

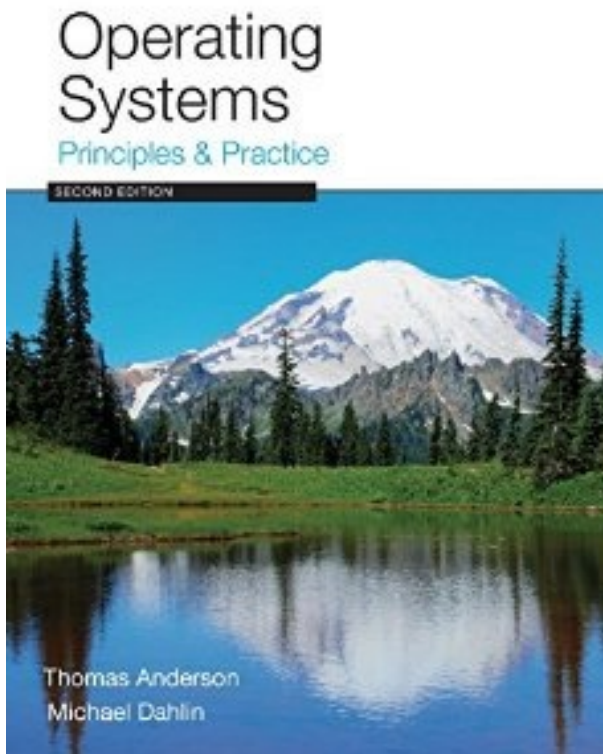


Sistemas Operativos

Capítulo 5 Acceso Sincronizado a Objetos Compartidos

Prof. Javier Cañas R.

Estos apuntes están tomados en parte del texto:
“Operating System: Principles and Practice” de T.
Anderson y M. Dalhin



Motivación

- Cuando dos o más threads concurrentemente leen/escriben memoria compartida, el comportamiento del programa queda indefinido
 - Si dos threads escriben concurrentemente la misma variable ¿cuál debería ganar?
- La itineración de threads es no determinística
 - El comportamiento puede cambiar cada vez que se corra el programa
- Tanto el compilador como el hardware pueden reordenar las instrucciones de máquina
- Las operaciones sobre un conjunto de palabras de memoria *no son atómicas*.

Pregunta: Pánico?

Thread 1

```
p = someComputation();  
pInitialized = true;
```

Thread 2

```
while (! pInitialized )  
    ;  
q = someFn(p);  
if (q != someFn(p))  
    panico
```

Thread 1

```
p = someComputation();  
pInitialized = true;
```

Thread 2

```
while (! pInitialized )  
    ;  
q = someFn(p);  
if (q != someFn(p))  
    panico
```

Para maximizar el paralelismo a nivel de instrucción (ILP), el HW o compilador pueden hacer pInitialized = true antes del cálculo de p

Reordenamiento de instrucciones

- ¿Por qué los compiladores reordenan instrucciones?
 - Los procesadores modernos pueden ejecutar varias instrucciones de máquina al mismo tiempo. Si el resultado de una instrucción se necesita para la siguiente, el procesador detiene la instrucción momentáneamente. Para evitar esto el compilador reordena las instrucciones.
- ¿Por qué la CPU reordena instrucciones?
 - Escritura en buffers: permite ejecutar la siguiente instrucción mientras se completa la escritura.

Condiciones de carrera

- Una condición de carrera (*race condition*) se produce cuando el comportamiento de un programa depende de la forma como se intercalan las operaciones de los diferentes threads.
- Los threads hacen una carrera entre sus operaciones. El resultado del programa depende del thread que gana.

Ejemplo 1

- El siguiente ejemplo es el programa más simple del mundo donde dos threads cooperan:

Thread 0
`x=1;`

Thread 1
`x=2;`

- ¿Cuáles son los posibles valores finales de x?

Ejemplo 2

- El segundo programa más simple del mundo donde dos threads cooperan:

Inicialmente $y=2$

Thread 0

$x=y+1;$

Thread 1

$y=y*2;$

- ¿Cuáles son los posibles valores finales de x ?

Ejemplo 3

- El tercer programa más simple del mundo donde dos threads cooperan:

Thread 0

`x=x+1;`

Thread 1

`x=x+2;`

- ¿Cuáles son los posibles valores finales de x?

Respuesta Ejemplo 3

load r1,x	
add r2,r1,1	
store x, r2	
	load r1,x
	add r2,r1,2
	store x, r2

Final: x==3

load r1,x	
	load r1,x
add r2,r1,1	
	add r2,r1,2
	store x, r2
store x, r2	

Final: x==1

load r1,x	
	load r1,x
add r2,r1,1	
	add r2,r1,2
store x, r2	
	store x, r2

Final: x==2

Atomicidad de Operaciones

- En el ejemplo 3, el código fue desarmado para analizarlo. ¿Qué pasa si el código no se puede desarmar?.
- Se denominan operaciones atómicas a operaciones indivisibles. Una operación atómica no puede dividirse.
- En arquitecturas modernas load y store son indivisibles para palabras de 32bits.

Demasiada leche

	Alumno A	Alumna B
12:30	Abrir refrigerador: no hay leche	
12:35	Salir a comprar	
12:40	Llegada al almacén	Abrir refrigerador: no hay leche
12:45	Comprar leche	Salir a comprar
12:50	Llegada al Depto. Guardar la leche	Llegada al almacén
12:55		Comprar leche
13:00		Llegada al Depto. Guardar la leche
		Oh!

Definiciones

- **Condición de carrera:** la salida de un programa concurrente depende en el orden en que se realizan las operaciones entre threads.
- **Exclusión mutua:** sólo un thread realiza una tarea particular a la vez.
- **Sección crítica:** parte de un código donde sólo un thread puede ejecutarse a la vez. Acceso a datos compartidos
- **Lock:** previene a un thread de hacer algo

Lock

- **Lock:** previene a un thread de hacer algo
 - Lock antes de entrar a una sección crítica, antes de acceder datos compartidos
 - Unlock al abandonar, después de acceder los datos compartidos.
 - Esperar si la sección crítica está con Lock. Todo sincronismo siempre requiere espera

Propiedades de un Código

- Son dos las propiedades que debe cumplir un código **correcto**:
 1. **Seguridad**: El programa nunca entra en un ***mal estado***.
 2. **Vivacidad**: El programa eventualmente entra en un ***buen estado***.

Demasiada leche: solución 1

- Una solución es que algunos de los estudiantes dejen una nota en el refrigerador antes de salir a comprar leche. En el código una nota es un flag (nota).
- Propiedades del código:
 - **Vivacidad**: Si se necesita leche, alguien compra
 - **Seguridad**: Nunca más de una persona compra
- Entonces cada thread ejecuta el siguiente código:

Demasiada leche: solución 1

- Entonces cada thread ejecuta el siguiente código:

```
if (milk==0) { //no hay leche
    if (nota ==0) {// no hay nota
        nota=1; //dejar nota
        milk++; //comprar leche
        nota=0; //remover nota
    }
}
```

Análisis solución 1

- Esta solución viola el criterio de seguridad:

```
if (milk==0) {
```

```
    if (nota ==0) {  
        nota=1;  
        milk++;  
        nota=0;  
    }
```

```
if (milk==0) {  
    if (nota ==0) {  
        nota=1;  
        milk++;  
        nota=0;  
    }
```

A veces funciona y
a veces falla (Heisenbug)

Demasiada leche: solución 2

- Se dejan dos notas. Éstas se crean antes de verificarlas

Hebra A

```
notaA=1; // se deja nota
if (notaB==0) {
    if (milk ==0) {
        milk++;
    }
}
notaA=0; //se elimina nota
```

Hebra B

```
notaB=1; // se deja nota
if (notaA==0) {
    if (milk ==0) {
        milk++;
    }
}
notaB=0; //se elimina nota
```

- La idea es dejar una nota “Voy a comprar leche”.

Análisis solución 2

- La solución es sólida y cumple criterio de seguridad. Lamentablemente no cumple el criterio de vivacidad (al menos una estudiante debe comprar si falta leche)
- Es posible que ambos threads dejen sus respectivas notas, y ambos deciden no comprar leche!

Demasiada leche: solución 3

- Nos aseguraremos que al menos uno de los threads determine si el otro ha comprado o no antes de decidir la compra

Hebra A

```
notaA=1; // se deja nota
while (notaB==1) {
    ;
}
if (milk ==0) {//línea M
    milk++;
}
notaA=0; //se elimina nota
```

Hebra B

```
notaB=1; // se deja nota
if (notaA==0) {
    if (milk ==0) {
        milk++;
    }
}
notaB=0; //se elimina nota
```

Discusión

- La solución 3 funciona correctamente. Observar que el código *Hebra B* no tiene loops, lo que significa que B termina la ejecución y $\text{NotaB}=0$.
- La solución tiene vivacidad y seguridad, pero la solución es:
 - Compleja
 - Asimétrica: códigos diferentes
 - Ineficiente: A está en “busy waiting” consumiendo CPU
 - Falla si el SW o HW reordenan instrucciones

La mejor solución

- Se desarrollará un método para generar programas donde múltiples threads pueden acceder estados compartidos. La idea consiste en crear **objetos compartidos** que utilizan **objetos de sincronización** para la coordinación
- La primitiva `lock` asegura que sólo un thread a la vez puede obtener un `lock`.

Demasiada leche: lock

Clase Kitchen

Método: buyIfNeeded()

```
Kitchen::buyIfNeeded(){  
    lock.acquire();  
    if (milk ==0) {  
        milk++;  
    }  
    lock.release();  
}
```

Objetos Compartidos

- Los objetos compartidos pueden ser accedidos en forma segura por múltiples threads.
- Todos sus estados compartidos: heap, variables estáticas y globales deben encapsularse en uno o más objetos.
- Los objetos compartidos extienden la orientación orientada a objeto. Objetos compartidos ocultan detalles de sincronización a través de una interfaz limpia y simple.

Implementación de Objetos Compartidos

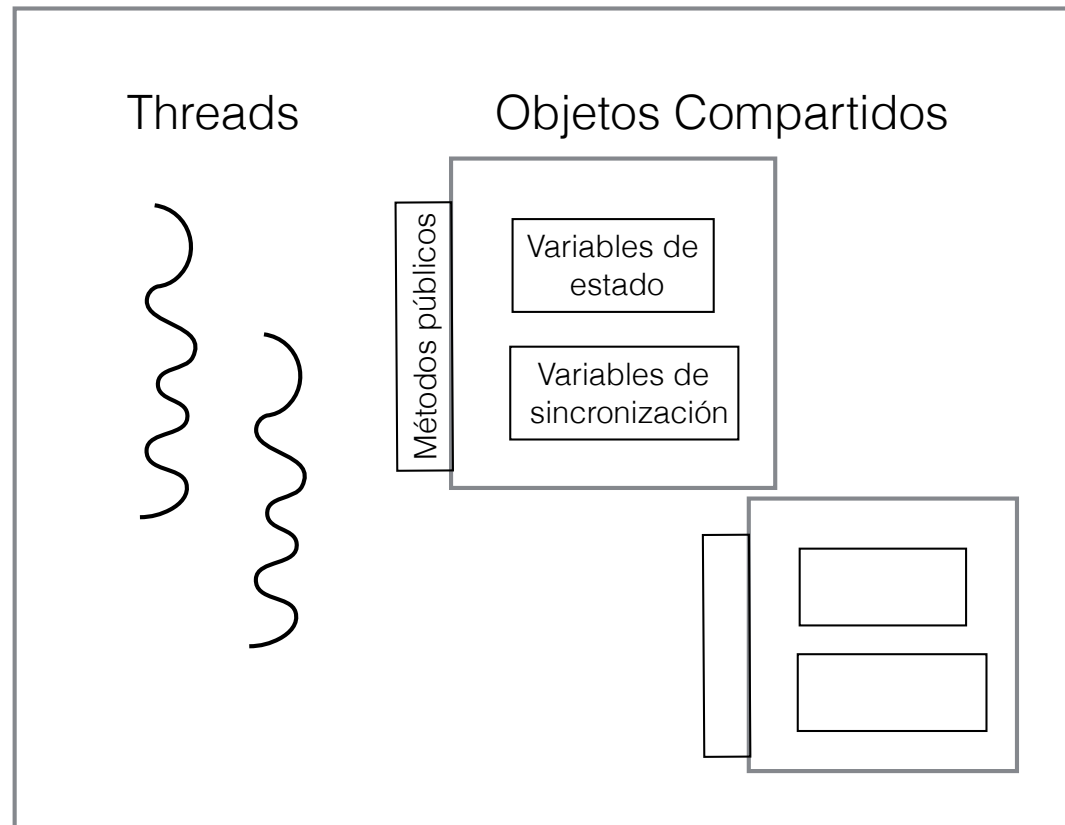
- Internamente, objetos compartidos deben manejar detalles de sincronización.
- Los objetos compartidos definen la lógica específica de la aplicación y ocultan detalles de implementación.
- Objetos compartidos incluyen variables de sincronización como variables miembros (variables asociadas a objetos específicos y accesibles por métodos)
- La atomicidad de instrucciones es necesaria para la implementación.

Objetos Compartidos y Variables de Sincronización

Hoja de Ruta

Aplicaciones Concurrentes
Objetos Compartidos Ej.: Buffers Limitados, Peluqueros, Filósofos, ...
Objetos de Sincronización Semáforos Locks Variables de Condición
Atomicidad de Instrucciones Inhibir interrupciones Test&Set, Swap
Aspectos de Hardware Multiprocesadores Interrupciones de HW

Objetos Compartidos



Exclusión Mutua: Lock

- Lock es una variable de sincronización que proporciona ***exclusión mutua***. Si un thread retiene un lock, ningún otro thread lo puede hacer.
- El conjunto de operaciones que se realizan reteniendo el lock aparecen como atómicas (indivisibles) para los otros threads.
- Un lock tiene dos métodos: `Lock::acquire()` y `Lock::release()`.
- Un lock sólo puede estar: BUSY o FREE

Métodos del Lock

- `Lock::acquire()` espera hasta que lock esté FREE, y entonces, atómicamente queda BUSY.
- Si hay múltiples threads esperando, a lo más uno tiene éxito.
- `Lock::release()` el Lock queda en estado FREE. Si hay threads con acquire pendientes, otro thread retiene el lock.

Ejemplo

```
lock.acquire();  
if (milk ==0) {  
    milk++;  
}  
lock.release();
```


Propiedades

- Una solución al problema del acceso sincronizado a objetos compartidos debe cumplir con 3 propiedades:
 1. **Exclusión Mutua**: A lo más un thread retiene el Lock.
 2. **Progreso**: Si ningún thread retiene el Lock y algún thread intenta adquirirlo, entonces algún thread tendrá éxito en adquirir el Lock.
 3. **Espera acotada**: Si el thread T intenta adquirir el Lock, entonces existe una cota en el número de veces que otros threads adquieren el Lock, antes que lo haga T.

Discusión

- La propiedad 1, exclusión mutua es la propiedad de *seguridad*.
- Las propiedades 2 y 3 son propiedades de *vivacidad*.
- La tercera propiedad define la *equidad*.
- Una **no propiedad**: No se asegura que los threads adquieran el Lock en algún orden establecido en el tiempo.

Secciones Críticas

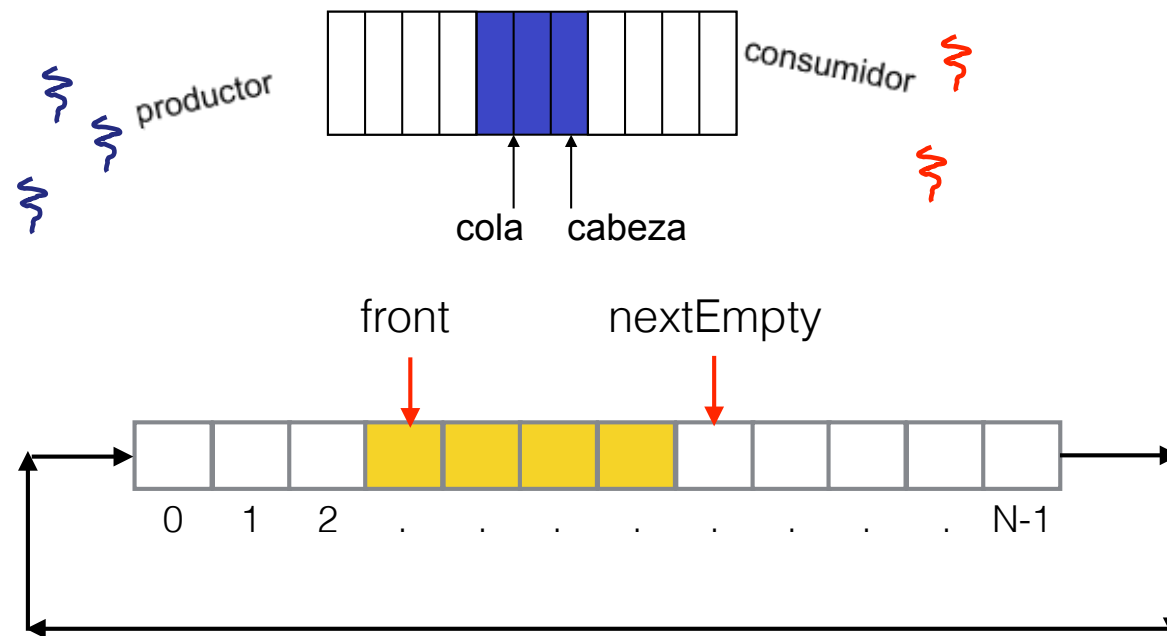
- Una sección crítica es una secuencia de código que opera sobre un estado compartido.
- Una sección crítica es una secuencia de código que atómicamente accesa un estado compartido

Ejemplo: Cola delimitada con threads seguros

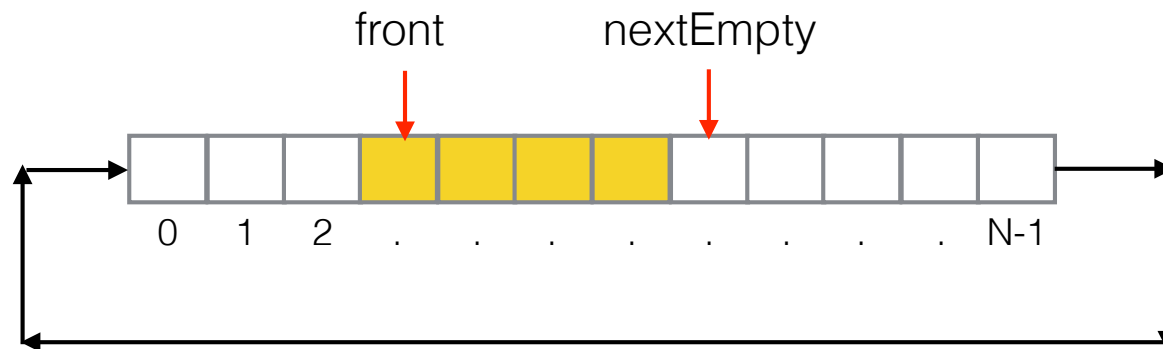
- Cada objeto compartido es una instancia de una clase que define sus estados y los métodos que operan sobre ese estado.
- Los estados incluyen variables (`int`, `float`, `array`, ...) y variables de sincronización (`locks`).
- En cualquier momento se utiliza un constructor de clase para producir otra instancia del objeto compartido.

Cola delimitada

Productor-Consumidor



Invariantes



- ▶ El número total de elementos que se han insertado en la cola es **nextEmpty**
- ▶ El número total de elementos que se han removido de la cola es **front**
- ▶ $\text{front} \leq \text{nextEmpty}$
- ▶ El número actual de items en la cola es $\text{nextEmpty} - \text{front}$
- ▶ $\text{nextEmpty} - \text{front} \leq \text{MAX}$
- ▶ front y nextEmpty van creciendo en el tiempo pero las inserciones son: $\text{item}[\text{nextEmpty} \% \text{MAX}]$ y remociones $\text{item} = \text{item}[\text{front} \% \text{MAX}]$

TSQueue.h

```
// TSQueue.h
// Thread-safe queue interface

const int MAX = 10;

class TSQueue {
    // Synchronization variables
    Lock lock;

    // State variables
    int items[MAX];
    int front;
    int nextEmpty;

public:
    TSQueue();
    ~TSQueue(){};
    bool tryInsert(int item);
    bool tryRemove(int *item);
};
```

TSQueue.cc

```
// Try to insert an item. If the queue is
// full, return false; otherwise return true.
bool
TSQueue::tryInsert(int item) {
    bool success = false;

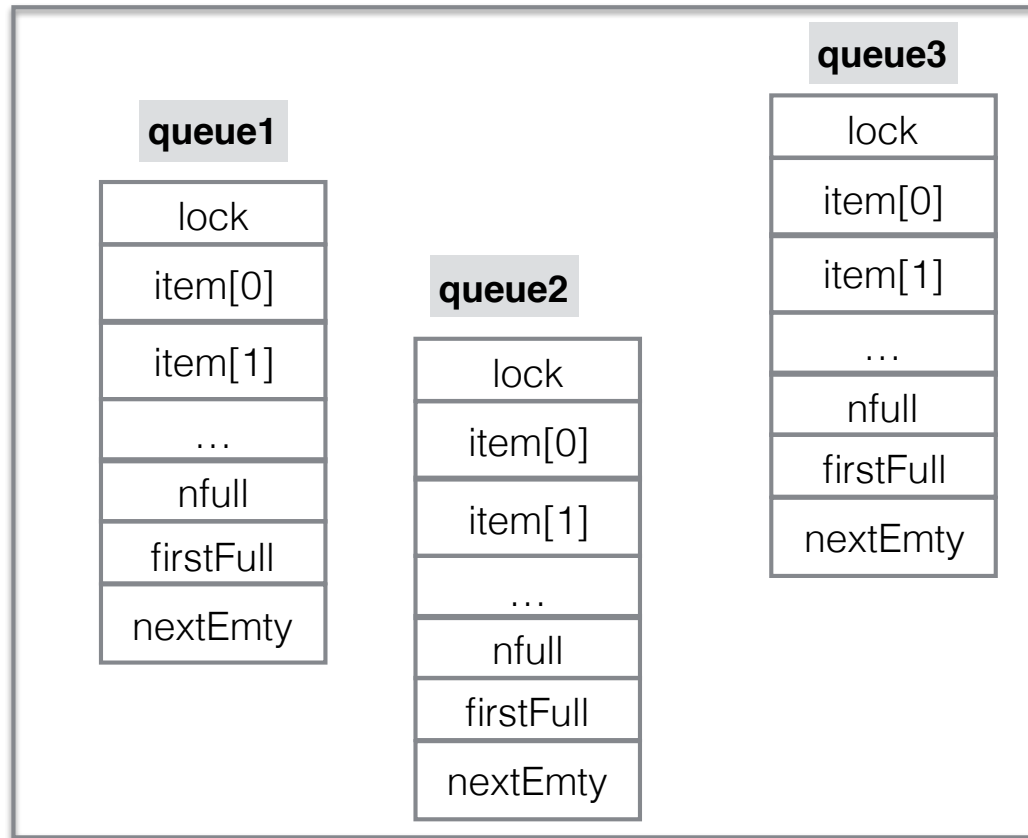
    lock.acquire();
    if ((nextEmpty - front) < MAX) {
        items[nextEmpty % MAX] = item;
        nextEmpty++;
        success = true;
    }
    lock.release();
    return success;
}

// Try to remove an item. If the queue is
// empty, return false; otherwise return true.
bool
TSQueue::tryRemove(int *item) {
    bool success = false;

    lock.acquire();
    if (front < nextEmpty) {
        *item = items[front % MAX];
        front++;
        success = true;
    }
    lock.release();
    return success;
}

// Initialize the queue to empty
// and the lock to free.
TSQueue::TSQueue() {
    front = nextEmpty = 0;
}
```

Tres instancias de la clase TSQueue



Programa que crea tres colas de la clase TSQueue y threads que insertan

```
#include <assert.h>
#include <stdio.h>
#include "shtread.h"
#include "TSQueue.h"

void *putSome(void *tsqueuePtr)
{
    int ii;
    TSQueue *queue = (TSQueue *)tsqueuePtr;

    for(ii = 0; ii < 100; ii++){
        queue->tryInsert(ii);
    }
    return NULL;
}
```

```
int main(int argc, char **argv)
{
    TSQueue *queues[3];
    shtread_t workers[3];
    int ii, jj, ret;
    bool success;

    // Start the worker threads
    for(ii = 0; ii < 3; ii++){
        queues[ii] = new TSQueue();
        shtread_create_p(&workers[ii], putSome,
                        queues[ii]);
    }

    shtread_join(workers[0]);

    // Remove from the queues
    for(ii = 0; ii < 3; ii++){
        printf("Queue %d:\n", ii);
        for(jj = 0; jj < 20; jj++){
            success = queues[ii]->tryRemove(&ret);
            if(success){
                printf("Got %d\n", ret);
            }
            else{
                printf("Nothing there\n");
            }
        }
    }
}
```

Análisis del Código

- El programa utiliza los siguientes archivos:

- `sthread.c`
 - `sthread.h`
 - `Lock.cc`
 - `Lock.h`
 - `TSQueue.cc`
 - `TSQueue.h`
 - `TSQueueMain.cc`

- Estos archivos se pueden bajar de:

<http://www.cs.utexas.edu/users/dahlin/osbook/code/>

Código TSQueueMain.cc

```
int main(int argc, char **argv)
{
    TSQueue *queues[3];
    pthread_t workers[3];
    int ii, jj, ret;
    bool success;

    // Start the worker threads
    for(ii = 0; ii < 3; ii++){
        queues[ii] = new TSQueue();
        pthread_create_p(&workers[ii], putSome,
                        queues[ii]);
    }

    pthread_join(workers[0]);

    // Remove from the queues
    for(ii = 0; ii < 3; ii++){
        printf("Queue %d:\n", ii);
        for(jj = 0; jj < 20; jj++){
            success = queues[ii]->tryRemove(&ret);
            if(success){
                printf("Got %d\n", ret);
            }
            else{
                printf("Nothing there\n");
            }
        }
    }
}
```

```
int main(int argc, char **argv)
{
    TQueue *queues[3];
    pthread_t workers[3];
    int ii, jj, ret;
    bool success;

    // Start the worker threads
    for(ii = 0; ii < 3; ii++){
        queues[ii] = new TQueue();
        pthread_create_p(&workers[ii], putSome,
        queues[ii]);
    }
}
```

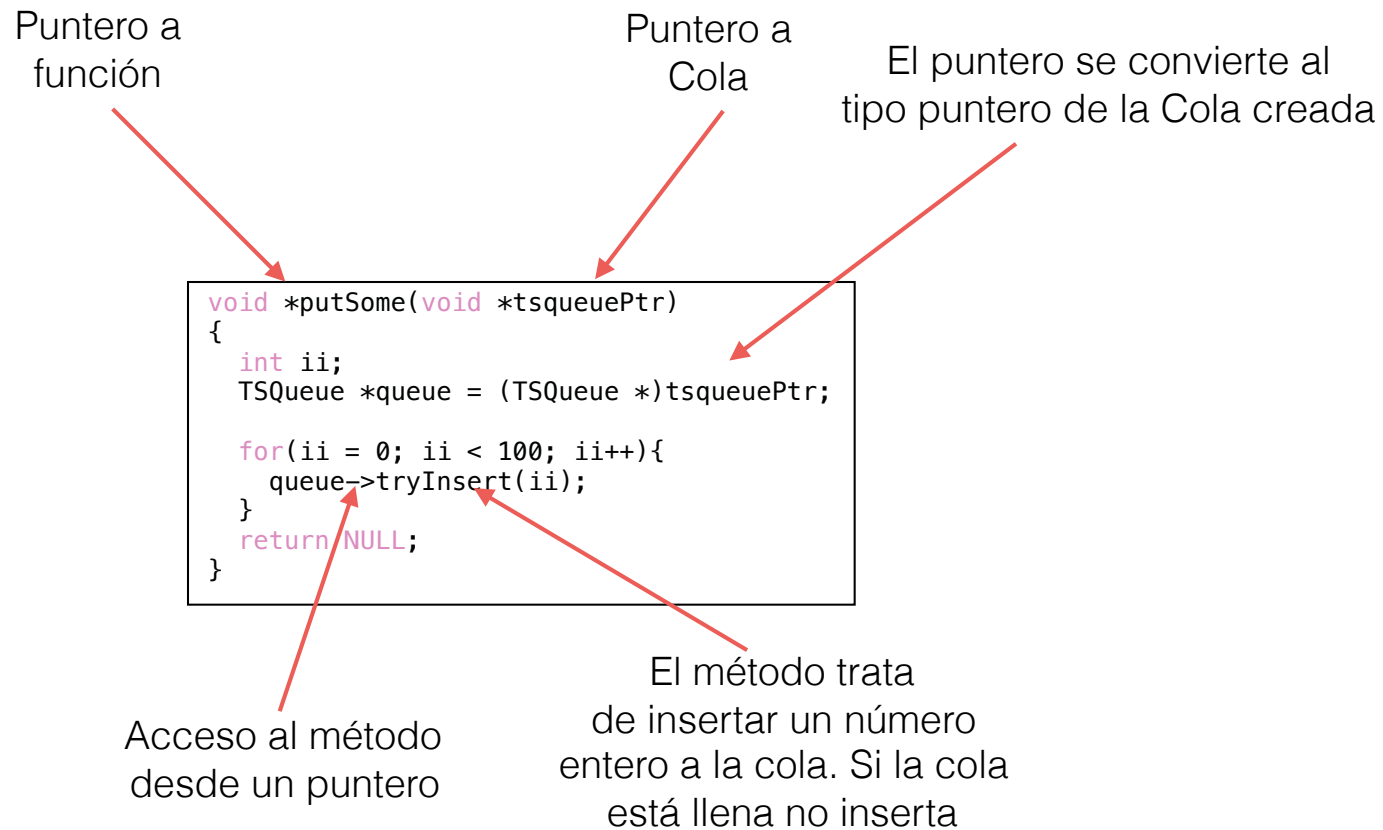
Se definen tres colas y tres threads

Se crean tres colas con el constructor de la Clase. Cada cola se accesa por un puntero

Se crean y activan tres threads, cada uno asociado a una de las colas


usa `pthread_create_p` en vez de `pthread_create` para pasar un puntero a la cola que utilizará

putSome intenta insertar 100 enteros a una cola



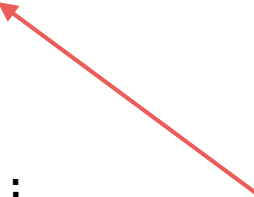
```
pthread_join(workers[0]);
```

Nos aseguramos que al menos un trabajador (thread) haya terminado su intento de insertar 100 números a su cola



```
// Remove from the queues
for(ii = 0; ii < 3; ii++){
    printf("Queue %d:\n", ii);
    for(jj = 0; jj < 20; jj++){
        success = queues[ii]->tryRemove(&ret);
        if(success){
            printf("Got %d\n", ret);
        }
        else{
            printf("Nothing there\n");
        }
    }
}
}
```

Sólo el thread principal (main) intenta remover 20 números de cada una de las colas



Compilación

```
g++ -g -Wall -Werror -D_POSIX_THREAD_SEMANTICS TSQueueMain.cc -c -o TSQueueMain.o
gcc -g -Wall -Werror -D_POSIX_THREAD_SEMANTICS pthread.c -c -o pthread.o
g++ -g -Wall -Werror -D_POSIX_THREAD_SEMANTICS Lock.cc -c -o Lock.o
g++ -g -Wall -Werror -D_POSIX_THREAD_SEMANTICS TSQueue.cc -c -o TSQueue.o
g++ -lpthread TSQueueMain.o TSQueue.o pthread.o Lock.o -o TSMain
```

Ejecución:

./TSMain

Resultados

Queue 0:

Got 0

Got 1

Got 2

Got 3

Got 4

Got 5

Got 6

Got 7

Got 8

Got 9

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Queue 2:

Got 0

Got 1

Got 2

Got 3

Got 4

Got 5

Got 6

Got 7

Got 8

Got 9

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Queue 2:

Got 0

Got 1

Got 2

Got 3

Got 4

Got 5

Got 6

Got 7

Got 8

Got 9

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Nothing there

Variables de Condición

- Las variables de condición permiten a un thread esperar que otro thread realice alguna acción.
- En el caso de la solución al problema de la cola delimitada, en vez de retornar un error cuando se intenta sacar un ítem de una cola vacía, es mejor esperar hasta que la cola tenga un ítem y removerlo. Las variables de condición permiten esta espera.

Variables de Condición: definición

- Una variable de condición es un objeto de sincronización que permite que un thread eficientemente espere que ocurra un cambio en un estado compartido que está protegido por un lock.
- Una variable de condición tiene tres métodos:
 - **CV::wait(Lock *lock)**
 - **CV::signal()**
 - **CV::broadcast()**

Métodos de variables de condición

- Métodos:
 - **CV::wait(Lock *lock)**: Atómicamente libera el lock y suspende la ejecución del thread que lo invoca, poniendo el thread en una cola de espera asociada a la variable. Posteriormente cuando el thread es despertado, re-adquiere el lock antes de retornar del wait.
 - **CV::signal()**: Toma **un** thread que está en la cola de espera y lo marca como elegible para ejecutarse (pasa a la Ready List)
 - **CV::broadcast()**: Toma **todos** los thread que están en la cola de espera y los marca como elegibles para ejecutarse (pasan a la Ready List)

Observaciones

- Una variable de condición está siempre asociada a un lock.
- El thread que espera, está permanentemente esperando por un cambio en el estado del objeto compartido. Debe inspeccionar el estado del objeto en un loop.
- La única razón de un thread para hacer un signal o broadcast es que acaba de cambiar un estado, de una forma que puede ser de interés para los threads que esperan.
- Un thread siempre debe hacer un signal o broadcast reteniendo el lock.

Patrón de diseño para wait() y signal()

```
SO::methodThatWait() {  
  lock.acquire();  
  // read/write shared state  
  
  while (!testOnSharedState()) {  
    cv.wait(&lock);  
  }  
  assert(testOnSharedState());  
  // read/write shared state  
  lock.release();  
}
```

```
SO::methodThatSignals() {  
  lock.acquire();  
  // read/write shared state  
  
  // if change shared state so  
  // that testOnSharedState is true  
  cv.signal();  
  
  // read/write shared state  
  lock.release();  
}
```

SO: SharedObject

Observaciones

1. Una variable de condición ***no tiene memoria***. En si misma esta variable no tiene estado interno. Solo tiene asociada una cola.
2. ***wait()*** atómicamente ***libera el lock***. Siempre se ejecuta un wait reteniendo el lock y la llamada al wait lo libera y pone el thread en la cola asociada a la variable de condición.
3. Un thread que es despertado vía signal o broadcast ***no se ejecuta inmediatamente***. Es puesto en la cola Ready sin ninguna prioridad especial.

... Observaciones

4. `wait()` siempre debe ser llamada desde el **interior de un loop**. Esto se hace para asegurar la atomicidad entre el signal/broadcast y el retorno de una llamada a `wait()`. Así se verifica el estado hasta que el predicado cambia:

```
.....  
while(predicado_sobre_variable_de_estado(...)) {  
    wait(&lock);  
}
```

Ejemplo: Cola Delimitada

- La nueva versión es BBQ (Blocking Bounded Queue).
- BBQ.h define la interfaz y los métodos públicos.
BBQ.cc define la implementación.

BBQ.h: define interfaz y variables

```
#include "Lock.h"
#include "CV.h"
#include "thread.h"

// BBQ.h
// Thread-safe blocking queue.

const int MAX = 10;

class BBQ{
private:
    // Synchronization variables
    Lock lock;
    CV itemAdded;
    CV itemRemoved;

    // State variables
    int items[MAX];
    int front;
    int nextEmpty;

public:
    BBQ();
    ~BBQ() {};
    void insert(int item);
    int remove();
};
```

BBQ.cc: define la implementación

```
#include <assert.h>
#include <pthread.h>
#include "BBQ.h"

// BBQ.cc
// thread-safe blocking queue

// Wait until there is room and
// then insert an item.
void
BBQ::insert(int item) {
    lock.acquire();

    while ((nextEmpty - front) == MAX) {
        itemRemoved.wait(&lock);
    }
    items[nextEmpty % MAX] = item;
    nextEmpty++;

    itemAdded.signal();
    lock.release();
}
```

```
// Wait until there is an item and
// then remove an item.
int
BBQ::remove() {
    int item;

    lock.acquire();

    while (front == nextEmpty) {
        itemAdded.wait(&lock);
    }
    item = items[front % MAX];
    front++;

    itemRemoved.signal();
    lock.release();
    return item;
}

// Initialize the queue to empty,
// the lock to free, and the
// condition variables to empty.
BBQ::BBQ() {
    front = nextEmpty = 0;
}
```

Ejemplo clásico: Lectores y Escritores

- **Problema:** Base de datos que contiene registros que pueden ser escritos y leídos. Múltiples threads pueden leer simultáneamente, pero sólo un thread puede escribir. Si un thread escribe, ninguno puede leer.
- Primero se implementará readers/writers lock (RWLock) para proteger datos compartidos. La exclusión mutua es entre:
 - cualquier escritor \longleftrightarrow cualquier escritor
 - cualquier escritor \longleftrightarrow conjunto de lectores
- Estos locks son frecuentes en bases de datos.

RWLock

- Para generalizar la exclusión mutua se implementará un nuevo objeto compartido llamado RWLock para proteger acceso y forzar sus reglas. RWLock se construye con locks y variables de condición.

RWLock.h

Interfaz y variables miembros de la clase

```
class RWLock{
private:
    // Synchronization variables
    Lock lock;
    CV readGo;
    CV writeGo;

    // State variables
    int activeReaders;
    int activeWriters;
    int waitingReaders;
    int waitingWriters;
```

```
public:
    RWLock();
    ~RWLock() {};
    void startRead();
    void doneRead();
    void startWrite();
    void doneWrite();

private:
    bool readShouldWait();
    bool writeShouldWait();
};
```

Un thread que quiera atómicamente leer o escribir

```
RWLock-> startRead();
//leer
RWLock -> doneRead();
```

```
RWLock-> startWrite();
//leer y escribir
RWLock -> doneWrite();
```

Predicados

- Los lectores deben esperar si hay escritores activos o pendientes mientras que los escritores esperan sólo si hay lectores o escritores activos.
- De esta forma se evita que los escritores caigan en inanición si hay un flujo grande de lectores.

```
bool  
RWLock::readShouldWait() {  
    return (activeWriters > 0 || waitingWriters > 0); }  

```

```
bool  
RWLock::writeShouldWait() {  
    return (activeWriters > 0 || activeReaders > 0); }  

```

Variables de sincronización

- Ahora hay que agregar las variables de sincronización. La pregunta es ¿cuándo los métodos deben esperar?
 1. Agregar los locks de exclusión mutua.
 2. `startRead` o `startWrite` deben esperar y se agrega la variable de condición `readGo` y `writeGo`.
 3. `doneRead` y `doneWrite` no necesitan esperar y no requieren variables de condición.

StartRead y doneRead

```
void RWLock::startRead() {  
    lock.acquire();  
    waitingReaders++;  
    while (readShouldWait()) {  
        readGo.Wait(&lock);  
    }  
    waitingReaders--;  
    activeReaders++;  
    lock.release();  
}
```

```
// Done reading. If no other active  
// reads, a write may proceed.  
void RWLock::doneRead() {  
    lock.acquire();  
    activeReaders--;  
    if (activeReaders == 0  
        && waitingWriters > 0) {  
        writeGo.signal();  
    }  
    lock.release();  
}
```

Los códigos para `startWrite` y `doneWrite` son similares. Para `doneWrite`, por si hay escrituras pendientes se señala con `writeGo`. Si no, se hace un broadcast con `readGo`


```

RWLock::RWLock()
{
    activeReaders = 0;
    activeWriters = 0;
    waitingReaders = 0;
    waitingWriters = 0;
}

```

```

// Wait until no active or waiting
// writes, then proceed.
void RWLock::startRead() {
    lock.acquire();
    waitingReaders++;
    while (readShouldWait()) {
        readGo.Wait(&lock);
    }
    waitingReaders--;
    activeReaders++;
    lock.release();
}

// Done reading. If no other active
// reads, a write may proceed.
void RWLock::doneRead() {
    lock.acquire();
    activeReaders--;
    if (activeReaders == 0
        && waitingWriters > 0) {
        writeGo.signal();
    }
    lock.release();
}

// Read waits if any active or waiting
// write ("writers preferred")
bool
RWLock::readShouldWait() {
    return (activeWriters > 0
        || waitingWriters > 0);
}

```

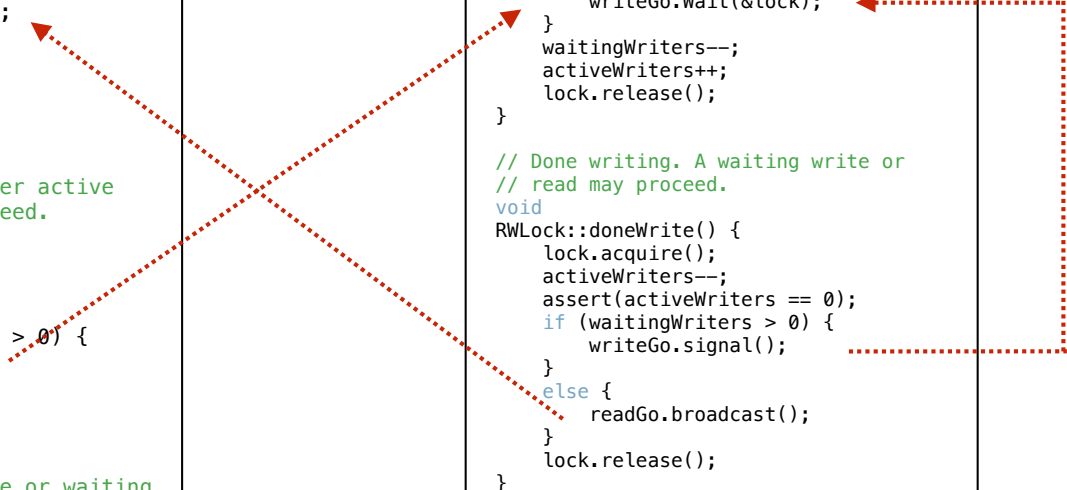
```

// Wait until no active read or
// write then proceed.
void RWLock::startWrite() {
    lock.acquire();
    waitingWriters++;
    while (writeShouldWait()) {
        writeGo.Wait(&lock);
    }
    waitingWriters--;
    activeWriters++;
    lock.release();
}

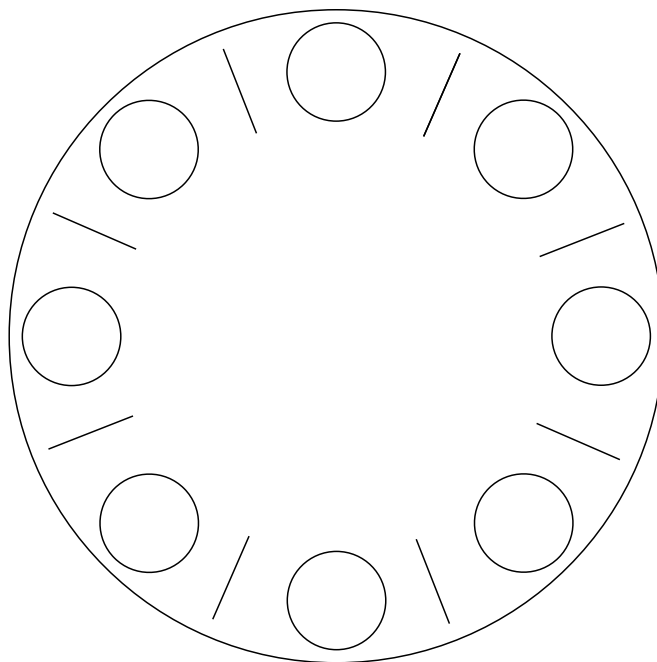
// Done writing. A waiting write or
// read may proceed.
void
RWLock::doneWrite() {
    lock.acquire();
    activeWriters--;
    assert(activeWriters == 0);
    if (waitingWriters > 0) {
        writeGo.signal();
    }
    else {
        readGo.broadcast();
    }
    lock.release();
}

// Write waits for active read or write
bool
RWLock::writeShouldWait() {
    return (activeReaders > 0
        || activeWriters > 0);
}

```



Filósofos



```

const int N=5; // número de filósofos
Class Filósofos {
    Lock lock;
    int estado[N]; //0:pensando, 1: comiendo
    cv libre[N];

    public:
    void tomar(int i); //filósofo i toma palillos
    void dejar (int i); //filósofo i deja palillos
    bool test(int i); //filósofo i ¿puede tomar palillos?
    Filosofo::Filosofo {}; //constructor
}

bool
Filosofo::test(int i)
    {return((estado[(i+4) % N] !=1) && (estado[(i+1) % N] !=1));}

void
Filosofo::tomar(int i) {
    lock.acquire();
    while (!test(i))
        libre[i].wait(&lock);
    estado[i]=1;
    lock.release();}

void
Filosofo::dejar(int i) {
    lock.acquire();
    estado[i]=0;
    libre[(i+4) % N].signal();
    libre[(i+1) % N].signal();
    lock.release();}

```

Filosofo → tomar(i);
 ...
 comer

 Filosofo → dejar(i);

Semáforos

- Un semáforo es, al igual que los objetos compartidos con locks y variables de condición, es una primitiva de sincronización.
- Un semáforo S contiene una variable entera.
- Se definen dos operaciones que modifican S:
 - P()
 - V()

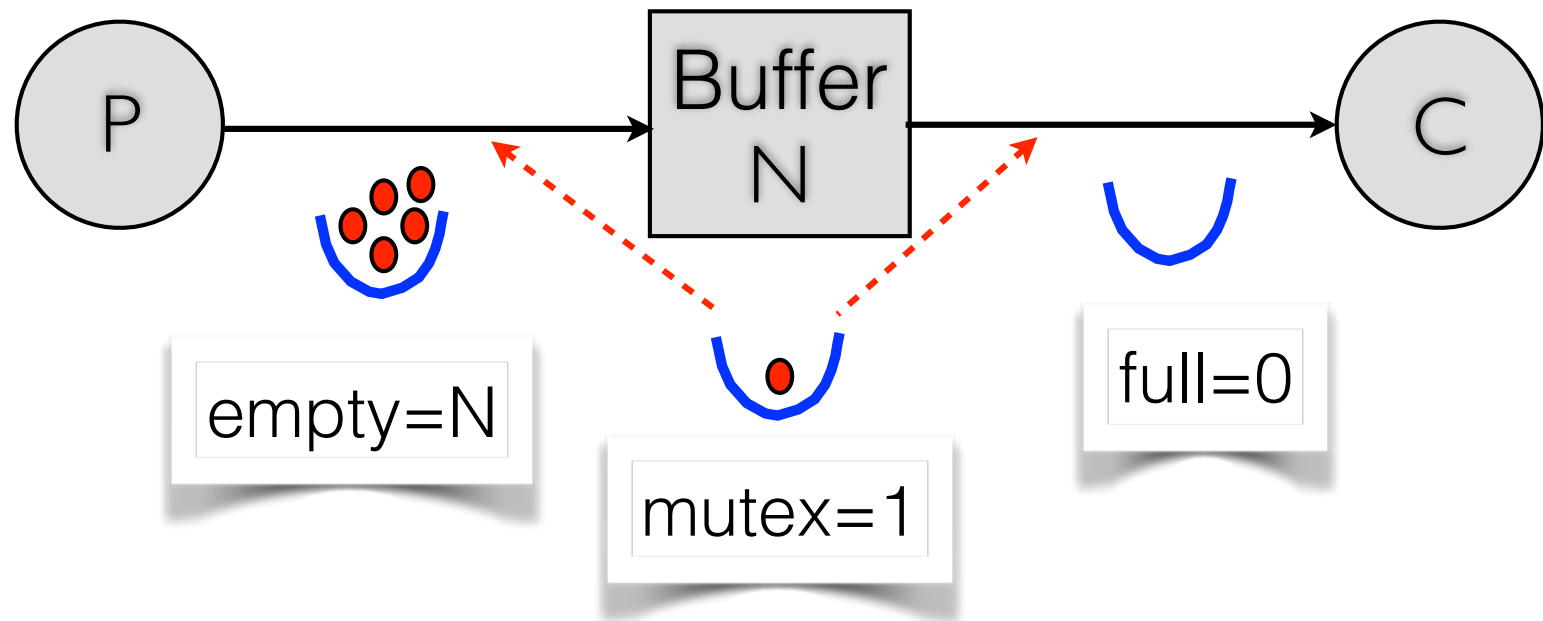
... Semáforos

- Variables semáforos sólo son accesibles vía dos operaciones indivisibles (atómicas):

- ```
P(S) {
 while (S <= 0)
 ; // no hace nada
 S--;
}
```

- ```
V(S) {  
    S++;  
}
```

Cola Delimitada: Solución gráfica



... Buffer: Producer

```
do {  
    //produce an item in nextp  
    P(empty);  
    P(mutex);  
    //add the item to the buffer  
    V(mutex);  
    V(full);  
} while (TRUE);
```

... Buffer: Consumidor

```
do {  
    P(full);  
    P(mutex);  
    //remove an item from buffer to nextc  
    V(mutex);  
    V(empty);  
    //consume the item in nextc  
} while (TRUE);
```


Discusión: Semáforos considerados perjudiciales

- Semáforos pueden utilizarse para exclusión mutua (inicializando en 1) o en general esperando otro thread que haga algo (variable de condición).
- La programación con locks y variables de condición es más segura, el código queda mejor documentado y es más fácil de leer, sin embargo, código basado en semáforos es común en sistemas operativos.

Implementación de Objetos de Sincronización

- Locks y variables de condición tienen estados.
 - Locks: estado={FREE, BUSY} y cola de cero o más threads esperando que lock esté FREE
 - Variables de Condición: el estado es la cola de threads esperando ser señalizados
 - ¿Cómo atómicamente modificar estas estructuras?
- Soluciones requieren apoyo de hardware:
 - Deshabilitar interrupciones: Se puede ocupar en uniprocesadores.
 - Instrucciones atómicas de lectura-modificación-escritura. Por ejemplo `test_and_set` y `swap`.

Deshabilitar Interrupciones

- Esta solución sólo sirve para uniprosesadores y también para threads del kernel.
- En un uniprosesador, un thread puede hacer una secuencia de instrucciones atómicas desabilitando el sistema de interrupciones.
- Esta solución sólo sirve para el kernel porque no se puede dejar mucho tiempo las interrupciones deshabilitadas.

Deshabilitar Interrupciones

Lock::acquire() { disable interrupts }
Lock::release() { enable interrupts }

```
Lock::acquire(){
    disableInterrupts ();
    if(value == BUSY){
        waiting.add(current TCB);
        suspend();
    } else {
        value = BUSY;
    }
    enableInterrupts ();
}
```

```
Lock::release() {
    disableInterrupts ();
    if (!waiting.Empty()){
        thread = waiting.remove();
        readyList.append(thread);
    } else {
        value = FREE;
    }
    enableInterrupts ();
}
```

La instrucción Test and Set

- Definición:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

- $v \leftarrow \text{test_and_set}(x)$ (IBM/360)
- El valor de x se copia en v y el valor TRUE se asigna a x dentro del mismo ciclo de lectura escritura.

SpinLocks

```
class SpinLock {  
    private:  
        int value=0; // 0 =Free; 1= BUSY  
  
    public:  
        void acquire() {  
            while (testAndSet(&value))  
                ; /spin  
        }  
  
        void release() {  
            value = 0;  
        }  
}
```



Sistemas Operativos

Capítulo 5 Acceso Sincronizado a Objetos Compartidos

Prof. Javier Cañas R.