



Sistemas Operativos

Acceso Sincronizado a Objetos Compartidos

Viktor Andrés Tapia Vásquez

Segundo Semestre 2021

Departamento de Informática, Campus SSJJ.

1. Conceptos Básicos
2. Problema: Demasiado Ron
3. Herramientas

Conceptos Básicos

- Consideremos un proceso con dos hebras.

Thread 0	Thread 1
<code>x=1;</code>	<code>x=2;</code>

- ¿Cuáles son los posibles valores para x?
- *Houston, tenemos un problema...*
- La itineración de hebras es no determinista.

- Consideremos otro proceso con dos hebras.
- Inicialmente $x = 0$.

Thread 0	Thread 1
$x = x + 1;$	$x = x + 2;$

- ¿Cuáles son los posibles valores para x ?
- Es necesario descomponer el código anterior.
- La idea es analizar las operaciones atómicas que lo componen.
- lw y sw son operaciones atómicas.

Conceptos Básicos

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

Figure 1: Ejemplo: Posibles escenarios

- Problema: Acceso concurrente a objetos compartidos.
 - ✓ Comportamiento del programa indefinido.
 - ✓ El resultado puede variar en cada ejecución.
 - ✓ Incoherencia de datos.

Condición de carrera

Se produce cuando el comportamiento del programa **depende** de la forma en que se intercalan las operaciones que se deben ejecutar.

Conceptos Básicos

Ojo: no es solo para hebras.

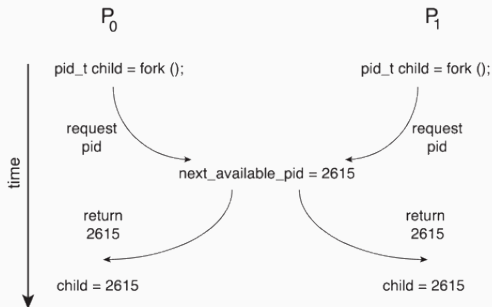


Figure 2: Condición de carrera entre dos procesos

- **Sección Crítica:** Código donde se accede al objeto compartido.
- Toda solución a un problema de sincronización debe cumplir con:
 - ✓ **Exclusión mutua:** Solo 1 entra a la sección crítica.
 - ✓ **Progreso:** Las solicitudes de acceso deben avanzar.
 - ✓ **Espera acotada:** Si se solicita entrar, la espera es acotada.
- Exclusión mutua es la propiedad de **seguridad**.
- Progreso y espera acotada son la propiedad de **vivacidad**.

- Construir la solución es pega del programador.
- Esto puede llevar a errores y otros problemas.
- SO's modernos entregan las herramientas para sincronizar.
- No resuelven el problema ellos!

Problema: Demasiado Ron

Demasiado Ron

	Thread A	Thread B
22:00	Verificar Bar: No hay ron	
22:05	Salir a Comprar	
22:10	Llegada a la boti.	Verificar Bar: No hay ron
22:15	Comprar ron	Salir a Comprar
22:20	Llegada al depto. Guardar ron	Llegada a la boti.
22:25		Comprar ron
22:30		Llegada al depto. Guardar ron
		Oh!

Figure 3: Problema propuesto para la sincronización

Demasiado Ron - Primera Solución

- Cada hebra deja una nota (flag) en el Bar. Ejecutan:

```
if (ron == 0) //no hay ron
{
    if (nota == 0) // no hay nota
    {
        nota = 1; //dejar nota
        ron++; //comprar ron
        nota = 0; // sacar nota
    }
}
```

- **Seguridad:** Nunca se puede comprar más.
- **Vivacidad:** Si se requiere ron, alguien compra.

Demasiado Ron - Primera Solución

- El criterio de seguridad no se cumple:

Hebra A `if (ron == 0)`
`{`

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

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

Hebra B

Demasiado Ron - Segunda Solución

- Se dejan dos notas. Se crean antes de validar:

Hebra A

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

Hebra B

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

- La idea es dejar una nota: Voy a comprar ron.

Demasiado Ron - Segunda Solución

- Seguridad OK: Nunca se compra de más.
- Ambas hebras podrían dejar sus notas y decidir no comprar.
- Falla con la vivacidad.
- La idea es que si se requiere, al menos una compre.

Demasiado Ron - Tercera Solución

- Al menos una hebra se asegura si se ha comprado o no.

Hebra A

```
notaA = 1; //se deja nota
while (notaB == 1){
    //do something
    ;
}
if (ron == 0){
    ron++;
}

notaA = 0; //se elimina nota
```

Hebra B

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

Demasiado Ron - Tercera Solución

- Funciona correctamente.
- La hebra B no tiene loops. Si termina $\text{notaB} = 0$.
- La solución tiene vivacidad y seguridad pero es:
 - ✓ Compleja.
 - ✓ Asimétrica: Códigos diferentes.
 - ✓ Ineficiente: A produce espera ociosa.
 - ✓ Falla ante un reordenamiento de instrucciones.

Herramientas

- Aplicaciones concurrentes acceden a ellos de forma segura.
- Todos sus estados se comparten.
- Deben encapsular, en uno o más objetos:
 - ✓ Heap.
 - ✓ Variables estáticas y globales.
- Extienden la orientación a objetos.

- Se ocultan detalles de sincronización.
- La interfaz es limpia y simple.
- Incluyen variables de sincronización.
- Se asocian a objetos específicos.
- Accesibles a través de métodos.
- La atomicidad de instrucciones es necesaria.

- Locks.
- Variables de Condición.
- Semáforos.

- Proporcionan exclusión mutua.
- Si una hebra retiene un lock, ninguna más puede tomarlo.
- Tiene dos métodos:
 - ✓ `acquire()`
 - ✓ `release()`
- Tiene dos estados:
 - ✓ BUSY
 - ✓ FREE

- `acquire()`:
 - ✓ Espera hasta que lock esté FREE.
 - ✓ Cambia automáticamente a BUSY.
 - ✓ Varias hebras esperando: A lo más una tiene éxito.
- `release()`:
 - ✓ Lock queda en estado FREE.
 - ✓ `acquire()` pendientes: A lo más una tiene éxito.

- Ejemplo: Para el problema del ron.
- Definimos Lock lock;
- La idea es proteger la sección crítica.

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

- Esta solución cumple con:
 - ✓ **Exclusión mutua:** A lo más una hebra retiene el lock.
 - ✓ **Progreso:** Si nadie lo retiene y se solicita, se obtiene.
 - ✓ **Espera acotada:** El tiempo de espera queda acotado.
- Ojo: No existe orden de asignación del lock.

- Ejemplo: Problema cola delimitada.
- Utilizaremos hebras para resolverlo.
- Cada **objeto compartido** es una instancia de una clase.
- Se definen sus estados y métodos que operan sobre éste.
- Los estados incluyen variables (int, float, etc.) y de sincronización.

Productor-Consumidor

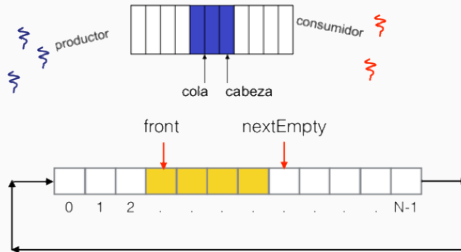


Figure 4: Modelo del problema de la cola delimitada

- Total de elementos insertado es **nextEmpty**.
- Total de elementos removidos es **front**.
- $\text{front} \leq \text{nextEmpty}$.
- Elementos en la cola: $\text{nextEmpty} - \text{front}$.
- La solución utiliza los códigos disponibles en:
moodle.inf.utfsm.cl

Herramientas - Variables - Locks

```
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");
            }
        }
    }
}
```

Se definen 3 colas y 3 threads

Se crean 3 colas con el constructor de la clase. Cada cola se accesa por un puntero.

Se crean 3 threads, cada uno asociado a una de las colas.

Usa `pthread_create_p` para pasar un puntero a la cola que usará

Figure 5: TSQueueMain.cc

Herramientas - Variables - Locks

putSome intenta insertar 100 enteros a una cola

Puntero se convierte al tipo puntero de la cola creada

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

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

Acceso al método desde un puntero

Método trata de insertar un número entero a la cola. Si la cola está llena no inserta

Figure 6: TSQueueMain.cc

Herramientas - Variables - Locks

```
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");  
        }  
    }  
}
```

Al menos una hebra
debe terminar su
intento de insertar 100
números a su cola.

Solo el thread principal
intenta remover 20
números de cada una
de las colas

Figure 7: TSQueueMain.cc

Herramientas - Variables - Locks

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;
}
```

Figure 8: Librerías

Herramientas - Variables - Locks

Queue 0:	Queue 2:	Queue 2:
Got 0	Got 0	Got 0
Got 1	Got 1	Got 1
Got 2	Got 2	Got 2
Got 3	Got 3	Got 3
Got 4	Got 4	Got 4
Got 5	Got 5	Got 5
Got 6	Got 6	Got 6
Got 7	Got 7	Got 7
Got 8	Got 8	Got 8
Got 9	Got 9	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
Nothing there	Nothing there	Nothing there
Nothing there	Nothing there	Nothing there
Nothing there	Nothing there	Nothing there
Nothing there	Nothing there	Nothing there
Nothing there	Nothing there	Nothing there

Figure 9: Resultado

Herramientas - Variable de Condición

- Permite que una hebra espere eficientemente que ocurra un cambio en un estado compartido que está protegido por un lock.
- **Siempre** está asociada a un lock.
- Ejemplo: En el problema anterior, en vez de producir un error al momento de intentar sacar un elemento de la cola vacía, es mejor que las hebras esperen que la cola tenga un ítem.
- Tiene tres métodos:
 - ✓ `wait(Lock *lock)`
 - ✓ `signal()`
 - ✓ `broadcast()`

- `wait(Lock *lock)`:
 - ✓ Atómicamente libera el lock.
 - ✓ Suspende la ejecución de la hebra que lo invoca.
 - ✓ La hebra se va a una cola de espera asociada a la variable.
 - ✓ Cuando despierta, adquiere el lock antes de retornar el wait.

- Para mover las hebras desde la cola de espera hasta la cola ready:
- `signal()`:
 - ✓ Solo una hebra.
- `broadcast()`:
 - ✓ Todas las hebras.

```
S0::wait(){
    lock.acquire();
    //leer o escribir estado compartido

    while(!testOnSharedState()){
        cv.wait(&lock);
    }

    assert(testOnSharedState());
    //leer o escribir estado compartido
    lock.release();
}
```

Figure 10: Ejemplo uso wait

```
S0::signal(){  
    lock.acquire();  
    //leer o escribir estado compartido  
  
    //Si existen cambios (testOnSharedState()==true)  
    cv.signal();  
  
    //leer o escribir estado compartido  
    lock.release();  
}
```

Figure 11: Ejemplo uso signal

Herramientas - Variable de Condición

- Una variable de condición no tiene memoria.
- Solo tiene una cola asociada.
- `wait()` siempre se ejecuta:
 - ✓ Reteniendo un lock.
 - ✓ Desde el interior de un loop.
- Una hebra despertada por `signal()` o `broadcast()` no se ejecuta inmediatamente. Simplemente es dejada en la cola ready.
- Ejemplo: una mejor solución del problema de la cola delimitada.

Herramientas - Variable de Condición

```
#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();
};
```

Figure 12: Interfaz y variables

Herramientas - Variable de Condición

```
#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;
}
```

Figure 13: Implementación

- Es una variable entera.
- Se accede mediante dos operaciones atómicas.
- `p()` / `wait()`:

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

- `v()` / `signal()`:

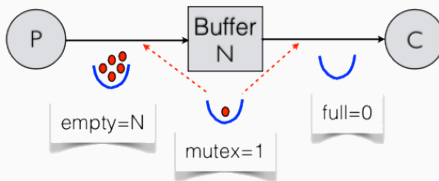
```
signal(S) {  
    S++;  
}
```

- So's modernos diferencian dos tipos:
 - ✓ **Contadores:** Definido en un dominio no restringido.
 - ✓ **Binarios:** Valor 0 o 1. Conocidos como Mutex.
- Mutex generalmente protege la sección crítica.
- El problema es que presentan *busy waiting*.
- Al menos no producen cambios de contexto innecesarios.
- Se les conoce como **spin locks**.

- Para evitar el *busy waiting* se modifican los semáforos.
- A cada semáforo se le asigna una cola de espera.
- Se definen dos operaciones:
 - ✓ **Block:** Lleva al proceso a la cola de espera.
 - ✓ **WakeUp:** Lleva al proceso a la cola ready.
- La idea es simple, en vez de esperar se bloquean.

- Uso incorrecto:
 - ✓ `signal(mutex) wait(mutex)`
 - ✓ `wait(mutex) wait(mutex)`
 - ✓ Omitir un `wait(mutex)` y/o un `signal(mutex)`.
- Estos errores pueden llevar a deadlock y/o starvation.

- Ejemplo: Nuestro clásico problema de la cola delimitada.



```
do{
    //produce un ítem
    p(empty);
    p(mutex);
    // agregar ítem
    v(mutex);
    v(full);
} while(TRUE);

do{
    p(full);
    p(mutex);
    // remover ítem
    v(mutex);
    v(empty);
} while(TRUE);
```

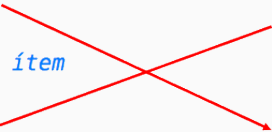


Figure 14: Productor v/s Consumidor

- Los procesos puede que tengan que esperar mucho para obtener una herramienta de sincronización.
- Espera indefinida viola las propiedades de progreso y espera acotada.
- **Liveness:** Propiedades de un SO moderno que aseguran el progreso de los procesos.



Sistemas Operativos

Acceso Sincronizado a Objetos Compartidos

Viktor Andrés Tapia Vásquez

Segundo Semestre 2021

Departamento de Informática, Campus SSJJ.