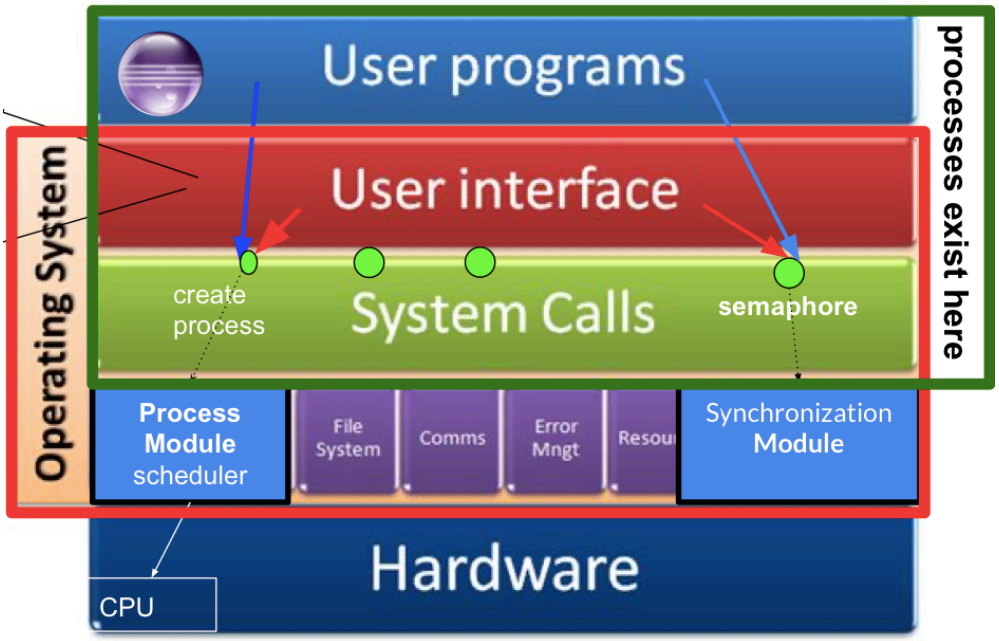
Tema 4: Semáforos

Programación Concurrente y Avanzada

Jorge Llorente Alguacil

1. Concepto de Semáforo



Las llamadas del sistema operativo será lo que utilicemos a la hora de utilizar semáforos.

Como recordatorio, los procesos son programas en ejecución y un proceso se describe como:

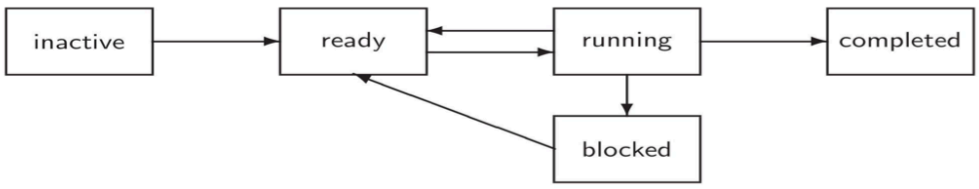
- **Contexto de Memoria:** Valores de variables, pila, espacio del programa
- **Contexto de CPU:** Contador del programa, registros,
- **Bloque de control de proceso:** descriptor de procesos.

Los procesos ya no van a tener que hacer espera activa, los semáforos van a poder bloquear los procesos dentro del sistema operativo. El modulo de procesos del sistema operativo ofrece **llamas del sistema** para crear y gestionar las llamadas de procesos.

Internamente el modulo de procesos del sistema operativo:

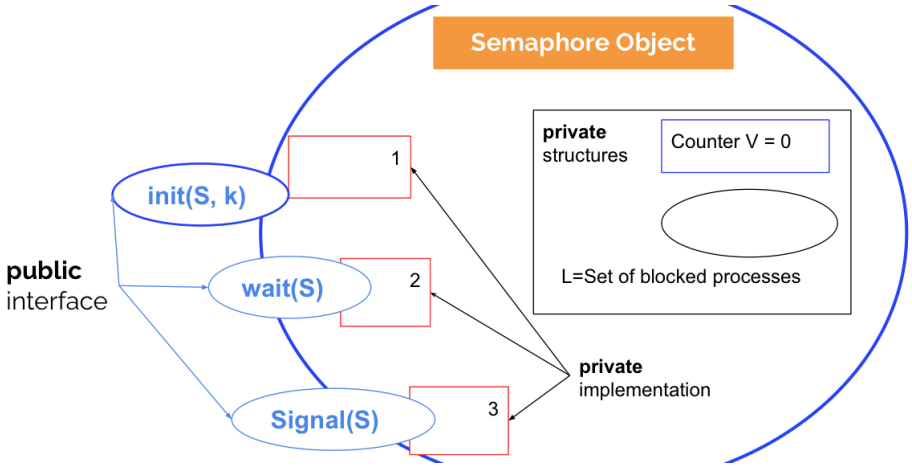
- **Carga los programas en memoria.**
- **Mantiene información sobre los procesos.**
- **Tiene el planificador de los procesos que selecciona el proceso a ejecutar en la CPU.**

El planificador va a tener dos colas, la de **listos** y la de **bloqueados**. En nuestro modelo, el siguiente proceso que ejecutará el planificador, será arbitrario. Los procesos podrán estar listos, ejecutando o bloqueados.



Los semáforos son una de las herramientas que ofrece el sistema operativo, y son objetos. El objeto semáforo tiene el **estado privado** y las **operaciones públicas** para modificar el estado.

- **Operaciones publicas:** init, wait, signal (todas estas operaciones son atómicas)
- **Estado interno del semáforo:** Formado por contador V de señales, y una lista de procesos bloqueados L.



- **`init(S,K):`** inicializa el contador V con K y el conjunto L, lo inicializa a conjunto vacío; todo esto del semáforo S.
- **`wait(S):`** si V > 0 se decrementa uno y p continua en exclusión. Si v = 0 se bloquea el proceso que llama el semáforo. (Consume señales)

```
if    S.V > 0
      S.V <- S.V -1
else
      S.L <- S.L U {p}
      p.state <- blocked
```

- **`signal(S):`** Si no hay procesos bloqueados en L, el contador V se incrementa. Si hay procesos bloqueados en L, se escoge un proceso q de manera arbitraria, y se desbloquea. El proceso que hace la llamada sigue ejecutándose, simplemente deja listo el siguiente proceso. (Genera señales)

```
if    S.L = ø
      S.V <- S.V + 1
else
      let q be an arbitrary process of S.L
      S.L <- S.L - {q}
      q.state <- ready
```

Cabe destacar que si estamos haciendo un signal o un wait sobre un semáforo, no podemos hacer otras operaciones sobre ese semáforo hasta que termine la operación.

2. Semáforos Binarios

Son un tipo de semáforos particulares, ya que el valor de K solo se puede inicializar con 0 o con 1. Además V, no puede tener valores mayor que 1. Esto afectará a la semántica del signal.

- **init(S,K):**
```
init V <- k
set L <- ø
```
- **wait(S):**
```
if   S.V > 0
     S.V <- S.V -1
else
     S.L <- S.L U {p}
     p.state <- blocked
```
- **signal(S):**
```
if   S.V = 1
     //undefined
else if   S.L = ø
     S.V <- 1
else
     let q be an arbitrary process of S.L
     S.L <- S.L - {q}
     q.state <- ready
```

Como hemos mencionado anteriormente, vamos a tener modificaciones en el signal. Esto es debido a que el signal una vez se cambia a uno, no se acepta más señales, por tanto, el método dará una excepción…

Algorithm 6.1: Critical section with semaphores (two processes)	
binary semaphore S ← (1, Ø)	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wait(S)	q2: wait(S)
p3: critical section	q3: critical section
p4: signal(S)	q4: signal(S)

Ya no nos harán falta los preprotocolos y los postprotocolos, ya que directamente los sustituiremos con los semáforos directamente.

El proceso, cuando quiera entrar en la sección crítica, lanzará el wait(S), por tanto, el primero que entre en el wait(S) será quien podrá entrar en la sección crítica, ya que no se bloqueará.

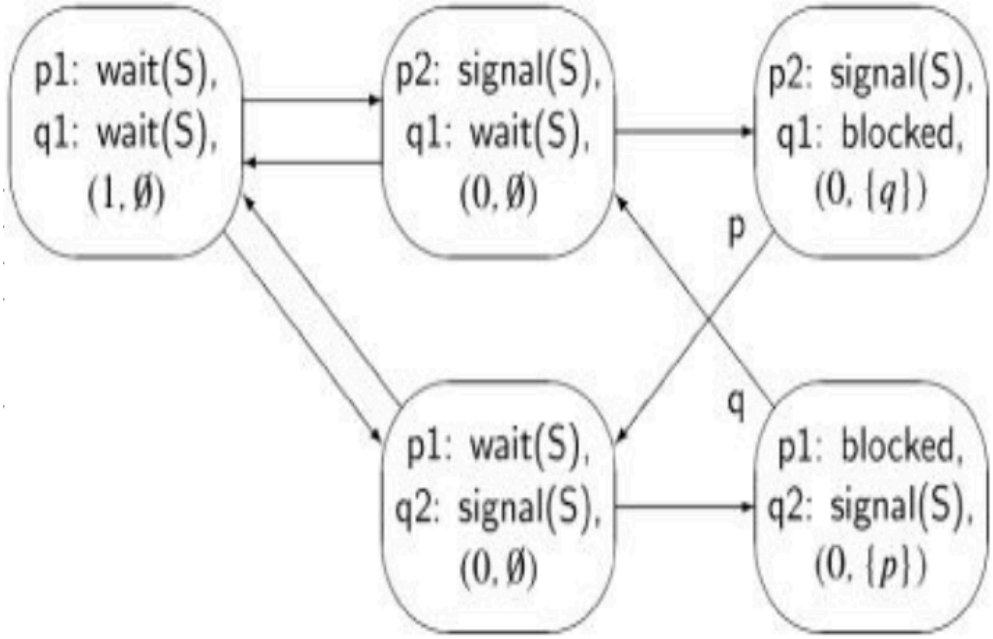
Si el otro proceso quisiese entrar en la sección crítica, al lanzar el wait(S), se quedará bloqueado **(espera pasiva)** hasta que otro proceso haga el signal(S).

Lo que ocurrirá al salir de la sección crítica, será que al ejecutar el signal(S), si quedan procesos bloqueados, al salir de la sección crítica, se bloqueará el siguiente.

En el tema anterior veiamos que “resumiamos” la seccion crítica en el pre-protocolo y el post-protocolo. Nuestro algorítmo quedaría:

Algorithm 6.2: Critical section with semaphores (two proc.,	
binary semaphore S ← (1, ∅)	
p	**q**
loop forever	loop forever
p1: wait(S)	q1: wait(S)
p2: signal(S)	q2: signal(S)

En la FSM los estados nos quedarían:



Cabe destacar que cuando un proceso ejecuta el signal, y otro proceso estaba bloqueado, lo que ocurre es que AMBOS procesos avanzan, es decir, el proceso que estaba bloqueado, se desbloqueará.

Viendo el diagrama de estados anterior, podemos ver que se cumple la propiedad de **EXCLUSIÓN MUTUA** ya que no podemos llegar a (p2, q2).

Respecto a la propiedad de **LIBRE DE INANICIÓN** esta también se cumple, ya que siempre, tarde o temprano salimos del bloqueo para entrar en la sección crítica. También está **LIBRE DE INTERBLOQUEO** ya que los procesos siempre pueden avanzar.

3. Problemas de inanición

El problema que tenemos es que para más de 2 procesos, podemos tener inanición con semáforos binarios, ya que el proceso a desbloquear se selecciona de manera arbitraria

(por la semántica del propio signal). Es lo que se conoce como **Weak Semaphore** o semáforo débil.

Para solucionar esto, utilizaremos una cola (FIFO) para solventar este problema, de tal manera que convertiremos nuestros semáforos en **Strong Semaphores** o semáforos fuertes.

4. Problema de la sección crítica

Algorithm 6.18: Critical section problem (k out of N processes)

```
                    binary semaphore S ← 1, delay ← 0
                    integer count ← k
    integer m
    loop forever
p1:     non-critical section
p2:     wait(S)
p3:     count ← count − 1
p4:     m ← count
p5:     signal(S)
p6:     if m ≤ −1 wait(delay)
p7:     critical section
p8:     wait(S)
p9:     count ← count + 1
p10:    if count ≤ 0 signal(delay)
p11:    signal(S)
```

N procesos intentan entrar en la sección crítica, pero en nuestra sección crítica solo pueden entrar K procesos.

En el preprotocolo restaremos uno a la variable count, y posteriormente, consultamos m y vemos si nos tenemos que bloquear o no en el segundo semáforo. (Si m es menor que -1 nos tendremos que bloquear).

En el postprotocolo incrementaremos el contador, y si este es menor que 0, significa que hay algún proceso bloqueado, por tanto, haremos un signal para dar paso a otro de los procesos que están esperando.

Con **semáforos generales** la cosa se nos simplifica bastante.

semaphore S <- k//(N° de secciones criticas)	
p	q
Loop	Loop

`Non-Critical Section`	`Non-Critical Section`
`Wait(S)`	`Wait(S)`
`Critical Section`	`Critical Section`
`Signal(S)`	`Signal(S)`
`End Loop`	`End Loop`

Vamos a resolver un problema muy interesante con semáforos. **Ejemplo:**

Algorithm 6.5: Mergesort		
integer array A binary semaphore S1 ← (0, ∅) binary semaphore S2 ← (0, ∅)		
sort1	**sort2**	**merge**
p1: sort 1st half of A	q1: sort 2nd half of A	r1: wait(S1)
p2: signal(S1)	q2: signal(S2)	r2: wait(S2)
p3:	q3:	r3: merge halves of A

Hasta que no hayan terminado los procesos sort1, sort2, tenemos una barrera de sincronización que impide que se ejecute el merge. Para ello utilizaremos dos semáforos inicializados a cero.

- **`sort1`:** Ordena la primera parte del array y después indica que ya ha terminado.
- **`sort2`:** Ordena la segunda parte del array y después indica que ya ha terminado.
- **`merge`:** Espera a que se ejecute sort1, una vez el primero hace el signal, merge se desbloquea y continua. Después en wait(S2) espera a que sort2 se ejecute y haga el signal, por tanto el proceso se vuelve a desbloquear y continua. Puede ocurrir que el signal(S2) ya se haya ejecutado, pero este valor se guarda. Por último merge juntará las mitades de A.

Vamos a resolverlo ahora con un semáforo general:

`semaphore S <- 0`		
`sort1`	**`sort2`**	**`merge`**
`sort 1st half of A`	`sort 2nd half of A`	`wait(S)`
`Signal(S)`	`Signal(S)`	`wait(S)`

		`merge halves of A`

5. Problema de consumidores y productores

Casi todo el software de comunicaciones es concurrente, este problema es básico para este tipo de software. Un **productor** produce un dato y lo envía al consumidor mediante un canal de comunicaciones. El **consumidor** se bloquea hasta que recibe datos del productor.

El canal de comunicación lo modelares como una cola FIFO **(buffer)** para los datos. Cabe destacar que hay dos tipos de comunicación:

- **Síncrona:** Cuando el buffer tiene tamaño 0. Por este motivo ambos se tienen que sincronizar en la transferencia del dato, por tanto, si el productor o el consumidor llegan antes tendrán que esperar al otro **(cita).**
- **Asíncrona:** Cuando el buffer tiene tamaño mayor que 0, de manera que el productor dejará **(append)** el dato en el buffer, y el consumidor lo cogerá **(take)** del buffer.

Nuestro problema va a tener **restricciones de sincronización**. Estás son:

- **Un consumidor no puede coger datos de un buffer vacío.**
- **Si el tamaño del buffer es finito, el productor no puede meter un dato si este buffer está lleno.** Se tendrá que quedar bloqueado hasta que haya huecos en el búffer. Si no hacemos eso podemos o **perder datos** o tener riesgo de **sobreescritura.**

6. Problema de consumidores y productores con buffer infinito

Algorithm 6.6: Producer-consumer (infinite buffer)

```
infinite queue of dataType buffer ← empty queue
semaphore notEmpty ← (0, Ø)
```

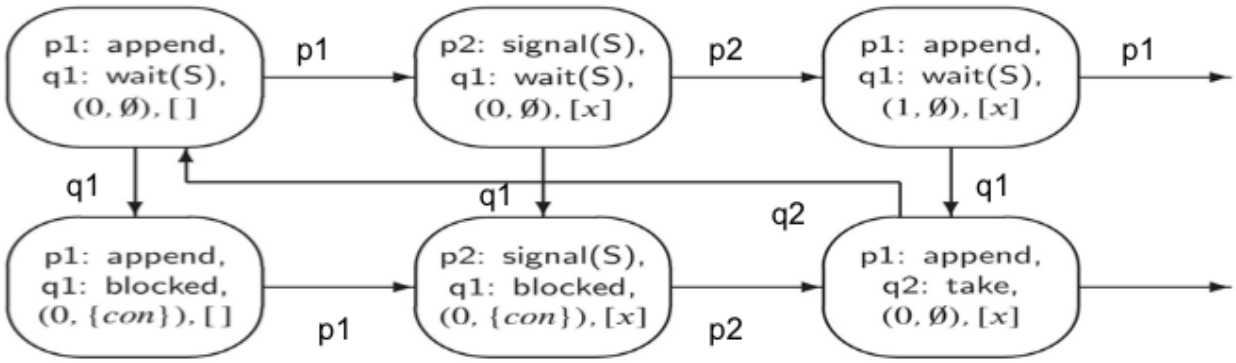
	producer		consumer
	dataType d		dataType d
	loop forever		loop forever
p1:	d ← produce	q1:	wait(notEmpty)
p2:	append(d, buffer)	q2:	d ← take(buffer)
p3:	signal(notEmpty)	q3:	consume(d)

Iniciamos el buffer a una cola vacia y un semáforo. El productore generá el dato y lo mete en la variable d, y hacemos un append en el buffer (lo que deberíamos poner en una sección crítica), y por último hacemos un signal para avisar que hay un dato en el buffer.

En el **consumidor** comprobaremos a ver si hay un dato en el buffer con un wait. Después cogeremos el dato del buffer y lo consumiremos posteriormente.

Con el semáforo respetamos las restricciones de sincronización del buffer vacío. Ambos procesos se ejecutan exclusión mutua **para el mismo buffer**.

La maquina de estados de este problema, utilizaremos la version abreviada solo con el append y el signal:



Como el buffer es infinito, solo vamos a ver los primeros estados. La correción de este algoritmo nos permite:

- **No tener condiciones de carrera,** el consumidor nunca saca un elemento de un buffer vacío.
- **Libre de inanición,** solo un proceso puede ser bloqueado.
- **Libre de interbloqueo,** un productor solo genera señales.

6. Problema de consumidores y productores con buffer finito

Para un buffer finito vamos a utilizar el semáforo que hemos utilizado anteriormente, y uno nuevo que llamaremos notFull (con el tamaño del buffer). Se tiene que cumplir que:

notEmpty = notFull = N

Algorithm 6.8: Producer-consumer (finite buffer, semaphores)

```
finite queue of dataType buffer ← empty queue
semaphore notEmpty ← (0, Ø)  # Occupied positions in the buffer
semaphore notFull ← (N, Ø)   # Free positions in the buffer
```

producer	consumer
dataType d	dataType d
loop forever	loop forever
p1: d ← produce	q1: wait(notEmpty)
p2: wait(notFull)	q2: d ← take(buffer)
p3: append(d, buffer)	q3: signal(notFull)
p4: signal(notEmpty)	q4: consume(d)

El algoritmo para el productor después de producir un dato mandará un wait al semáforo notFull, y si esté está vacío hará el append en el buffer, y después hará un signal para decir que hay datos en el buffer.

En el caso de los consumidores, el algoritmo comprobará si el semáforo no está vacío, de tal manera que cuando no lo esté cogerá un elemento del buffer y hará un signal para decir que ha consumido un dato, y posteriormente consume el dato que ha cogido.

Los buffers circuilares tienen el índice **in** para indezar el primer espacio vacío, y el **out** para indexar el primer elemento para consumir.
El semáforo **notEmpty** cuenta elementos en el buffer y el **notFull** cuenta espacios vacíos

Algorithm 6.19: Producer-consumer (circular buffer)

```
dataType array [0..N] buffer
integer in, out ← 0
semaphore notEmpty ← (0, Ø)
semaphore notFull ← (N, Ø)
```

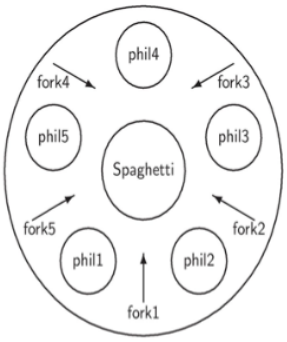
producer	consumer
dataType d	dataType d
loop forever	loop forever
p1: d ← produce	q1: wait(notEmpty)
p2: wait(notFull)	q2: d ← buffer[out]
p3: buffer[in] ← d	q3: out ← (out+1) modulo N
p4: in ← (in+1) modulo N	q4: signal(notFull)
p5: signal(notEmpty)	q5: consume(d)

en el buffer.

7. El problema de la cena de los filósofos

Algorithm 6.9: Dining philosophers (outline)

```
loop forever
p1: think
p2: preprotocol   //take the right and left forks (only one at a time)
p3: eat
p4: postprotocol  // leave the right and left forks
```



Los filósofos para comer deben tener **dos tenedores** y dos filósofos nos pueden tener el mismo tenedor **(exclusión mutua)**. Por último no tendremos **interbloqueos** ni i**nanición.**

Algorithm 6.10: Dining philosophers (first attempt)

```
semaphore array [0..4] fork ← [1,1,1,1,1]
loop forever
p1: think
p2: wait(fork[i])
p3: wait(fork[i+1])
p4: eat
p5: signal(fork[i])
p6: signal(fork[i+1])
```

En este primer intento vamos a utilizar un array de semáforos, todo ellos inicializados a uno. Vamos a asignar cada semáforo una sección crítica, por lo que cada uno tendrá que tener un semáforo. La primera sección crítica gestiona el tenedor i, y la segunda sección crítica el tenedor i+1.

El problema está en que si todos los filósofos se sentasen a la vez habría un interbloqueo, ya que todos solo tendrían un tenedor.

La primera posible solución:

Algorithm 6.11: Dining philosophers (second attempt)

```
semaphore array [0..4] fork ← [1,1,1,1,1]
semaphore room ← 4   // only four enter to eat
```

```
    loop forever
p1:     think
p2:     wait(room)
p3:     wait(fork[i])
p4:     wait(fork[i+1])
p5:     eat
p6:     signal(fork[i])
p7:     signal(fork[i+1])
p8:     signal(room)
```

Lo que vamos a hacer es meter el "comedor" en una habitación, para la cual, a la hora de entrar, solo podrán, como mucho, tres filósofos para entrar. De esta manera, todos los procesos de la derecha podrán coger el "tenedor" de la izquierda.

Podemos asegurar que al menos un filósofo podrá utilizar los dos tenedores. El proceso que se quede fuera, simplemente tendrá que esperar a que uno de los procesos termine.

Respecto al código lo que vamos a hacer va a ser, primero inicializar un semáforo con el array de tenedores, y un semáforo para la habitación. Si este es 0, no podrá entrar en la habitación, si tiene otro valor, podrá entrar en la habitación y coger los dos tenedores. Una vez tiene los dos tenedores, ya puede comer, y cuando haga esto lo que va a ocurrir es que el proceso dejará los tenedores, y saldrá de la habitación.

Esta solución es correcta, ya que **no hay inanición, interbloqueo y hay exclusión mutua.** Un detalle importante es que debemos utilizar aritmética modular para que el filósofo N pueda utilizar el tenedor N y el tenedor 0.

Segunda solución posible:

Algorithm 6.10: Dining philosophers (first attempt)

```
semaphore array [0..4] fork ← [1,1,1,1,1]
    loop forever
p1:     think
p2:     wait(fork[i])
p3:     wait(fork[i+1])
p4:     eat
p5:     signal(fork[i])
p6:     signal(fork[i+1])
```

Algorithm 6.12: Dining philosophers (third attempt)

```
semaphore array [0..4] fork ← [1,1,1,1,1]
    philosopher 4
    loop forever
p1:     think
p2:     wait(fork[0])
p3:     wait(fork[4])
p4:     eat
p5:     signal(fork[0])
p6:     signal(fork[4])
```

M. Ben-Ari. Principles of Concurrent and

Una manera de romper el interbloqueo de la solución incorrecta, es utilizar asimetría. Lo que haremos será utilizar el código de la primera solución para todos los filósofos excepto para el cuarto, que este tendrá un código especifico para él mismo.

De esta manera rompemos la simetría que produce el interbloqueo, lo que hace que este no se produzca.

Esta solución, la podremos aplicar a más problemas, ya que, vamos cogiendo de manera creciente. En el esquema podemos ver como gracias a que al invertir el proceso del filósofo 4, siempre al menos uno de los filósofos puede coger un tenedor.