# SISTEMAS OPERATIVOS

## Ingeniería civil informática

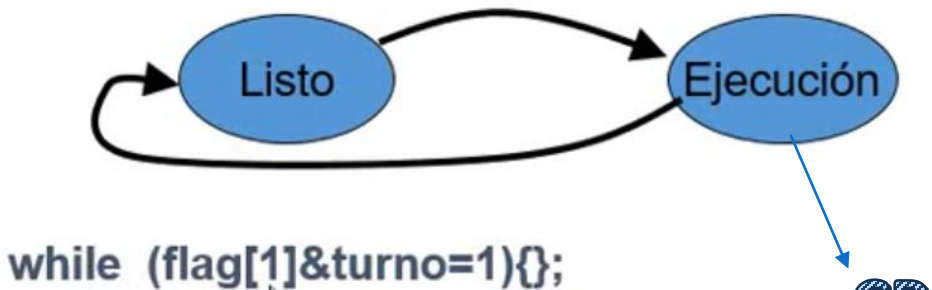
GONZALO CARREÑO

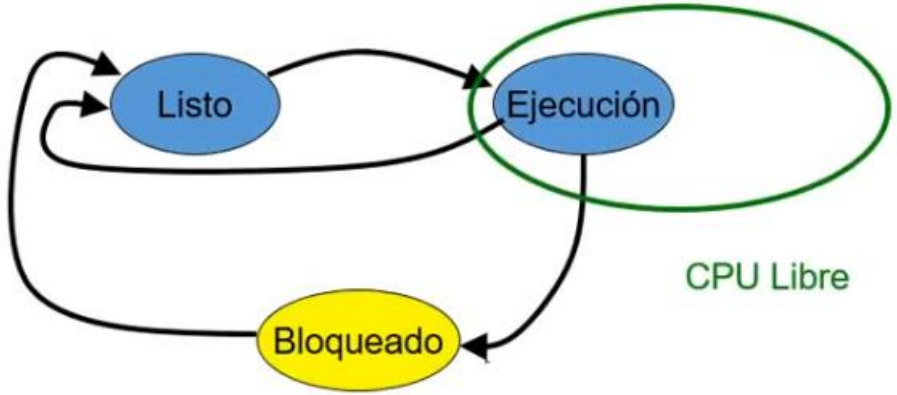
GONZALOCARRENOB@GMAIL.COM

# Desventajas de soluciones

Los procesos que están solicitando entrar en su sección critica están en espera ocupada.

- Consumiendo tiempo del procesador
- Lo mas eficiente es bloquear el proceso



# Soluciones por hardware

1era. Solución: Inhabilitación de interrupciones:

Si no hay interrupciones, no se invoca el planificador a corto plazo por lo que al proceso que esta en ejecución no se le puede quitar el procesador. Con esta acción nos aseguramos que el proceso que se encuentra en la sección critica no la perderá temporalmente el uso del procesador, no se le asignara el procesador a ningún otro proceso.

```
Proceso(Pi)
{
      while(forever)
      {
            inhabilita interrupciones
            sección crítica
            habilita interrupciones
            sección restante
      }
}
```

# Soluciones por hardware

2da solución: Instrucciones maquina especiales.

- El acceso a una posición de memoria excluye otros accesos a la misma posición.
- Los diseñadores han propuesto instrucciones maquina que ejecutan 2 acciones atómicas(indivisibles) en la misma posición de memoria.
- La ejecución de tales instrucciones es también mutuamente exclusiva incluso con varios CPUs.
- Pueden usarse para proveer exclusión mutua pero necesitan complementarse con otros mecanismos para satisfacer los otros 3 requisitos del problema de la sección critica.

# La instrucción test and set

Recibe 2 parámetros que pueden ser registros o direcciones de memoria

```
testandset x,y
```

- Si se ejecuta con x=0, cambia x a 1 y establece y con 1
- Si se ejecuta con x=1, solo establece y con 0

```
                          x   y
mov 0,x                   0
testandset x,y            1   1

testandset x,y                0
```

# La instrucción test and set

```
int b=0;

Proceso Pi:
int r;
repeat
      do{ testandset b,r } while(r==0);
      CS;
      b:=0;
      RS
forever
```

Se establece la variable global b con el valor 0

Si después de hacer testandset con b, si el resultado es 1, puede continuar

Usa la sección critica

Al liberar la sección critica reestablece b con 0 para que otro proceso pueda entrar

# Ejemplo de exclusión mutua con test and set

| P0 | P1 | P2 | b | $r_0$ | $r_1$ | $r_2$ |
|---|---|---|---|---|---|---|
| | | | 0 | | 0 | 0 |
| do{ testandset b,r } while(r==0); | | | 1 | 1 | | |
| | do{ testandset b,r } while(r==0); | | 1 | | 0 | |
| | | do{ testandset b,r } while(r==0); | 1 | | | 0 |
| SC | | | | | | |
| b=0; | | | 0 | | | |
| | do{ testandset b,r } while(r==0); | | 1 | | 1 | |
| | | do{ testandset b,r } while(r==0); | 1 | | | 0 |
| | SC | | 1 | | | |

# La instrucción test and set

¿La exclusión mutua se preserva?

# La instrucción Xchg

La instrucción xchg recibe dos argumento que pueden ser registro o direcciones de memoria, que llamaremos x e y

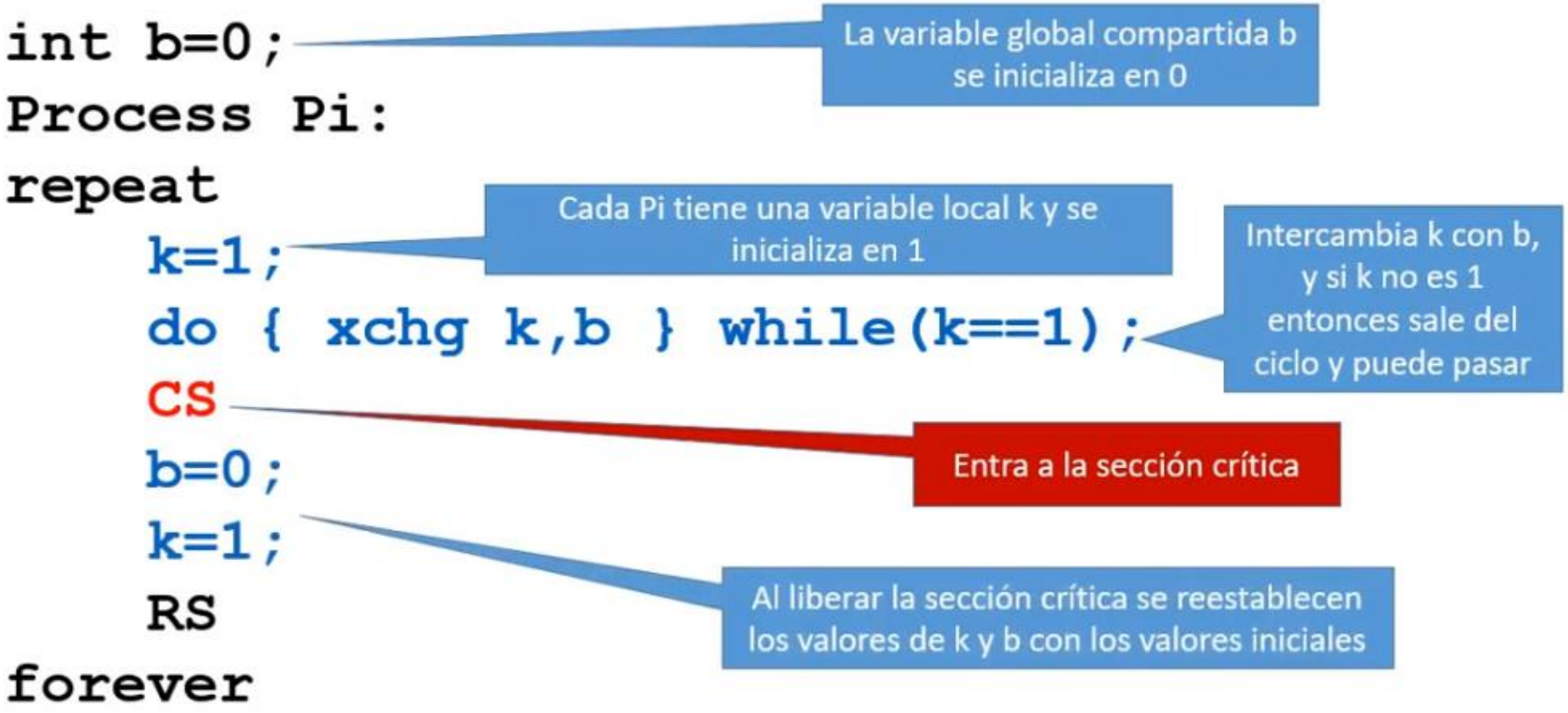
`xchg x,y`

- Intercambia los valores de *x* y *y* de manera atómica

Si lo ejecuto nuevamente los valores se intercambian

| | x | y |
|---|---|---|
| mov 0,x | 0 | |
| mov 1,y | | 1 |
| xchg x,y | 1 | 0 |
| xchg x,y | 0 | 1 |

# Usando xchg para exclusión mutua



# Ejemplo de instrucción xchg

| P0 | P1 | $k_0$ | $k_1$ | b |
|---|---|---|---|---|
| | | 0 | 0 | 0 |
| k=1 | | 1 | 0 | 0 |
| | k=1 | 1 | 1 | 0 |
| do { xchg k,b } while(k==1); | | 0 | 1 | 1 |
| | do { xchg k,b } while(k==1); | 0 | 1 | 1 |
| SC | | 0 | 1 | 1 |
| b=0; k=1 | | 1 | 1 | 0 |
| | do { xchg k,b } while(k==1); | 1 | 0 | 1 |

# Soluciones del sistema

Sol1. Semáforos

Herramienta de sincronización que provee el sistema operativo que no requiere espera ocupada.

Un semáforo S es una variable que, aparte de la inicialización, solo se puede acceder por medio de 2 operaciones atómicas y mutuamente exclusivas:

**Wait(s)**

- P(s), Down(s)

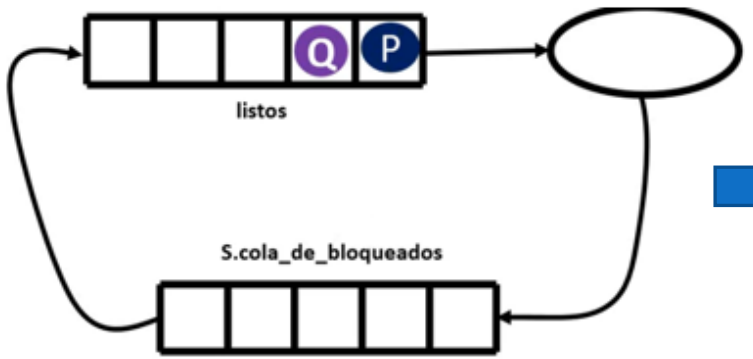
**Signal(S)**

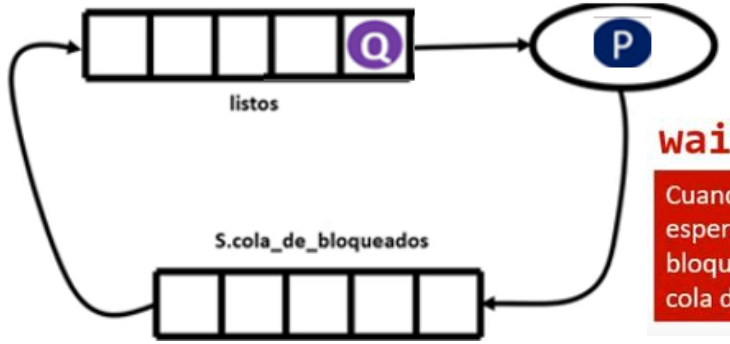
- V(s), Up(s), Post(s) o Release (s)

Para evitar la espera ocupada cuando un proceso tiene que esperar, se pondrá en una cola de procesos bloqueados esperando un evento y de esta manera no usara la CPU

# Semáforos

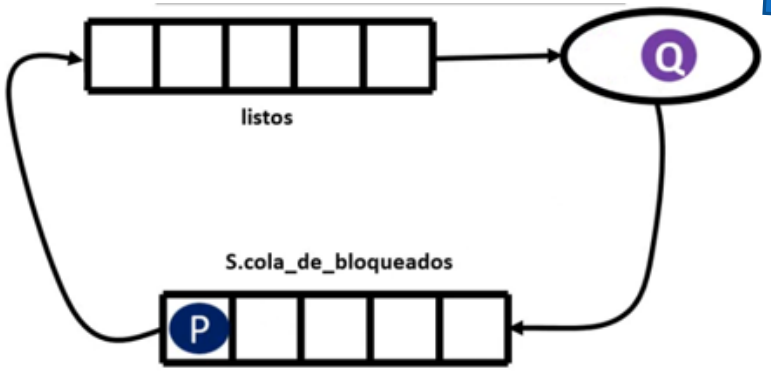
```
struct SEMAPHORE {
    int count;
    queue cola_de_bloqueados;
} S;
```



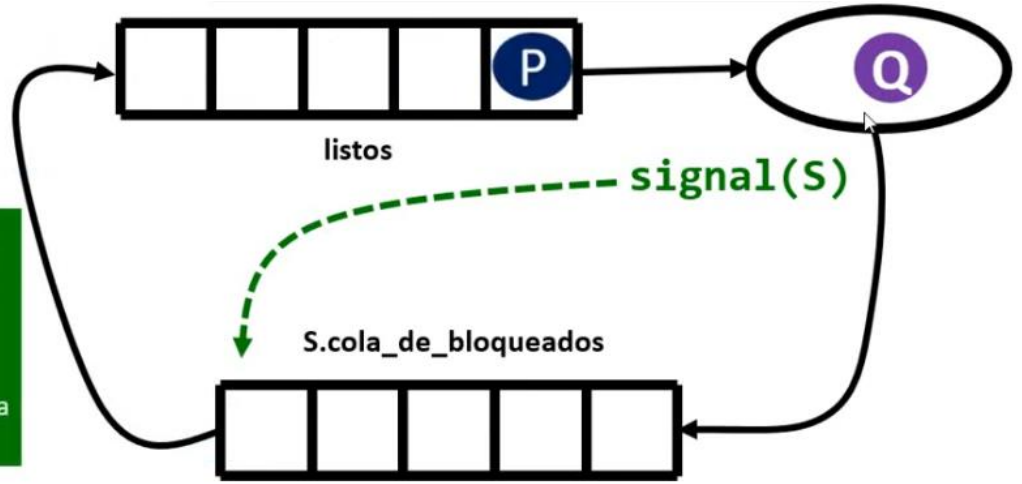




**wait(S)**

Cuando un proceso debe esperar a un semáforo S, se bloquea y se pone en la cola del semáforo S.




La operación signal quita (con la política FIFO) un proceso de la cola y lo pone en la cola de procesos listos y puede pasar a ejecución.



# Semáforos binarios y semáforos generales

## Semáforo binario

- Solo pueden tener dos valores, 0 y 1.
- En Windows se llaman mutex

## Semáforo general o entero

- Pueden tomar muchos valores positivos

# Semáforo binario

```
struct SEMAPHORE {
    int valor; (0,1)
    queue cola_de_bloqueados;
} s;
```

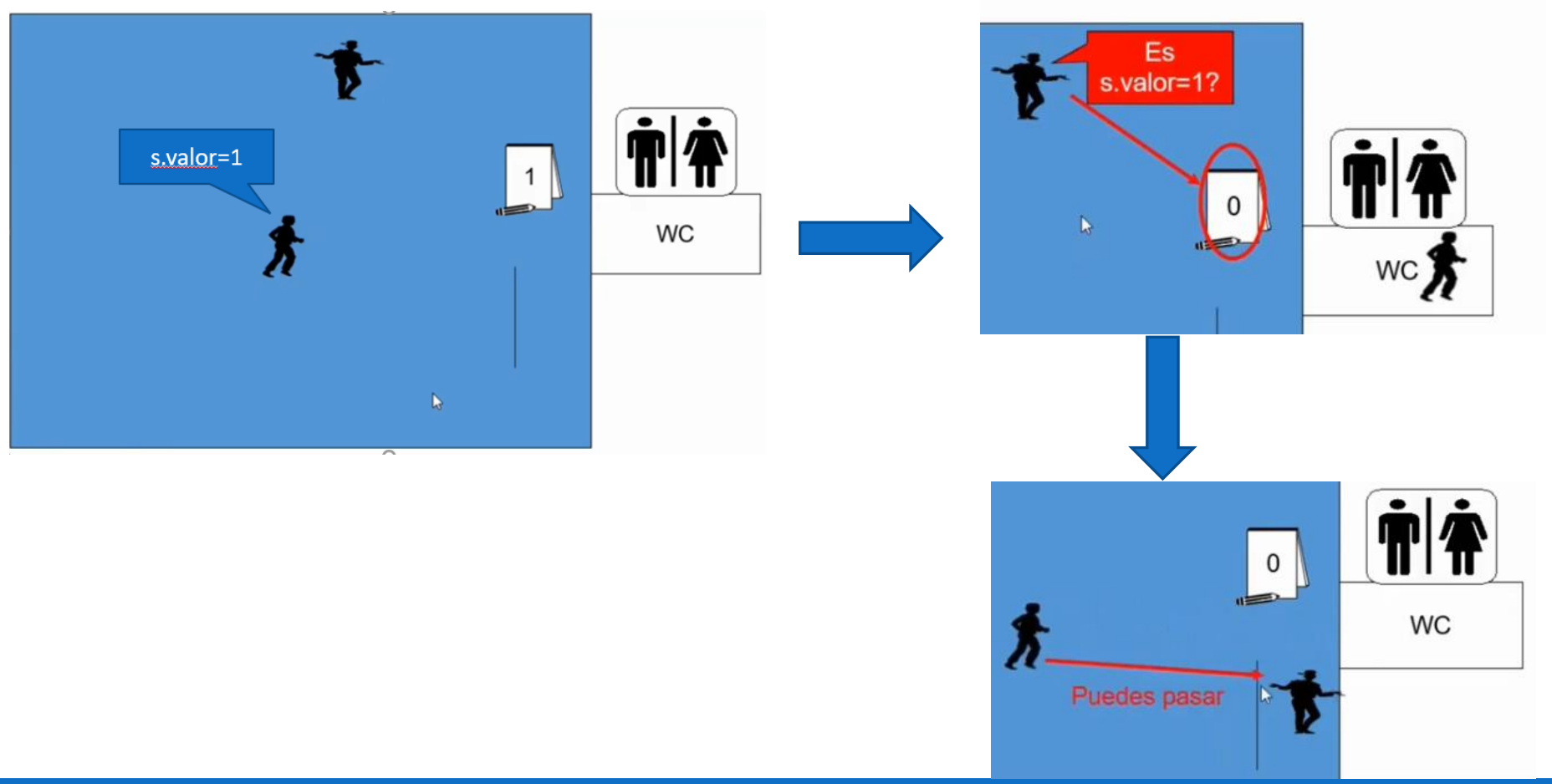
```
WaitB(s):
  if s.valor=1
    s.valor=0
  else {
    poner este proceso en s.cola_de_bloqueados;
    bloquear este proceso
  };
```

Atómica y mutuamente exclusiva

```
SignalB(s):
  If s.cola_de_bloqueados está vacía
    s.valor=1
  else {
    quitar un proceso P de s.cola_de_bloqueados;
    poner el proceso P en la cola de listos
  };
```

Atómica y mutuamente exclusiva

# Ejemplo



# Semáforo general o entero

Este tipo de semáforos pueden tomar muchos valores positivos a diferencia de los semáforos binarios que solo pueden tomar 2 valores.

En una definición de semáforos enteros, los semáforos pueden ser negativos

```
struct SEMAPHORE {
    int contador;
    queue cola_de_bloqueados;
} s;
```

```
Wait(s):
  s.contador--;
  if s.contador<0 then
  {
    poner este proceso en s.cola_de_bloqueados;
    bloquear este proceso
  }
```

Atómica y mutuamente exclusiva

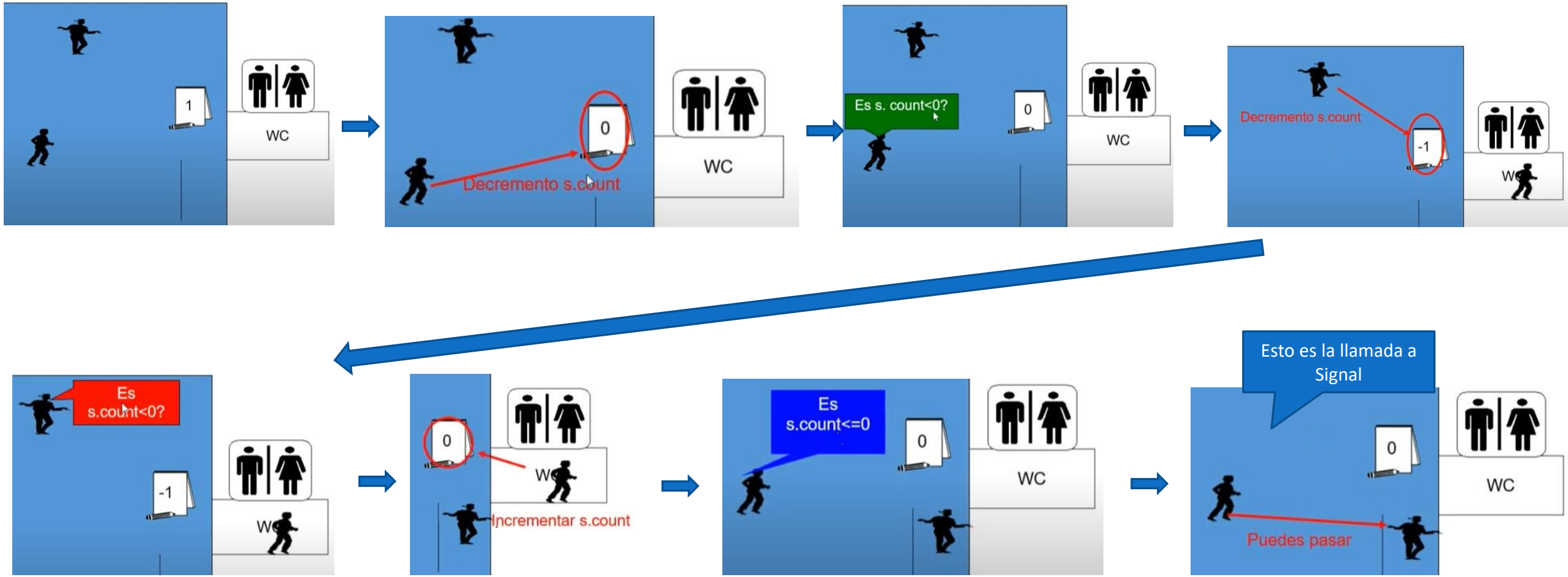
```
Signal(s):
  s.contador++;
  if s.contador <=0
  {
    quitar un proceso P de s.cola_de_bloqueados;
    poner el proceso P en la cola de listos
  }
```

Atómica y mutuamente exclusiva

Ahora que significado le podemos dar a contador, si contador tiene un numero mayor o igual a 0 este significa la cantidad de procesos que podrían ejecutar wait sin que se bloqueen, o lo que es igual, la cantidad de procesos que pueden ejecutar wait sin que se bloquee es igual al contador.

En cambio si contador tiene un numero negativo este contador significa el numero de procesos que están esperando en el semáforo y esto es el valor absoluto del contador, por ejemplo si contador es -4 quiere decir que hay 4 procesos que están bloqueados en el semáforo.

# Ejemplo


1
WC
0
Decremento s.count
WC
Es s. count<0?
0
WC
Decremento s.count
-1
WC
Es
s.count<0?
-1
WC
0
WC
Incrementar s.count
Es
s.count<=0
0
WC
Esto es la llamada a
Signal
0
WC
Puedes pasar

# Definición de primitivas de semáforos para semáforos enteros

```
struct SEMAPHORE {
    unsigned int contador;
    unsigned int bloqueados;
    queue cola_de_bloqueados;
} s;
```

```
Wait(s):
  if s.contador==0 then
  {
    s.bloqueados++;
    poner este proceso en s.cola_de_bloqueados;
    bloquear este proceso;
  }
  else
    s.contador--;
```

```
Signal(s):
 if s.bloqueados==0 then
     s.contador++;
 else
 {
    quitar un proceso P de s.cola_de_bloqueados;
    poner el proceso P en la cola de listos
    s.bloqueados--;
 }
```

El contador es el número de procesos que pueden ejecutar wait sin que se bloqueen y el contador de bloqueados es el número de procesos que van a estar esperando en el semáforo.

Con las soluciones del sistema damos solución al problema de la sincronización de procesos con semáforos.

# Usando semáforos para resolver problemas de selección critica

```
main()
{

      cobegin {
            P(0);P(1);P(2)… P(N);
      }
}
```

```
Semaphore S;

Process P(int i)
{
      while(1)
      {
            wait(S);
            CS
            signal(S);
            RS
      }
}
```

Se necesita:

- Inicializa S.count=1
- Solo 1 proceso se le permite entrar a la sección critica(Exclusión mutua)

# Ejecución

| P(0) | P(1) | P(2) | S.count |
|---|---|---|---|
| | | | 1 |
| wait(S); | | | 0 |
| | wait(S); | | -1 |
| | | wait(S); | -2 |
| CS | | | |
| signal(S); | | | -1 |
| | CS | | |
| | signal(S); | | 0 |
| | | CS | |
| | | signal(S); | 1 |
| RS | | | |
| | RS | | |

| P(0) | P(1) | P(2) | S.count |
|---|---|---|---|
| | | | 2 |
| wait(S); | | | 1 |
| | wait(S); | | 0 |
| | | wait(S); | -1 |
| CS | | | |
| | CS | | |
| signal(S); | | | 0 |
| | | CS | |
| | signal(S); | | 1 |
| | | signal(S); | 2 |
| | RS | | |

# Uso de semáforos para sincronizar procesos

Tenemos los procesos cocinero() y mesero() que inician concurrentemente

- Necesitamos que:
  - `preparar_comida()` en `cocinero()` sea ejecutado antes que `servir_comida()` en `mesero()`

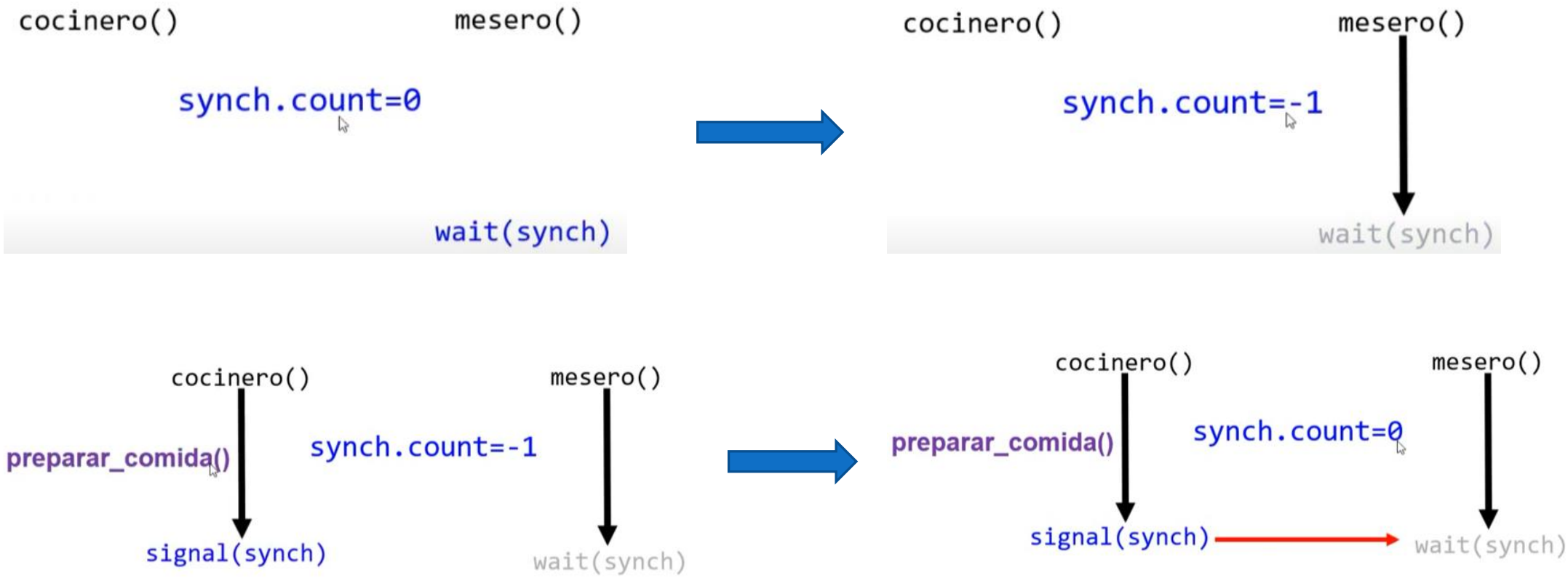
```
Process cocinero()
{
    preparar_comida();
    signal(synch);
}
```

```
semaphore synch;
```

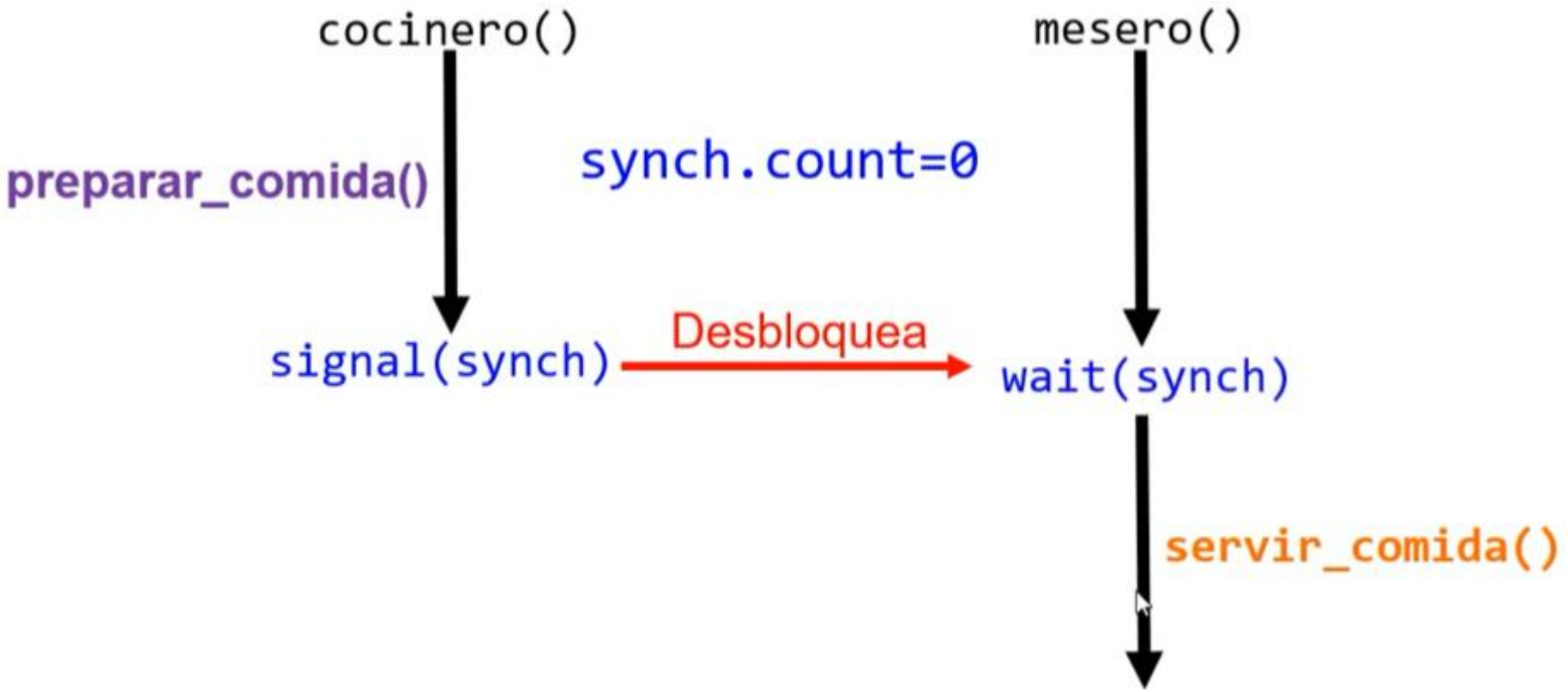
```
main()
{
        initsem(synch,0);
        cobegin {
                cocinero();
                mesero();
        }
}
```

```
Process mesero()
{
    wait(synch);
    servir_comida();
}
```

# Ejemplo de uso de semáforos para sincronizar procesos

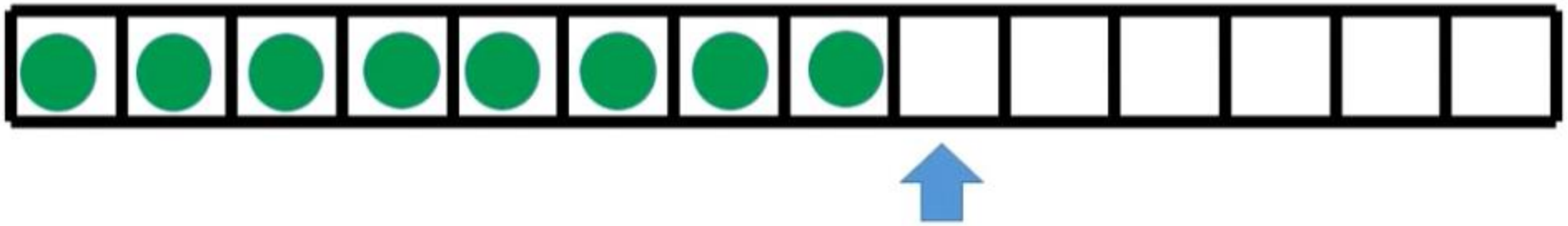


# Ejemplo de uso de semáforos para sincronizar procesos

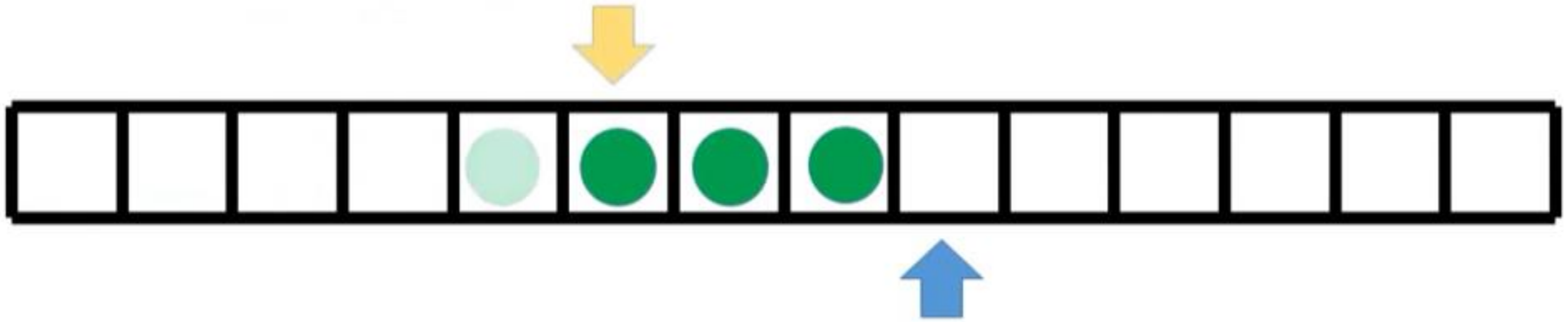


# El problema del productor consumidor

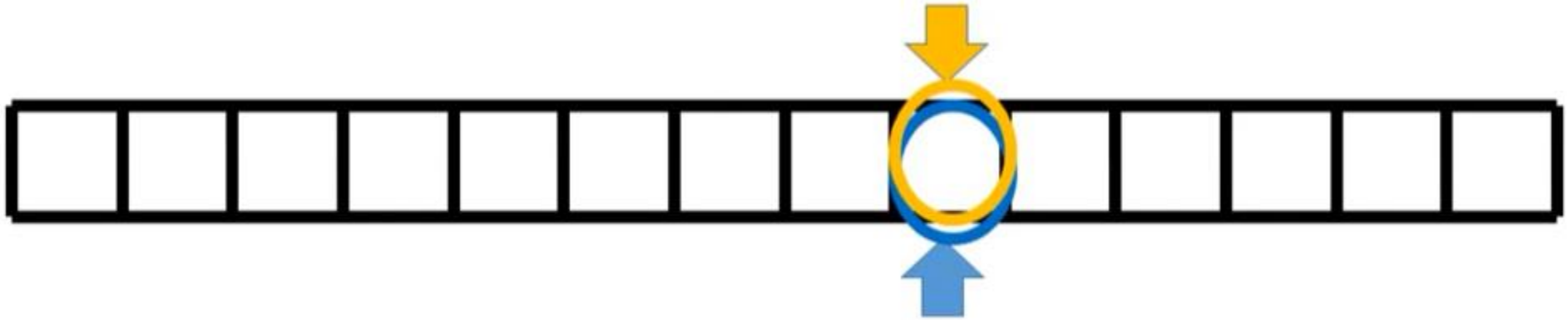
Paradigma de procesos cooperantes, productor produce información que se consume por un consumidor.



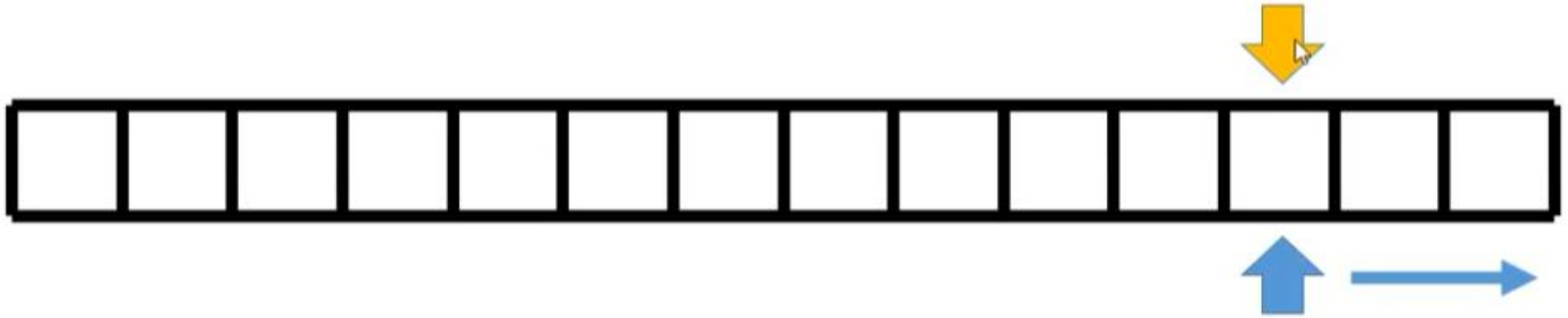
Uno o más productores generan cierto tipo de datos (registros, caracteres) y los sitúan en un buffer



El único consumidor saca elementos del buffer de uno en uno.



El sistema está obligado a impedir la superposición de operaciones sobre el buffer.



El productor puede generar elementos y almacenarlos en el buffer a su propio ritmo

El consumidor procede de manera similar, pero debe estar seguro de que no intenta leer de un buffer vacío.

Por lo tanto, el consumidor debe asegurarse de que el productor ha progresado por delante de él (ent>sal) antes de continuar.

# El problema del productor consumidor (primer intento)

```
int n;
SemaphoreBin s;
SemaphoreBin retraso;

Productor()
{
    while(forever)
    {
1:      producir;
2:      waitB(s);
3:      añadir;
4:      n++;
5:      if (n==1) signalB(retraso);
6:      signalB(s)
    }
}
```

```
Consumidor()
{
    waitB(retraso);
    while(forever)
    {
7:      waitB(s);
8:      tomar;
9:      n--;
10:     signalB(s);
11:     consumir;
12:     if (n==0) waitB(retraso);
    }
}
```

```
main()
{
    n=0;
    initbsem(s,1);
    initbsem(retraso,0);
    cobegin {
        Productor();
        Consumidor();
    }
}
```

```
n=1   retraso.valor=0  s.valor=0
```

```
int n;
SemaphoreBin s; // 1
SemaphoreBin retraso; // 0
```

```
Productor()
{
    while(forever)
    {
1:      producir;
2:      waitB(s);
3:      añadir;
4:      n++;
5:      if (n==1) signalB(retraso);
6:      signalB(s)
    }
}
```

```
Consumidor()
{
    waitB(retraso);
    while(forever)
    {
7:      waitB(s);
8:      tomar;
9:      n--;
10:     signalB(s)
11:     consumir;
12:     if (n==0) waitB(retraso);
    }
}
```

```
main() {
    n=0;
    initbsem(s,1);
    initbsem(retraso,0);
    cobegin {
        Productor();Consumidor();
    }
}
```

Cambios de estado por iteraciones

```
n=0   retraso.valor=0  s.valor=1
n=0   retraso.valor=0  s.valor=0
n=1   retraso.valor=1  s.valor=0
n=1   retraso.valor=1  s.valor=1
n=1   retraso.valor=1  s.valor=0
n=0   retraso.valor=1  s.valor=0
n=0   retraso.valor=1  s.valor=1
n=0   retraso.valor=0  s.valor=1
n=0   retraso.valor=0  s.valor=0
n=-1  retraso.valor=0  s.valor=0
n=-1  retraso.valor=0  s.valor=1
```

```
int n;
SemaphoreBin s; // 1
SemaphoreBin retraso;  // 0
Productor()
{
    while(forever)
    {
1:      producir;
2:      waitB(s);
3:      añadir;
4:      n++;
5:      if (n==1) signalB(retraso);
6:      signalB(s)
    }
}
```

```
Consumidor()
{
    waitB(retraso);
    while(forever)
    {
7:      waitB(s);
8:      tomar;
9:      n--;
10:     signalB(s);
11:     consumir;
        if (n==0) waitB(retraso);
12:
    }
}
```

```
int n;
SemaphoreBin s; // 1
SemaphoreBin retraso;  // 0
Productor()
{
    while(forever)
    {
1:      producir;
2:      waitB(s);
3:      añadir;
4:      n++;
5:      if (n==1) signalB(retraso);
6:      signalB(s)
    }
}
```

```
Consumidor()
{
    waitB(retraso);
    while(forever)
    {
7:      waitB(s);
8:      tomar;
9:      n--;
10:     consumir;
11:     if (n==0) waitB(retraso);
        signalB(s);
12:
    }
}
```

```
int m,n;
SemaphoreBin s; // 1
SemaphoreBin retraso; // 0

Productor()
{
    while(forever)
    {
        producir;
        waitB(s);
        añadir;
        n++;
        if (n==1) signalB(retraso);
        signalB(s);
    }
}
```

```
Consumidor()
{
    waitB(retraso);
    while(forever)
    {
        waitB(s);
        tomar;
        n--;
        m=n;
        signalB(s);
        consumir;
        if (m==0) waitB(retraso);
    }
}
```

# El problema del productor consumidor con semáforos generales enteros

```
Semaphore n;
Semaphore s;

Productor()
{
    while(forever)
    {
        producir;
        wait(s);
        añadir;
        signal(s);
        signal(n);
    }
}
```

```
Consumidor()
{
    while(forever)
    {
        wait(n);
        wait(s);
        tomar;
        signal(s);
        consumir;
    }
}
```

```
main()
{
    initbsem(s,1);
    initbsem(n,0);
    cobegin {
        productor();consumidor();
    }
}
```

# Ejecución

**n.count = 0** **s.count = 1**

```
Semaphore n;
Semaphore s;

Productor()
{
    while(forever)
    {
        producir;
        wait(s);
        añadir;
        signal(s);
        signal(n);
    }
}
```

```
Consumidor()
{
    while(forever)
    {
        wait(n);
        wait(s);
        tomar;
        signal(s);
        consumir;
    }
}
```

```
main()
{
    initbsem(s,1);
    initbsem(n,0);
    cobegin {
        productor();consumidor();
    }
}
```

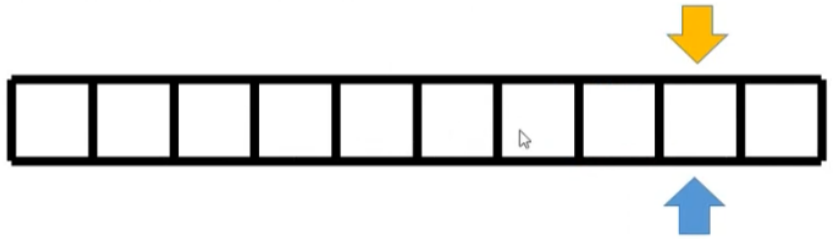
**n.count =-1** **s.count = 1**

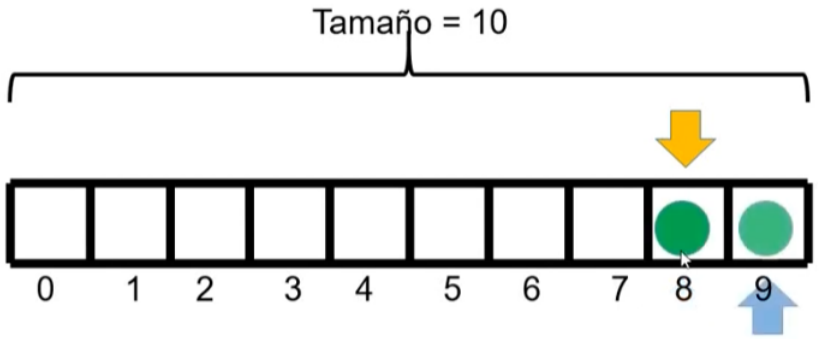
**n.count =-1** **s.count = 0**

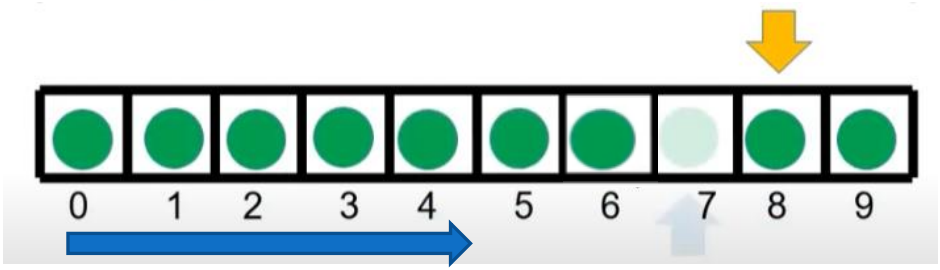
**n.count =-1** **s.count = 1**

**n.count = 0** **s.count = 0**

**n.count = 0** **s.count = 1**




Tamaño = 10
0 1 2 3 4 5 6 7 8 9


0 1 2 3 4 5 6 7 8 9

# Solución con semáforos al problema del productor consumidor con buffer limitado

```
Semaphore s;
Semaphore n;
Semaphore e;

Productor()
{
    while(forever)
    {
        producir;
        wait(e);
        wait(s);
        añadir;
        signal(s);
        signal(n);
    }
}
```

```
Consumidor()
{
    while(forever)
    {
        wait(n);
        wait(s);
        tomar;
        signal(s);
        signal(e);
        consumir;
    }
}
```

```
main() {
    initbsem(s,1);
    initbsem(n,0);
    initbsem(e,TAMAÑO_BUFFER);
    cobegin {
        productor();consumidor()
    }
}
```

# Ejecución

e.count = 6  n.count =0  s.count =1

```
Semaphore s;
Semaphore n;
Semaphore e;

Productor()
{
    while(forever)
    {
        producir;
        wait(e);

        wait(s);
        añadir;
        signal(s);

        signal(n);
    }
}
```

```
Consumidor()
{
    while(forever)
    {
        wait(n);

        wait(s);
        tomar;
        signal(s);

        signal(e);
        consumir;
    }
}
```

```
main() {
    initbsem(s,1);
    initbsem(n,0);
    initbsem(e,TAMAÑO_BUFFER);
    cobegin {
        productor();consumidor()
    }
}
```

e.count = 5  n.count =1  s.count =1

e.count = 4  n.count =2  s.count =1

e.count = 3  n.count =3  s.count =1

e.count = 2  n.count =4  s.count =1

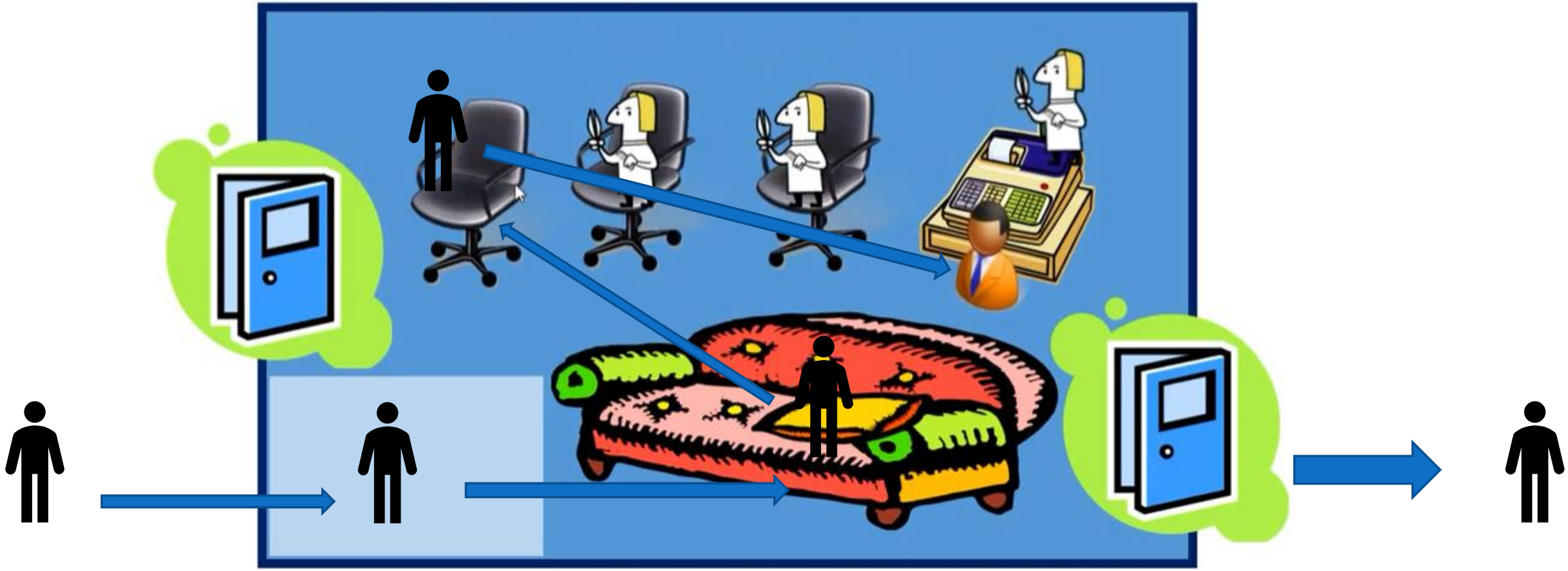
e.count = 1  n.count =4  s.count =1

e.count = 0  n.count =6  s.count =1

e.count =-1  n.count =6  s.count =1

# El problema de la barbería

# El problema de la barbería



# Una barberia no equitativa

```
Semaphore max_capacidad, sofá,
silla_barbero, coord, cliente_listo,
terminado, dejar_silla_b, pago,
recibo;
```

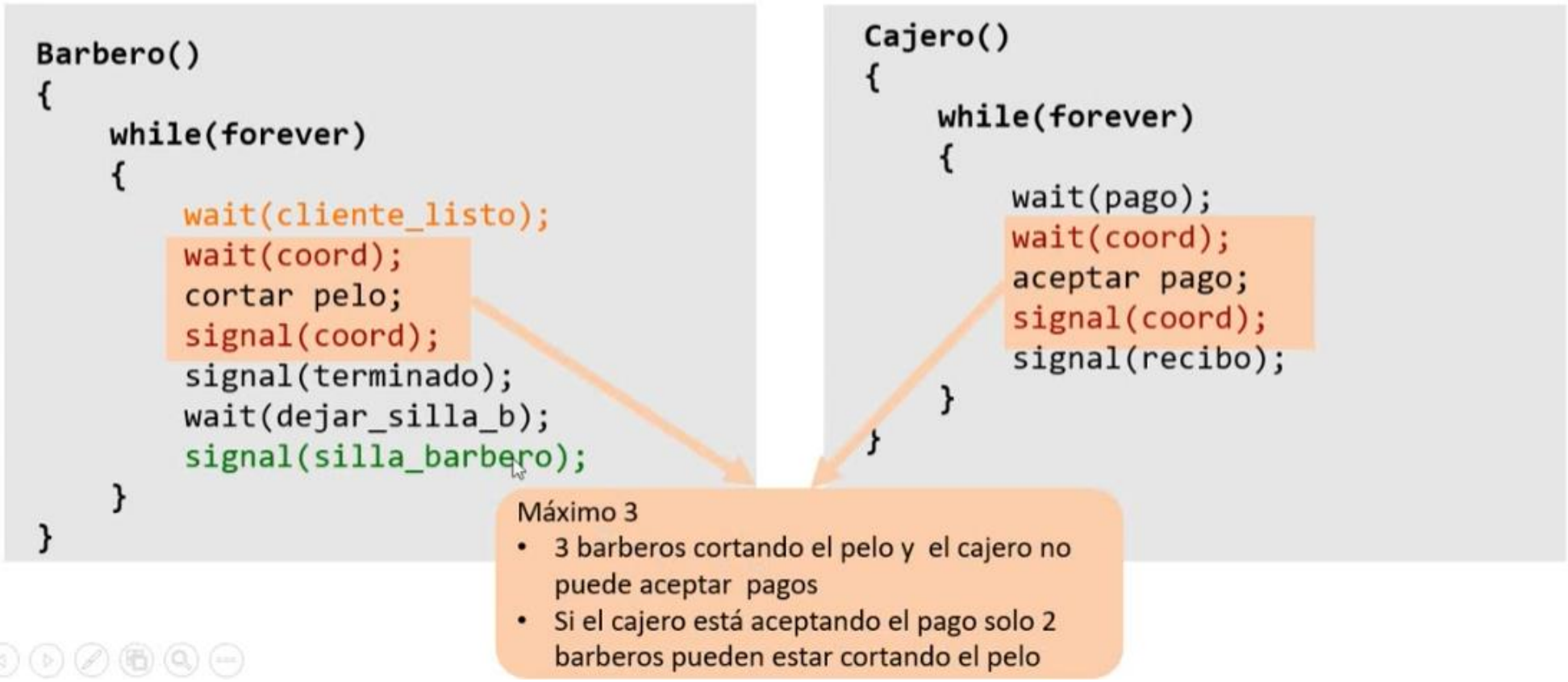
```
main()  // programa principal
{
    initsem(max_capacidad,20);
    initsem(sofá,4);
    initsem(silla_barbero,3);
    initsem(coord,3);
    initsem(cliente_listo,0);
    initsem(terminado,0);
    initsem(dejar_silla_b,0);
    initsem(pago,0);
    initsem(recibo,0);
```

```
cobegin
    {
        cliente();..50 veces..cliente();
        barbero();barbero();barbero();
        cajero();
    }
}
```

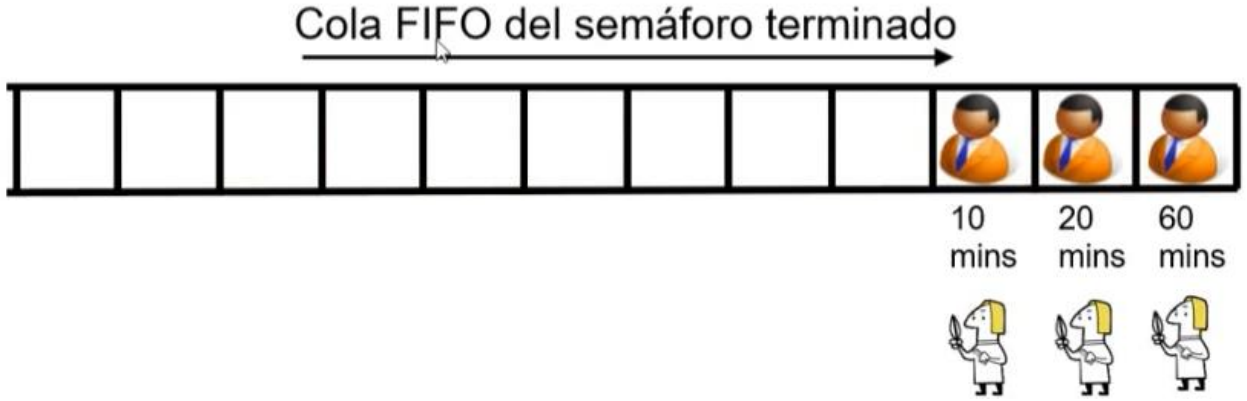
```
Cliente()
{
    wait(max_capacidad);
    entrar en tienda;
    wait(sofá);
    sentarse en sofá;
    wait(silla_barbero);
    levantarse del sofá;
    signal(sofá);
    sentarse en silla del barbero;
    signal(cliente_listo);
    wait(terminado);
    levantarse de la silla del barbero;
    signal(dejar_silla_b);
    pagar;
    signal(pago);
    wait(recibo);
    salir de la tienda;
    signal(max_capacidad);
}
```

```
Barbero()
{
    while(forever)
    {
        wait(cliente_listo);
        wait(coord);
        cortar pelo;
        signal(coord);
        signal(terminado);
        wait(dejar_silla_b);
        signal(silla_barbero);
    }
}
```

```
Cajero()
{
    while(forever)
    {
        wait(pago);
        wait(coord);
        aceptar pago;
        signal(coord);
        signal(recibo);
    }
}
```


Barbero()
{
while(forever)
{
wait(cliente_listo);
wait(coord);
cortar pelo;
signal(coord);
signal(terminado);
wait(dejar_silla_b);
signal(silla_barbero);
}
}
Cajero()
{
while(forever)
{
wait(pago);
wait(coord);
aceptar pago;
signal(coord);
signal(recibo);
}
}
Máximo 3
• 3 barberos cortando el pelo y el cajero no puede aceptar pagos
• Si el cajero está aceptando el pago solo 2 barberos pueden estar cortando el pelo

Tres clientes sentados en las tres sillas de barbero y estarán bloqueados en wait (terminado) y serán liberados en el orden en que se sentaron en las sillas del barbero.




¿Qué ocurre si los clientes requieren diferentes tiempos en sus cortes o uno de los barberos es muy rápido?

# Solución

```
Semaphore max_capacidad, sofá,
  silla_barbero, coord, exmut1,
  exmut2, cliente_listo,
  dejar_silla_b,
  pago,recibo,terminado[50];
int contador;
```

```
main()
{
    initsem(max_capacidad,20);
    initsem(sofá,4);
    initsem(silla_barbero,0);
    initsem(coord,3);
    initsem(exmut1,1);
    initsem(exmut2,1);
    initsem(cliente_listo,0);
    initsem(dejar_silla_b,0);
    initsem(pago,0);
    initsem(recibo,0);
    for(i=0;i<50;i++)
      initsem(terminado[i],0);
```

```
    cobegin
    {

      Cliente();.50 veces..Cliente();
      Barbero();Barbero();Barbero();
      Cajero();
    }
}
```

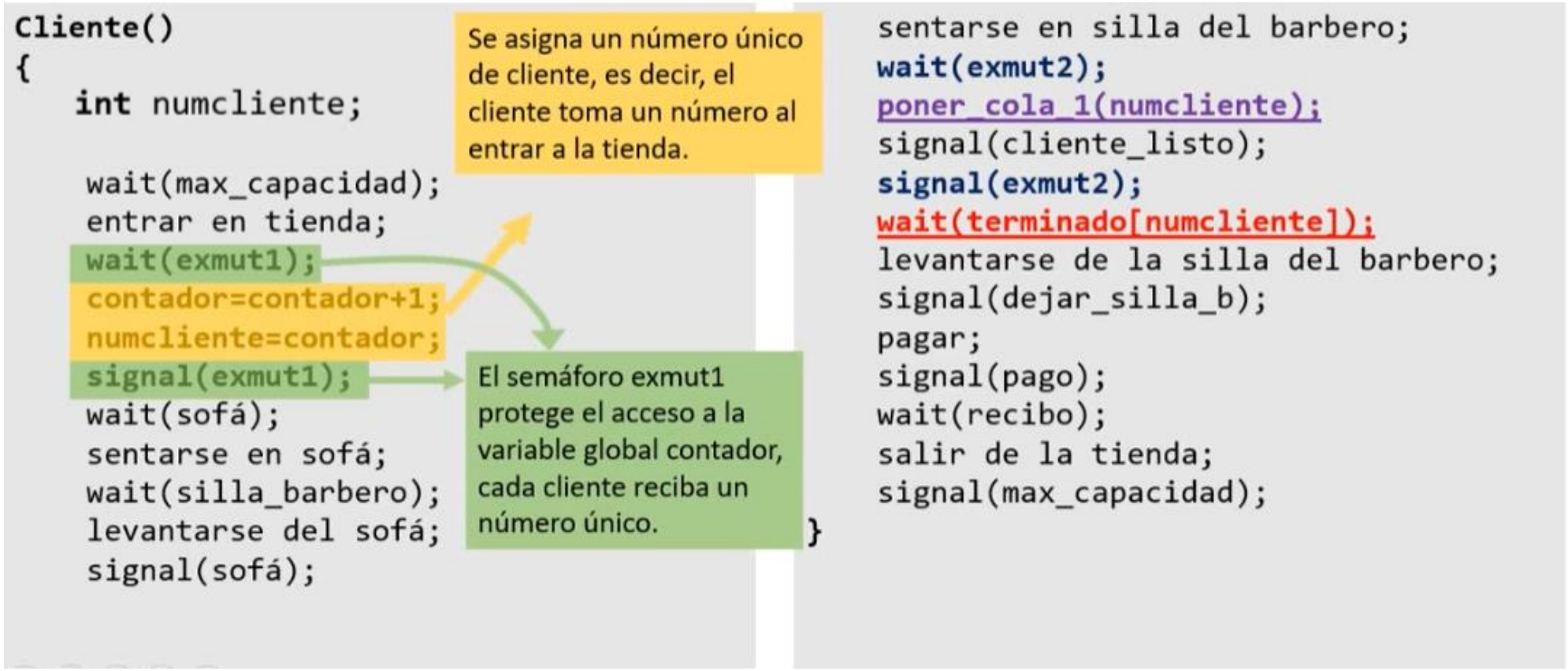
```
Semaphore max_capacidad, sofá,
  silla_barbero, coord, exmut1,
  exmut2, cliente_listo,
  dejar_silla_b,
  pago,recibo,terminado[50];
int contador;
```

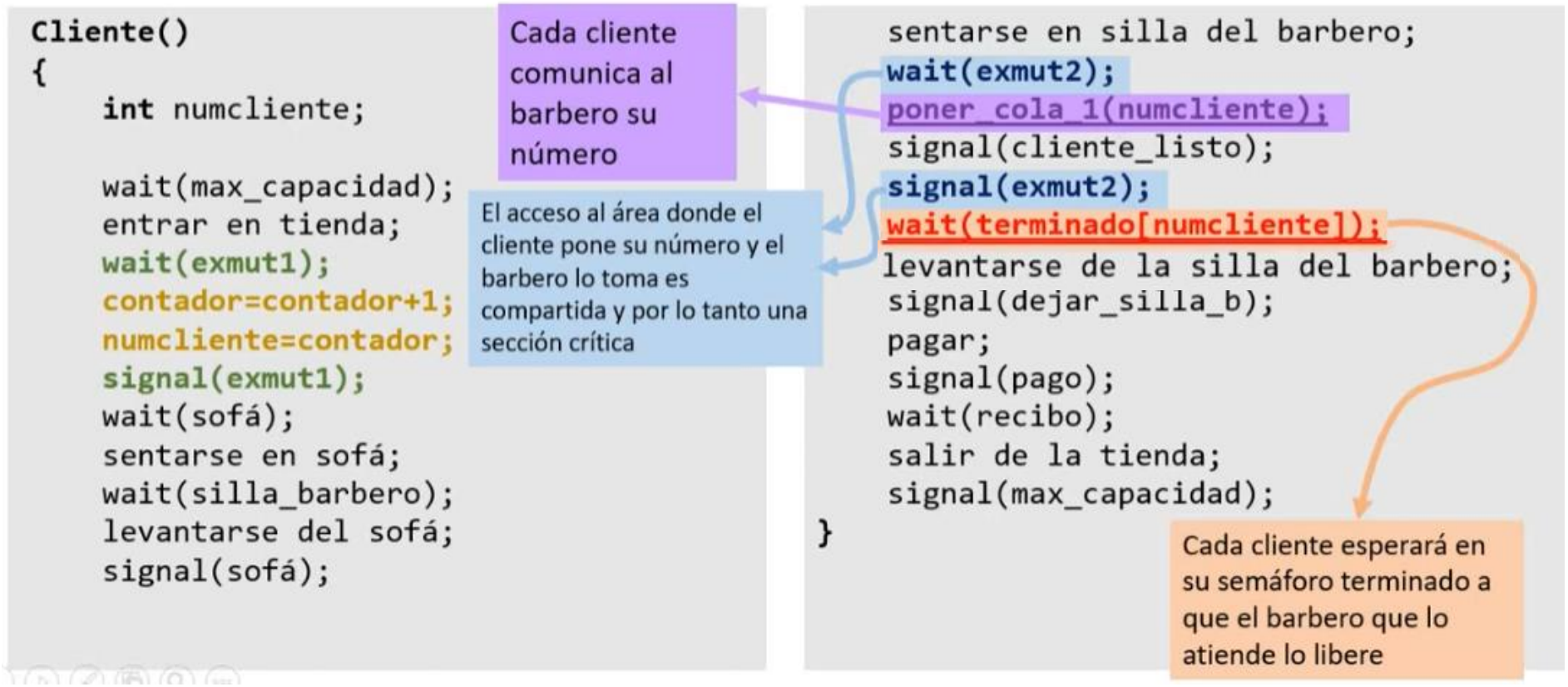
```
main()
{
    initsem(max_capacidad,20);
    initsem(sofá,4);
    initsem(silla_barbero,0);
    initsem(coord,3);
    initsem(exmut1,1);
    initsem(exmut2,1);
    initsem(cliente_listo,0);
    initsem(dejar_silla_b,0);
    initsem(pago,0);
    initsem(recibo,0);
    for(i=0;i<50;i++)
      initsem(terminado[i],0);
```

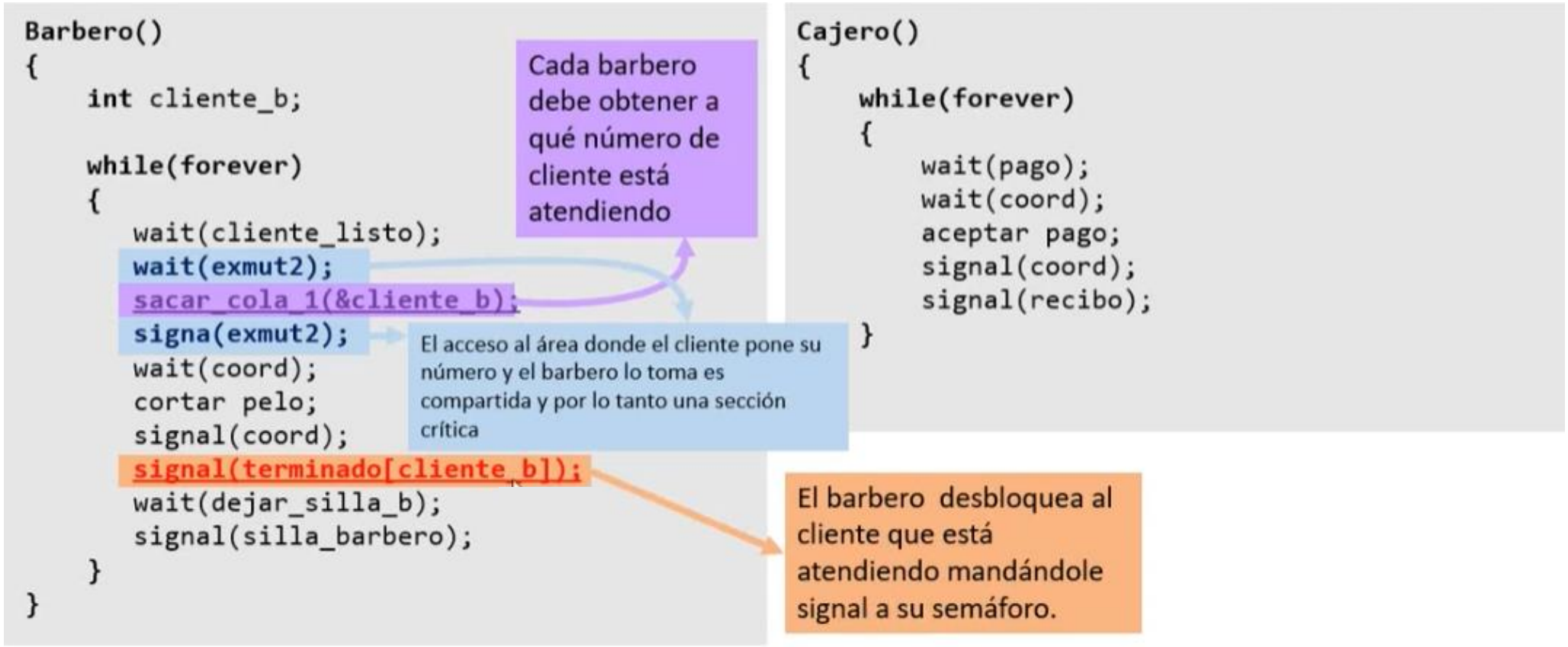
```
    cobegin
    {

      Cliente();.50 veces..Cliente();
      Barbero();Barbero();Barbero();
      Cajero();
    }
}
```

Cada cliente debe tener su propio semáforo terminado inicializado en cero, ahí esperará su signal del barbero que le está cortando el pelo


Cliente()
{
int numcliente;
wait(max_capacidad);
entrar en tienda;
wait(exmut1);
contador=contador+1;
numcliente=contador;
signal(exmut1);
wait(sofá);
sentarse en sofá;
wait(silla_barbero);
levantarse del sofá;
signal(sofá);
Se asigna un número único de cliente, es decir, el cliente toma un número al entrar a la tienda.
El semáforo exmut1 protege el acceso a la variable global contador, cada cliente reciba un número único.
sentarse en silla del barbero;
wait(exmut2);
poner_cola_1(numcliente);
signal(cliente_listo);
signal(exmut2);
wait(terminado[numcliente]);
levantarse de la silla del barbero;
signal(dejar_silla_b);
pagar;
signal(pago);
wait(recibo);
salir de la tienda;
signal(max_capacidad);
}


Cliente()
{
int numcliente;
wait(max_capacidad);
entrar en tienda;
wait(exmut1);
contador=contador+1;
numcliente=contador;
signal(exmut1);
wait(sofá);
sentarse en sofá;
wait(silla_barbero);
levantarse del sofá;
signal(sofá);
Cada cliente comunica al barbero su número
El acceso al área donde el cliente pone su número y el barbero lo toma es compartida y por lo tanto una sección crítica
sentarse en silla del barbero;
wait(exmut2);
poner_cola_1(numcliente);
signal(cliente_listo);
signal(exmut2);
wait(terminado[numcliente]);
levantarse de la silla del barbero;
signal(dejar_silla_b);
pagar;
signal(pago);
wait(recibo);
salir de la tienda;
signal(max_capacidad);
}
Cada cliente esperará en su semáforo terminado a que el barbero que lo atiende lo libere


Barbero()
{
int cliente_b;
while(forever)
{
wait(cliente_listo);
wait(exmut2);
sacar_cola_1(&cliente_b);
signa(exmut2);
wait(coord);
cortar pelo;
signal(coord);
signal(terminado[cliente_b]);
wait(dejar_silla_b);
signal(silla_barbero);
}
}
Cada barbero debe obtener a qué número de cliente está atendiendo
El acceso al área donde el cliente pone su número y el barbero lo toma es compartida y por lo tanto una sección crítica
Cajero()
{
while(forever)
{
wait(pago);
wait(coord);
aceptar pago;
signal(coord);
signal(recibo);
}
El barbero desbloquea al cliente que está atendiendo mandándole signal a su semáforo.