Productores/Consumidores

Concurrencia y Paralelismo

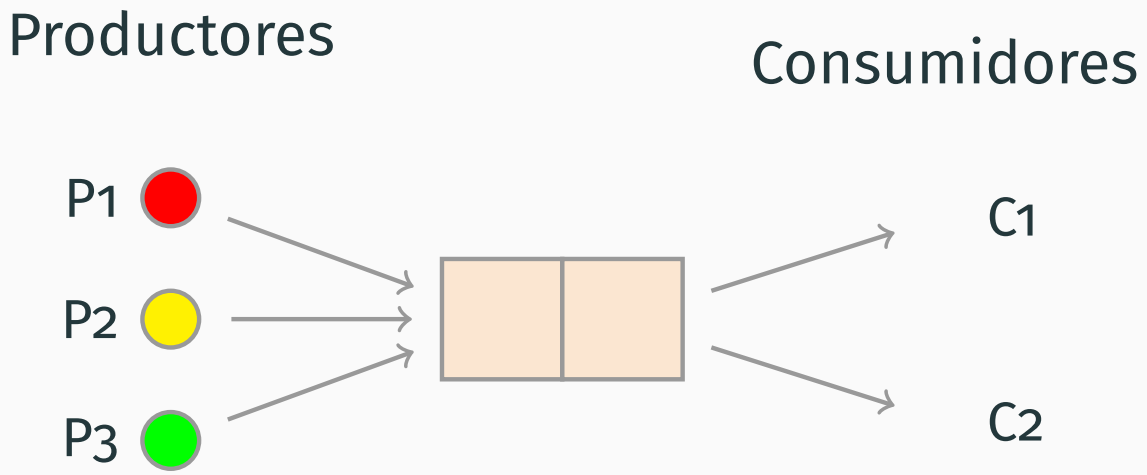
Juan Quintela
quintela@udc.es

Javier París
javier.paris@udc.es

Descripción

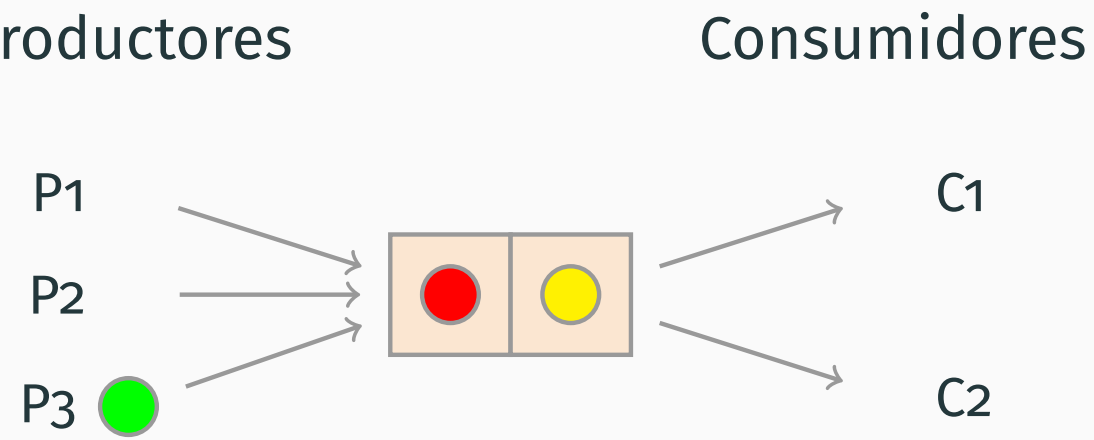
- Buffer compartido entre procesos que insertan elementos (productores), y procesos que eliminan elementos (consumidores).
- Hay que controlar el acceso al buffer compartido para mantener los datos consistentes.
- Si el buffer se llena los productores deben esperar a que los consumidores eliminen elementos.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

Descripción: Ejemplo



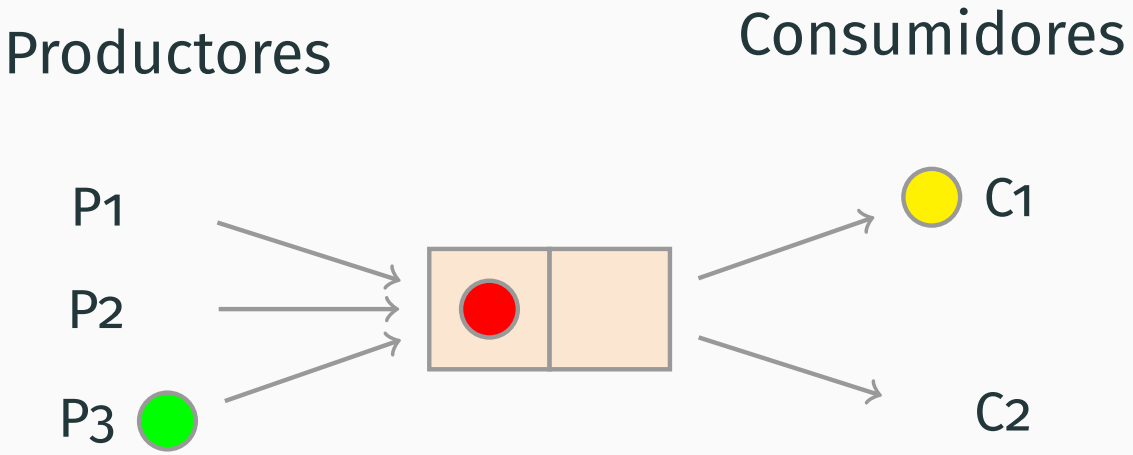
3 Productores intentan insertar en el buffer compartido

Descripción: Ejemplo



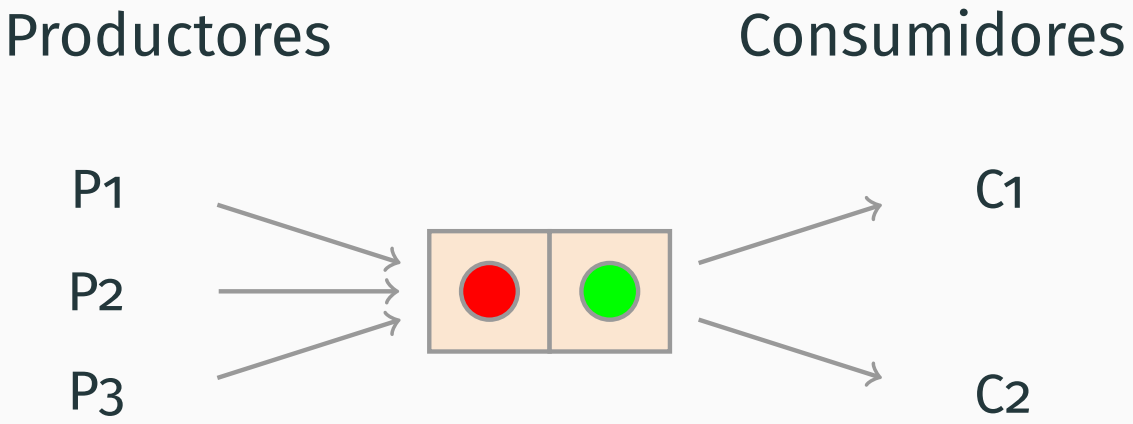
Los dos primeros insertan sus productos. El tercero tiene que esperar a que se libere alguna posición.

Descripción: Ejemplo



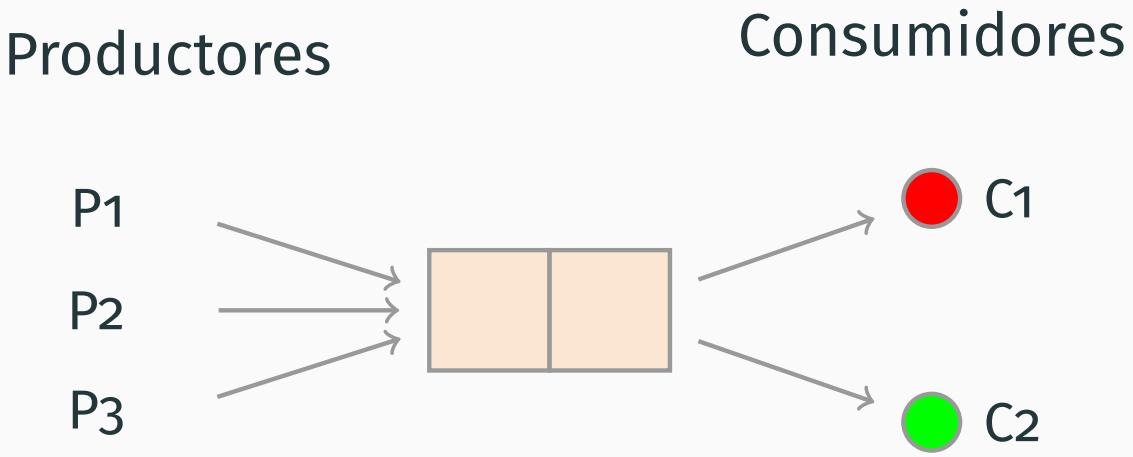
Un consumidor retira un producto del buffer y lo hace algo con él (lo consume).

Descripción: Ejemplo



El tercer productor puede insertar.

Descripción: Ejemplo



Los consumidores retiran los dos últimos productos.

Aplicaciones

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:

Funciones buffer

```
void insert(elemento e);
elemento remove();
int elements(); // Número de elementos en el buffer
int buffer_size(); // Capacidad máxima del 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

Productor

```
pthread_mutex_t buffer_lock;

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

Solución con Threads: Consumidor

Consumidor

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

Buffer limitado

- Vamos a añadir el caso de que el buffer tenga tamaño finito y pueda estar vacío.
- 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 con buffer limitado

Productor

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

Consumidor con buffer limitado

Consumidor

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

Problemas

- 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:

Condiciones en pthread

```
pthread_cond_t cond;

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:

condiciones

```
pthread_cond_t buffer_full;
pthread_cond_t buffer_empty;
```

Productor con Condiciones

Productor

```
while(1) {
  elemento e = crear_elemento();
  pthread_mutex_lock(buffer_lock);
  while(elements()==buffer_size()) { // Esperar por sitio
    pthread_cond_wait(buffer_full, buffer_lock);
  }
  insert(e);
  if(elements()==1) pthread_cond_broadcast(buffer_empty);
  pthread_mutex_unlock(buffer_lock);
}
```

Consumidor con Condiciones

Consumidor

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

Wait es atómico

Internamente wait se hace de forma atómica:

Wait

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

Wait es atómico

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

Wait no atómico

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

Este problema se llama **lost wakeup**

Usando Condiciones

Para usar correctamente condiciones **siempre** hay que:

- comprobar antes y después de esperar usando un while el estado del programa para ver si podemos continuar o no. Solo podemos omitir la comprobación de después si estamos completamente seguros de que no va a causar problemas.
- tener bloqueado el mutex que protege ese estado antes de comprobarlo, y hasta después de terminar la espera y pasar por la sección crítica.
- pasar ese mutex al llamar a wait para que otro thread pueda cambiarlo.
- evitar mantener otros mutex bloqueados mientras esperamos salvo que estemos absolutamente seguros que no va a provocar un problema.

Usando Condiciones

Thread que comprueba y espera

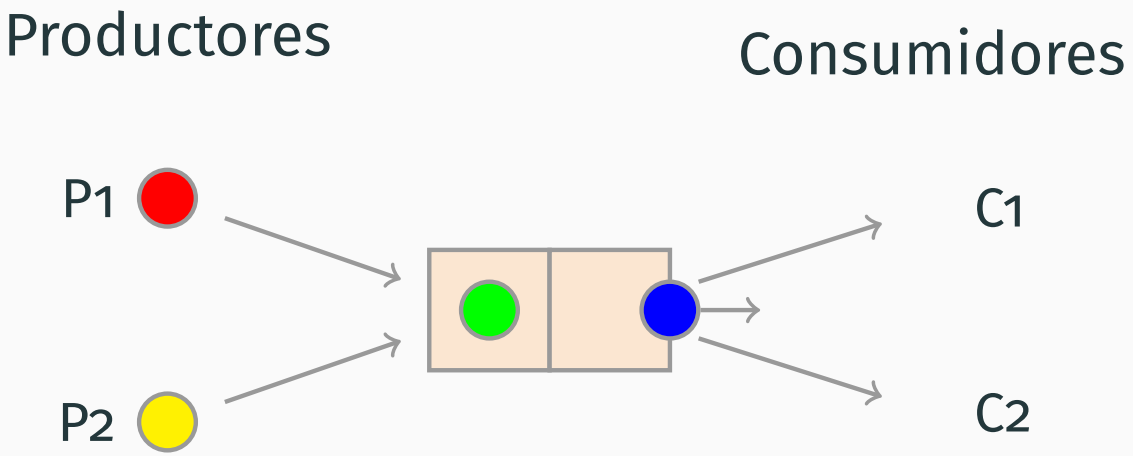
```
lock(mutex_que_protege_estado);
while(!estado es valido para continuar)
    wait(condicion, mutex_que_protege_estado);
// seccion critica
unlock(mutex_que_protege_estado);
```

Thread que cambia el estado y notifica

```
lock(mutex_que_protege_estado);
Cambiar estado;
if(estado no era valido para thread que
           espera pero con el cambio sí) {
    broadcast(condicion);
}
unlock(mutex_que_protege_estado);
```

While/If para comprobar la condición

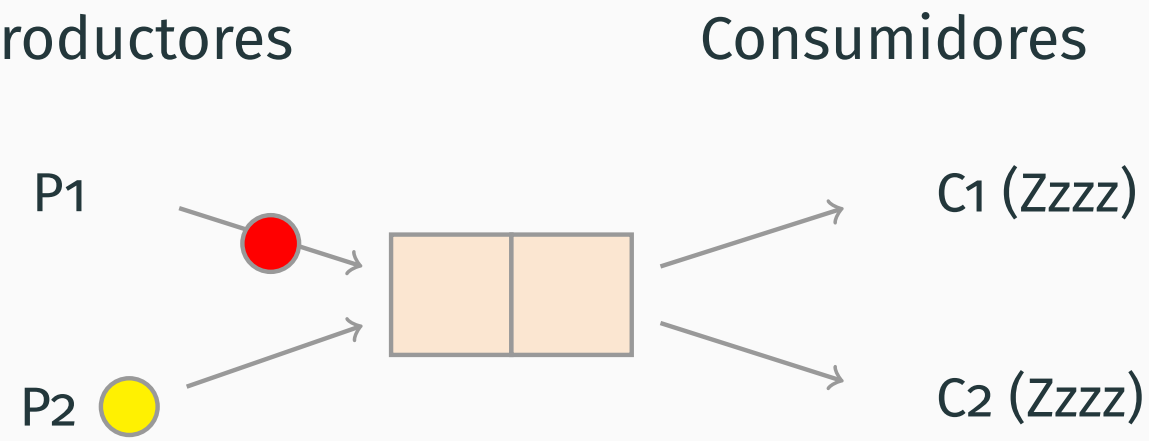
¿Por que se usa un while para esperar en vez de un if? => Estamos usando un broadcast para despertar, por lo que no sabemos cuantos productores despiertan. Si solo se ha retirado 1 elemento y despierta más de 1, se van a intentar insertar elementos con el buffer lleno.



Al retirar el elemento, si despertamos a los dos productores y no comprueban el estado del buffer insertarán los dos.

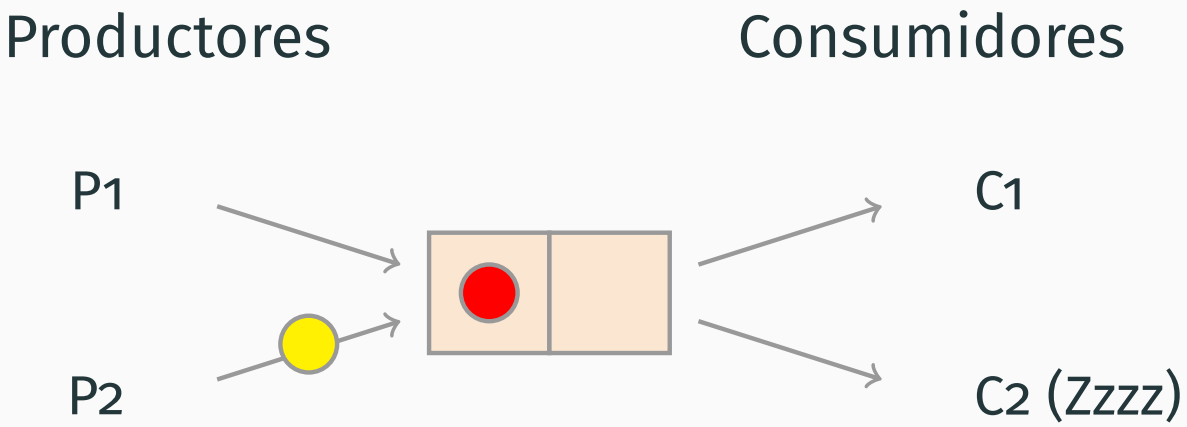
¿Broadcast o Signal?

¿Por que usar broadcast en vez de signal?



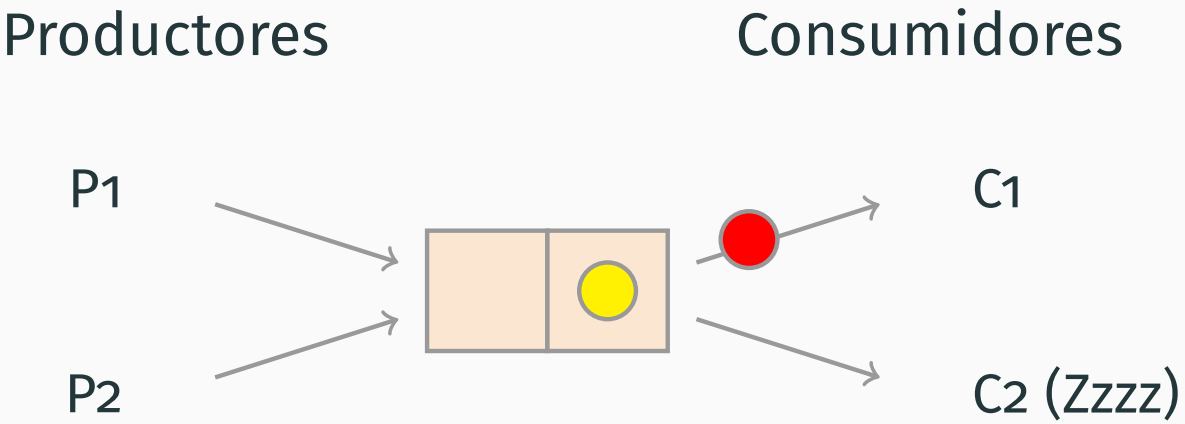
El primer productor inserta y hace signal. Se despierta el primer consumidor.

¿Broadcast o Signal?



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

¿Broadcast o Signal?



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

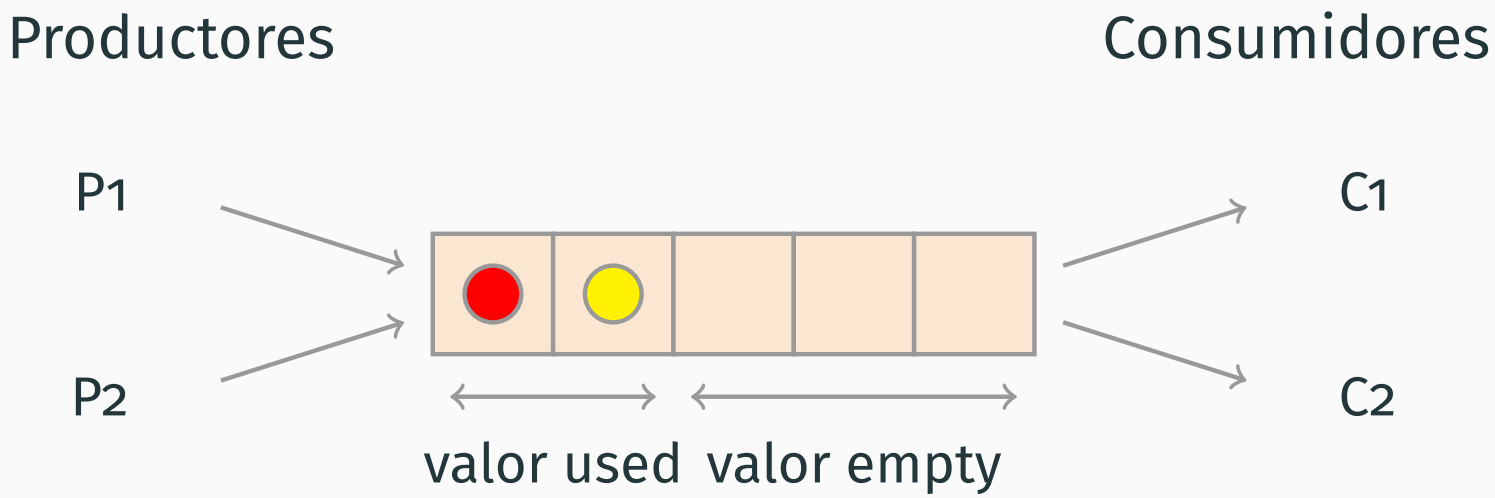
Con semáforos

- 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:

Semáforos

```
sem_t empty;
sem_t used;
```

Con semáforos



Con semáforos

- Esas variables las inicializaremos con el buffer vacío:

Inicialización del los semáforos

```
sem_init(&empty, 1, buffer_size());
sem_init(&used, 1, 0);
```

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

Semáforo de acceso al buffer

```
sem_t mutex;
sem_init(&mutex, 1, 1);
```

Productor con semáforos

Productor con semáforos

```
while(1) {
  elemento e=crear_elemento();
  sem_wait(&empty); // empty-- o espera
  sem_wait(&mutex);
  insert(e);
  sem_post(&mutex);
  sem_post(&used); // used++ o despertar consumidor
}
```

Consumidor con semáforos

Consumidor con semáforos

```
while(1) {
  elemento e;
  sem_wait(&used); // used-- o esperar
  sem_wait(&mutex);
  e=remove();
  sem_post(&mutex);
  sem_post(&empty); // empty++ o despertar productor
  // hacer algo con e
}
```