

# Sistemas Operativos

Fernando R. Rannou  
Departamento de Ingeniería Informática  
Universidad de Santiago de Chile

May 8, 2014

# Concurrencia y sincronización

- Cuando dos o más tareas (procesos o hebras) comparten algún recurso en forma concurrente o paralela, debe existir un mecanismo que sincronice sus actividades
- Riesgos:
  - corrupción del recurso compartido
  - soluciones erróneas
- Ejemplos:
  - ① Acceso a impresora en un sistema distribuido
  - ② Consulta y actualización de una base de datos
  - ③ Ejecución paralela de tareas para resolver un problema en conjunto
- **Los problemas de sincronización no sólo ocurren en sistemas multiprocesadores, sino también (y fundamentalmente) en monoprocesadores**

# Beneficios de concurrencia

- Aumento en el rendimiento mediante múltiples procesadores
  - Paralelismo
- Aumento del throughput de las aplicaciones
  - Un llamado a una operación de I/O sólo bloquea una hebra
- Mejora en interactividad de las aplicaciones
  - Asignar mayor prioridad a hebras que atiendan al usuario
- Mejora en la estructuración de aplicaciones
- Sistemas distribuidos

# Ejemplo de acceso concurrente a datos compartidos

Considere dos hebras que ejecutan concurrentemente la función `insert(item)` que inserta un elemento en una lista compartida

```
typedef struct list {  
    int data;  
    struct list *next;  
} Lista;  
  
void insert(int item) {  
    struct list *p;  
  
    p = malloc(sizeof(struct list));  
    p->data = item;  
    p->next = Lista;  
    Lista = p;  
}
```

Ambas hebras procuran e insertan correctamente el item en un nodo, pero dependiendo del orden de ejecución, es probable que sólo un nodo se inserte en la Lista

# Condición de carrera

## Condición de carrera

Una condición de carrera (CC) es cuando el resultado de una operación depende del orden de ejecución de dos o más hebras

- A veces, el orden de ejecución produce resultados correctos o consistentes. Por ejemplo, si una hebra inserta el valor 8 y la otra el valor 5, pueden haber dos resultados correctos:
  - 5, 8
  - 8, 5
- Pero sólo el 8 o sólo el 5 es incorrecto

De ahora en adelante, una CC siempre representa un error de concurrencia

# Sección crítica

## Sección crítica

Una sección crítica (SC) es un trozo de código, ejecutado por múltiples hebras, en el cual se accede a datos compartidos y, al menos una de las hebras escribe sobre los datos

- Note que si ninguna hebra modifica los datos (write) no hay SC
- Si no hay recursos compartidos, no hay SC
- Si es mono hebra no hay SC

¿Cuál es la SC?

```
void insert(int item) {  
    struct list *p;  
  
    p = malloc(sizeof(struct list));  
    p->data = item;  
    p->next = Lista;  
    Lista = p;  
}
```

# Exclusión mutua

## Exclusión mutua

Exclusión mutua (EM) es un requerimiento sobre una SC, que dice que sólo una hebra puede estar ejecutando dicha SC

- EM se define por los datos que se acceden en ella, no tanto por el código
- Por ejemplo, en un Stack concurrente, los métodos `pop()` y `push()` son mutuamente excluyentes

```
void pop() {  
    if (! empty())  
        top--;  
}
```

```
void push(int item) {  
    if (! full()) {  
        top++;  
        data[top] = item  
    } else error()  
}
```

# Modelo de concurrencia

- Asumimos una colección de tareas del tipo

```
Process(i) {  
    while (true) {  
        codigo no critico  
        ...  
        enterCS();  
        SC();  
        exitSC();  
        ..  
        codigo no critico  
    }  
}
```

- El código es ejecutado concurrentemente por múltiples tareas (hebras o procesos)
- `enterCS()` representa las acciones que la hebra debe realizar para poder entrar a su SC en forma segura
- `exitSC()` representa las acciones que una hebra debe realizar para salir la SC y dejarla habilitada a otras hebras
- El código no crítico es cualquier código donde no se accede recursos compartidos



# Requerimientos para una solución de EM

## Requerimientos

Una solución al problema de la EM es correcta cuando satisface:

- 1 **Exclusión mutua:** Cuando  $T_i$  está en su SC, ninguna otra hebra  $T_j$  está en su correspondiente SC. Esto significa que la ejecución de operaciones de las secciones críticas no son concurrentes
- 2 **Sin deadlock:** Deadlock es cuando todas las hebras quedan ejecutando `enterCS()` indefinidamente (ya sea bloqueadas o en busy-waiting), y por lo tanto, ninguna entra.
- 3 **Sin inanición:** Una hebra entra en inanición si nunca logra entrar a su SC.
- 4 **Progreso:** Si una hebra  $T_i$  ejecuta `enterCS()`, y ninguna otra  $T_j$  está en la SC está, se debe permitir a  $T_i$  entrar a la SC.

## Notas

- Una hebra puede tener más de una SC
- Aunque deadlock implica inanición, inanición no implica deadlock
- **No se debe realizar suposiciones sobre la velocidad relativa y orden de ejecución de las hebras**

# Locks

La función `enterCS()` puede ser implementada de varias formas:

- ① Por hardware: Usar instrucciones nativas del procesador
- ② Por software: Usar algoritmos e implementarlo en lenguaje de alto nivel
- ③ Por SO: Usar servicios del SO

A veces, se usa el término **lock** en forma genérica. Es decir

- ① `lock.acquire()` es equivalente a `enterCS()`
- ② `lock.release()` es equivalente a `exitCS()`

# Atomicidad

- 1 Supongamos que en una oficina se juega al amigo secreto, y hay una caja en una sala donde se deposita los regalos
- 2 Para que nadie sepa cuál es el regalo que aporté, me fijo que no haya nadie en la sala antes de depositar mi regalo
- 3 Pero, ... siempre es posible que dos o más personas vean la sala vacía, para encontrar que cuando llegan hay alguien más
- 4 Las operaciones de mirar y depositar el regalo son dos operaciones distintas, y en conjunto NO son atómicas

## Atomicidad

- Una operación es atómica si no puede dividirse en partes
- Una operación atómica se ejecuta completamente o no se ejecuta
- Una operación atómica se ejecuta sin ser interrumpida

# La ilusión de atomicidad

Considere el siguiente código ejecutado por dos hebras (asuma  $i=5$ ):

<code>inc(int &amp;i) {</code>	<code>LOAD i, ACC</code>	<code>// carga el acumulador</code>
<code>  i = i + 1;</code>	<code>INC ACC</code>	<code>// incrementa el acumulador</code>
<code>}</code>	<code>STORE ACC, i</code>	<code>// almacena el resultado en memoria</code>

Los posibles resultados son:

- 7, resultado válido
- 6, resultado inválido

Note que toda instrucción assembler es atómica, pero hasta la instrucción más simple de lenguaje de alto nivel **puede no ser atómica**.

# Solución por hardware 1

- **Deshabilitación de interrupciones** : Recordar que una hebra se ejecuta hasta que invoca un llamado al SO o es interrumpida, por ejemplo por quantum. Entonces para garantizar EM, la hebra deshabilita las interrupciones justo antes de entrar en su SC

```
while (true)
{
    deshabilitar_interrupciones();
    SC();
    habilitar_interrupciones();
}
```

- Sencillo de entender
- Habilidad del procesador para hacer context-switch queda limitada
- No sirve para multiprocesadores
- Funciona para cualquier número de hebras
- No está libre de inanición

## Solución por hardware 2, testset()

- **Uso de instrucciones especiales de máquina** que realizan en forma **atómica** más de una operación.
- Por ejemplo testset()

```
boolean testset(int &i) {  
    if (i == 0) {  
        i = 1;  
        return true;  
    } else return false;  
}  
  
T(i) {  
    while (true) {  
        ...  
        while (!testset(bolt)); //busy-waiting  
        SC();  
        bolt = 0;  
        ...  
    }  
}  
  
int bolt; // shared  
void main() {  
    bolt = 0;  
    parbegin(T(1), T(2), ..., T(n));  
}
```

**busy-waiting** es lo mismo que **spin-lock**: Cuando una tarea espera poder entrar a la SC ejecutando instrucciones.

## Solución por hardware 3, exchange()

- `exchange(a, b)` o `(swap(a, b))` intercambia los valores contenidos en las direcciones de memoria `a` y `b`, en forma atómica.

```
void exchange(int a, int b) {  
    int tmp;  
    tmp = a;  
    a = b;  
    b = tmp;  
}  
  
T(i) {  
    int keyi;  
    while (true) {  
        keyi = 1;  
        while (keyi != 0)  
            exchange(keyi, bolt); //busy-waiting  
        SC();  
        exchange(keyi, bolt);  
        ...  
    }  
}  
  
int bolt; // shared  
void main() {  
    bolt = 0;  
    parbegin(T(1), T(2), ..., T(n));  
}
```

# EM por instrucciones de máquina

- Ventajas
  - Aplicable a cualquier número de hebras y procesadores que comparten memoria física
  - Simple y fácil de verificar
  - Puede ser usada con varias SC, cada una controlada por su propia variable
- Desventajas
  - *Busy-waiting* (o *spin-lock*) consume tiempo de procesador
  - Es posible inanición



# Solución por software 1

- Dos hebras comparten la variable turno

```
int turno;
T0 {                                T1 {
    while (true) {                  while (true) {

        while (turno != 0);          while (turno != 1);
        SC();                        SC();
        turno = 1;                  turno = 0;

    }                                }
}                                    }

void main() {
    turno = 0;
    parbegin(T0, T1);
}
```

- Garantiza exclusión mútua
- Sólo aplicable a dos procesos
- Alternación estricta de ejecución
- No satisface progreso

## Solución por software 2

- El problema con la solución anterior es que almacenamos el PID de la tarea que puede entrar a la SC
- Intentemos solucionar este problema almacenando el PID de la tarea que está en su SC
- Arreglo Booleano `flag[]` compartido

```
boolean flag[2];
T0 {
    while (TRUE) {
        while (flag[1]);
        flag[0] = true;
        SC();
        flag[0] = FALSE;
        ...
    }
}

T1 {
    while (TRUE) {
        while (flag[0]);
        flag[1] = TRUE;
        SC();
        flag[1] = FALSE;
        ...
    }
}

void main() {
    flag[0] = flag[1] = FALSE;
    parbegin(T0, T1);
}
```

- No satisface el requerimiento de EM

## Solución por software 3

- El problema más evidente de la solución anterior es que una tarea puede cambiar su estado después que otra tarea haya consultado dicho estado, pero antes que el primero haya entrado a la SC.
- Intentemos solucionar este problema cambiando el orden de las dos instrucciones involucradas

```
boolean flag[2];
T0 {                                T1 {
    while (TRUE) {                  while (TRUE) {
        flag[0] = TRUE;              flag[1] = TRUE;
        while (flag[1]);             while (flag[0]);
        SC();                        SC();
        flag[0] = FALSE;            flag[1] = FALSE;
        ...                          ...
    }                                }
}                                    }

void main() {
    flag[0] = flag[1] = FALSE;
    parbegin(T0, T1);
}
```

- SE satisface el requerimiento de EM, pero...
- Existe posibilidad de **deadlock**

# Solución de Peterson

```
boolean flag[2];  
int turno;
```

```
T0 {  
    while (true) {  
        ...  
        flag[0] = true;  
        turno = 1;  
        while (flag[1] && turno == 1);  
        SC();  
        flag[0] = false;  
        ...  
    }  
}  
void main(){  
    flag[0] = flag[1] = false;  
    parbegin(T0, T1);  
}
```

```
T1 {  
    while (true) {  
        ...  
        flag[1] = true;  
        turno = 0;  
        while (flag[0] && turno == 0);  
        SC();  
        flag[1] = false;  
        ...  
    }  
}
```

- Satisface todos los requerimientos
- Es compleja de entender
- Funciona sólo para dos tareas
- Usa busy-waiting

# Algoritmo de la panadería (simplificado)

La solución de software para  $N$  hebras utiliza la lógica de una panadería (o farmacia), en la que los clientes son atendidos de acuerdo a un ticket que sacan al momento de llegar

- 1 Cada hebra  $T_i$  tiene una variable entera  $\text{num}[i]$ , inicialmente inicializada en 0. Este es el ticket.
- 2 Todas las hebras pueden leer cualquier  $\text{num}[j]$ , pero solo  $T_i$  puede escribir  $\text{num}[i]$ .
- 3 Cuando  $T_i$  desea entrar a la SC, setea  $\text{num}[i]$  a un valor más grande que todos los otros  $\text{num}[j]$ . Esto corresponde a sacar el ticket.
- 4 Para cada hebra  $j$ ,  $T_i$  espera hasta que  $\text{num}[j]$  es cero o  $\text{num}[j]$  es mayor que  $\text{num}[i]$ , para todo  $j \neq i$ . Esto corresponde a la espera del turno.
- 5 Luego que lo anterior se hizo cierto para todas las  $T_j$ ,  $T_i$  entra a la SC
- 6 Al salir de la SC,  $T_i$  setea  $\text{num}[i]$  en cero.

# Algoritmo de la panadería (simplificado)

```
int num[N] = 0;

1 T(int i) {
2   num[i] = max(num[0], num[1], ..., num[N-1]) + 1;
3   for (p=0, p < N; p++) {
4     while (num[p] != 0 && num[p] < num[i]) ;
5   }
6   SC();
7   num[i] = 0;
8 }
```

Para que esta solución sea correcta, se requiere que

- 1 Las operaciones de lectura y escritura de variables sean atómicas
- 2 La operación `max()+1` sea atómica

# Algoritmo de la panadería (original)

- Si `max()+1` no es atómica, dos o más hebras pueden tomar el mismo número. ¿Cómo?
- Para romper la igualdad, se usa el par `(num[i], i)`. Es decir, de todas las hebras que hayan elegido el mismo número, la hebra con ID menor tiene la prioridad
- Aún se requiere atomicidad de read-write de variables enteras y booleanas

```
int num[N] = 0;
boolean choosing[N] = False;

1  T(int i) {
2      choosing[i] = true;
3      num[i] = max(num[0],num[1],...,num[N-1]) + 1;
4      choosing[i] = false;
5      for (p=0, p < N; p++) {
6          while (choosing[p]);
7          while (num[p] != 0 && (num[p],p) < (num[i],i)) ;
8      }
9      SC();
10     num[i] = 0;
11 }
```

# Solución de SO: Semáforos y Locks

- Un **semáforo** es una variable especial usada para que dos o más procesos se señalicen mutuamente
- Un semáforo es una variable entera, pero
- Accesada sólo por dos operaciones
  - ① **wait(s)** decrementa el semáforo; si el valor resultante es negativo, el proceso se bloquea; sino, continúa su ejecución
  - ② **signal(s)** incrementa el semáforo; si el valor resultante es menor o igual que cero, entonces se despierta un proceso que fue bloqueado por wait()
- Se denominan **semáforos contadores** a aquellos semaforos que pueden tomar valores mayor o menor que cero.
- El valor de inicialización de un semáforo contador depende de la aplicación.



# Primitivas sobre semáforos contadores

```
struct semaphore {  
    int count;  
    queueType queue;  
}
```

```
void wait(semaphore s)  
{  
    s.count--;  
    if (s.count < 0) {  
        place this process  
        in s.queue;  
        block this process  
    }  
}
```

```
void signal(semaphore s)  
{  
    s.count++;  
    if (s.count <= 0) {  
        remove a process P  
        from s.queue;  
        place process P on ready state  
    }  
}
```

El SO garantiza que estas operaciones se ejecutan cómo si fueran atómicas

# Semáforo como objeto

La siguiente es una posible representación de un semáforo como una clase:

```
class Semaphore {  
    int val;  
    Queue<Process> Q;  
public:  
    Semaphore()      : val(0), Q(NULL) {}  
    Semaphore(int v) : val(v), Q(NULL) {}  
    ~Semaphore() { delete Q; }  
  
    void signal();  
    void wait();  
};
```

A esta clase aún le falta mucho para ser una implementación correcta.

# Semáforo binario o mutex o lock en C

- Un semáforo binario sólo puede tomar los valores 0 o 1
- Sería un error inicializar un semáforo binario con una valor distinto de 0 ó 1

```
wait(s)
{
    if (s.value == 1)
        s.value = 0;
    else {
        place this process
            in s.queue
        block this process
    }
}
```

```
signal(s)
{
    if (s.queue is empty())
        s.value = 1
    else {
        remove a process P
            from s.queue
        place P on the ready list;
    }
}
```

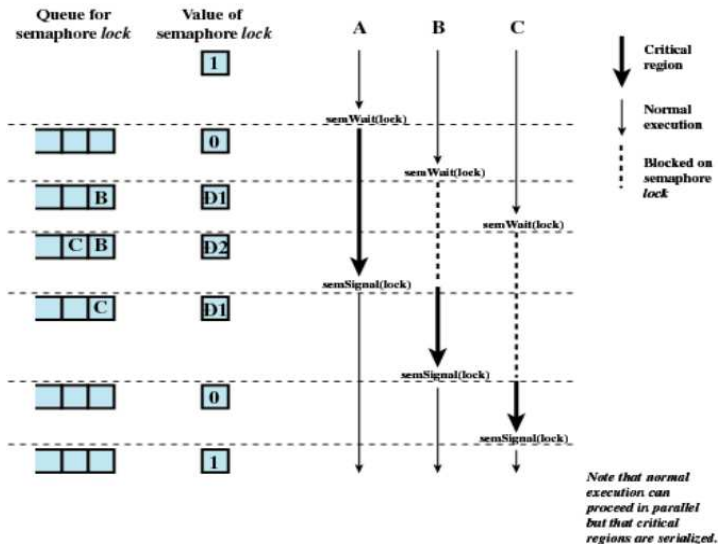
- Otro nombre dado a un semáforo binario es **lock** o **mutex**.
- Típicamente, las operaciones son  
acquire() = lock() = wait()  
release() = unlock() = signal()

# EM usando semáforos

```
semaphore s;  
Process(i) {  
    while (true) {  
        codigo no critico  
        ...  
        wait(s)  
        SC();  
        signal(s);  
        ..  
        codigo no critico  
    }  
}  
  
void main() {  
    s = 1;  
    parbegin(P(1), P(2),...,P(n));  
}
```

Note que para esta solución se puede usar semáforos binario o semáforos contadores

# Ejemplo



# El problema del productor/consumidor

- Uno o más tareas **productoras** generan datos y los almacenan en un buffer compartido
- Una tarea **consumidor** toma (consume) los datos del buffer uno a la vez
- Claramente, sólo un agente (productor o consumidor) puede acceder el buffer a la vez

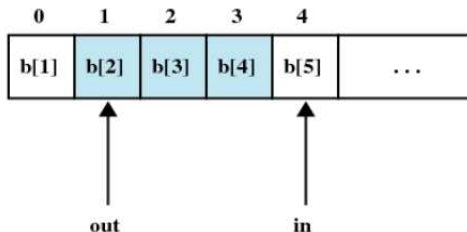
Aquí va dibujo

# Productos/consumidor buffer infinito

- Si el buffer es infinito, el buffer nunca se llena (Duh!)
- Sin embargo, la tarea consumidora puede encontrar el buffer vacío

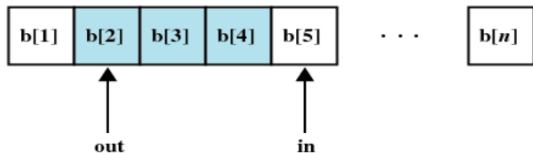
```
producer() {  
    v = produce();  
    b[in] = v;  
    in++;  
}
```

```
consumer() {  
    while (in == out);  
    w = b[out];  
    out++;  
    consume(w);  
}
```

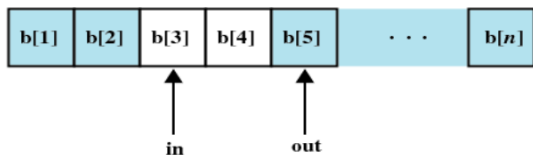


# Productor/consumidor buffer finito

Ahora, el productor debe preocuparse que el buffer no esté lleno



(a)





# Productor/consumidor buffer finito

```
producer() {  
    while (true) {  
        v = produce();  
        while ((in + 1) % n == out);  
        b[in] = v;  
        in = (in + 1) % n;  
    }  
}
```

```
consumer() {  
    while (true) {  
        while (in == out);  
        w = b[out];  
        out = (out + 1) % n;  
        consume(w);  
    }  
}
```

- El buffer es de largo  $n$  y tratado circularmente
- $in$  indica la siguiente posición libre dónde producir y  $out$  la posición del siguiente elemento a consumir
- Note que aún no damos una solución al problema

# Productor/consumidor: buffer infinito, solución errónea, semáforos binarios

```
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;

producer() {
    while(true) {
        v = produce();
        wait(s);
        put_in_buffer(v);
        n++;
        if (n == 1)
            signal(delay);
        signal(s);
    }
}

consumer() {
    wait(delay);
    while(true) {
        wait(s);
        w = take_from_buffer();
        n--;
        signal(s);
        consume();
        if (n == 0) wait(delay);
    }
}

main() {
    n = 0;
    parbegin(producer, consumer);
}
```

- Semáforo binario **s** se usa para asegurar EM
- Semáforo **delay** se usa para hacer que el consumidor espere si el buffer está vacío

# Productor/consumidor: buffer infinito, solución correcta, semáforos binarios

```
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;

producer() {
    while(true) {
        v = produce();
        wait(s);
        put_in_buffer(v);
        n++;
        if (n == 1)
            signal(delay);
        signal(s);
    }
}

consumer() {
    int m; //variable local
    wait(delay);
    while(true) {
        wait(s);
        w = take_from_buffer();
        n--;
        m = n;
        signal(s);
        consume();
        if (m == 0) wait(delay);
    }
}

main() {
    n = 0;
    parbegin(producer, consumer);
}
```

- Semáforo binario **s** se usa para asegurar EM
- Semáforo **delay** se usa para hacer que el consumidor espere si el

# Productor/consumidor, buffer infinito, solución con semáforos contadores

```
semaphore s = 1;  
semaphore n = 0;
```

```
producer()  
{  
    while(true) {  
        v = produce();  
        wait(s);  
        put_in_buffer(v);  
        signal(s);  
        signal(n);  
    }  
}
```

```
consumer()  
{  
    while(true) {  
        wait(n);  
        wait(s);  
        w = take_from_buffer();  
        signal(s);  
        consume();  
    }  
}
```

# Productor/consumidor, buffer finito, solución con semáforos contadores

```
binary_semaphore s = 1;
semaphore full = 0;
semaphore empty = sizeofbuffer;
```

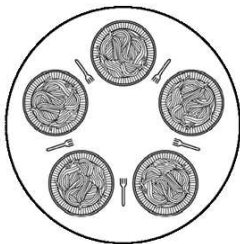
```
producer()
{
    while(true) {
        v = produce();
        wait(empty);
        wait(s);
        put_in_buffer(v);
        signal(s);
        signal(full);
    }
}
```

```
consumer()
{
    while(true) {
        wait(full);
        wait(s);
        w = take_from_buffer();
        signal(s);
        signal(empty);
        consume(w);
    }
}
```

- Semáforo **empty** usado para contar el número de espacios vacíos
- Semáforo **full** usado para contar el número de espacios usados
- Semáforo **s** usado para EM

## El problema de los filósofos comensales

- Cinco filósofos se sientan alrededor de una mesa con una fuente de spaghetti.
- Cuando un filósofo no está comiendo, está filosofando.
- Como carecen de habilidades manuales, necesitan de dos tenedores para comer, pero sólo hay cinco.
- Cuando uno o dos de los tenedores están siendo ocupados por el filósofo de al lado, el filósofo vuelve a filosofar.
- Cuando un filósofo deja de comer spaghetti, coloca los tenedores sobre la mesa.



# filósofos: solución 1

```
Semaphore fork[5] = {1}
void philosopher(int i) {
    while (true) {
        think();
        Wait(fork[i]);
        Wait(fork[(i+1)mod 5]);
        eat();
        Signal(fork[(i+1) mod 5]);
        Signal(fork[i]);
    }
}

main()
{
    parbegin(philosopher(0),philosopher(1),
             philosopher(2),philosopher(3),
             philosopher(4));
}
```

Deadlock!!!

# filósofos: solución 1

Una posible solución al problema de deadlock es controlar que sólo 4 tenedores puedan ser usados simultáneamente

```
Semaphore fork[5] = {1}
Semaphore room = 4;
void philosopher(int i) {
    while (true) {
        think();
        Wait(room);
        Wait(fork[i]);
        Wait(fork[(i+1)mod 5]);
        eat();
        Signal(fork[(i+1) mod 5]);
        Signal(fork[i]);
        Signal(room);
    }
}

main() {
    parbegin(philosopher(0),philosopher(1),
            philosopher(2),philosopher(3),
            philosopher(4));
}
```



# El problema de los lectores/escritores

- Uno o más lectores pueden estar simultáneamente leyendo un archivo
- Pero sólo un escritor puede estar escribiendo el archivo
- Si un escritor está escribiendo el archivo, ningún lector puede estar leyendo el archivo. Ej: sistema de biblioteca
- Note que se garantiza que los lectores sólo leen
- Si lectores y escritores leyeran y escribieran, volveríamos al problema general de exclusión mutua

# Lectores/escritores: solución 1

```
int readcount;
semaphore x = 1, wsem = 1;

void reader() {
    while (true) {
        Wait(x);
        readcount++;
        if (readcount == 1)
            Wait(wsem);
        Signal(x);
        READUNIT();
        Wait(x);
        readcount--;
        if (readcount == 0)
            Signal(wsem);
        Signal(x);
    }
}

void writer() {
    while (true) {
        Wait(wsem);
        WRITEUNIT();
        Signal(wsem);
    }
}

main() {
    readcount = 0;
    parbegin(reader, counter);
}
```

- Lectores tienen prioridad
- **Inanición escritores**

## Lectores/escritores: solución 2

```
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;

void reader() {
    while (true) {
        Wait(z);
        Wait(rsem);
        Wait(x);
        readcount++;
        if (readcount == 1)
            Wait(wsem);
        Signal(x);
        Signal(rsem);
        Signal(z);
        READUNIT();
        Wait(x);
        readcount--;
        if (readcount == 0)
            Signal(wsem);
        Signal(x);
    }
}

void writer() {
    while (true) {
        Wait(y);
        writecount++;
        if (writecount == 1)
            Wait(rsem);
        Signal(y);
        Wait(wsem);
        WRITEUNIT();
        Signal(wsem);
        Wait(y);
        writecount--;
        if (writecount == 0)
            Signal(rsem);
        Signal(y);
    }
}

main() {
    readcount = writecount = 0;
    parbegin(reader, counter);
}
```

// Escritores tiene prioridad

# Problema típico con semáforos

- Los semáforos no implementan la solución al problema de la EM
- Es posible que el programador de la solución se equivoque al inicializar o usar los semáforos. Por ejemplo:

```
semaphore n = 0
semaphore s = 1
void consumer() {
    while (true) {
        Signal(s);
        Wait(s);
        take();
        Wait(n);
    }
}

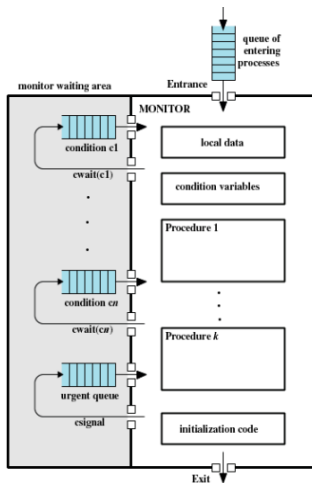
void producer() {
    while (true) {
        produce();
        Wait(s);
        append();
        Signal(s);
        Wait(n);
    }
}
```

- Existen muchas formas de implementar un solución errónea al simple problema de la EM.
- Los monitores nacieron como un apoyo al programador para evitar dichos errores

# Monitor

- Un monitor es un módulo de software que evita errores de EM
- Características principales:
  - Variables locales son accesadas sólo dentro del monitor; es decir por la funciones del monitor
  - Para ingresar al monitor, un proceso invoca algunas de las funciones del monitor
  - Sólo un proceso puede estar ejecutándose en el monitor: exclusión mútua
- Los monitores necesitan **variables de condición**:
  - Suponga que un proceso dentro del monitor necesita suspenderse hasta que una condición se haga verdadera
  - Las variables de condición proveen el mecanismo de sincronización para suspender un proceso dentro del monitor y reanudar su ejecución en otro momento

# Estructura de un monitor



# Variables de condición

Las variables de condición (VC) pueden accederse sólo usando dos funciones:

- `Wait(c)`: bloquea la ejecución del proceso llamador en la VC `c`. El monitor queda libre para ser usado por otro proceso
- `Signal(c)`: resume la ejecución de algún proceso bloqueado por un `Wait()` sobre la misma VC `c`. Si hay varios procesos bloqueados, elegir cualquiera; si no hay ninguno bloqueado, no hacer nada
- Note que las funciones `Wait()` y `Signal()` de los monitores se parecen a las funciones `Wait()` y `Signal()` de los semáforos, **PERO NO son iguales** y cumplen una función distinta
- Las VC no tiene valor asociado.
- Las operacion `Wait()` y `Signal()` sobre VC no afectan o cambian a las VC asociadas

# Productor/consumidor con monitor

- Idea: poner el buffer dentro del monitor
- Dos métodos principales del monitor:
  - ① `append(x)`: agrega el item `x` al buffer
  - ② `v = take()`: saca un item del buffer y lo retorna en `v`
- Dos procesos clientes del monitor: `Producer()` y `Consumer()`
- Sólo uno de los procesos puede estar ejecutando métodos del monitor a la vez

```
void producer() {  
    while (true) {  
        x = produce();  
        M.append(x);  
    }  
}  
  
void consumer() {  
    while (true) {  
        v = M.take();  
        consume(v);  
    }  
}  
  
main() {  
    parbegin(producer, consumer);  
}
```



# Productor/consumidor con monitor, a lo C++

```
monitor M {  
    char buffer[N];  
    int in, out;  
    int count;  
    cond notfull, notempty;  
public:  
    M() : in(0), out(0), count(0) {}  
    void append(char x);  
    char take();  
};
```

```
void M::append(char x) {  
    if (count == N)  
        Wait(notfull);  
    buffer[in] = x;  
    in = (in+1) % N;  
    count++;  
    Signal(notempty);  
}
```

```
char M::take() {  
    if (count == 0)  
        Wait(notempty);  
    v = buffer[out];  
    out = (out+1) % N;  
    count--;  
    Signal(notfull);  
}
```

# Filósofos con monitor

```
main() {  
    parbegin(philosopher(0), philosopher(1), philosopher(2), philosopher(3), philosopher(4))  
}  
  
void philosopher(int i) {  
    while (true) {  
        think();  
        P.get_forks();  
        eat();  
        P.put_forks();  
    }  
}
```

# Filósofos con monitor

Monitor P

```
state[5] of {THINKING, HUNGRY, EATING};  
cond self[5];
```

public:

```
P() : state[0] (THINKING), state[1] (THINKING), state[2] (THINKING),  
      state[3] (THINKING), state[4] (THINKING) {}
```

```
void get_forks(int i) {  
    state[i] = HUNGRY;  
    test(i);  
    if (state[i] != EATING then Wait(self[i]));  
}
```

```
void put_forks(int i) {  
    state[i] = THINKING;  
    test((i+4) mod 5);  
    test((i+1) mod 5);  
}
```

private:

```
void test(int i) {  
    if (state[i+4 mod 5] != EATING && state[i] == HUNGRY && state[i+1 mod 5] != EATING) {  
        state[i] = EATING;  
        Signal(self[i]);  
    }  
}
```

}

# Paso de mensajes

- Los mecanismos de sincronización que hemos visto hasta ahora están basados en variables compartidas (globales), es decir la comunicación es implícita
- El paso de mensajes es un mecanismo de sincronización donde la comunicación es explícita.
- Paso de mensajes significa que un proceso envía información a otro proceso con el cual no comparten memoria
- Dos primitivas básicas de comunicación:
  - ❶ `send(destination, message)`
  - ❷ `receive(source, message)`

# Sincronización

- El paso de mensajes implica necesariamente un nivel básico de sincronización: el receptor de la información no puede recibirla antes que el emisor la envíe.
- **Blocking send, blocking receive** Tanto el emisor como el receptor se bloquean hasta que el mensaje llegue al receptor
- **Nonblocking send, blocking receive** El emisor continúa su ejecución después de enviarla El receptor se bloquea hasta que llegue
- **Nonblocking send, nonblocking receive** Ninguno se bloquea

# Tipos de direccionamiento

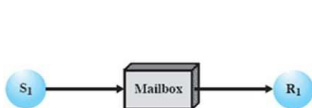
- **Direccionamiento directo**

- La primitiva `send()` incluye el identificador del proceso destino
- La primitiva `receive()` podría (o no) saber quien le envía el mensaje y tomaría la información del mensaje fuente
- Así, podría enviar una confirmación de recepción del mensaje

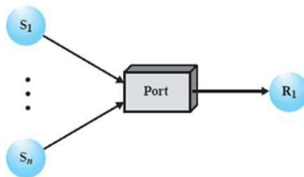
- **Direccionamiento indirecto**

- Los mensajes no se envía de un proceso a otro, sino a una estructura de datos compartida consistente de colas que almacenan temporalmente mensajes
- Mailboxes
- Así, un proceso deposita un mensaje en el mailbox y otro lo recupera. Permite tener más modelos de comunicación, como se presenta a continuación

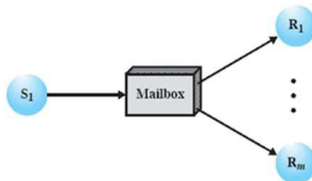
## Indirect Process Communication (...)



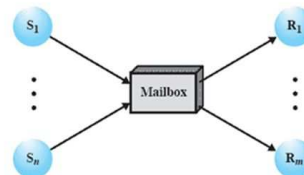
(a) One to one



(b) Many to one



(c) One to many



(d) Many to many

# Exclusión mutua con paso de mensajes

- Asumimos blocking `receive()` y nonblocking `send()`
- Se crea un “mailbox” mutex
- Para entrar a la SC, un proceso debe recibir un mensaje
- Cuando sale de la SC pone devuelta el mensaje en el mailbox
- Asumimos que si dos o más procesos hacen un `receive()` al mismo tiempo, sólo uno de ellos lo recibe

```
int n = ... // number of processes
void P(int i){
    message msg;
    while (true) {
        receive(mutex, msg);
        SC();
        send(mutex, msg);
    }
}

void main() {
    mailbox mutex = new mailbox();
    send(mutex, NULL);
    parbegin{P(1), P(2), ..., P(n)};
}
```



# Productor/Consumidor con paso de mensajes

```
int size, null;
int i;
void producer() {
    message pmsg;
    while (true) {
        receive(mayproduce, pmsg);
        pmsg = produce();
        send(mayconsume, pmsg);
    }
}
```

```
void main() {
    mailbox mayproduce = new mailbox();
    mailbox mayconsume = new mailbox();
    for (int i=1; i <= size; i++)
        send(mayproduce, null);
    parbegin(produce, consumer);
}
```

```
void consumer() {
    message cmsg;
    while (true) {
        receive(mayconsume, cmsg);
        consume(cmsg);
        send(mayproduce, null);
    }
}
```

# Sincronización con hebras POSIX

- Hebras POSIX definen funciones para la sincronización de hebras
  - ① *mutexes*: parecidos a los semáforos binarios y sirven para proveer exclusión mútua a cualquier número de hebras de un proceso
  - ② *variables de condición*: un mecanismo asociado a los mutexes. Permiten que una hebra se bloquee y libere la SC en la que se encuentra
- Estos mecanismos no sirven para sincronizar hebras de distintos procesos

# Mutex

- Un mutex es la forma más básica de sincronización y provee el mismo servicio que un semáforo binario
- Generalmente se utilizan para implementar acceso exclusivo a SC de las hebras

```
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);  
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Si una hebra invoca `pthread_mutex_lock(m)` y el mutex `m` ya está tomado, entonces la hebra se bloquea (en ese mutex), sino toma (cierra) el mutex y continúa su ejecución
- Si una hebra invoca `pthread_try_lock(m)` y el mutex `m` ya está tomado, entonces `pthread_try_lock(m)` retorna un código de error `EBUSY`; la hebra no se bloquea y puede continuar su ejecución (fuera de la SC)

# Variables de condición

- ¿ Qué sucedería si una hebra se bloquea dentro de una SC?
- Una variable de condición permite que una hebra se bloquee dentro de una SC y al mismo tiempo libere la sección crítica

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t * mutex);  
int pthread_cond_signal(pthread_cond_t *cond);  
int pthread_cond_broadcast(pthread_cond_t *cond);  
  
int pthread_cond_destroy(pthread_cond_t *cond);  
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t * attr;  
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- La hebra que invoca `pthread_cond_wait(c, m)` se bloquea hasta que la condición `c` se *señalice*; además libera el mutex `m`
- `pthread_cond_signal(c)` señala o despierta alguna otra hebra que está bloqueada en la variable de condición `c`
- `pthread_cond_broadcast(c)` desbloquea todas las hebras esperando en `c`

# Producer/consumidor usando pthreads

```
typedef struct {
    int buf[QUEUESIZE];
    int head, tail;
    int full, empty;
    pthread_mutex_t mutex;
    pthread_cond_t notFull, notEmpty;
} buffer_t;

void main() {
    buffer_t *buffer;
    pthread_t pro, con;

    buffer = bufferInit();
    pthread_mutex_init(&buffer->mutex, NULL);
    pthread_cond_init(&buffer->notFull, NULL);
    pthread_cond_init(&buffer->notEmpty, NULL);

    pthread_create(&pro, NULL, producer, (void *) buffer);
    pthread_create(&con, NULL, consumer, (void *) buffer);

    pthread_join (pro, NULL);
    pthread_join (con, NULL);
    exit 0;
}
```

# El Productor

```
void *producer(void *arg)
{
    int v;
    buffer_t *buffer;
    buffer = (buffer_t *) arg;

    while (true) {
        v = produce();
        pthread_mutex_lock (&buffer->mutex);
        while (buffer->full) {
            pthread_cond_wait (&buffer->notFull, &buffer->mutex);
        }
        put_in_buffer(buffer, v);
        pthread_cond_signal(&buffer->notEmpty);
        pthread_mutex_unlock(&buffer->mutex);
    }
}
```

# El Consumidor

```
void *consumer (void *arg)
{
    int v;
    buffer_t *buffer;
    buffer = (buffer_t *) arg;

    while (true) {
        pthread_mutex_lock (&buffer->mutex);
        while (buffer->empty) {
            pthread_cond_wait (&buffer->notEmpty, &buffer->mutex);
        }
        v = take_from_buffer(buffer);
        pthread_cond_signal(&buffer->notFull);
        pthread_mutex_unlock(&buffer->mutex);
    }
}
```

# Barreras para hebras

- Una barrera es un mecanismo de sincronización detener (bloquear) la ejecución de un grupo de hebras en un mismo punto del programa
- Cuando la barrera está a bajo las hebras se detienen en ella en el orden en que llegan
- Cuando la barrera se levanta, las hebras continúan su ejecución
- Las barreras son útiles para implementar *puntos de sincronización* globales en un algoritmo
- Usamos mutex y variables de condición para implementar barreras
- Implementamos las siguientes funciones
  - ❶ `barrier_init()`
  - ❷ `barrier_wait()`
  - ❸ `barrier_destroy()`



# Implementación barreras con pthreads

- Una posible forma de implementación es usando dos mutexes y dos variables de condición
- Cada par representa una barrera

```
struct _sb {  
    pthread_cond_t  cond;  
    pthread_mutex_t mutex;  
    int runners; // numero de hebras que aun no llegan a la barrera  
};  
  
typedef struct {  
    int maxcnt; // numero de hebras participando en la barrera  
    struct _sb sb[2];  
    struct _sb *sbp;  
} barrier_t;
```

# Inicialización de barrera

```
int barrier_init(barrier_t *bp, int count)
{
    int i;
    if (count < 1) {
        printf("Error: numero de hebras debe ser mayor que 1\n");
        exit(-1);
    }

    bp->maxcnt = count;
    bp->sbp = &bp->sb[0];

    for (i=0; i < 2; i++) {
        struct _sb *sbp = &(bp->sb[i]);
        sbp->runners = count;
        pthread_mutex_init(&sbp->mutex, NULL);
        pthread_cond_init(&sbp->cond, NULL);
    }
    return(0);
}
```

# Espera con barreras

```
int barrier_wait(barrier_t *bp)
{
    struct _sb *sbp = bp->sbp;

    pthread_mutex_lock(&sbp->mutex);
    if (sbp->runners == 1) {
        if (bp->maxcnt != 1) {
            sbp->runners = bp->maxcnt;
            bp->sbp = (bp->sbp == &bp->sb[0])? &bp->sb[1] : &bp->sb[0];
            pthread_cond_broadcast(&sbp->cond);
        }
    } else {
        sbp->runners--;
        while (sbp->runners != bp->maxcnt)
            pthread_cond_wait(&sbp->cond, &sbp->mutex);
    }
    pthread_mutex_unlock(&sbp->wait_mutex);
}
```

# Uso de barreras

```
void *proceso(void *arg) {
    int *tidptr, tid;
    tidptr = (int *) arg; tid = *tidptr;
    printf("hola justo antes de la barrera%d\n", tid);
    barrier_wait(&barrera);
}

int main(int argc, char *argv[])
{
    int i, status;
    pthread_t hebra[10];

    barrier_init(&barrera, 10);
    for (i=0; i < 10; i++){
        taskids[i] = (int *) malloc(sizeof(int));
        *taskids[i] = i;
        pthread_create(&hebra[i], NULL, &proceso, taskids[i]);
    }

    for (i=0; i < 10; i++)
        pthread_join(hebra[i], (void **) &status);

    barrier_destroy(&barrera);
}
```

# Semáforos POSIX

POSIX define semáforos con nombre y semáforos sin nombre.

- **semáforos con nombre:** Para crear un semáforo con nombre invocamos la función:

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag, mode_t mode ,
```

- **name:** nombre dado al semáforo.
- **oflag:** Puede tomar dos valores: `O_CREAT` para crea un semáforo, y `O_EXCL`, en cuyo caso la función falla si el semáforo ya existe.
- **mode\_t:** Setea los permisos sobre el semáforo.
- **value:** Especifica el valor inicial del semáforo.

- Las funciones para incrementar y decrementar el semáforo son:

```
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
```

- Finalmente, para cerrar y eliminar un semáforo:

```
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

# Semáforos POSIX (cont)

- **semáforos sin nombre:** Para crear un semáforo sin nombre invocamos la función:

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- **sem:** puntero al semáforo (inicializado por `sem_init()`)
- **pshared:**
  - Si `pshared` es cero, el semáforo debe ser visible (compartido) con todas la hebras del proceso.
  - Si `pshared` es distinto de cero, el semáforo debe ser ubicado en un área de memoria compartida, de tal forma que se comparta entre procesos.
  - **value:** valor inicial del semáforo.
- Para incrementar y decrementar el semáforo, usamos `sem_wait()` y `sem_post()`.
- Un semáforo sin nombre se destruye con:

```
int sem_destroy(sem_t *sem);
```

## Semáforos POSIX (cont)

- Los semáforos con nombre nos permiten sincronizar procesos no relacionados; en este caso los procesos deben abrir el semáforo con el mismo nombre.
- Los semáforos sin nombre nos permiten sincronizar hebras de un mismo procesos y procesos relacionados (por ejemplo padre e hijo).

# Memoria compartida System V

- Un objeto de memoria compartida representa memoria que puede ser mapeada concurrentemente en el espacio de direcciones de varios procesos.
- Para obtener un segmento compartido, invocamos `shmget()`

```
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
int shmget(key_t key, int size, int shmflg);
```

- `key`: es una llave única que identifica al segmento
  - `size`: es el tamaño en bytes del segmento
  - `shmflg`: es una palabra de flags que se construye con los permisos de acceso al segmento y con la operación OR sobre `IPC_CREAT` y `IPC_EXCL`.
- Una forma conveniente de generar una llave única es mediante:  

```
key_t ftok (char *path, char proy );
```

    - `path`: es el camino absoluto de un archivo existente
    - `proy`: es un caracter cualquiera.



# Ejemplo memoria compartida

- Las siguientes líneas crean un segmento de memoria compartida de 1Kbytes

```
...  
key_t key;  
int shmid;  
  
key = ftok("/dev/null", '1');  
shmid = shmget(key, 1024, IPC_CREAT | 0644);
```

- El archivo especificado puede ser cualquiera y no será realmente usado para almacenar la información contenida en el segmento.
- En este caso los flags de creación corresponden a
  - ① Permisos de acceso: `rw-r--r--`, es decir, el dueño (creador) del segmento puede leer y escribir, y todos los otros sólo pueden leer.
  - ② El segmento se creará si no existe o conectará el proceso al segmento si ya existe.

## Memoria compartida (cont)

- Una vez creado el segmento el proceso debe acoplarlo a su espacio de direcciones:

```
void *shmat(int shmid, void *shmaddr, int shmflg);
```

- `shmid` es el identificador del segmento
  - `shmaddr` especifica la dirección donde deseamos acoplar el segmento. Generalmente especificamos el valor 0 y así dejar al SO decidir donde realizar el acoplamiento.
- `shmat` incrementa el número de procesos que han mapeado el segmento a su espacio de direcciones.
- `shmat` retorna la dirección del segmento (puntero a `void`) la cual puede ser *casteado* a cualquier tipo de datos.

## Memoria compartida (cont)

- Cuando un proceso ya no necesita el segmento compartido, se desacopla de él con:

```
int shmdt(const void *shmaddr);
```

- `shmaddr` es la dirección retornada por `shmat`.
- Esta operación decrementa el número de procesos que han mapeado el segmento a su espacio de direcciones.
- Para destruir un segmento compartido, usamos:

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- Esta función implementa varias operaciones sobre segmentos de memoria compartida.
- Usar `cmd = IPC_RMID` y `buf = NULL` para desacoplar el segmento identificado por `shmid`.
- El segmento será realmente destruido cuando ya no exista ningún proceso que lo tenga acoplado.

# Memoria compartida POSIX

- El procedimiento para usar memoria compartida en POSIX es levemente diferente a System V
- Primero, obtenemos un segmento de memoria compartida con `shm_open()`

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int shm_open(const char *name, int oflag, mode_t mode);
```

- `name`: es el nombre (único) del segmento
- `oflag`: es un flag con el modo de creación
- `mode`: *no afecta si el segmento es abierto para lectura, escritura o ambos*. Generalmente usamos el valor 0
- `shm_open()` retorna el descriptor del segmento

# Memoria compartida POSIX (cont)

- Luego, mapeamos el segmento al espacio virtual del proceso

```
#include <sys/mman.h>
```

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd,  
           off_t offset);
```

- `addr` es la dirección virtual del proceso donde el descriptor `fd` debiera ser mapeado. Lo usual es especificar `NULL` y dejar al SO decidir dónde mapearlo
- `offset` es el desplazamiento, a partir del inicio del descriptor, donde comienza el mapeo
- `length` es el largo (en bytes) del segmento a mapear
- `prot` especifica flags de protección del segmento. Generalmente usamos `PROT_READ` | `PROT_WRITE` para acceso de lectura y escritura
- `flags` especifica la visibilidad de los cambios al segmento
  - ① `MAP_SHARED`: los cambios se ven en todos los procesos
  - ② `MAP_PRIVATE`: los cambios son sólo visibles al proceso que los realiza
  - ③ `MAP_FIXED`: *mejor no usar*
- `mmap()` retorna la dirección donde se mapeó el segmento

# Ejemplo completo productor.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <fcntl.h>

#define SHMSZ 11200
char SEM_NAME[] = "tarea1";

int main() {
    int ch;
    int shmid;
    key_t key;
    int *shm,*s;
    sem_t *mutex;

    if ( (mutex = sem_open(SEM_NAME,O_CREAT,0666,1)) == SEM_FAILED) {
        perror("unable to create semaphore");
        sem_unlink(SEM_NAME);
        exit(-1);
    }
```

## producer.c (cont)

```
key = ftok("/home/rannou/junk", 'R');
if ( (shmidx = shmget(key, SHMSZ, IPC_CREAT|0666)) < 0 ) {
    perror("failure in shmget");
    exit(-1);
}

if ( *(shm = shmat(shmidx, NULL, 0)) == -1 ) {
    perror("failure in shmat");
    exit(-1);
}

s = shm;
for(ch=0; ch<=2700; ch++){
    sem_wait(mutex);
    *s++ = ch;
    sem_post(mutex);
}

while(*shm != -1) sleep(1);

sem_close(mutex);
sem_unlink(SEM_NAME);
shmctl(shmidx, IPC_RMID, 0);
exit(0);
```

# cliente.c

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>

#define SHMSZ 11200
char SEM_NAME[] = "tarea1";

int main()
{
    int ch;
    int shmid;
    key_t key;
    int *shm,*s;
    sem_t *mutex;

    if ( (mutex = sem_open(SEM_NAME,O_CREAT,0666,1)) == SEM_FAILED) {
        perror("reader:unable to execute semaphore");
        sem_close(mutex);
        exit(-1);
    }
```



## cliente.c (cont)

```
key = ftok("/home/rannou/junk", 'R');
if( (shmid = shmget(key, SHMSZ, 0666)) < 0) {
    perror("reader:failure in shmget");
    exit(-1);
}

shm = shmat(shmid, NULL, 0);
s = shm;
for(s=shm; *s != 2700; s++) {
    sem_wait(mutex);
    printf("%d\n", *s);
    sem_post(mutex);
}

*shm = -1;
sem_close(mutex);
shmctl(shmid, IPC_RMID, 0);
exit(0);
}
```