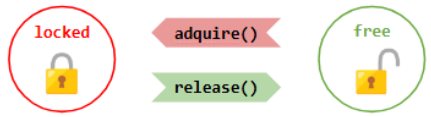
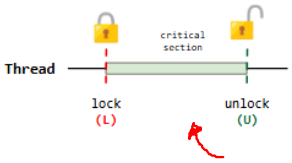
25/11/2025 - Sistemas Operativos (UdeC)

1. Primitivas de sincronización

1 Locks

- Un lock es un objeto de memoria que proporciona exclusión mutua.

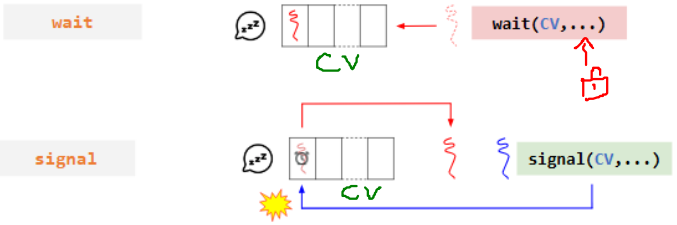





```
adquire(lock);
Región crítica
release(lock);
```

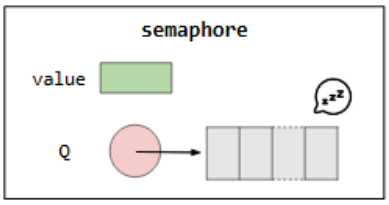
2 Variables de condición (CV)

- Primitiva de sincronización inventada por Dijkstra en 1968.
- Objeto con un valor entero no negativo (asociado al estado).



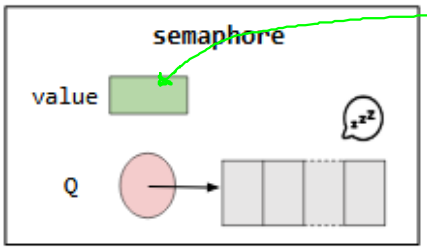
3 Semáforo

- Objeto con un valor entero no negativo (asociado al estado).
- Cola de hilos que se encuentran dormidos.




```
typedef struct {
  int value;
  struct process *Q;
} sem_t;
```

Semáforo




```
typedef struct {
  int value;
  struct process *Q;
} sem_t;
```

init

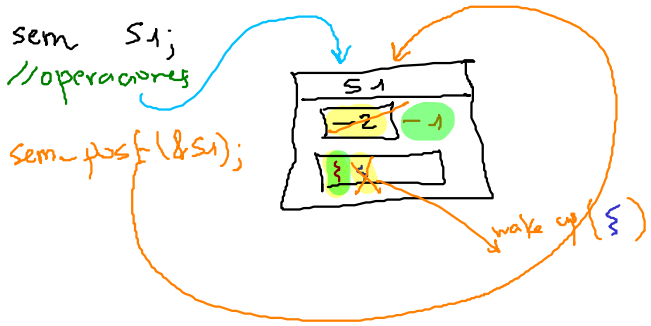
```
sem_init(sem_t *s, int value) {
  s->value = initval
}
```

wait

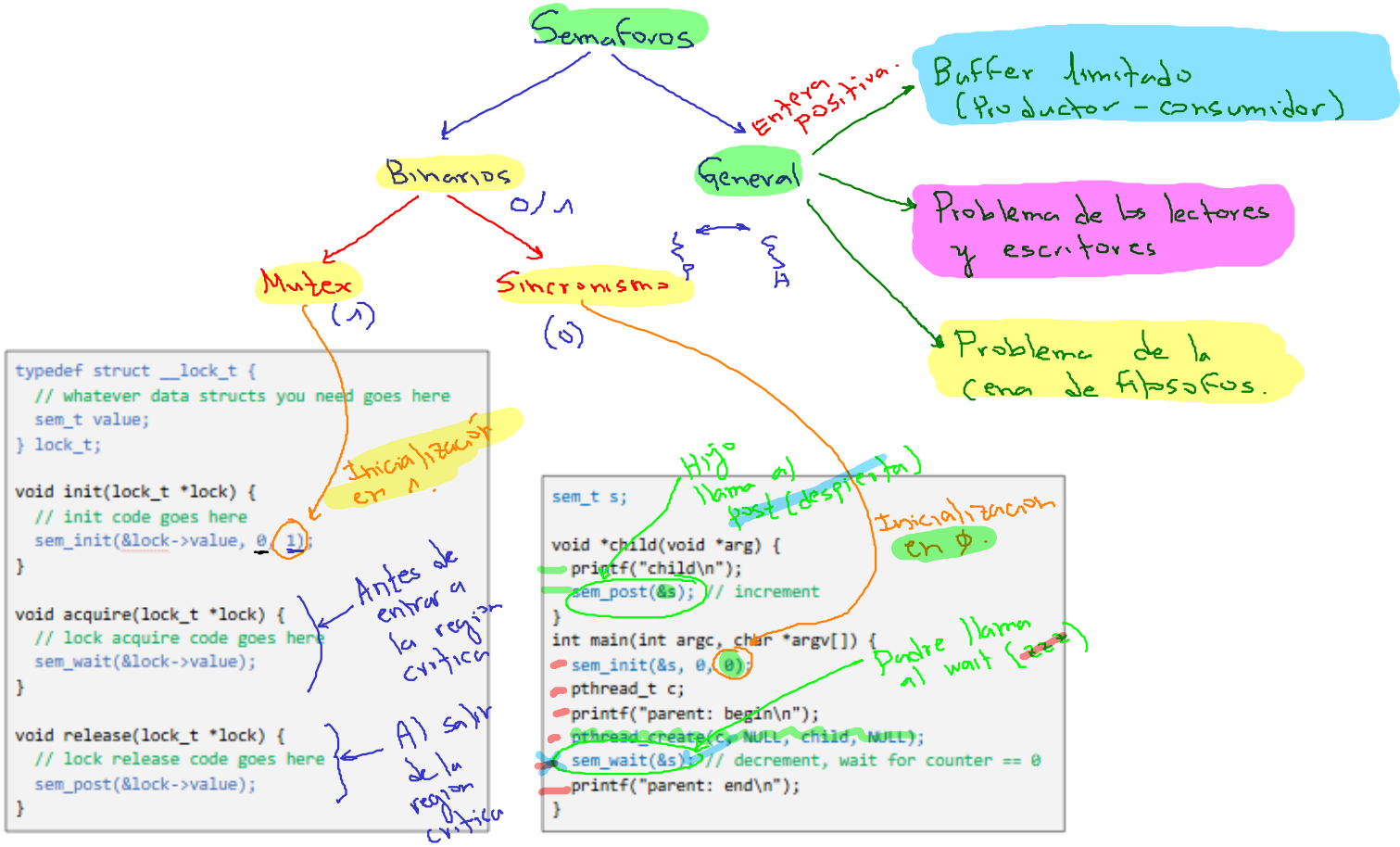
```
void sem_wait(sem_t *s) { // Must be executed atomically
  s->value--;
  if (S->value < 0) {
    add this process to s->Q;
    block();
  }
}
```

post

```
void sem_post(sem_t *s) { // Must be executed atomically
  s->value++;
  if (s->value <= 0) {
    remove a process P from s->Q;
    wakeup(P);
  }
}
```



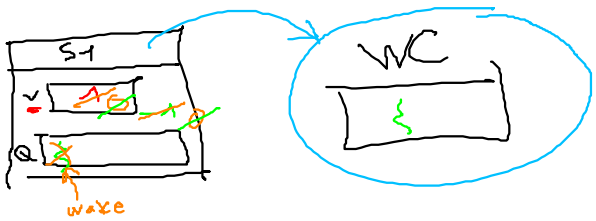
2. Ejemplos de aplicacion basicos empleando semaforos




```c
typedef struct __lock_t {
  // whatever data structs you need goes here
  sem_t value;
} lock_t;

void init(lock_t *lock) {
  // init code goes here
  sem_init(&lock->value, 0, 1);
}

void acquire(lock_t *lock) {
  // lock acquire code goes here
  sem_wait(&lock->value);
}

void release(lock_t *lock) {
  // lock release code goes here
  sem_post(&lock->value);
}
```

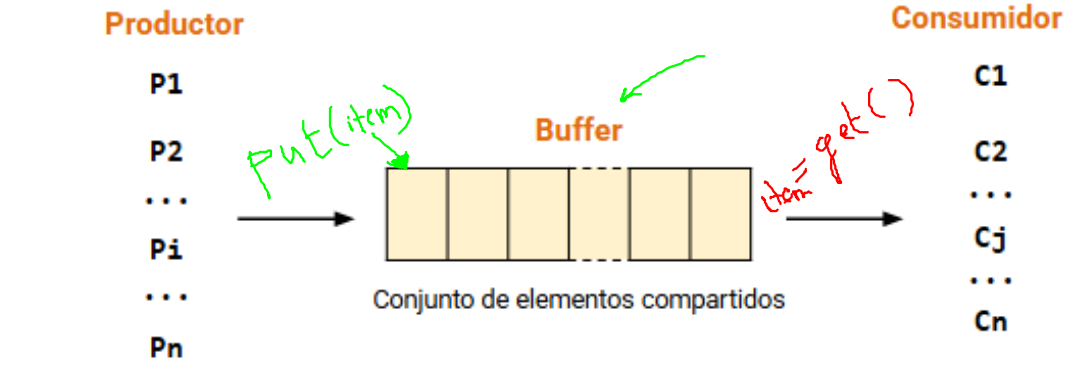
```c
sem_t s;

void *child(void *arg) {
  printf("child\n");
  sem_post(&s); // increment
}
int main(int argc, char *argv[]) {
  sem_init(&s, 0, 0);
  pthread_t c;
  printf("parent: begin\n");
  pthread_create(c, NULL, child, NULL);
  sem_wait(&s); // decrement, wait for counter == 0
  printf("parent: end\n");
}
```

```c
sem s1
sem_init(&s1, 1);

sem_wait(&s1);
Reg. Critica
sem_post(&s1);
```



3. Problema del productor consumidor

- También conocido como el problema del buffer limitado (Bounded Buffer Problem)
- El productor y el consumidor se ejecutan a tasas diferentes:
  - No hay serialización de uno detrás de otro.
  - Las tareas son independientes.



Como implementamos el problema — Plan

1. Caso base (sin pensar en concurrencia)

- [x] Estructuras de datos y funciones necesarias
  - [x] Buffer circular (Limitado) → Que puede pasar
  - [x] Poner elementos en el buffer
  - [ ] Sacar elementos del buffer

https://iximiuz.com/en/posts/nodejs-writable-streams-distilled/

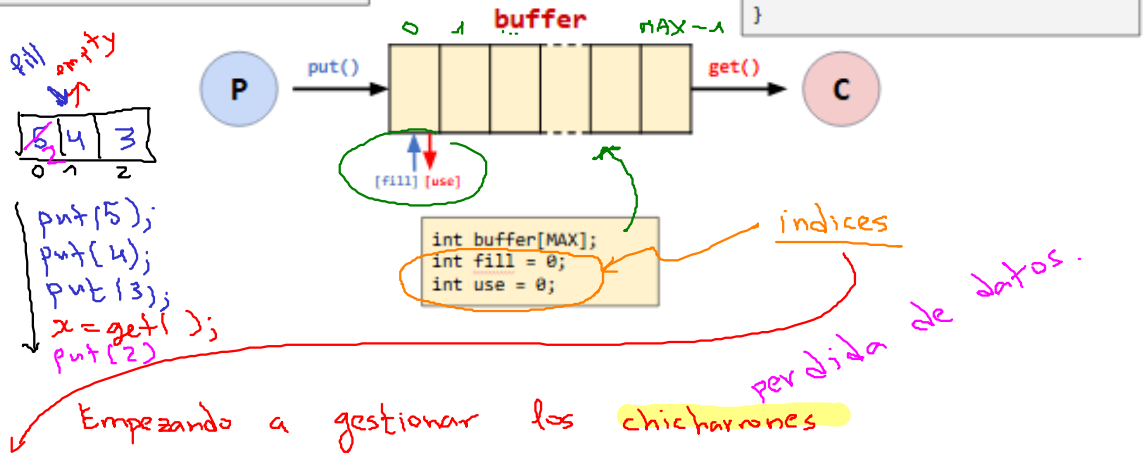
Rutinas put y get

put()

```
void put(int value) {
  buffer[fill] = value;   // Line F1
  fill = (fill + 1)%MAX;  // Line F2
}
```

get()

```
int get() {
  int tmp = buffer[use];  // Line G1
  use = (use + 1)%MAX;    // Line G2
  return tmp;
}
```

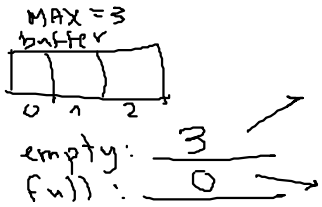


2. Secciones criticas y sincronizacion

Empleamos semaforos

empty = MAX

full = 0



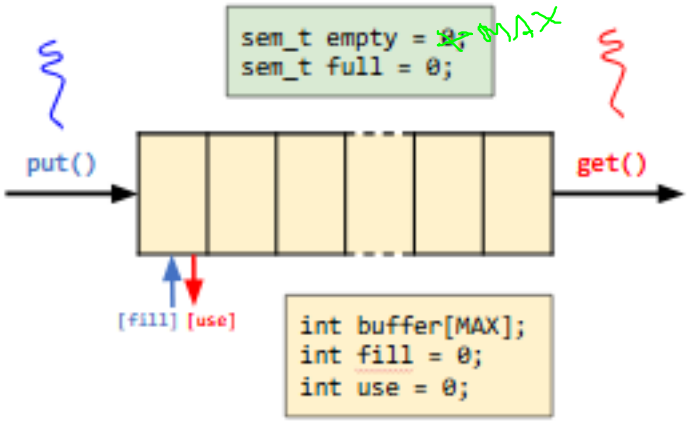
Inicializacion: Buffer vacio

MAX slots libres

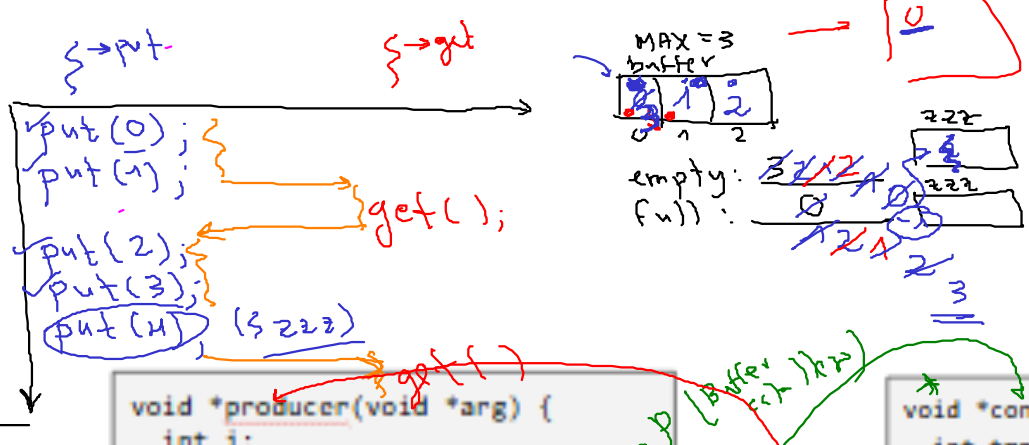
0 slots ocupados

```
sem_t empty - 0;
sem_t full - 0;

int main(int argc, char *argv[]) {
  // ...
  sem_init(&empty, 0, MAX); // MAX are empty
  sem_init(&full, 0, 0);    // 0 are full
  // ...
}
```

Declaracion de la estructura de datos



Tenemos que adaptar put() y get() para que sean inmunes a Perdida de datos



```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&empty);  // Line P1
    put(i);            // Line P2
    sem_post(&full);   // Line P3
  }
}
```

Productor → put()

P Espera a que el buffer se vacié para colocar datos en este

```
void *consumer(void *arg) {
  int tmp = 0;
  while (tmp != -1) {
    sem_wait(&full);   // Line C1
    tmp = get();       // Line C2
    sem_post(&empty);  // Line C3
    printf("%d\n", tmp);
  }
}
```

Consumidor → get()

C Espera a que el buffer se vuelva a llenar para usarlo

La solución es logica y tiene sentido pero recordemos que como hay concurrencia existen desafios

Preguntas: Sera que Funciona?

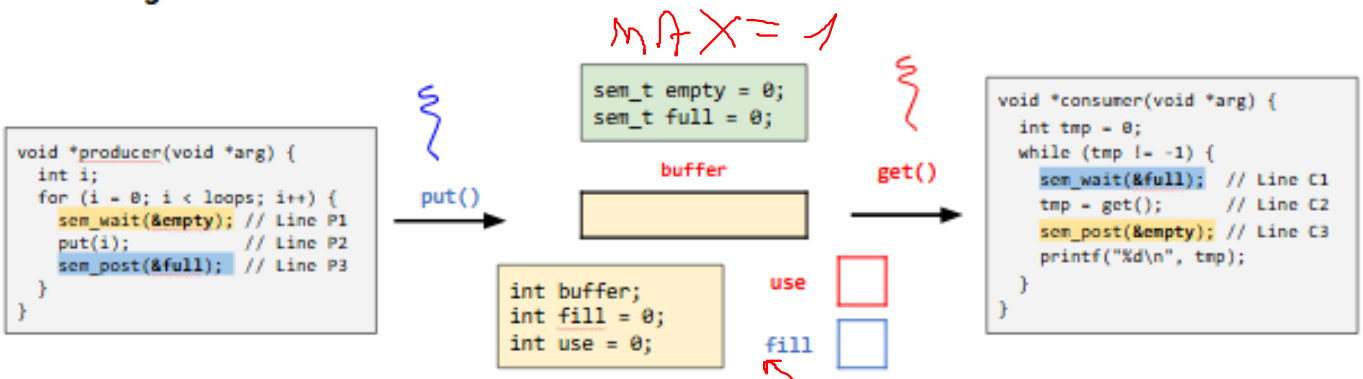
Analisis →
a. Que no haya Race condition.
b. Que haya sincronizacion (no bloqueo)

Implementación 1 - Caso 1

Caso 1: Single buffer

- **Dos hilos**:
  - Productor
  - Consumidor
- **Single buffer**: `MAX = 1`

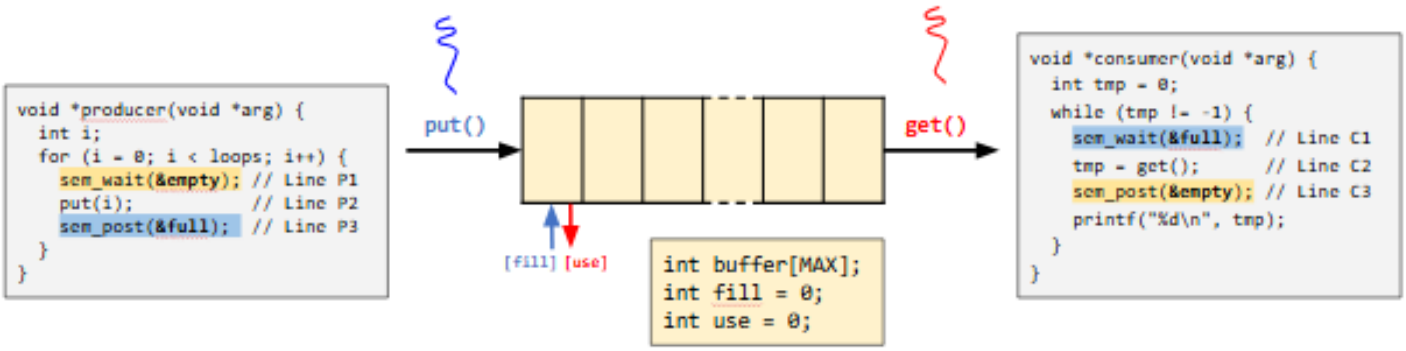
```
sem_t empty = 0;
sem_t full = 0;

int main(int argc, char *argv[]) {
  // ...
  sem_init(&empty, 0, 1);   // 1 is empty
  sem_init(&full, 0, 0);    // 0 is full
  // ...
}
```



Empleando varios slots MAX > 1 pero la logica viene a ser la misma (por ahora)

Caso 2: Buffer circular

- El buffer comparte varios elementos (**`MAX > 1`**) entre productores y consumidores.



Caso 2: Buffer circular

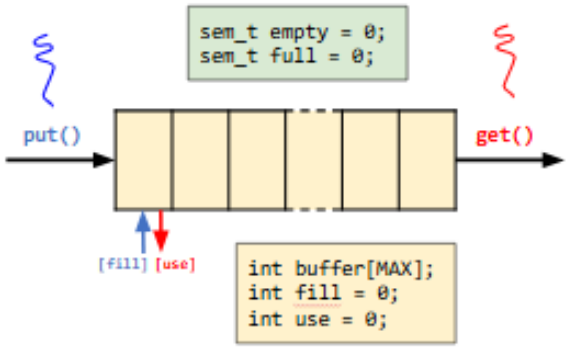
Requiere dos semáforos:

- **`empty`**: Que se inicializa a **`MAX`** (**`MAX`** espacios vacíos (empty buffers) de modo que el productor puede ejecutarse **`MAX`** veces primero).
- **`full`**: Que se inicializa a **`0`** (**`0`** espacios llenos (full buffers)).

```
sem_t empty = 0;
sem_t full = 0;

int main(int argc, char *argv[]) {
  // ...
  sem_init(&empty, 0, MAX); // MAX are empty
  sem_init(&full, 0, 0);    // 0 are full
  // ...
}
```

Caso 2: Buffer circular

- El buffer comparte varios elementos (`MAX > 1`) entre productores y consumidores.




```c
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&empty); // Line P1
    put(i);           // Line P2
    sem_post(&full);  // Line P3
  }
}
```

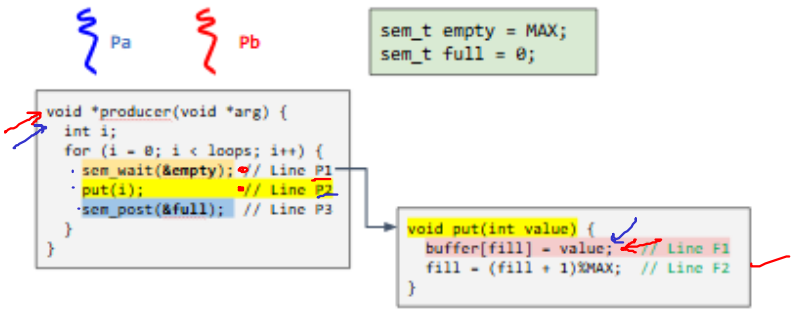
```c
void *consumer(void *arg) {
  int tmp = 0;
  while (tmp != -1) {
    sem_wait(&full);  // Line C1
    tmp = get();      // Line C2
    sem_post(&empty); // Line C3
    printf("%d\n", tmp);
  }
}
```

Chicharron

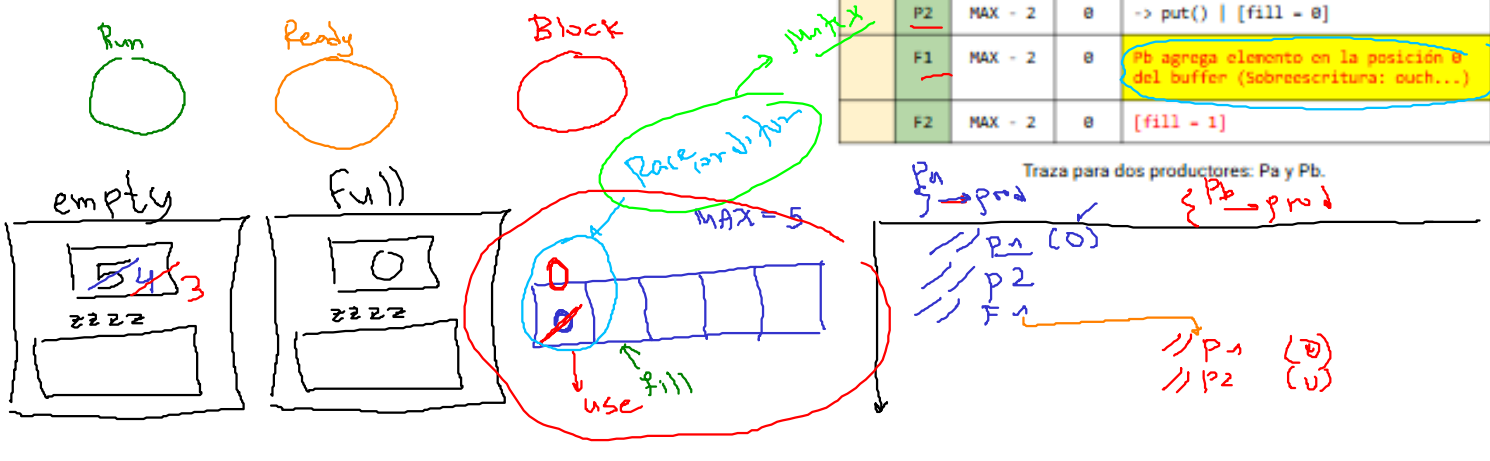
Caso 2: Buffer circular

Problema: Llenar el buffer e incrementar el contador es una sección crítica.

- Si MAX es mayor que 1 (MAX > 1):
  - **Race condición** en función **put** (línea F1).
  - El dato antiguo será sobrescrito.



Hilos		Semáforos		
Pa	Pb	empty	full	Comentarios
---	---	MAX	0	Valores iniciales: [fill = 0]
P1		MAX - 1	0	P1 -> wait(empty) | [fill = 0]
P2		MAX - 1	0	-> put() | [fill = 0]
F1		MAX - 1	0	Pa agrega elemento en la posición 0 del buffer
		MAX - 1	0	Interrupción: Pa → Pb
	P1	MAX - 2	0	P2 -> wait(empty)
	P2	MAX - 2	0	-> put() | [fill = 0]
	F1	MAX - 2	0	Pb agrega elemento en la posición 0 del buffer (Sobreescritura: ouch...)
	F2	MAX - 2	0	[fill = 1]



Problema: **Race condition**

Solución: **Agregar exclusión mutua** para proteger la sección crítica (Llenar el buffer e incrementar el índice en el buffer)

```
int main(int argc, char *argv[]) {
  // ...
  sem_init(&empty, 0, MAX); // MAX are empty
  sem_init(&full, 0, 0);    // 0 are full
  // ...
}
```

```
int main(int argc, char *argv[]) {
  // ...
  sem_init(&empty, 0, MAX); // MAX are empty
  sem_init(&full, 0, 0);    // 0 are full
  sem_init(&mutex, 0, 1);   // mutex = 1
  // ...
}
```

Caso 2: Buffer circular

Problema: **Race condition**

Solución: **Agregar exclusión mutua** para proteger la sección crítica (Llenar el buffer e incrementar el índice en el buffer)

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&empty); // Line P1
    put(i);           // Line P2
    sem_post(&full);  // Line P3
  }
}
```

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&mutex);  // line P0 (NEW LINE)
    sem_wait(&empty);  // Line P1
    put(i);            // Line P2
    sem_post(&full);   // Line P3
    sem_post(&mutex);  // line P4 (NEW LINE)
  }
}
```

```
sem_t empty = MAX;
sem_t full = 0;
sem_t mutex = 1;
```

```
void *consumer(void *arg) {
  int tmp = 0;
  while (tmp != -1) {
    sem_wait(&full);  // Line C1
    tmp = get();      // Line C2
    sem_post(&empty); // Line C3
    printf("%d\n", tmp);
  }
}
```

```
void *consumer(void *arg) {
  int tmp = 0;
  while (tmp != -1) {
    sem_wait(&mutex);  // line c0 (NEW LINE)
    sem_wait(&full);   // Line C1
    tmp = get();       // Line C2
    sem_post(&empty);  // Line C3
    sem_post(&mutex);  // line c4 (NEW LINE)
    printf("%d\n", tmp);
  }
}
```

* Volvamos a analizar si hay chicharron

Análisis de la solución que agregar exclusión mutua

Suponga que se tienen dos hilos uno productor y uno consumidor.

- El consumidor adquiere el **mutex**. (line c0).
- El consumidor llama **sem_wait()** en el semáforo full. (line C1)
- El consumidor está bloqueado y libera la CPU.
- El consumidor aún posee el **mutex**.
- El productor llama **sem_wait()** en semáforo **mutex** (line P0)
- El productor está bloqueado.

Interbloqueo (Deadlock)

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&mutex);  // line P0 (NEW LINE)
    sem_wait(&empty);  // Line P1
    put(i);            // Line P2
    sem_post(&full);   // Line P3
    sem_post(&mutex);  // line P4 (NEW LINE)
  }
}
```

```
void *consumer(void *arg) {
  int tmp = 0;
  while (tmp != -1) {
    sem_wait(&mutex);  // line c0 (NEW LINE)
    sem_wait(&full);   // Line C1
    tmp = get();       // Line C2
    sem_post(&empty);  // Line C3
    sem_post(&mutex);  // line c4 (NEW LINE)
    printf("%d\n", tmp);
  }
}
```

Análisis de la solución que agregar exclusión mutua

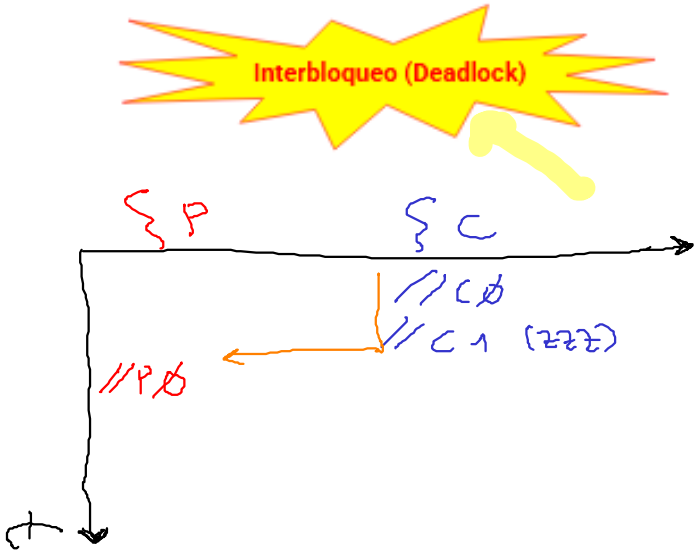
P

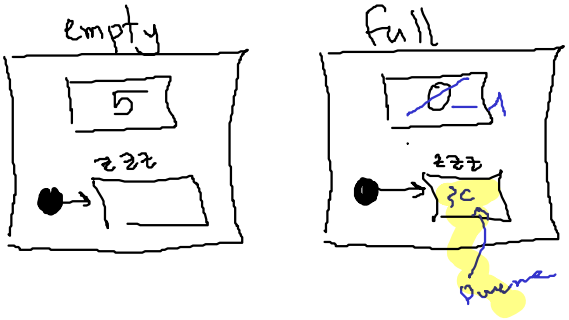
```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&mutex);  // line P0 (NEW LINE)
    sem_wait(&empty);  // Line P1
    put(i);            // Line P2
    sem_post(&full);   // Line P3
    sem_post(&mutex);  // line P4 (NEW LINE)
  }
}
```

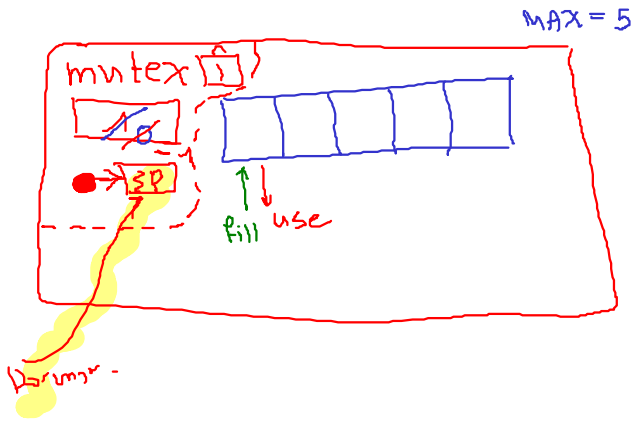
C

```
void *consumer(void *arg) {
  int tmp = 0;
  while (tmp != -1) {
    sem_wait(&mutex); // line c0 (NEW LINE)
    sem_wait(&full);  // Line C1
    tmp = get();      // Line C2
    sem_post(&empty); // Line C3
    sem_post(&mutex); // line c4 (NEW LINE)
    printf("%d\n", tmp);
  }
}
```

Hilos		Semáforos			
P	C	empty	full	mutex	Comentarios
---	---	MAX	0	1	Inicialización
	C0	MAX	0	0	C adquiere el mutex: C0 -> wait(mutex)
	C1	MAX - 1	-1	0	C se bloquea: C1 -> wait(empty)
		MAX - 1	0	0	Interrupción: C → P
P0		MAX - 1	0	-1	P se bloquea: P0 -> wait(mutex)







Solucion

Caso 2: Buffer circular

Problema: **Deadlock**

Solución: **Reducir el ámbito del lock** (reubicar la sección encargada de la exclusión mutua)

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&mutex);  // line P0 (NEW LINE)
    sem_wait(&empty);  // Line P1
    put(i);            // Line P2
    sem_post(&full);   // Line P3
    sem_post(&mutex);  // line P4 (NEW LINE)
  }
}
```

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&empty);  // Line P1
    sem_wait(&mutex);  // line P2 (MUTEX HERE)
    put(i);            // Line P3
    sem_post(&mutex);  // line P4 (MUTEX HERE)
    sem_post(&full);   // Line P4

  }
}
```

```
void *consumer(void *arg) {
  int tmp = 0;
  while (tmp != -1) {
    sem_wait(&mutex);  // line C0 (NEW LINE)
    sem_wait(&full);   // Line C1
    tmp = get();       // Line C2
    sem_post(&empty);  // Line C3
    sem_post(&mutex);  // line C4 (NEW LINE)
    printf("%d\n", tmp);
  }
}
```

```
void *consumer(void *arg) {
  int tmp = 0;
  while (tmp != -1) {
    sem_wait(&full);   // Line C1
    sem_wait(&mutex);  // line C2 (MUTEX HERE)
    tmp = get();       // Line C3
    sem_post(&mutex);  // line C4 (MUTEX HERE)
    sem_post(&empty);  // Line C5
    printf("%d\n", tmp);
  }
}
```

Implementación 1 - Caso 2

Análisis: Reubicación del mutex

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&empty);  // Line P1
    sem_wait(&mutex);  // line P2
    put(i);            // Line P3
    sem_post(&mutex);  // line P4
    sem_post(&full);   // Line P5

  }
}
```

P

```
void *consumer(void *arg) {
  int tmp = 0;
  while (tmp != -1) {
    sem_wait(&full);   // Line C1
    sem_wait(&mutex);  // line C2
    tmp = get();       // Line C3
    sem_post(&mutex);  // line C4
    sem_post(&empty);  // Line C5
    printf("%d\n", tmp);
  }
}
```

C

Hilos		Semáforos			
P	C	empty	full	mutex	Comentarios
---	---	MAX	0	1	Inicialización
	C1	MAX	-1	1	C se bloquea: C1 -> wait(full)
		MAX	-1	1	Interrupción: C → P
P1		MAX - 1	-1	1	P1 -> wait(empty)
P2		MAX - 1	-1	0	P adquiere el mutex: P2 -> wait(mutex)
P3		MAX - 1	-1	0	P entra en la región crítica
P4		MAX - 1	-1	1	P libera el mutex: P4 -> post(mutex)
P5		MAX - 1	0	1	P4 -> post(full) | C se despierta
		MAX - 1	0	1	Interrupción: P → C
	C2	MAX - 1	0	0	C adquiere el mutex: C2 -> wait(mutex)
	C3	MAX - 1	0	0	C entra en la región crítica
	C4	MAX - 1	0	1	P libera el mutex: C4 -> post(mutex)
	C5	MAX	0	1	C5 -> post(empty)

Implementación definitiva

```
sem_t empty;
sem_t full;
sem_t mutex;

int loops = 50; // Variable global

int main(int argc, char *argv[]) {
  // ...
  sem_init(&empty, 0, MAX); // MAX are empty
  sem_init(&full, 0, 0);    // 0 are full
  sem_init(&mutex, 0, 1);   // mutex = 1
  // ...
  pthread_t C, P;
  pthread_create(P, NULL, producer, NULL);
  pthread_create(C, NULL, consumer, NULL);
  // ...
  return 0;
}
```

3

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&empty);  // Line P1
    sem_wait(&mutex);  // line P2
    put(i);            // Line P3
    sem_post(&mutex);  // line P4
    sem_post(&full);   // Line P5

  }
}
```

RC

```
void *consumer(void *arg) {
  int tmp = 0;
  while (tmp != -1) {
    sem_wait(&full);   // Line C1
    sem_wait(&mutex);  // line C2
    tmp = get();       // Line C3
    sem_post(&mutex);  // line C4
    sem_post(&empty);  // Line C5
    printf("%d\n", tmp);
  }
}
```