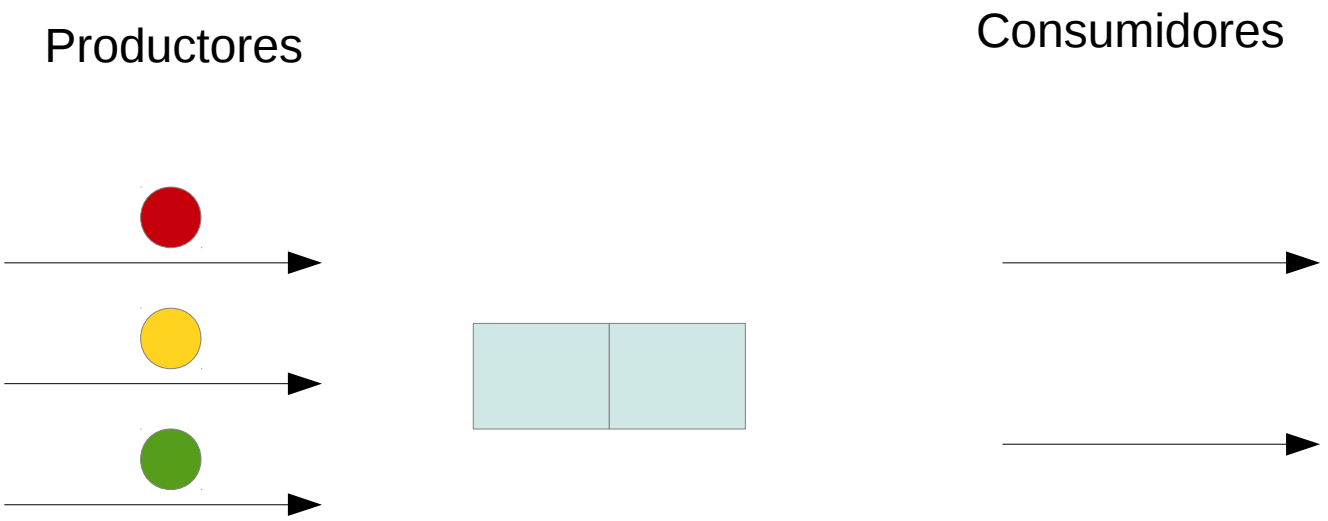
Productores / Consumidores
Condiciones

Juan Quintela, Javier París
{quintela, javier.paris}@udc.es

Descripción

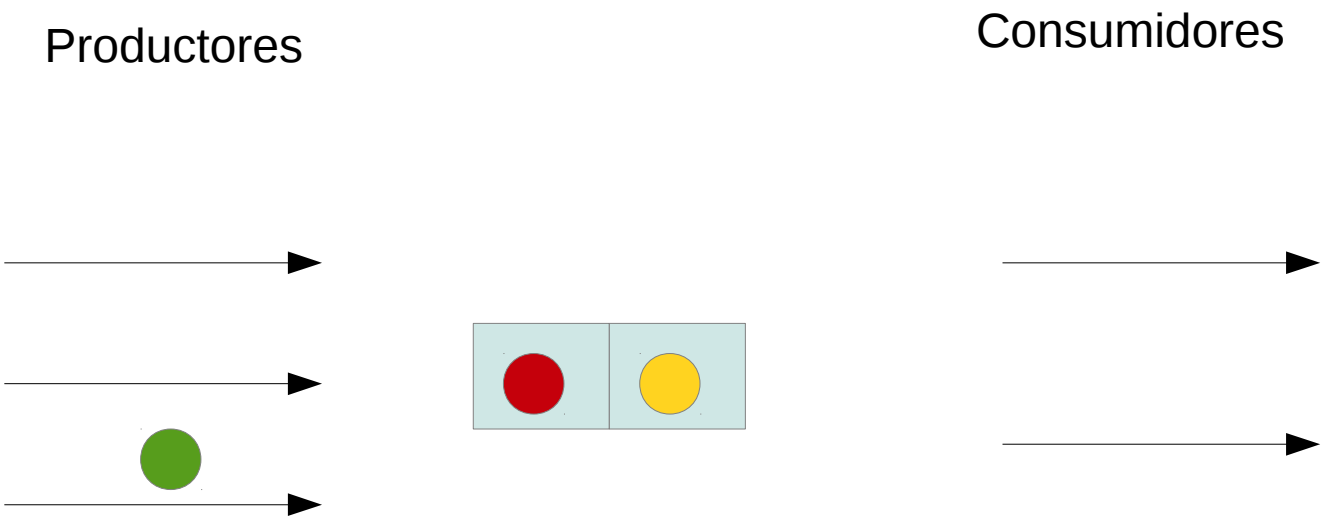
- Buffer compartido entre procesos que insertan elementos (productores), y procesos que eliminan elementos (consumidores).
- Hay que controlar el acceso al buffer compartido para mantener los datos consistentes.
- Si el buffer se llena los productores deben esperar a que los consumidores eliminen elementos.
- Si el buffer se vacía, los consumidores deben esperar a que los productores inserten elementos nuevos.

Descripción: Ejemplo



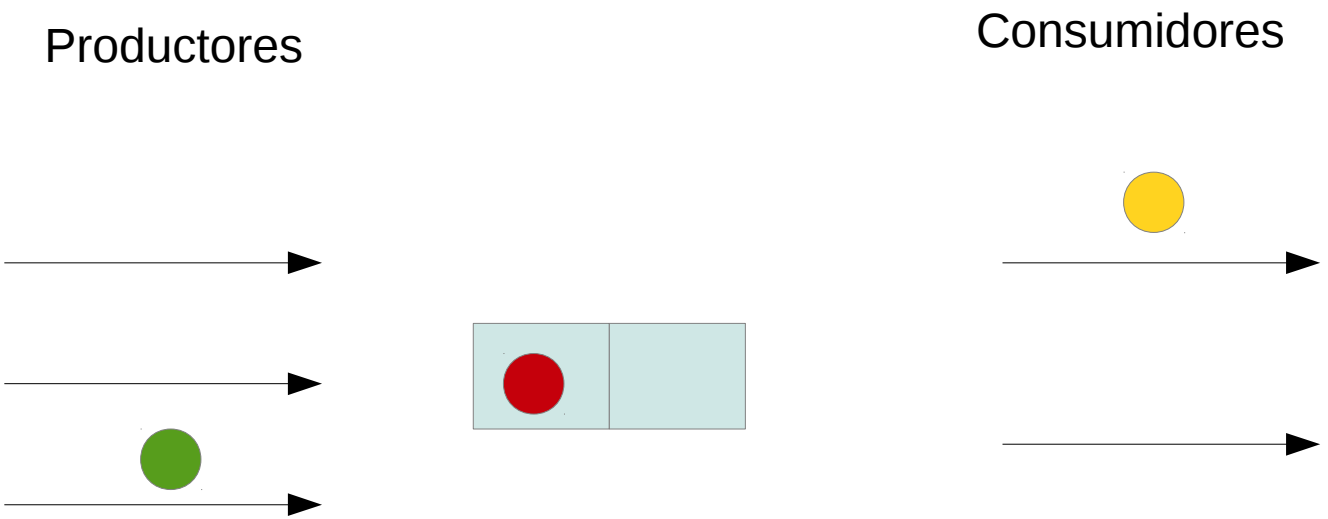
3 Productores intentan insertar en el buffer compartido

Descripción: Ejemplo



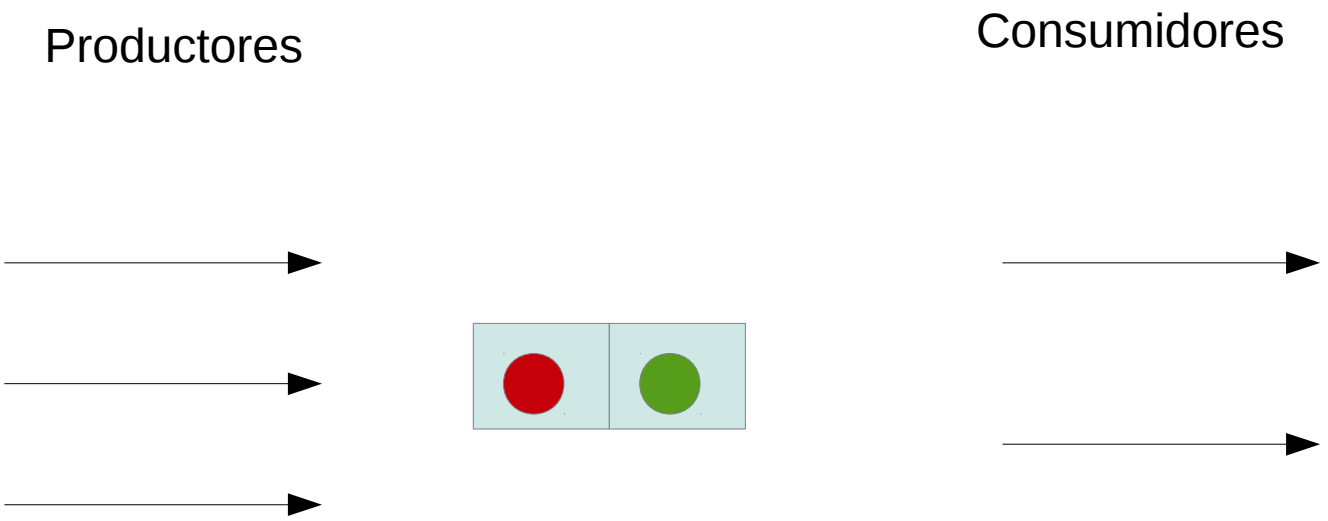
Los dos primeros insertan sus productos. El tercero tiene que esperar a que se libere alguna posición.

Descripción: Ejemplo



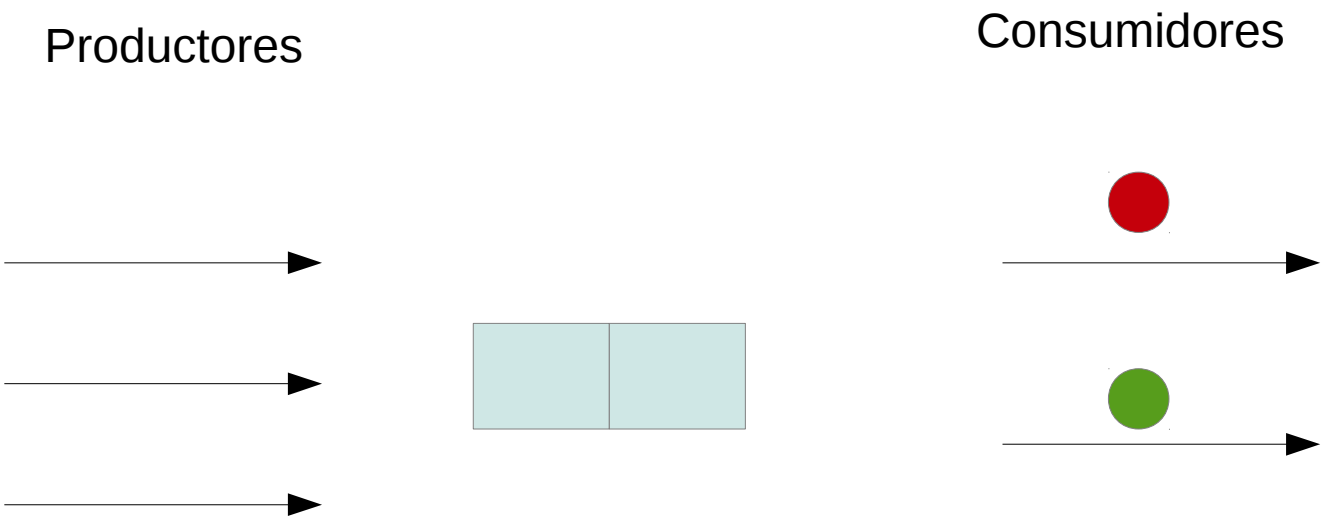
Un consumidor retira un producto del buffer y lo hace algo con él.

Descripción: Ejemplo



El tercer productor puede insertar.

Descripción: Ejemplo



Los consumidores retiran los dos últimos productos.

Ejemplos de uso

- Hay muchos problemas de concurrencia que funcionan como productores/consumidores:
  - Buffers en una conexión de red.
  - Buffers para la reproducción de video/audio.
  - Servidores web multithread, con un thread maestro que lee peticiones, y múltiples threads para procesarlas.

Solución con Threads: Estructuras.

- Vamos a asumir que tenemos funciones implementadas para acceder al buffer compartido:

  void insert(elemento);

  elemento remove();

  int count; // Número de elementos en el buffer
- En una solución real estas funciones podrían tener algún tipo de criterio para escoger algún elemento en concreto del buffer compartido.
- Vamos a empezar asumiendo que el buffer es infinito y no se vacía nunca.

Solución con Threads: Productor

```
pthread_mutex_t buffer_lock;


while(1) {
    elemento e = crear_elemento();
    pthread_mutex_lock(buffer_lock);
    insert(e);
    count++;
    pthread_mutex_unlock(buffer_lock);
}
```

Solución con Threads: Consumidor

```
while(1) {
    elemento e;
    pthread_mutex_lock(buffer_lock);
    e = remove();
    count--;
    pthread_mutex_unlock(buffer_lock);
    // Hacer algo con el elemento :)
}
```

Buffer Limitado

- Vamos a añadir el caso de que el buffer tenga tamaño finito(MAX_BUFFER) y pueda estar vacío. El número de elementos se guarda en count.
- El consumidor tiene que comprobar que haya elementos en el buffer antes de hacer remove()
- El productor tiene que comprobar que el buffer no esté lleno antes de hacer insert()

Productor

```
pthread_mutex_t buffer_lock;
while(1) {
    elemento e = crear_elemento(); int inserted;
    inserted =0;
    do {
        pthread_mutex_lock(buffer_lock);
        if(count<MAX_BUFFER) {
            insert(e);
            count++;
            inserted=1;
        }
        pthread_mutex_unlock(buffer_lock);
    } while(!inserted);
}
```

Consumidor

```
while(1) {
    elemento e; int removed;
    removed=0;
    do {
        pthread_mutex_lock(buffer_lock);
        if(count>0) {
            e = remove();
            count--;
            removed=1;
        }
        pthread_mutex_unlock(buffer_lock);
    } while(!removed);
    // Hacer algo con el elemento :)
}
```

Solución con buffer limitado

- Esta solución tiene el problema de que las esperas de productores y consumidores son **activas,** es decir, comprueban continuamente el valor de count hasta que tiene el valor correcto.
- Este tipo de esperas tiene un consumo alto de cpu, por lo que solo son viables si sabemos que la espera va a ser corta.
- Vamos a añadir un mecanismo que nos permita dormir a un thread hasta que el estado del problema cambie.

Sincronización por Condiciones

- Una condición permite a los procesos/threads suspender su ejecución hasta que se les despierte.
- Se diseñaron porque a veces es necesario interrumpir la ejecución en medio de una sección crítica hasta que el estado de un recurso compartido cambie por la acción de otro proceso.
- Ese otro proceso es el que debe encargarse de despertar a los que puedan estar esperando.

Sincronización por Condiciones

- En la librería pthread:

  pthread_cond_t

  int pthread_cond_init(pthread_cond_t *, pthread_condattr_t *);

  int pthread_cond_signal(pthread_cond_t *);

  int pthread_cond_broadcast(pthread_cond_t *);

  int pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *);

  int pthread_cond_timedwait(pthread_cond_t *,

  pthread_mutex_t *, const struct timespec *);

Productores/Consumidores con Condiciones

- Vamos a usar dos condiciones, una para hacer esperar a los consumidores con el buffer vacío, y otra para los productores con el buffer lleno:

  pthread_cond_t buffer_full;

  pthread_cond_t buffer_empty;

Productor

```
while(1) {
    elemento e = crear_elemento();
    pthread_mutex_lock(buffer_lock);
    while(count==MAX_BUFFER) { // Esperar hasta que haya hueco
        pthread_cond_wait(buffer_full, buffer_lock);
    }
    insert(e);
    count++;
    if(count==1) pthread_cond_broadcast(buffer_empty);
    pthread_mutex_unlock(buffer_lock);
}
```

Consumidor

```
while(1) {
    elemento e;
    pthread_mutex_lock(buffer_lock);
    while(count==0)
        pthread_cond_wait(buffer_empty, buffer_lock);
    e = remove();
    count--;
    if(count==BUFFER_MAX-1)
        pthread_cond_broadcast(buffer_full);
    pthread_mutex_unlock(buffer_lock);
    // Hacer algo con el elemento :)
}
```

Wait es atómico

- Internamente el wait hace de forma atómica:

```
wait(cond *c, mutex *m) {
    unlock(m);
    espera(m);
    lock(m);
}
```

Atómico

Wait es Atómico

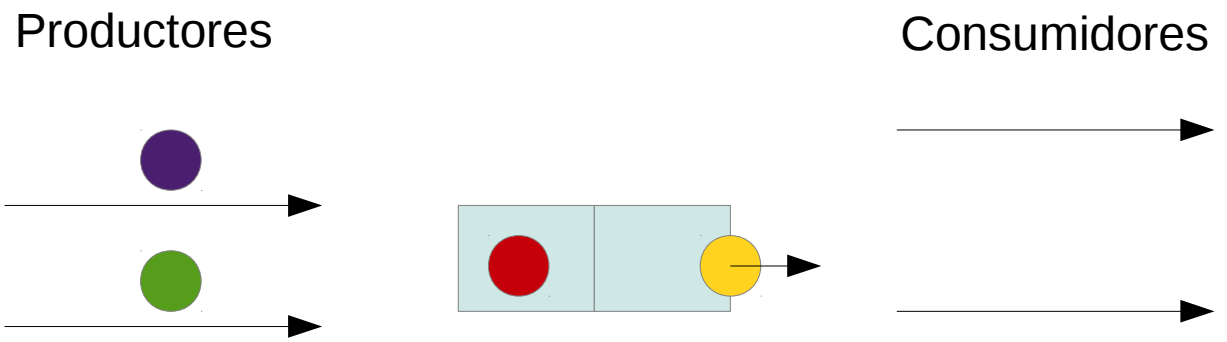
Con un wait no atómico el estado del buffer puede cambiar antes de que durmamos. Por ejemplo, en el productor:

```
pthread_mutex_lock(buffer_lock);
while(count==MAX_BUFFER) {
    unlock(m);
      <== Un consumidor elimina un elemento del buffer, y lanza un broadcast, pero este thread aun no está esperando y lo pierde.
    espera(m);
    lock(m);
}
```

Este problema se llama lost wakeup

Productor con buffer limitado

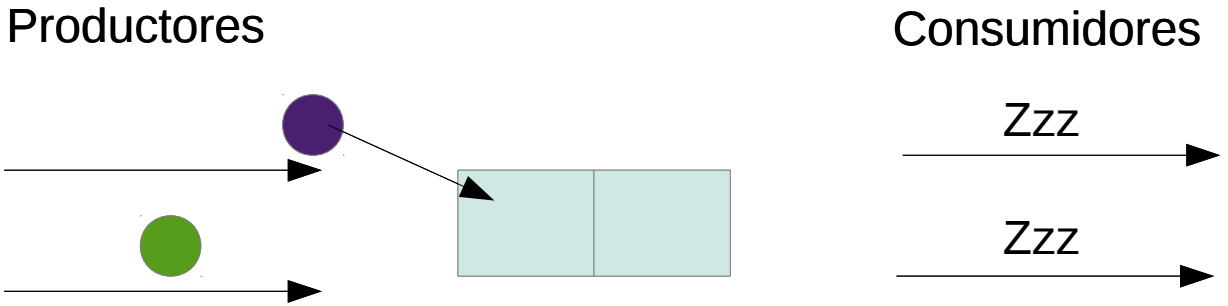
- ¿Por que se usa un while para esperar en vez de un if? => Estamos usando un broadcast para despertar, por lo que no sabemos cuantos productores despiertan. Si solo se ha retirado 1 elemento y despierta más de 1, se van a intentar insertar elementos con el buffer lleno.



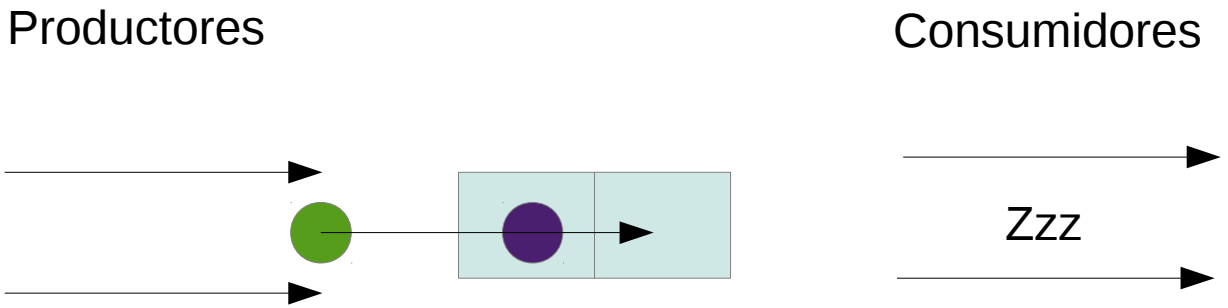
Al retirar el elemento, si despertamos a los dos productores que esperan y no Vuelven a comprobar el estado del buffer, intentarán insertar los dos.

Productor con buffer limitado

- ¿Por que usar broadcast en vez de signal?



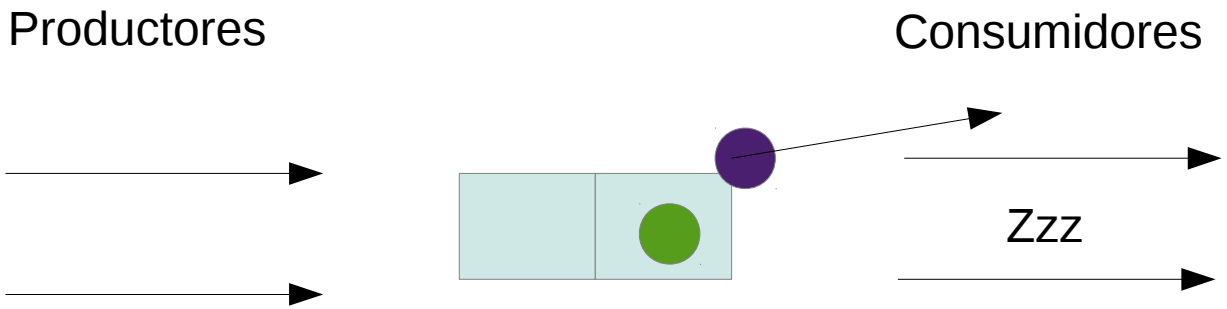
El primer productor inserta y hace signal. Se despierta el primer consumidor



Antes de que el consumidor quite el producto, el segundo productor inserta. Como el buffer no está vacío, no hace signal.

Productor con buffer limitado

- ¿Por que usar broadcast en vez de signal?



El primer consumidor retira un producto. El segundo consumidor duerme, a pesar de que el buffer no está vacío.

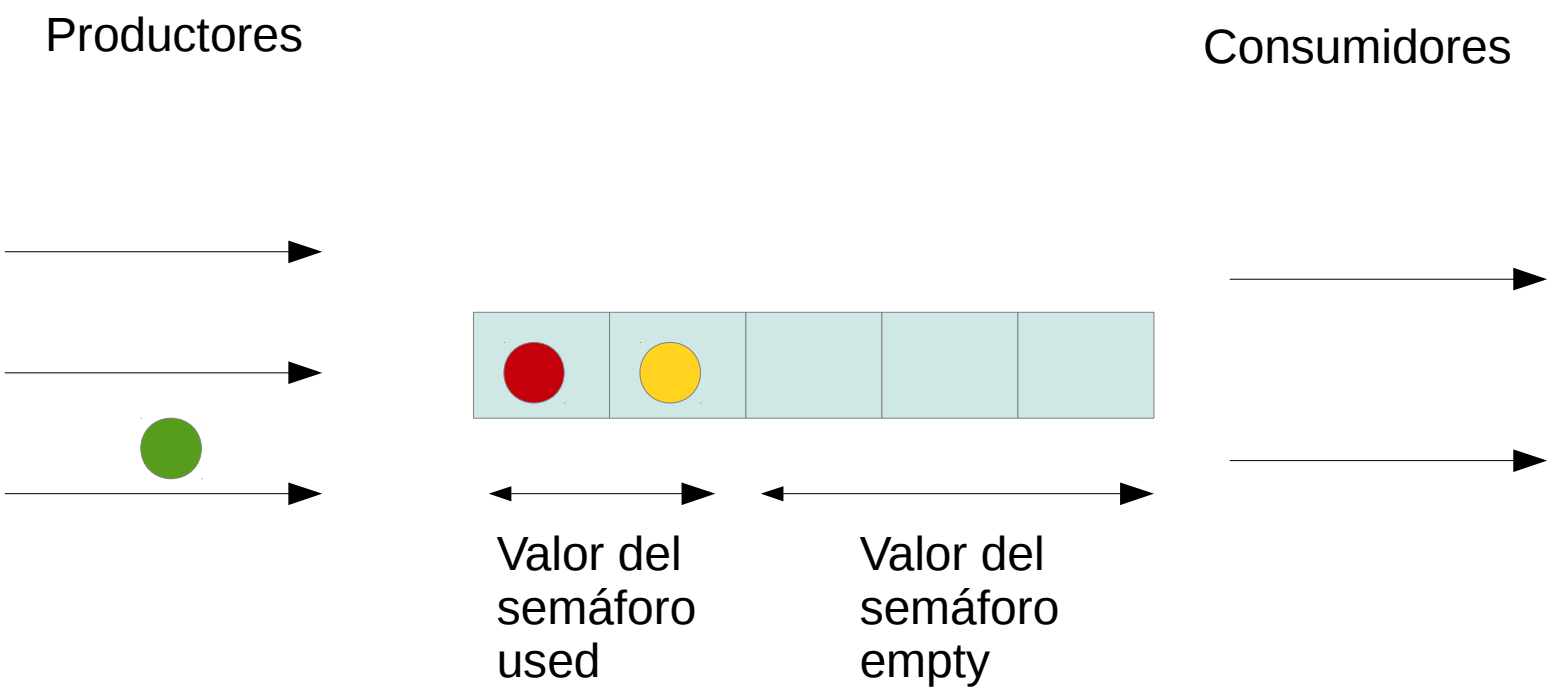
Con Semáforos

- Si no tenemos condiciones se puede implementar una solución con semáforos.
- Se usan dos semáforos para controlar el número de posiciones en el buffer que están llenas y vacías. El contador del semáforo representará el número de celdas en ese estado:

  sem_t empty;

  sem_t used;

Con semáforos



Con Semáforos

- Esas variables las inicializaremos con el buffer vacío:

  sem_init(&empty, 1, BUFFER_MAX);

  sem_init(&used, 1, 0);

- Además usaremos otro semáforo para controlar el acceso al buffer:

  sem_t mutex;

  sem_init(&mutex, 1, 1);

Productor con Semáforos

```
while(1) {
    elemento e=crear_elemento();
    sem_wait(&empty);
    sem_wait(&mutex);
    insert(e);
    count++;
    sem_post(&mutex);
    sem_post(&used);
}
```

Consumidor con Semáforos

```
while(1) {
    elemento e;
    sem_wait(&used);
    sem_wait(&mutex);
    e=eliminar();
    count--;
    sem_post(&mutex);
    sem_post(&empty);
    // hacer algo con e :)
}
```