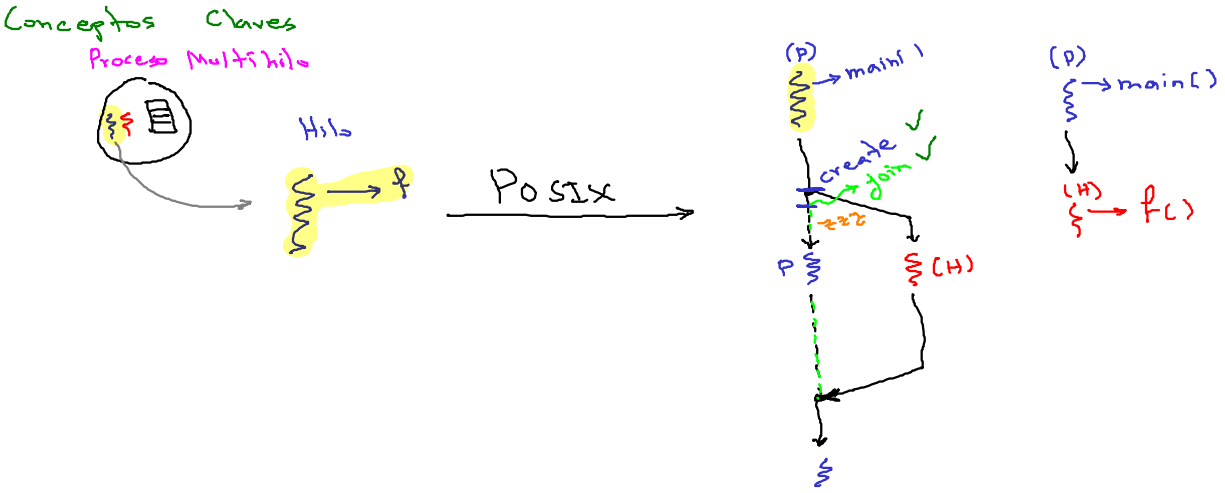
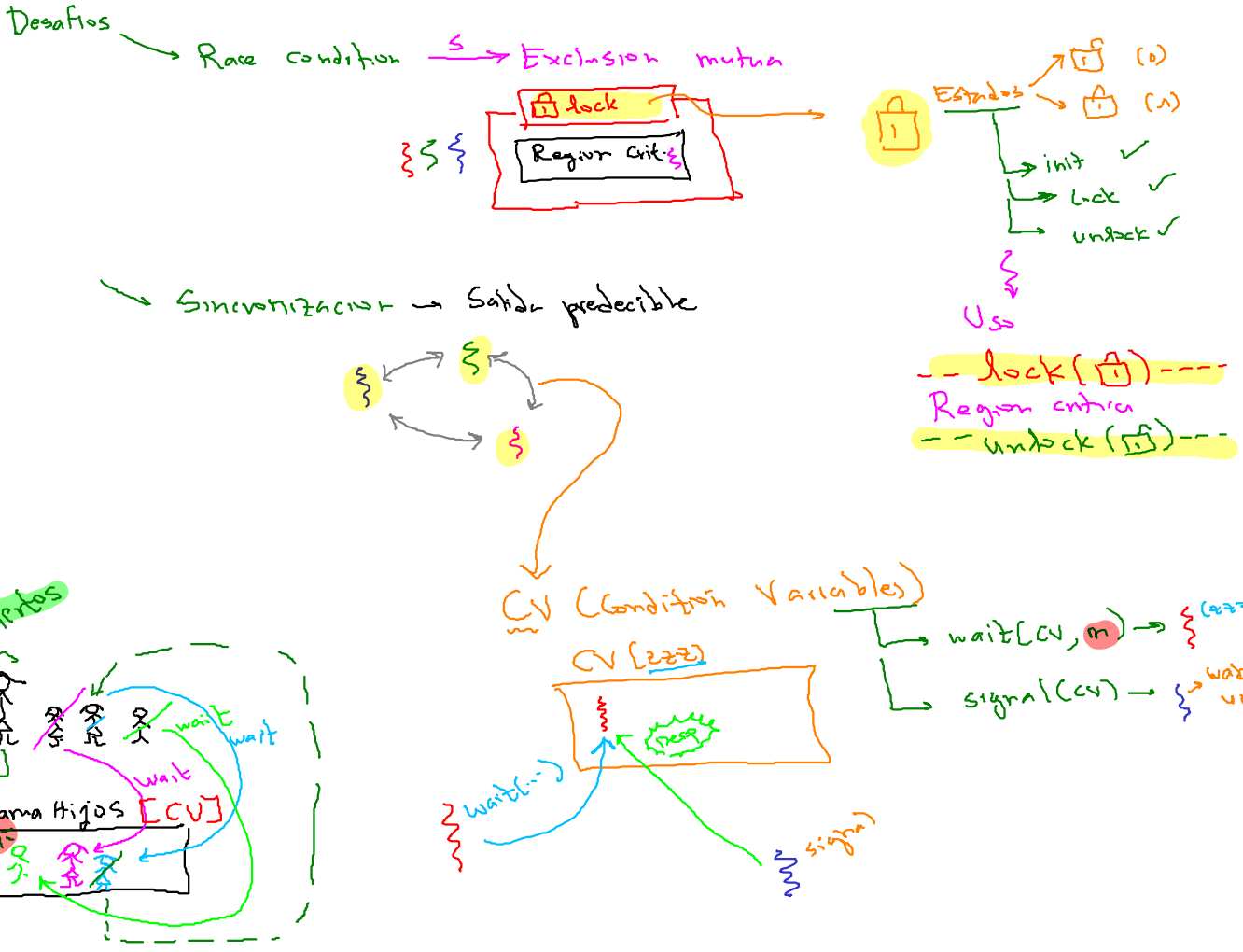
13/11/2025 – Sistemas Operativos (Ude@)

1. Repaso





2. Sincronización

Implementación del join - Caso 3: Los problemas de las 2 implementaciones anteriores ya es solucionado

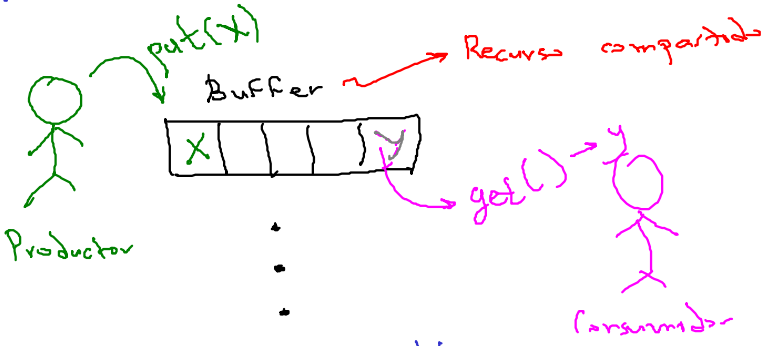

```c
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
void *child(void *arg) {
  printf("child\n");
  thr_exit();
  return NULL;
}
int main(int argc, char *argv[]) {
  printf("parent: begin\n");
  pthread_t p;
  pthread_create(&p, NULL, child, NULL);
  thr_join();
  printf("parent: end\n");
  return 0;
}
```

```c
void thr_exit() {
  pthread_mutex_lock(&m);
  done = 1;
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}
```

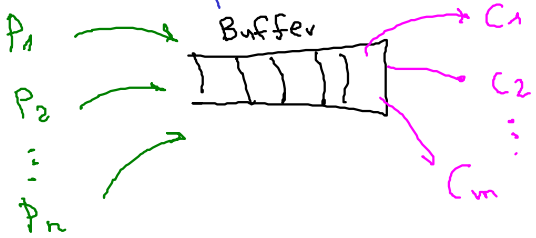
```c
void thr_join() {
  pthread_mutex_lock(&m);
  while (done == 0)
    pthread_cond_wait(&c, &m);
  pthread_mutex_unlock(&m);
}
```

Parent: Begin ... (zzz)
Child
Parent: End

3. Problema del Productor consumidor.

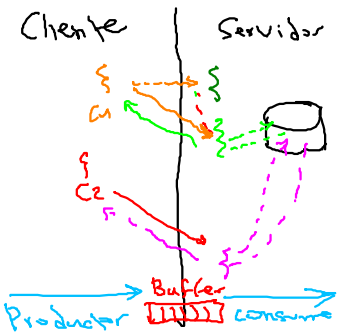


Generalización del problema



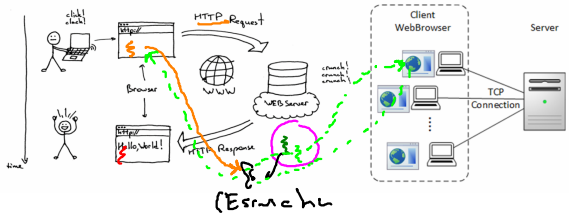
4. Casos de uso:

1. Supermercado ✓
2. Negocio de comidas Rapidas ✓
3. Servidor Multihilos ✓



Servidor multi-hilo

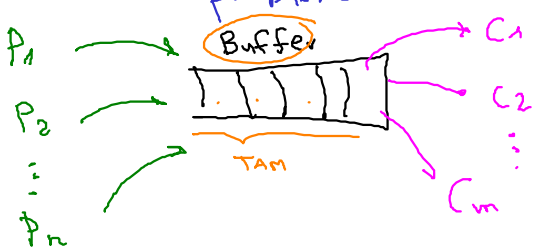
- Un **productor** ingresa **`http request`** en una **cola** de trabajo.
- El **consumidor** retira requerimientos de la **cola** y los procesa.



4. Consola de linea de comandos ✓

```
grep foo file.txt | wc l
```

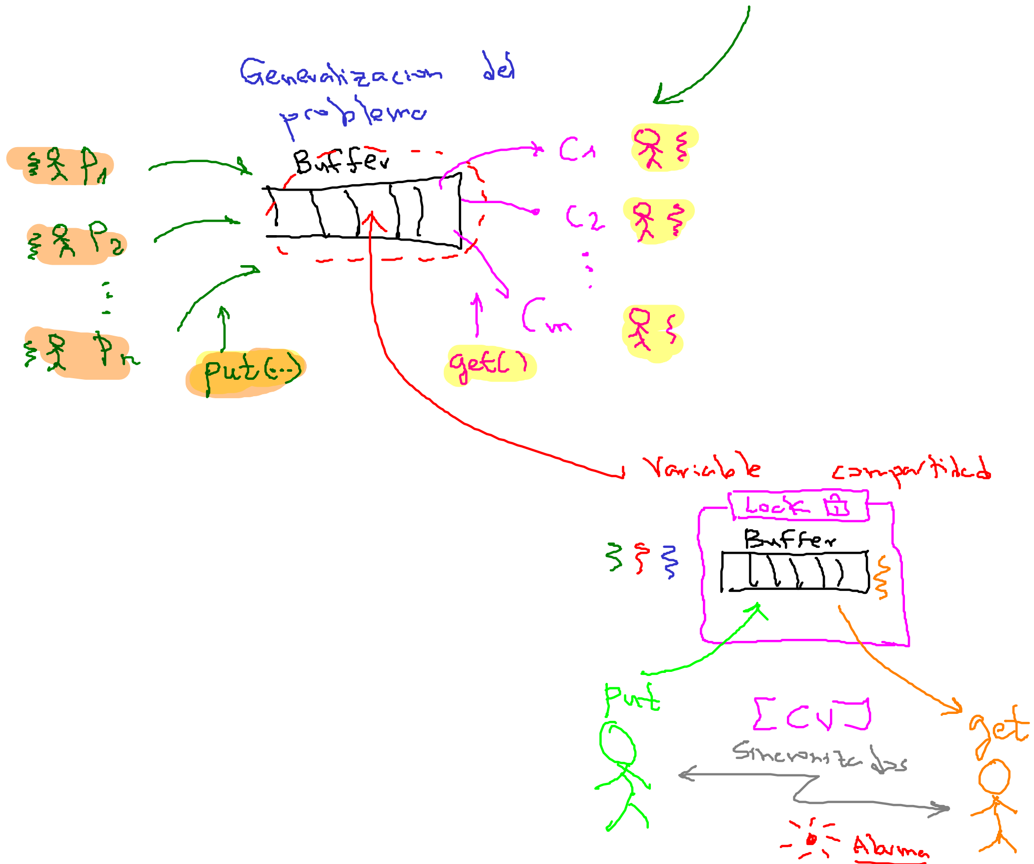
Generalización del problema



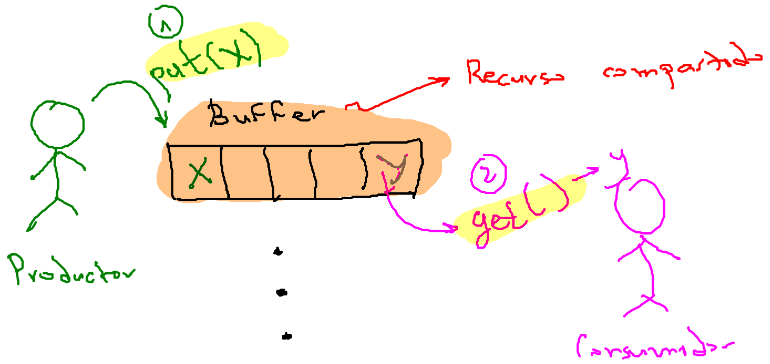
PROBLEMA DE PRODUCTOR - CONSUMIDOR CON BUFFER LIMITADO

5. Como llevar esto a codigo

[Tools: ① Hilo ② Lock (Excl. Mutua) ③ CV (Condition Variable)]

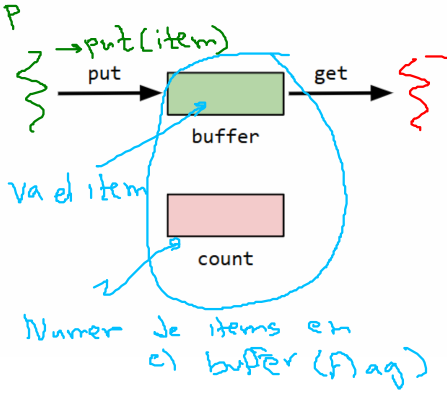


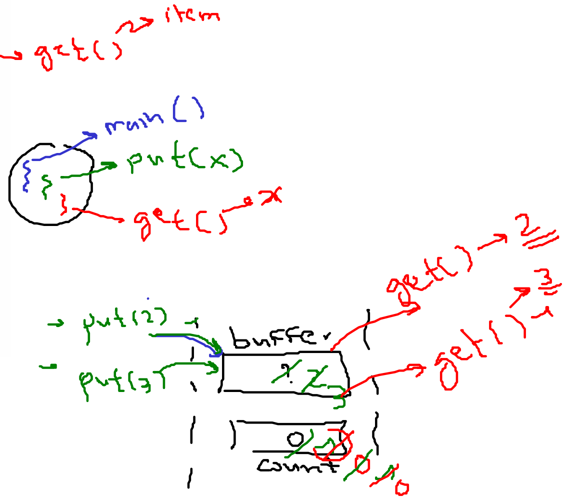
6. Implementación:

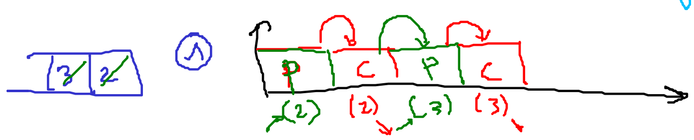


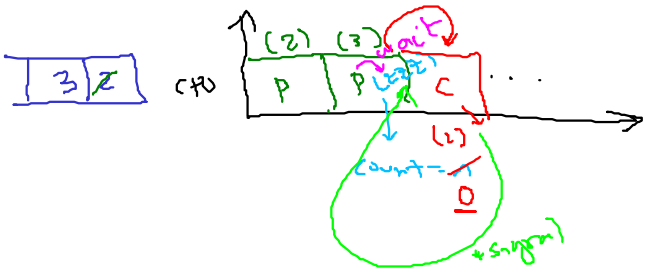
* Secuencia (1P – 1C)

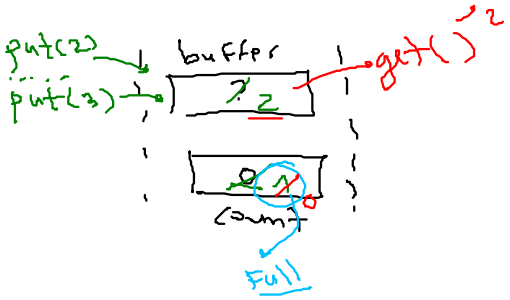

```
int buffer;
int count = 0; // initially, empty

void put(int value) {
  assert(count == 0);
  count = 1;
  buffer = value;
}

int get() {
  assert(count == 1);
  count = 0;
  return buffer;
}
```



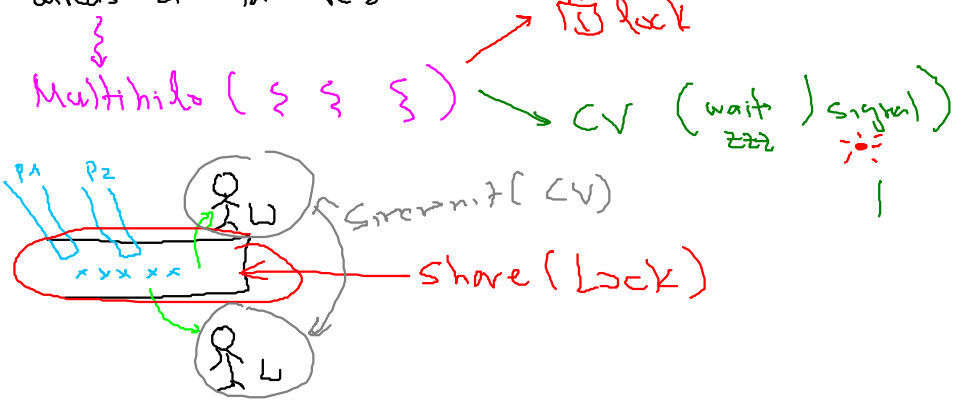


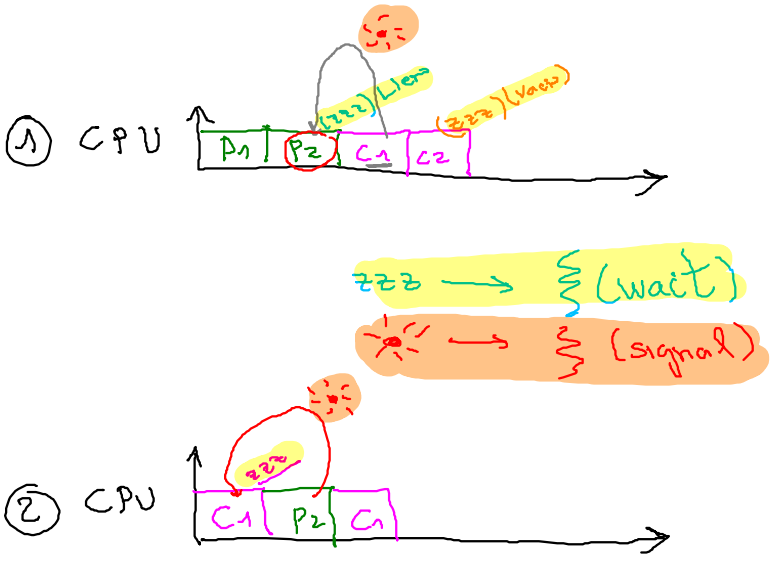






* Realidad → Varias tareas a la vez



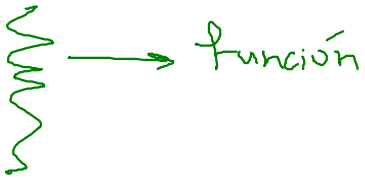


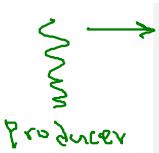
zzz → (wait)

→ (signal)

Al usar hilos

① Hilos ≠ Tarea



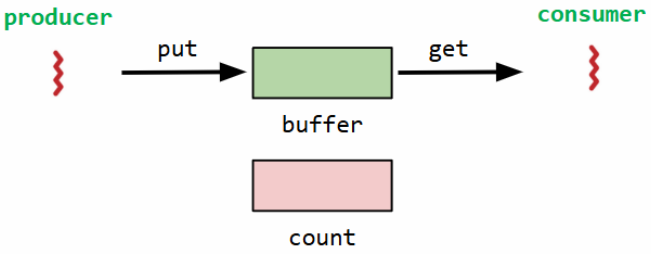



```c
void *producer(void *arg) {
  int i;
  int loops = (int) arg;
  for (i = 0; i < loops; i++) {
    put(i);
  }
}
```


```c
void *consumer(void *arg) {
  while (1) {
    int tmp = get();
    printf("%d\n", tmp);
  }
}
```

producer Ingresa datos enteros en el buffer compartido (uno por cada ciclo).

consumer Retira datos del buffer compartido.



¿Como implementamos el problema del productor-consumidor para que funcione bien con varios hilos

↓ 3 Formas

- **Productor/consumidor: Unica CV y sentencia if**
- Productor/consumidor: Unica CV y sentencia while
- Productor/consumidor – Single buffer: Usar 2 variables de condición y un while

Productor/consumidor: Unica CV y sentencia if

Estrategia general:

Use variables de condición (CV) para hacer que los consumidores esperen cuando no haya nada que consumir (buffer vacío) y los productores esperen cuando el buffer se encuentre lleno.

```c
int loops; // must initialize somewhere...
cond_t cond;
mutex_t mutex;
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);            // p1
    if (count == 1)                        // p2
      pthread_cond_wait(&cond, &mutex);    // p3
    put(i);                                // p4
    pthread_cond_signal(&cond);            // p5
    pthread_mutex_unlock(&mutex);          // p6
  }
}
```

```c
void *consumer(void *arg) {
  int i;
  while (1) {
    pthread_mutex_lock(&mutex);            // c1
    if (count == 0)                        // c2
      pthread_cond_wait(&cond, &mutex);    // c3
    tmp = get();                           // c4
    pthread_cond_signal(&cond);            // c5
    pthread_mutex_unlock(&mutex);          // c6
    printf("%d\n", tmp);
  }
}
```

(No es el que vamos a explicar)

- Se usa una única variable de condición (**cond**) y un lock (**mutex**) asociado.

Lógica de señalización entre el productor y el consumidor:

```c
int loops; // must initialize somewhere...
cond_t cond;
mutex_t mutex;
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);            // p1
    if (count == 1)                        // p2
      pthread_cond_wait(&cond, &mutex);    // p3
    put(i);                                // p4
    pthread_cond_signal(&cond);            // p5
    pthread_mutex_unlock(&mutex);          // p6
  }
}
```

```c
void *consumer(void *arg) {
  int i;
  while (1) {
    pthread_mutex_lock(&mutex);            // c1
    if (count == 0)                        // c2
      pthread_cond_wait(&cond, &mutex);    // c3
    tmp = get();                           // c4
    pthread_cond_signal(&cond);            // c5
    pthread_mutex_unlock(&mutex);          // c6
    printf("%d\n", tmp);
  }
}
```

producer `p1 - p3`:
El productor espera a que el ***buffer*** esté vacío.

consumer `c1 - c3`:
El Consumidor espera a que el ***buffer*** esté lleno.

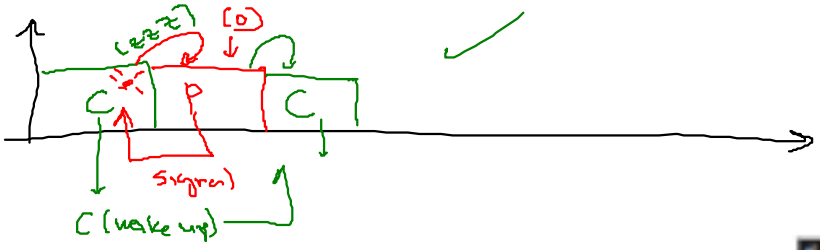
Análisis de la implementación

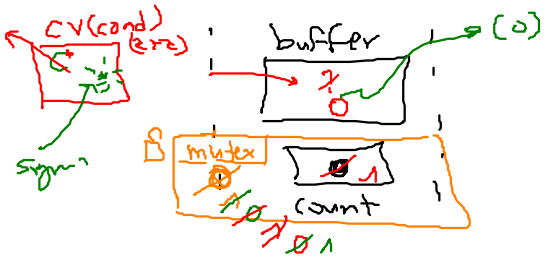
Caso	Escenario	Conclusiones
1	Un solo productor y un solo consumidor	El código funciona.
2	Un solo productor y varios consumidores	La solución tiene 2 problemas críticos (En breve veremos).

Caso 1: 1P - 1C

```c
int loops; // must initialize somewhere...
cond_t cond;
mutex_t mutex;
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);            // p1
    if (count == 1)                        // p2
      pthread_cond_wait(&cond, &mutex);    // p3
    put(i);                                // p4
    pthread_cond_signal(&cond);            // p5
    pthread_mutex_unlock(&mutex);          // p6
  }
}
```

```c
void *consumer(void *arg) {
  int i;
  while (1) {
    pthread_mutex_lock(&mutex);            // c1
    if (count == 0)                        // c2
      pthread_cond_wait(&cond, &mutex);    // c3
    tmp = get();                           // c4
    pthread_cond_signal(&cond);            // c5
    pthread_mutex_unlock(&mutex);          // c6
    printf("%d\n", tmp);
  }
}
```





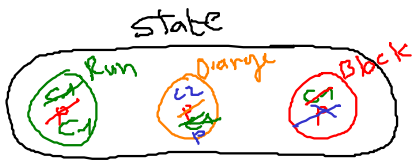

```
$ ./single_buffer-if1.out 10 1
0
1
2
3
4
5
6
7
8
9
^C
```

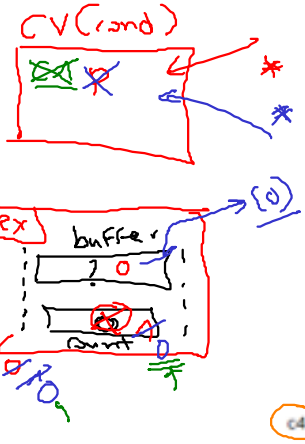
```
$ ./single_buffer-if1.out 20 1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
^C
```

Caso 2: 1P – 2C

```c
int loops; // must initialize somewhere...
cond_t cond;
mutex_t mutex;
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);            // p1
    if (count == 1)                        // p2
      pthread_cond_wait(&cond, &mutex);    // p3
    put(i);                                // p4
    pthread_cond_signal(&cond);            // p5
    pthread_mutex_unlock(&mutex);          // p6
  }
}
```

```c
void *consumer(void *arg) {
  int i;
  while (1) {
    pthread_mutex_lock(&mutex);            // c1
    if (count == 0)                        // c2
      pthread_cond_wait(&cond, &mutex);    // c3
    int tmp = get();                       // c4
    pthread_cond_signal(&cond);            // c5
    pthread_mutex_unlock(&mutex);          // c6
    printf("%d\n", tmp);
  }
}
```





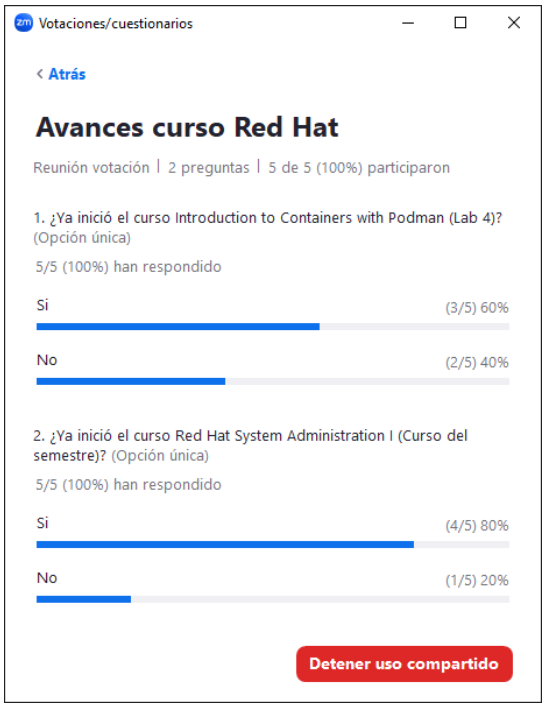
T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

Productor				p1	p2	p4	p5	p6	p1	p2	p3					
Consumidor 1	c1	c2	c3													c4
Consumidor 2												c1	c2	c4	c5	c6

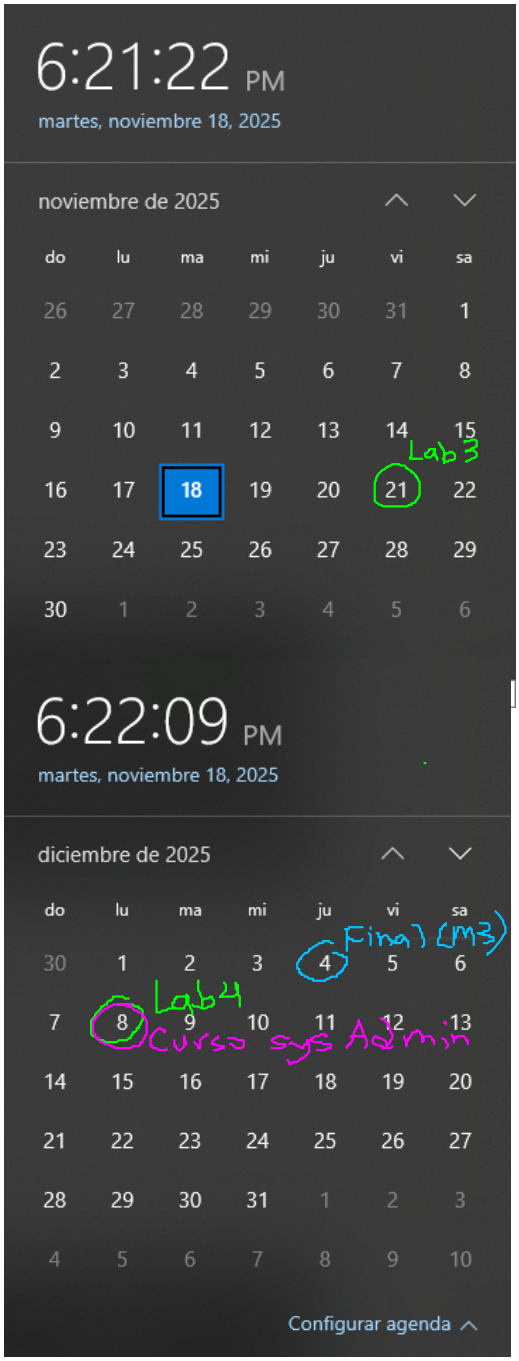
```
$ ./single_buffer-1f1.out 10 2
0
2
1
3
5
6
4
7
0
8
^C
```

```
$ ./single_buffer-1f1.out 30 2
0
2
1
3
4
5
7
8
6
10
9
11
13
14
15
16
12
18
17
19
single_buffer-1f1.out: main_singleBuffer-v1.c:72: get: Assertion `count == 1' failed.
Aborted (core dumped)
```

18/11/2025



Cierre: 08/12/2025



Productor/consumidor: Unica CV y sentencia while

Estrategia general:

Cambie la sentencia **`if`** por **`while`** en el consumidor y en el productor

```
int loops; // must initialize somewhere...
cond_t cond;
mutex_t mutex;
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);          // p1
    while (count == 1)                   // p2
      pthread_cond_wait(&cond, &mutex);  // p3
    put(i);                              // p4
    pthread_cond_signal(&cond);          // p5
    pthread_mutex_unlock(&mutex);        // p6
  }
}
```

```
void *consumer(void *arg) {
  int i;
  while(1) {
    pthread_mutex_lock(&mutex);          // c1
    while (count == 0)                   // c2
      pthread_cond_wait(&cond, &mutex);  // c3
    tmp = get();                         // c4
    pthread_cond_signal(&cond);          // c5
    pthread_mutex_unlock(&mutex);        // c6
    printf("%d\n", tmp);
  }
}
```

- Si el consumidor **`Tc1`** despierta y verifica nuevamente el estado de la variable.
  - Si el buffer está vacío, el consumidor vuelve a dormir.

Estrategia general:

- La idea es que el hilo en espera siempre debe volver a verificar la condición (**`cond`**) en un **`while`**, en lugar de hacerlo en un **`if`**.
  - Si no se vuelve a verificar, el hilo que está en espera seguirá pensando que la condición ha cambiado a pesar de que no ha hecho.

```
int loops; // must initialize somewhere...
cond_t cond;
mutex_t mutex;
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);          // p1
    while (count == 1)                   // p2
      pthread_cond_wait(&cond, &mutex);  // p3
    put(i);                              // p4
    pthread_cond_signal(&cond);          // p5
    pthread_mutex_unlock(&mutex);        // p6
  }
}
```

```
void *consumer(void *arg) {
  int i;
  while(1) {
    pthread_mutex_lock(&mutex);          // c1
    while (count == 0)                   // c2
      pthread_cond_wait(&cond, &mutex);  // c3
    tmp = get();                         // c4
    pthread_cond_signal(&cond);          // c5
    pthread_mutex_unlock(&mutex);        // c6
    printf("%d\n", tmp);
  }
}
```

Estrategia general:

- **Regla de oro**: Con variables de condición (**CV**), se debe **usar siempre ciclos while**.
  - Todavía persiste el **problema 2**

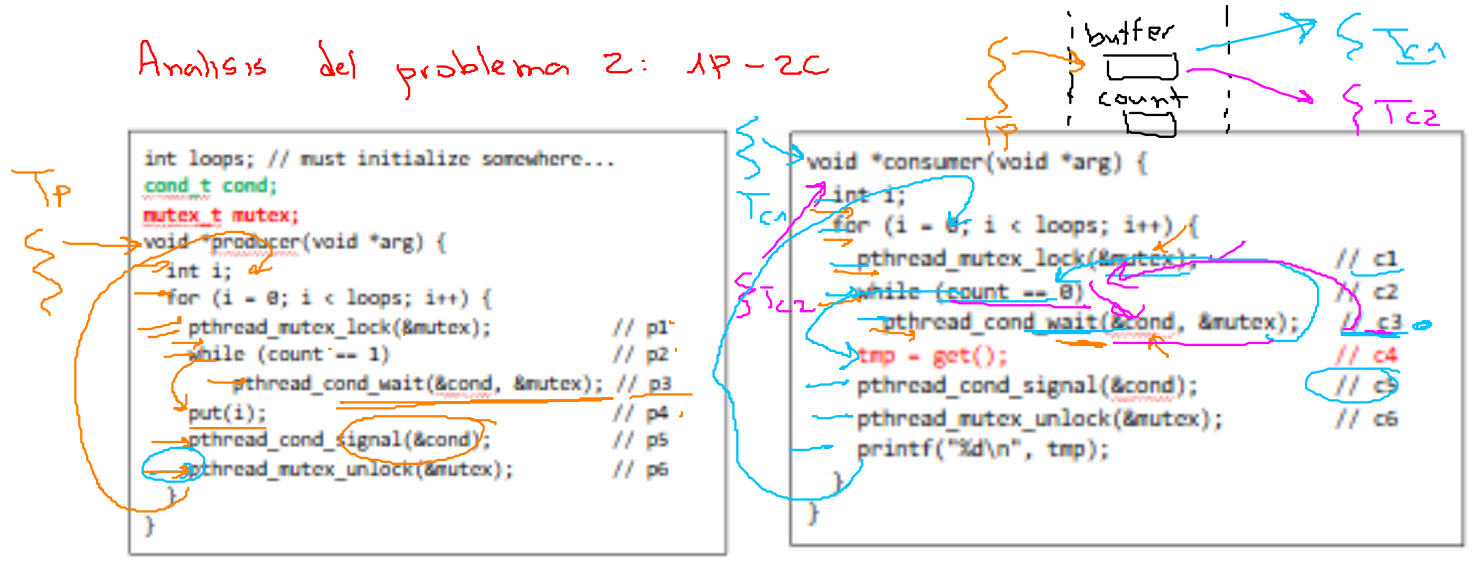
```
int loops; // must initialize somewhere...
cond_t cond;
mutex_t mutex;
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);          // p1
    while (count == 1)                   // p2
      pthread_cond_wait(&cond, &mutex);  // p3
    put(i);                              // p4
    pthread_cond_signal(&cond);          // p5
    pthread_mutex_unlock(&mutex);        // p6
  }
}
```

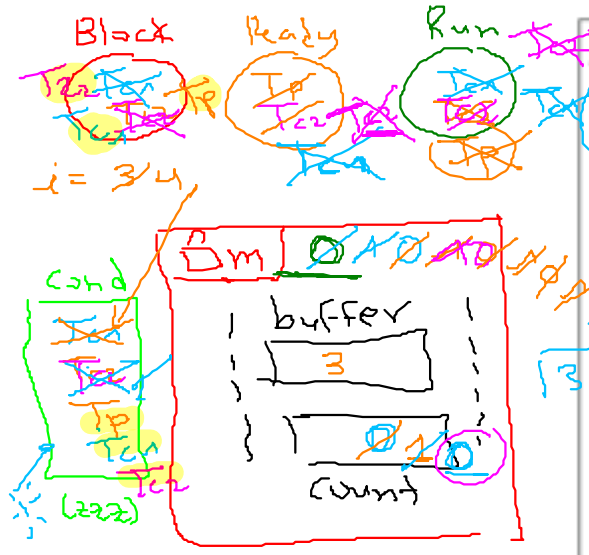
```
void *consumer(void *arg) {
  int i;
  while(1) {
    pthread_mutex_lock(&mutex);          // c1
    while (count == 0)                   // c2
      pthread_cond_wait(&cond, &mutex);  // c3
    tmp = get();                         // c4
    pthread_cond_signal(&cond);          // c5
    pthread_mutex_unlock(&mutex);        // c6
    printf("%d\n", tmp);
  }
}
```

Analisis del problema 2: 1P – 2C

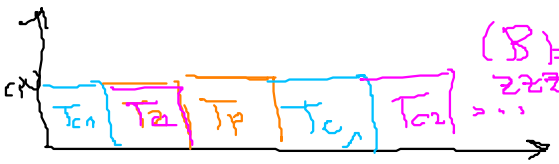
```
int loops; // must initialize somewhere...
cond_t cond;
mutex_t mutex;
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);            // p1
    while (count == 1)                     // p2
      pthread_cond_wait(&cond, &mutex);    // p3
    put(i);                                // p4
    pthread_cond_signal(&cond);            // p5
    pthread_mutex_unlock(&mutex);          // p6
  }
}
```

```
void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);            // c1
    while (count == 0)                     // c2
      pthread_cond_wait(&cond, &mutex);    // c3
    tmp = get();                           // c4
    pthread_cond_signal(&cond);            // c5
    pthread_mutex_unlock(&mutex);          // c6
    printf("%d\n", tmp);
  }
}
```





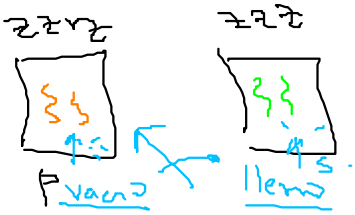
T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T_{c2}



T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
...	...	...	...	...	...	...	*(cont.)*
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep ...

Conclusión:

1. Señalizar es necesario. ✓
2. La señalización llevada a cabo debe ser **más específica**.



Productor							p1	p2	p4	p5	p6	p1	p2	p3							
Consumidor 1	c1	c2	c3												c2	c4	c5	c6	c1	c2	c3
Consumidor 2				c1	c2	c3														c2	c3

Escenario	Conclusiones
Un solo productor y varios consumidores	El código se puede bloquear

```
$ ./single_buffer-while1.out 1000 2
```

```
80
81
82
84
83
86
87
88
85
90
89
92
91
93
```

Productor/consumidor – Single buffer: Usar 2 variables de condición y un while

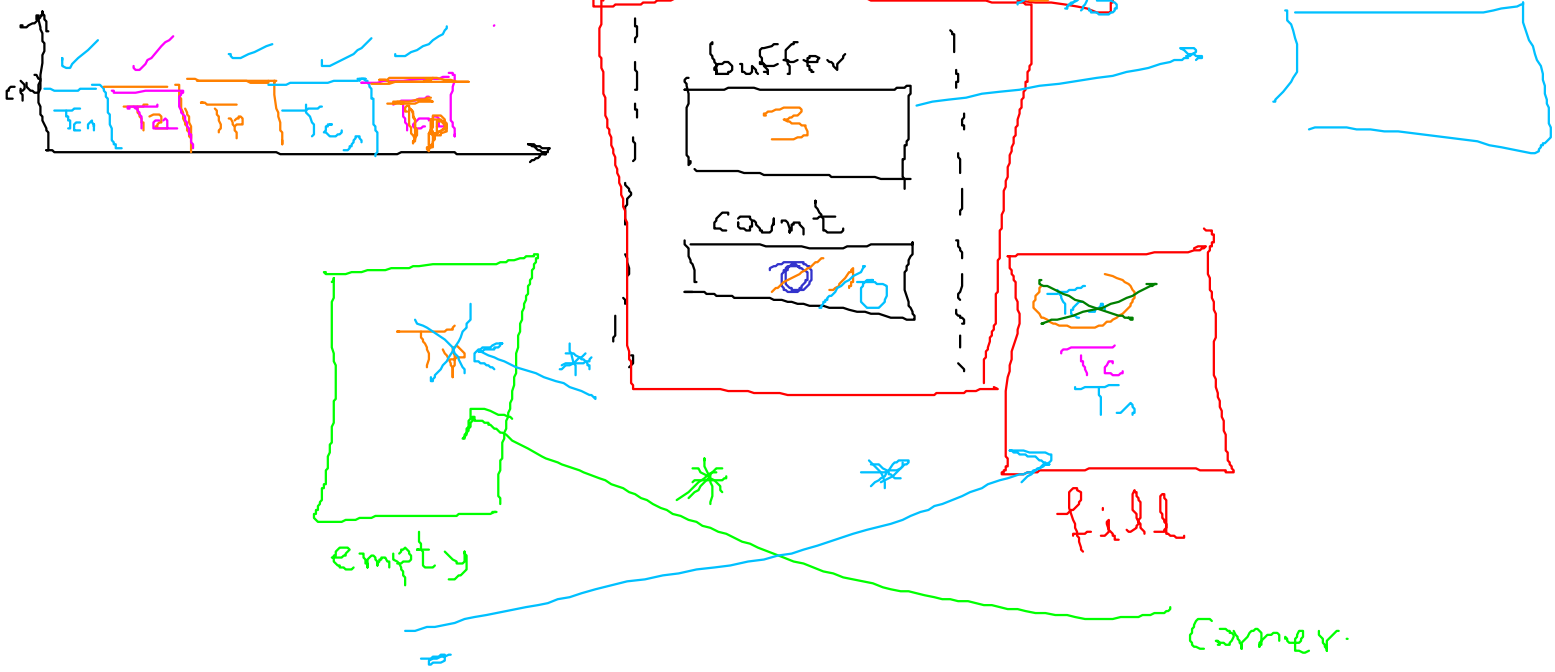
2 camas

Solución: Use dos variables de condición (empty y fill)

- **Productor**: va a dormir usando la variable **empty**, y señaliza **fill**
- **Consumidor**: va a dormir usando la variable **fill**, y señaliza **empty**

```
cond_t empty, fill;
mutex_t mutex;
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
    while (count == 1)
      pthread_cond_wait(&empty, &mutex);
    put(i);
    pthread_cond_signal(&fill);
    pthread_mutex_unlock(&mutex);
  }
}
```

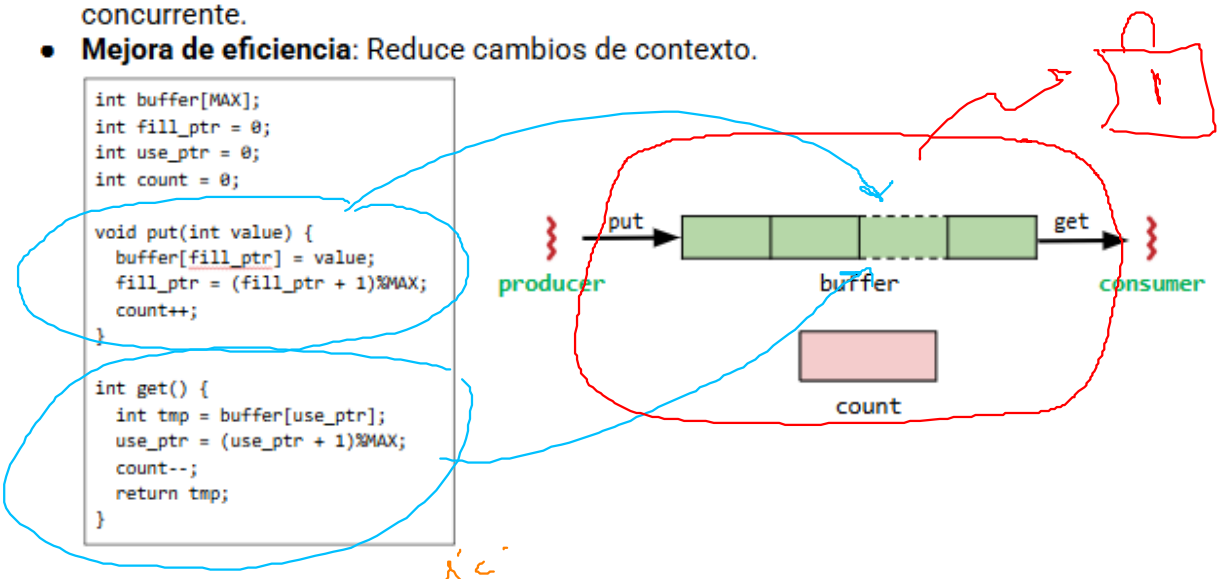
```
void *consumer(void *arg) {
  int i;
  while(1) {
    pthread_mutex_lock(&mutex);
    while (count == 0)
      pthread_cond_wait(&fill, &mutex);
    int tmp = get();
    pthread_cond_signal(&empty);
    pthread_mutex_unlock(&mutex);
    printf("%d\n", tmp);
  }
}
```

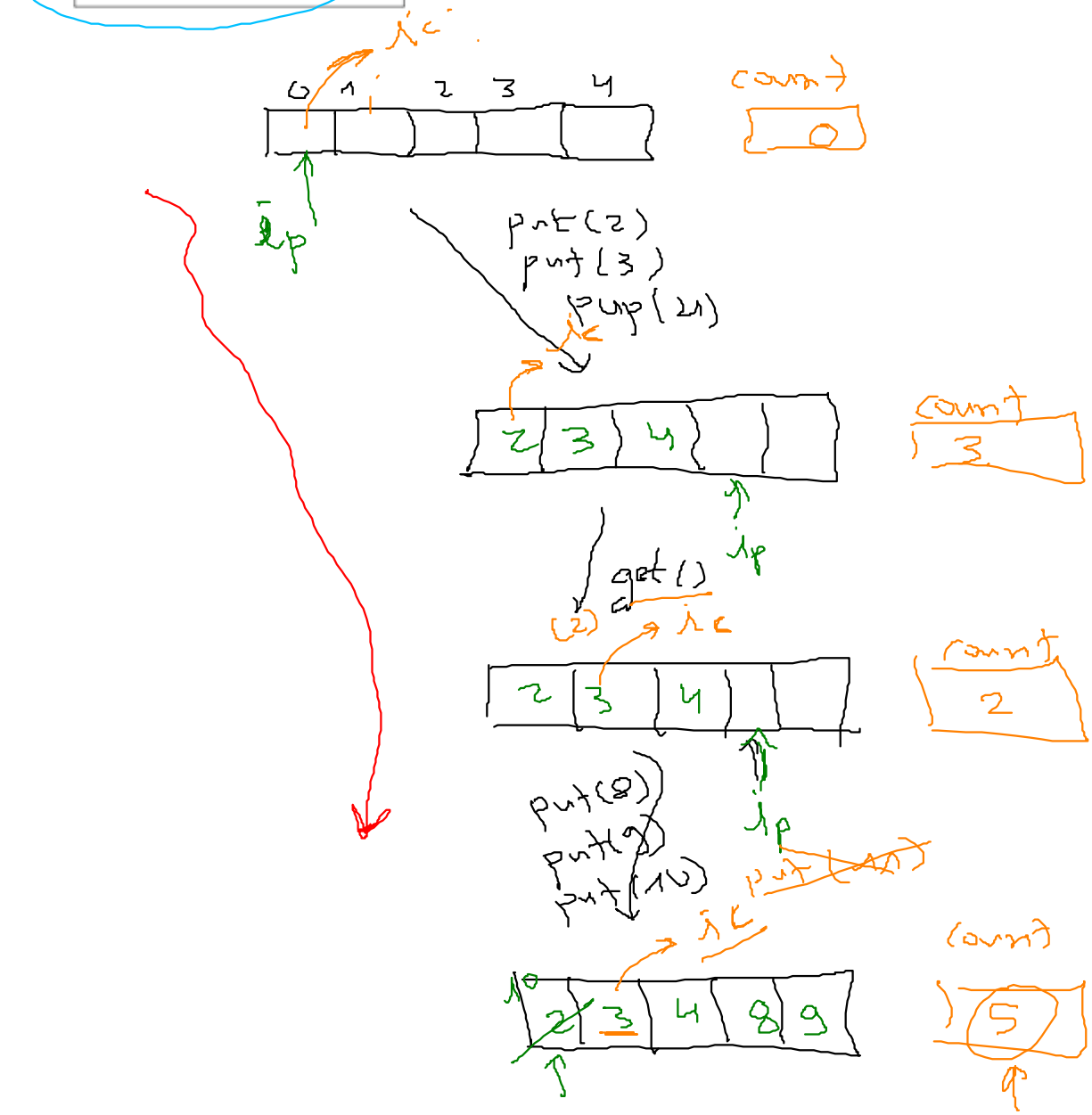


Solución: Agregar mas slots en el buffer

- **Mejora de concurrencia**: Permite que se produzca y se consuma de manera concurrente.
- **Mejora de eficiencia**: Reduce cambios de contexto.

```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr = 0;
int count = 0;

void put(int value) {
  buffer[fill_ptr] = value;
  fill_ptr = (fill_ptr + 1)%MAX;
  count++;
}

int get() {
  int tmp = buffer[use_ptr];
  use_ptr = (use_ptr + 1)%MAX;
  count--;
  return tmp;
}
```





Solución: Agregar mas slots en el buffer

```
cond_t empty, fill;
mutex_t mutex;
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);            // p1
    while (count == MAX)                   // p2
      pthread_cond_wait(&empty, &mutex);   // p3
    put(i);                                // p4
    pthread_cond_signal(&fill);            // p5
    pthread_mutex_unlock(&mutex);          // p6
  }
}
```

```
void *consumer(void *arg) {
  int i;
  while(1) {
    pthread_mutex_lock(&mutex);            // c1
    while (count == 0)                     // c2
      pthread_cond_wait(&fill, &mutex);    // c3
    int tmp = get();                       // c4
    pthread_cond_signal(&empty);           // c5
    pthread_mutex_unlock(&mutex);          // c6
    printf("%d\n", tmp);
  }
}
```

- Un **productor va a dormir** solo si el buffer está **lleno** (**p2**).
- Un **consumidor va a dormir** sólo si el buffer está **vacío** (**c2**).

Código 1

Aspectos a resaltar:

- A diferencia de los casos anteriores, este código ya tiene en cuenta que el buffer puede tener más de un elemento.
- No maneja marca de terminación de modo que aunque funciona el código se bloquea.

Archivos:

- mythreads.h
- main_buffer-v3.c

Escenario	Conclusiones
Un solo productor y varios consumidores	El código funciona pero se bloquea

```
$ ./bounded-buffer.out 10 1000 4
```

```
989
990
991
992
993
994
995
996
997
998
999
981
978
979
^C
```

Código 2

Aspectos a resaltar:

- A diferencia de los casos anteriores, este código ya tiene en cuenta que el buffer puede tener más de un elemento.
- Corrige el problema del código anterior agregando un -1 como condición de terminación al meter datos.

Archivos:

- mythreads.h
- main_buffer-v4.c

Escenario	Conclusiones
Un solo productor y varios consumidores	El código funciona

```
$ ./bounded-buffer.out 10 1000 4
```

```
993
972
994
995
996
997
998
999
-1
-1
974
-1
975
-1
```

Resumen: Reglas de juego

- Mantenga un **estado** además de las **variables condición** (CV).

count

```
cond_t done;
mutex_t mutex;
```

- Siempre espere/señale (**`wait/signal`**) el lock adquirido.
- Siempre que adquiera un lock, verifique el **estado**.

```
int loops; // must initialize somewhere...
cond_t cond;
mutex_t mutex;
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
     pthread_mutex_lock(&mutex);          // p1
     while (count == 1)                   // p2
     pthread_cond_wait(&cond, &mutex);    // p3
     put(i);                              // p4
     pthread_cond_signal(&cond);          // p5
     pthread_mutex_unlock(&mutex);        // p6
  }
}
```

```
void *consumer(void *arg) {
  int i;
  while(1) {
     pthread_mutex_lock(&mutex);          // c1
     while (count == 0)                   // c2
     pthread_cond_wait(&cond, &mutex);    // c3
     tmp = get();                         // c4
     pthread_cond_signal(&cond);          // c5
     pthread_mutex_unlock(&mutex);        // c6
     printf("%d\n", tmp);
  }
}
```