo y nSystem los ofrece mediante la siguiente API:

```
nMonitor nMakeMonitor( );
void nEnter(nMonitor m);
void nExit(nMonitor m);
void nWait(nMonitor m);
void nNotifyAll(nMonitor m);
void nNotify(nMonitor m);
void nDestroyMonitor(nMonitor m);
```

Un monitor se construye con *nMakeMonitor* y se destruye con *nDestroyMonitor*. La función *nEnter* se usa para anunciar el inicio de una sección crítica y *nExit* para indicar su final. La garantía que entrega un monitor *M* es que a lo más una tarea *T* puede estar entre un *nEnter(M)* y un *nExit(M)*, y en ese caso se dice que *T* tiene la propiedad de *M*. Cuando una tarea *T* invoca un *nEnter(M)* y *T'* ya tiene la propiedad de *M*, el *nEnter* de *T* espera hasta que *T'* libere el monitor con *nExit(M)*. Si hay varias tareas que compiten por la propiedad del mismo monitor, éste se asigna en un orden no especificado[1].

La función *nWait(M)* libera el monitor *M* y suspende la tarea que lo invoca hasta que otra tarea invoque *nNotifyAll(M)*. Antes de retornar, *nWait* debe esperar hasta adquirir la propiedad del monitor nuevamente.

Implementación de un Buffer con monitores

Cuando se va a resolver un problema usando monitores, es útil partir de una solución incorrecta sin sincronización y que recurre a *busy-waiting* y transformarla en una solución correcta agregándole la sincronización con monitores. Esta transformación es casi mecánica. Por ejemplo, la siguiente tabla muestra la similitud de ambas soluciones lado a lado:

<table>
<tr><th>Solución incorrecta</th><th>Solución correcta con monitores</th></tr>
<tr><td><pre>
Item buf[N];
int nextempty= 0, nextfull= 0;
int count= 0;
</pre></td><td><pre>
Item buf[N];
int nextempty= 0, nextfull= 0;
int count= 0;
nMonitor m; /* = nMakeMonitor() */
</pre></td></tr>
<tr><td><pre>
Item get() {
  Item x;

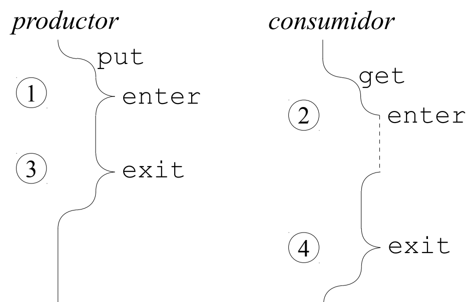
  while (count==0)
    ; /* busy-waiting */
  x= buf[nextfull];
  nextfull= (nextfull+1)%N;
  count--;

}
</pre></td><td><pre>
Item get() {
  Item x;
  nEnter(m);
  while (count==0)
    nWait(m);
  x= buf[nextfull];
  nextfull= (nextfull+1)%N;
  count--;
  nNotifyAll(m);
  nExit(m);
}
</pre></td></tr>
</table>

1 Aunque la implementación actual sí otorga el monitor por orden de llegada (FIFO).

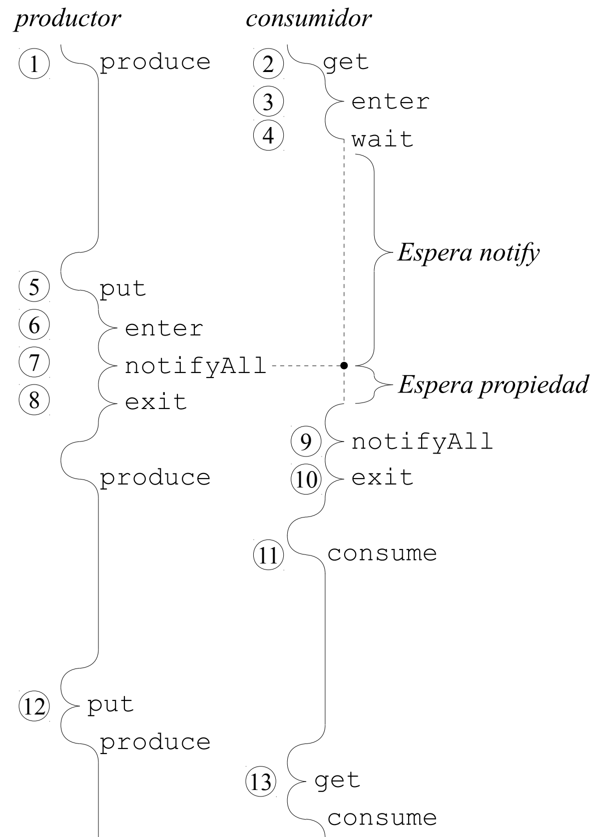
`void put(Item x) {`
 	
`  while (count==N)`
`    ; /* busy-waiting */`
`  buf[nextempty]= x;`
`  nextempty= (nextempty+1)%N;`
`  count++;`
 	
`}`
	`void put(Item x) {`
	`  `**`nEnter(m);`**
	`  while (count==N)`
	`    `**`nWait(m);`**
	`  buf[nextempty]= x;`
	`  nextempty= (nextempty+1)%N;`
	`  count++;`
	`  `**`nNotifyAll(m);`**
	`  `**`nExit(m);`**
	`}`

La siguiente figura muestra el funcionamiento de *nEnter* y *nExit*:



En 1, el productor acaba de invocar put y por lo tanto llama a nEnter para ganar la propiedad del monitor. En 2, el consumidor necesita obtener un ítem del buffer y llama a nEnter, pero debe esperar porque el monitor lo posee el productor. En 3, el productor libera el monitor y por lo tanto lo adquiere el consumidor, pudiendo extraer el ítem requerido (suponiendo que el buffer no está vacío). En 4, el consumidor libera el monitor. En este caso se dice que hubo *contención* por el monitor, es decir una tarea trató de obtener el monitor cuando era la propiedad de otra tarea.

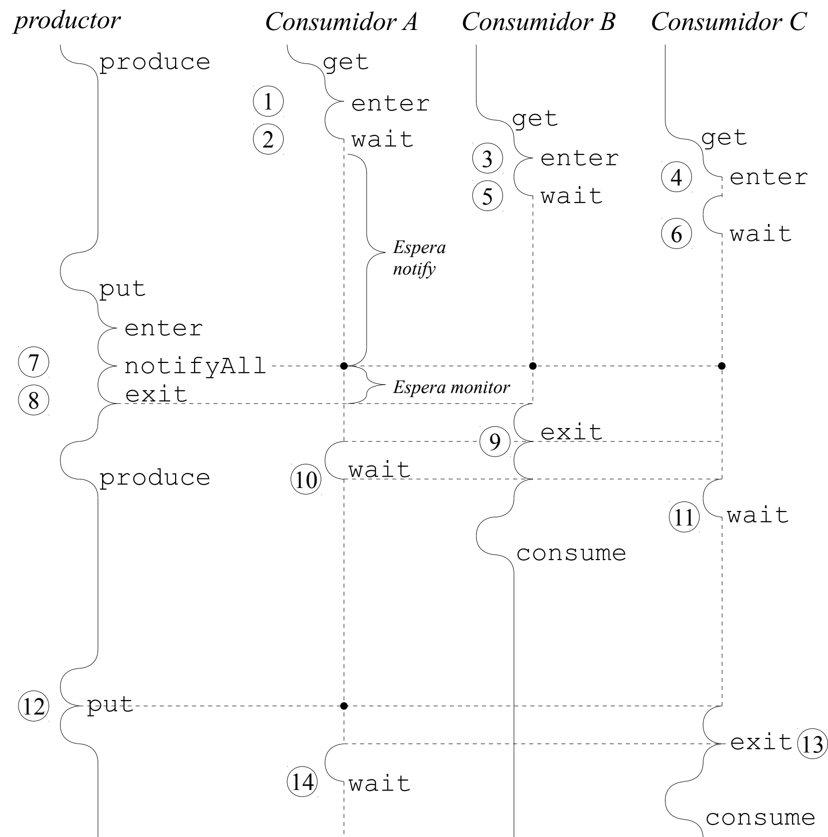
El siguiente ejemplo muestra qué pasa cuando el buffer está vacío:



En 1, el productor llama a *produce* para fabricar el primer ítem. En 2, el consumidor llama a get para obtener un ítem. Para ello adquiere el monitor en 3 llamando a *nEnter*, pero se encuentra con el buffer vacío y por lo tanto se duerme, llamando a *nWait* en 4, a la espera de una notificación. Observe que en 3 no hubo contención por el monitor, ya que este estaba libre cuando se solicitó. En 5, el productor terminó de fabricar el primer ítem y llama a *put* para depositarlo en el buffer. En 6, adquiere sin problemas el monitor porque el consumidor lo liberó en 4.

El productor llama a *nNotifyAll* en 7, lo que despierta al consumidor, pero este último continúa esperando ahora la propiedad del monitor. El consumidor *no puede continuar inmediatamente después de nNotifyAll, porque ya no habría exclusión mutua y podrían ocurrir dataraces*. En 8, el productor libera el monitor y por lo tanto lo adquiere el consumidor, el que extraerá el ítem recién depositado, sin peligro de dataraces porque el productor ya no está manipulando el buffer. Observe que en 9, el consumidor invoca un *nNotifyAll* que no despierta a ninguna tarea y luego libera el monitor en 10. Cuando se llama a *consume* en 11 se obtiene la concurrencia buscada por esta solución ya que una tarea produce un ítem al mismo tiempo que otra tarea lo consume. En 12, el productor deposita un segundo ítem en el buffer y el consumidor lo obtiene en 13 sin tener que esperar porque el buffer ya contiene el ítem solicitado.

Un punto importante a considerar es que los monitores no especifican un orden en el que se despiertan los threads después de una notificación. Esto es cierto con los monitores de Java y los de pthreads. Esto se demuestra en el siguiente diagrama en donde un productor despierta a 3 consumidores que esperan un ítem:



En 1, el consumidor A llamó a get, y por lo tanto pide el monitor, pero se encuentra con que el buffer está vacío de modo que espera llamando a *nWait* en 2 y liberando así el monitor. En 3, el consumidor B llamó a *get* y pide el monitor, que está libre. En 4, el consumidor C pide el monitor pero hay contención: no lo obtiene porque lo tiene el consumidor B y en consecuencia espera. En 5, el consumidor B se da cuenta que el buffer está vacío y llama a *nWait*, lo que libera el monitor y entonces el consumidor C adquiere el monitor. En 6, se da cuenta que está vacío y llama a *nWait*. Hay 3 consumidores que esperan un ítem.

Mientras tanto el productor estaba ocupado produciendo un ítem. En 7, llamó a *put* para depositar el ítem que acaba de producir y por lo tanto adquirió el monitor. Ahora llama a *nNotifyAll* para despertar a los 3 consumidores. Pero los consumidores no pueden continuar hasta adquirir la propiedad del monitor. En 8, el productor libera el monitor llamando a *nExit*.

En este punto, lo intuitivo es que el consumidor A adquiere el monitor, pero en general *la especificación de los monitores no garantiza el orden en que los threads adquieren el monitor después de un notify* (o *broadcast* en los caso de *pthreads*). Por eso, en este ejemplo se eligió que el consumidor B adquiriese el monitor en primer lugar (pero también pudo haberlo hecho el consumidor A o el consumidor C). Entonces éste extrae el ítem del buffer, dejándolo vacío, y libera el monitor en 9. Ahora adquiere el monitor el consumidor A (pero pudo haber sido el consumidor C) y encuentra que el buffer está vacío y por lo tanto llama a *nWait*, para continuar esperando.

Este ejemplo ***subraya la importancia de usar while en vez de if en la llamada a nWait.*** En general en las soluciones con monitores, suele ocurrir que al despertarse de un nWait, las condiciones para proceder con la operación todavía no se cumplen y se debe continuar esperando. Aún cuando parezca que en su solución sí se puede usar *if*, use *while* de todas formas porque es altamente probable que su intuición lo engañe. Y por último, el volver a evaluar la condición del *while* es un costo marginal en

tiempo de CPU.

El consumidor A libera el monitor en 10. Esto despierta al consumidor C que también encuentra vacío el buffer y llama a *nWait* en 11. En 12, el productor produce un segundo ítem y llama a *nNotifyAll*, lo que despierta a los consumidores A y C. Observe que el consumidor B está consumiendo el primer ítem. Cuando *put* llama a *nExit*, el consumidor C adquiere el monitor y extrae el segundo ítem y libera el monitor en 13. Ahora es el consumidor A el que adquiere el monitor pero continúa esperando en 14, hasta que el productor termine el 3er ítem.

Observe que nada impide que los consumidores B o C vuelvan a llamar a *get* y queden a la espera de un ítem. Cuando el productor deposite el 3er ítem, podría ocurrir que el consumidor A no sea el ganador de éste ítem y continuar así esperando.

Monitores estilo Hoare

En el diagrama anterior vimos que el *nNotifyAll* de 7 despierta 3 tareas. Una de ellas puede continuar pero las otras 2 se despertaron inútilmente. En nSystem existe la función *nNotify* que despierta una sola tarea. Esta función cumple el mismo rol que el método *notify* en Java o la función *pthread_cond_signal* de pthreads. Es tentador usar esta función para evitar despertar inútilmente a la tareas que no obtendrán un ítem. Sin embargo esto es incorrecto porque existe una muy rara condición de borde[2] en que pueden haber tanto productores como consumidores esperando en un *nWait*. Cuando el productor invoque *nNotify*, podría llegar a despertar a un productor en vez de un consumidor. En ese caso ningún consumidor se despertaría para extraer el ítem recién depositado. Tendrían que esperar a que se produzca un nuevo ítem.

Para optimizar efectivamente el problema del productor/consumidor se deben usar los monitores estilo Hoare (por el apellido de su inventor). En este estilo se introduce un nuevo tipo de datos: las *condiciones* cuya API en nSystem es la siguiente:

```
nCondition nMakeCondition(nMonitor mon);
void nDestroyCondition(nCondition cond);
void nWaitCondition(nCondition cond);
void nSignalCondition(nCondition cond);
```

La funciones *nMakeCondition* y *nDestroyCondition* construyen o destruyen una condición que es usada en conjunto con el monitor *mon*. La función *nWaitCondition* permite suspender a una tarea de la misma manera que lo hace *nWait*. La diferencia es que se puede elegir de qué condición se despertará una sola tarea con *nSignalCondition*. Con esta API se obtiene una solución eficiente del buffer del productor/consumidor:

```
Item buf[N];
int nextempty= 0, nextfull= 0;
int count= 0;
nMonitor m;        /* = nMakeMonitor() */
nCondition no_empty; /* = nMakeCondition(m); */
nCondition no_full;  /* = nMakeCondition(m); */
```

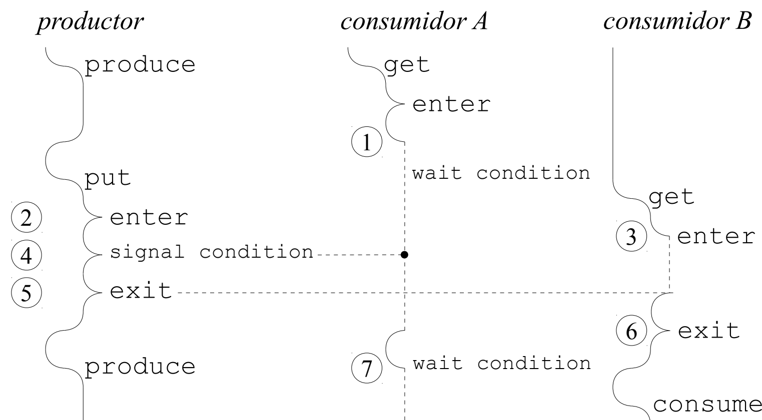
```
Item get() {
  Item x;
  nEnter(m);
```

2 Ver una explicación detallada en http://stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again

```
    while (count==0)
      nWaitCondition(no_empty); /* A */
    x= buf[nextfull];
    nextfull= (nextfull+1)%N;
    count--;
    nSignalCondition(no_full); /* B */
    nExit(m);
  }

  void put(Item x) {
    nEnter(m);
    while (count==N)
      nWaitCondition(no_full);  /* C */
    buf[nextempty]= x;
    nextempty= (nextempty+1)%N;
    count++;
    nSignalCondition(no_empty); /* D */
    nExit(m);
  }
```

En esta solución, en B se despierta a lo más un solo productor que esperaba en C, el que podrá depositar un ítem en la casi recién desocupada. Del mismo modo en D se despierta a lo más un solo consumidor que esperaba en A, que será el que extraiga el ítem recién depositado. Esto evita despertar inútilmente tareas que no podrán continuar.

Hay que destacar que se debe mantener el *while* de A y C. ***Piénselo muy bien si va a usar if antes de llamar a nWait o nWaitCondition***. En el siguiente diagrama se muestra que el consumidor que espera en A, al despertarse podría no obtener el ítem recién depositado, lo que demuestra la importancia de usar *while*.

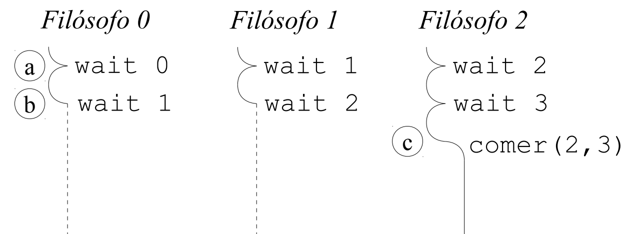


En 1 el consumidor A encuentra el buffer vacío y por lo tanto espera llamando a *nWaitCondition*. En 2, el productor llamó a *put* para depositar un ítem y obtiene el monitor. En 3, el consumidor B solicita el monitor para extraer un ítem del buffer, pero hay contención, es decir el monitor está ocupado y por lo tanto espera. El consumidor B no espera porque llamó a *nWaitCondition*, si no que espera en el *nEnter* por la propiedad del monitor. En 4, el productor llama a nSignalCondition y despierta al consumidor A, la única tarea que esperaba en la condición *no_empty*. En este instante tanto el consumidor A como el consumidor B compiten por obtener el monitor. Cuando en 5, el productor libera el monitor, no está especificado quién obtiene el monitor. Elegimos que lo obtenga el consumidor B, el que encuentra un ítem en el buffer y lo extrae. En 6 libera el monitor y ahora sí lo

recibe el consumidor A, el que encuentra el buffer vacío y por lo tanto debe llamar nuevamente a *nWaitCondition*. De haberse usado *if* en vez de *while* en el código de más arriba, el consumidor A hubiese continuado erróneamente con la extracción de un ítem que ya fue extraído por el consumidor B, entregando así basura.

Implementación de los filósofos con monitores

Ya vimos una implementación satisfactoria de los filósofos con semáforos. Pero esa solución tiene un problema de eficiencia: puede ocurrir que un filósofo tenga que esperar aún cuando los 2 palitos que necesita para comer están libres. Esta situación se muestra en el siguiente diagrama:



En *a*, los filósofos 0, 1 y 2 solicitan con éxito el primero de los palitos que necesitan para comer. En *b* solicitan el segundo palito, pero solo el filósofo 2 lo logra. El filósofo 1 espera porque el palito 2 lo tiene reservado el filósofo 2 y el filósofo 0 espera porque el palito 1 lo tiene reservado el filósofo 1. En *c* el filósofo 2 comienza a comer. El problema es que en esta situación el filósofo 0 podría comer porque el filósofo 1 no está comiendo con el palito 1. Solo lo tiene reservado.

Lo que se necesita es un mecanismo en donde un filósofo pidiese ambos palitos y reserve ambos si están libres o ninguno si alguno de ellos está reservado. Esto es difícil de lograr con semáforos porque no existe un mecanismo para pedir un semáforo pero no bloquearse si el semáforo no tiene tickets. Sin embargo esto sí se puede lograr de manera sencilla con monitores, como veremos a continuación.

La idea es mantener un arreglo de 5 valores booleanos que digan si una palito está reservado o no y un solo monitor que garantiza la exclusión mutua al acceder a ese arreglo.

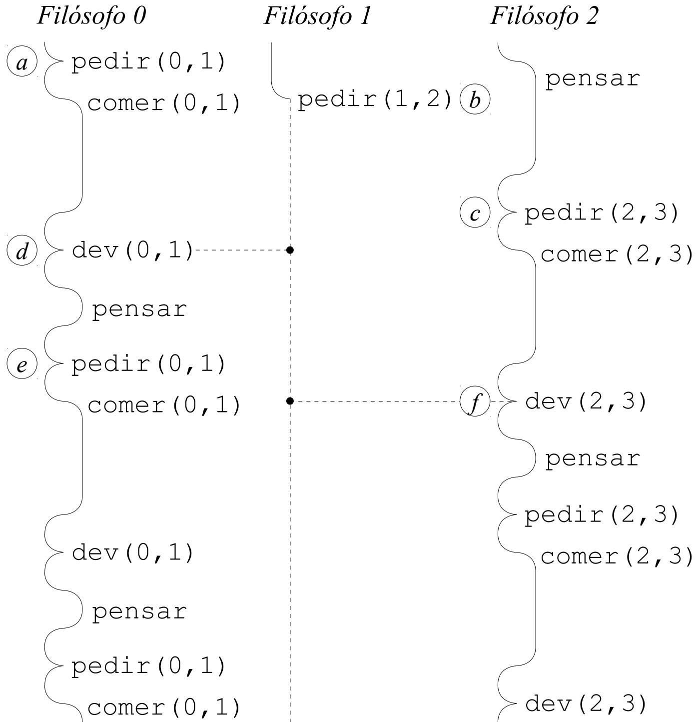
```
int palitos[5]= { FALSE, FALSE, FALSE, FALSE, FALSE};
nMonitor m; /* = nMakeMonitor(); */
void filosofo(int i) {
  for(;;) {
    pedir(i, (i+1)%5);
    comer(i, (i+1)%5);
    devolver(i, (i+1)%5);
    pensar();
} }

void pedir(int i, int j) {
  nEnter(m);
  while (palitos[i] || palitos[j])
    nWait(m);
  palitos[i]= palitos[j]= TRUE;
  nExit(m);
}

void devolver(int i, int j) {
  nEnter(m);
  palitos[i]= palitos[j]= FALSE;
  nNotifyAll(m);
```

```
    nExit(m);
  }
```

Esta solución logra obtener la eficiencia buscada, pero introduce el defecto que veremos a continuación.

Hambruna

El problema de esta solución es que un par de filósofos se pueden poner de acuerdo para que un tercer filósofo no pueda comer nunca. Esto se denomina hambruna o *starvation* en inglés. Esto se demuestra en el siguiente diagrama:



En *a*, el filósofo 0 obtiene los palitos 0 y 1. En *b*, el filósofo 1 solicita los palitos 1 y 2, pero no obtiene ninguno de los 2 porque el palito 1 está ocupado. En *c*, el filósofo 2 obtiene los palitos 2 y 3, justo antes de que el filósofo 0 libere en *d* los palitos 0 y 1. Entonces, el filósofo 1 se despierta y ve que el palito 1 está libre, pero esta vez es el palito 2 el que está ocupado de modo que continúa esperando. En *e*, el filósofo 0 le da hambre nuevamente y quiere comer. Obtiene los palitos 0 y 1, justo antes de que el filósofo esté satisfecho y devuelva en *f* los palitos 2 y 3. Nuevamente se despierta el filósofo 1 para descubrir que el palito 1 está libre pero no el palito 0, y continúa esperando. Y así cada vez que se despierta el filósofo 1 descubre que uno de los palitos está ocupado porque siempre habrá un filósofo que comience a comer justo antes de que el otro libere sus palitos. De esta forma, el filósofo 1 se morirá de hambre.

Cuando una solución se programa de tal forma que ningún thread puede sufrir hambruna se dice en Inglés que la solución es *fair*, o que posee *fairness* como atributo. Se acepta traducir al español este término señalando que una solución es justa o equitativa, pero es una mala traducción porque en ningún caso significa que los threads tengan la misma probabilidad de que su solicitud sea satisfecha. El significado de *fairness* es menos exigente: una solicitud será satisfecha tarde o temprano, pero mientras para un thread podría ser temprano, para otro podría ser tarde.

Observe que la hambruna es distinta de un deadlock porque afecta a un solo thread. En el caso del deadlock, todos los filósofos se mueren de hambre, pero porque todos esperan que otro filósofo libere el palito que necesitan, aunque esto nunca ocurrirá.

La situación de hambruna se da en muchas soluciones y se considera un aspecto negativo, pero a veces se acepta porque evitarla puede ser complejo. En este curso se explicitará en el enunciado de los controles o tareas cuando se debe evitar la hambruna. Si el enunciado no dice nada, su solución sí puede sufrir de hambruna.

También es importante hacer notar que usualmente no se considera hambruna cuando un thread no obtiene el recurso que solicita debido a la implementación específica de una herramienta de sincronización (semáforos, monitores, etc.). Por ejemplo, en la implementación recién vista del buffer con monitores para el productor/consumidor, puede ocurrir que un consumidor espere indefinidamente debido a la manera en que se despiertan después del *notify*. En este curso esto no se considera hambruna porque no son los demás consumidores los que se pusieron de acuerdo de alguna forma para que espere eternamente. Sin embargo podrían haber otros profesores o libros de sistemas operativos que sí consideran esto como una forma de hambruna atribuible a la herramienta de sincronización.

Lectores/escritores

En este problema varios procesos concurrentes comparten una misma estructura de datos y necesitan consultarla o actualizarla (modificarla). Consideremos el caso de un diccionario global por ejemplo con operaciones para definir, consultar o eliminar una palabra. Un thread que define o elimina una palabra se denomina *escritor*, mientras que si consulta se llama un *lector*.

Desde un punto de vista de la implementación es claro que si dos threads eliminan o definen una palabra concurrentemente se pueden producir dataraces comprometiendo la consistencia del diccionario. Por lo tanto la implementación de definir y eliminar requiere que se garantice la exclusión mutua. Por otra parte si varios threads consultan el diccionario concurrentemente y la implementación de la consulta solo lee la estructura del diccionario, entonces no habrá ningún datarace. Entonces, es tentador no garantizar la exclusión mutua para el caso de la operación de consulta. Sin embargo esto sería un grave error porque también se puede consultar al mismo tiempo que se modifica el diccionario. Esto puede gatillar un *segmentation fault* o respuestas inverosímiles como que una palabra existe en el diccionario, pero en realidad nunca estuvo presente.

En consecuencia la implementación trivial de la concurrencia en este diccionario global consistiría en usar por ejemplo un monitor para garantizar la exclusión mutua en todas las operaciones, incluyendo las consultas. En el problema de los lectores/escritores se descarta esta solución y se exigen las siguientes propiedades:

- Se debe permitir varios procesos lectores al mismo tiempo.
- Los escritores deben proceder en exclusión mutua con otros escritores o lectores.

Primera solución con hambruna

Veamos una primera solución de este problema usando los monitores de nSystem. La implementación de los lectores debe solicitar permiso para comenzar una consulta llamando al procedimiento *enterRead* y notificar su finalización llamando a *exitRead*. De manera análoga los escritores deben solicitar permiso para comenzar una modificación con *enterWrite* y notificar su término llamando a *exitWrite*. Esto se ejemplifica con este bosquejo de la implementación de consultar y definir:

```
char *consultar(char *key) {
  char *ret;
  enterRead();
  ... implementación de la consulta ...
  ret= …;
  exitRead();
  return ret;
}
```

```
void definir(char *key, char *val)
{
  enterWrite();
  ... implementación de la definición ...
  exitWrite();
}
```

La siguiente es una implementación de los procedimientos de solicitud de entrada y notificación de salida usando los monitores de nSystem.

```
nMonitor m;
int readers= 0;
int writing= FALSE;
```

```
void enterRead() {
  nEnter(m);
  while (writing)
    nWait(m);
  readers++;
  nExit(m);
}

void exitRead() {
  nEnter(m);
  readers--;
  if (readers==0)
    nNotifyAll(m);
  nExit(m);
}
```

```
void enterWrite() {
  nEnter(m);
  while (readers>0 || writing)
    nWait(m);
  writing= TRUE;
  nExit(m);
}

void exitWrite() {
  nEnter(m);
  writing= FALSE;
  nNotifyAll(m);
  nExit(m);
}
```

Ejercicio: confeccione un diagrama de threads que muestre que esta solución sufre de hambruna. Para ello muestre que 2 lectores se pueden poner de acuerdo para que nunca se otorgue el permiso de entrada a un escritor.

La metáfora de la isapre

Una forma de resolver el problema de la hambruna es satisfacer las solicitudes de entrada en orden de llegada. Luego las operaciones pueden ocurrir concurrentemente. Para lograr esto se opera de manera similar al otorgamiento de números de atención en servicios como la isapre, la farmacia, registro civil, etc. En este sistema, una persona saca un número de atención de un distribuidor de tickets a la entrada y espera a que su número aparezca en un visor digital en la sala de atención. La nueva solución queda así:

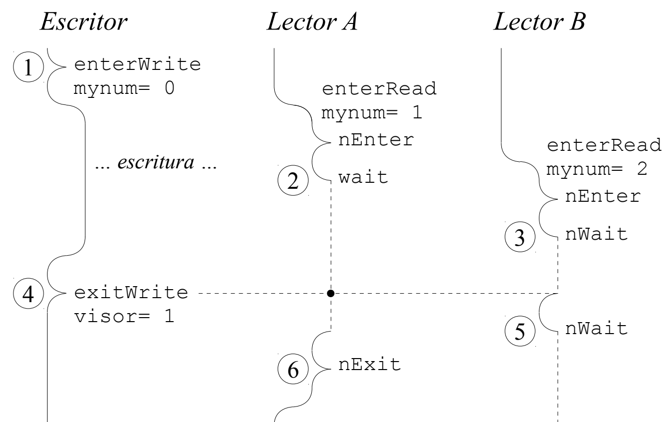
```
nMonitor m;
int readers= 0;
int dist= 0, visor= 0;
```

```
void enterRead() {
  int mynum;
  nEnter(m);
```

```
void enterWrite() {
  int mynum;
  nEnter(m);
```

<pre>   mynum= dist++;
    while (mynum!=visor)
      nWait(m);
    readers++;
    visor++;
    nNotifyAll(m); /* A */
    nExit(m);
  }
 	
  void exitRead() {
    nEnter(m);
    readers--;
    if (readers==0)
      nNotifyAll(m);
    nExit(m);
  }</pre>
	<pre> mynum= dist++;
	    while (readers>0 || mynum!=visor)
	      nWait(m);
	    nExit(m);
	  }
	
	  void exitWrite() {
	    nEnter(m);
	    visor++;
	    nNotifyAll(m);
	    nExit(m);
	  }</pre>

Observe que en A se necesita despertar a todos los threads en espera. De no hacerlo podría ocurrir la situación del siguiente diagrama:



En 1, el escritor obtiene el permiso de escritura. En 2, un lector A llamó a enterRead, pidió el monitor pero no puede continuar porque su número no coincide con el que aparece en el visor. En 3 le ocurre lo mismo a al lector B. En 4, el escritor notifica su salida, despertando a ambos lectores. No determinísticamente recibe el monitor el lector B, pero no puede continuar porque su número no coincide con el del visor. En buenas cuentas no puede continuar porque llegó después que el lector A. En 5, el lector B libera el monitor llamando a *nWait* y lo recibe el lector A, el que sí puede continuar porque su número coincide con el del visor. El problema es que al no llamar a *nNotifyAll*, el lector 2 contínua en espera, cuando sí está en condiciones de continuar. Esto disminuye el paralelismo de la solución. Por eso se necesita el *nNotifyAll* marcado con la nota A en el código de más arriba. Al estar presente, el Lector 2 se despierta nuevamente y ahora sí descubre que su número coincide con el del visor y procede con la lectura.

Análisis

La solución anterior resuelve el problema de la hambruna para el caso de los lectores/escritores. Sin embargo puede llegar a ser muy ineficiente cuando hay *n* lectores pendientes y el único escritor termina su escritura. Para que todos los lectores comiencen a trabajar se pueden llegar a requerir $O(n^2)$ cambios de contexto entre tareas. Esto puede suceder porque se podrían despertar los *n-1* threads equivocados antes de encontrar que el *n*-ésimo thread le toca su turno. Pero esto recién permitiría que trabajase el

primer lector. Para encontrar el próximo lector, nuevamente se podrían despertar *n-2* threads equivocados. Y así, si se realiza la suma, en el peor caso se llega a $O(n^2)$ ocasiones en que los threads se despiertan equivocadamente.

Solución eficiente con monitores estilo Hoare

La hambruna se puede resolver de una manera más eficiente usando las condiciones del estilo de monitores de Hoare para otorgar las autorizaciones de entrada en orden de llegada. La solución es más compleja y usa un cola *q* de requerimientos de lectura/escritura (tipo *Request*). Además se define el tipo *Ctrl* que encapsula toda la información necesaria para controlar el flujo de lectores/escritores. De esta forma, esta solución se puede usar para múltiples instancias de estructuras de datos que exhiban el patrón lector/escritor.

```
typedef struct {
  nMonitor m;
  FifoQueue q;
  int readers, writing;
} Ctrl;

#define READER 1
#define WRITER 2
typedef struct {
  int kind; /* READER || WRITER */
  nCondition w;
} Request;
```

```
Ctrl *makeCtrl() {
  Ctrl *c= (Ctrl)nMalloc(sizeof(*c));
  c->m= nMakeMonitor();
  c->q= MakeFifoQueue();
  c->readers= 0;
  c->writing= FALSE;
  return c;
}
```

```
void await(Ctrl *c, int kind) {
  Request r;
  r.kind= kind;
  r.w= nMakeCondition(m);
  PutObj(c->q, &r); /* al final de q */
  nWaitCondition(r.w);
  nDestroyCondition(r.w);
}
```

```
void wakeup(Ctrl *c) {
  Request *pr= (Request*)GetObj(c->q);
  if (pr==NULL)
    return;
  if (pr->kind==READER && !c->writing)
    nSignalCondition(pr->w);
  else if (pr->kind=WRITER &&
      c->readers==0 && !c->writing)
    nSignalCondition(pr->w);
  else /* se devuelve al comienzo de q */
    PushObj(c->q, pr);
}
```

```
void enterRead(Ctrl *c) {
  nEnter(c->m);
  if (c->writing ||
      !EmptyFifoQueue(c->q)) /* A */
    await(c, READER);
  c->readers++;
  wakeup(c); /* C */
  nExit(c->m);
}

void exitRead(Ctrl *c) {
  nEnter(c->m);
  c->readers--;
  if (c->readers==0)
    wakeup(c);
  nExit(c->m);
```

```
void enterWrite(Ctrl *c) {
  nEnter(m);
  if (c->readers>0 || c->writing ||
      !EmptyFifoQueue(c->q)) /* B */
    await(c, WRITER);
  c->writing= TRUE;
  nExit(c->m);
}

void exitWrite(Ctrl *c) {
  nEnter(c->m);
  c->writing= FALSE;
  wakeup(c);
  nExit(c->m);
}
```

}	

En esta solución es la cola *q* la que garantiza un orden FIFO en la atención de las tareas. Observe que en C, nuevamente es necesario llamar a *wakeup* para así lograr que otros lectores que vienen a continuación puedan ser autorizados para realizar su lectura concurrentemente con otras lecturas.

Este es uno de los pocos casos en donde en A y B *hay que usar if en vez de while*. Para convencerse resuelva los siguientes ejercicios:

- Suponga que en A y B de la solución anterior se reemplazan los *if* por *while*. Haga un diagrama de tareas mostrando un deadlock.
- Suponga que en A y B de la solución anterior se elimina la verificación de que la cola no esté vacía. Haga un diagrama de tareas mostrando que en ese caso puede existir una condición de borde en que se otorgue erróneamente una autorización de entrada. Piense en una situación en que hay contención: una primera tarea acaba de llamar a *nEnter* y espera obtener el monitor porque una segunda tarea está notificando su salida y por lo tanto llamará a *wakeup*.
- Suponga que en A y B de la solución anterior se elimina la verificación de que la cola no esté vacía y además se reemplazan los *if* por *while*. Haga un diagrama mostrando que una tarea podría recibir la autorización de entrada a pesar de que otra tarea lleva *un buen rato*[3] esperando entrar.

El patrón de uso de monitores

La mayoría de los problemas de sincronización se resuelven usando el siguiente patrón:

```
nEnter(m);
while ( … )
  nWait(m);     /* A */
… consulta/actualización de la sincronización …
nNotifyAll(m); /* B */
nExit(m);
```

Según el problema podría no ser necesario A o B. Sin embargo, en problemas en donde se requiere atender solicitudes por orden de llegada, tome como ejemplo la solución de los lectores/escritores con monitores de Hoare.

Ejercicios

- Resuelva el problema de los lectores/escritores otorgando los permisos de entrada por orden de llegada pero considerando la siguiente excepción: cuando un escritor libera el monitor y le toca el turno a un lector, entonces se da permiso de entrada a todos los lectores pendientes hasta ese instante, aún cuando hayan llegado después que otros escritores que todavía no reciben permiso de entrada. Observe que si un lector llega después de haber dado ese permiso, entonces debe esperar si habían escritores pendientes previamente porque se aplica el principio que llegó más tarde que otras tareas pendientes.
- Resuelva el control 1 del semestre Otoño 2009 de sistemas operativos, publicado en: http://users.dcc.uchile.cl/~lmateu/CC4302/controles/c1-092.pdf.

3 Si 2 tareas llegan casi al mismo tiempo, pero el monitor está ocupado, podría ocurrir trivialmente que reciban autorización en el orden inverso de su llegada. Esto es inevitable debido a que el monitor no necesariamente se entrega en orden de llegada. Esta es una de la razones por las cuales en sus soluciones debe minimizar el tiempo en que las tareas poseen el monitor.

- Resuelva la pregunta 1 del examen del semestre Otoño 2012 de sistemas operativos, publicado en: http://users.dcc.uchile.cl/~lmateu/CC4302/controles/ex-121.pdf.
- Resuelva el problema 2 del control 1 del semestre Otoño 2012 de sistemas operativos, publicado en: http://users.dcc.uchile.cl/~lmateu/CC4302/controles/c1-121.pdf.