

Productor – Consumidor

Cambios realizados

Los cambios realizados son pequeños , pero decisivos para el correcto funcionamiento del programa.

1º Cambio (Entorno)

En la parte de definición de los tags e identificadores

Se han cambiado los valores de productor a 1 y consumidor a 2 (aunque es indiferente). Y el valor de buffer a 5, este si es importante ya que no lo vamos a usar como un tag sino como un identificador de la hebra.

```
#define Productor      1//No es numero del proceso, sino la etiqueta(MPI_TAG)
#define Buffer          5//Numero del proceso(hebra destino)
#define Consumidor     2//No es numero del proceso, sino la etiqueta(MPI_TAG)
```

2º Cambio (Main)

Este cambio se ha realizado en el main a la hora de llamar a las funciones , ya que necesitan que le pasemos su id para poder identificarlas.

Lo primero , hemos cambiado ha sido la verificación del numero de hebras a lazar de 3 a 10.

Y después también tenemos que controlar el numero de ellas que vamos a crear, productor(del 0 al 4), buffer(5), consumidor(6..9) = Total 10

```
// verificar el identificador de proceso (rank), y ejecutar la
// operación apropiada a dicho identificador
if ( rank < Buffer )
    productor(rank);
else if ( rank == Buffer )
    buffer();
else
    consumidor(rank);
```

3º Cambio (Funciones)

Dentro de la función ‘Consumidor’ en el envío de la señal tenemos que cambiar el cero por el tag ‘consumidor’ para poder identificar que función tenemos que llevar a cabo.

```

for (unsigned int i=0;i<5;i++){
    MPI_Ssend( &peticion, 1, MPI_INT, Buffer, Consumidor, MPI_COMM_WORLD );
    MPI_Recv ( &value, 1, MPI_INT, Buffer, 0, MPI_COMM_WORLD,&status );
    cout << "Consumidor " << rank << " recibe valor " << value << " de Buffer

```

En la función de recibir añadimos el parámetro MPI_ANY_SOURCE para que reciba la petición de cualquiera que este llamando y le ponemos el tag de productor indicándole que solo esperamos la llamada que tenga ese tag.

Mas abajo dentro del case 1 le quitamos la llamada 'consumidor' y le ponemos el ID extraído del status de la hebra entrante .

```

switch(rama){
    case 0:
        MPI_Recv( &value[pos], 1, MPI_INT, MPI_ANY_SOURCE, Productor, MPI_COMM_WORLD, &status);
        cout << "Buffer recibe " << value[pos] << " de Productor " << status.MPI_SOURCE << endl << flush;
        pos++;
        break;
    case 1:
        MPI_Recv( &peticion, 1, MPI_INT, MPI_ANY_SOURCE, Consumidor, MPI_COMM_WORLD, &status);
        MPI_Ssend( &value[pos-1], 1, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
        cout << "Buffer envía " << value[pos-1] << " a Consumidor " << status.MPI_SOURCE << endl << flush;
        pos--;
        break;
}

```

La var status es inicializada por MPI_Probe que nos sirve para 'sondear' que llamadas quieren entrar y a partir de los resultados obtenidos proceder de la forma mas apropiada.

```

if ( pos==0 ){ // el consumidor no puede consumir
    rama = 0 ;
}else if (pos==TAM){ // el productor no puede producir
    rama = 1 ;
}else{ // ambas guardas son ciertas
    // leer 'status' del siguiente mensaje (esperando si no hay)
    MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status );

    // calcular la rama en función del origen del mensaje
    if (status.MPI_TAG == Productor){
        rama = 0 ;
    }else {
        rama = 1 ;
    }
}
}

```

Por ultimo en la función de 'productor' en el MPI_Ssend le hemos cambiado el 0 que tenia inicialmente por el tag de productor. Para que se pueda identificar el mensaje facilmente.

Muestra de ejecución:

```
angel@Predator-G3610: /media/angel/D8E80201E801DE9E/Universidad/2º año/1º Cuatrimestre/[SCD] Sistemas
angel@Predator-G3610: /media/angel/D8E80201E801DE9E/Universidad/2º año/1º Cuatrimestre/[SCD]
Sistemas Concurrentes y Distribuidos/Practicas/Practica 3/Ejemplos_practica_3$ mpirun -np
10 ./prodcons
Productor 4 produce valor 0
Productor 3 produce valor 0
Productor 1 produce valor 0
Productor 0 produce valor 0
Productor 2 produce valor 0
Productor 4 produce valor 1
Buffer recibe 0 de Productor 4
Consumidor 7 recibe valor 0 de Buffer
Buffer envia 0 a Consumidor 7
Buffer recibe 0 de Productor 0
Productor 0 produce valor 1
Buffer recibe 0 de Productor 2
Productor 2 produce valor 1
Buffer recibe 0 de Productor 1
Productor 1 produce valor 1
Buffer recibe 0 de Productor 3
Buffer envia 0 a Consumidor 8
Productor 3 produce valor 1
Consumidor 8 recibe valor 0 de Buffer
Buffer envia 0 a Consumidor 6
Buffer envia 0 a Consumidor 9
Consumidor 6 recibe valor 0 de Buffer
Consumidor 9 recibe valor 0 de Buffer
Productor 4 produce valor 2
Buffer recibe 1 de Productor 4
Buffer recibe 1 de Productor 0
Productor 0 produce valor 2
Buffer recibe 1 de Productor 2
Productor 2 produce valor 2
Productor 1 produce valor 2
Buffer recibe 1 de Productor 1
Buffer envia 1 a Consumidor 7
Buffer recibe 1 de Productor 3
Consumidor 7 recibe valor 1 de Buffer
Productor 3 produce valor 2
Consumidor 8 recibe valor 1 de Buffer
Buffer envia 1 a Consumidor 8
Buffer recibe 2 de Productor 0
Productor 0 produce valor 3
Buffer envia 2 a Consumidor 9
Buffer recibe 2 de Productor 2
Consumidor 9 recibe valor 2 de Buffer
Productor 2 produce valor 3
Buffer envia 2 a Consumidor 6
Buffer recibe 2 de Productor 1
Consumidor 6 recibe valor 2 de Buffer
Productor 1 produce valor 3
Productor 3 produce valor 3
```

Explicación Filósofos

Aspectos mas desatacados

Las partes mas significativas para la solución de este problema, es realmente tener bien claro como debe de funcionar el programa con la secuencia de sucesos que deben de ir aconteciendo:

- 1º El filosofo envía petición a tenedor
- 2º Tenedor recibe petición
- 3º Filosofo envía petición para tomar otro tenedor
- 4º Tenedor recibe otra petición
- 5º Come

6º Filósofo termina de usar los tenedores y envía una señal a cada uno de ellos.

Aparte de lo arriba mencionado, hay que tener cuidado para que no se produzca interbloqueo. En este caso se produciría interbloqueo si cada filósofo tomase un tenedor a la vez, por lo que se quedarían esperando a que alguno de ellos cediese algún tenedor.

Para evitar este fallo tenemos que cambiar el orden en que toma el tenedor un proceso de ellos. Se ha elegido el proceso(filósofo) cero. Lo que hace es que tomara el tenedor contrario con respecto a sus compañeros filósofos.

Según especifica el problema los filósofos toman el tenedor primero de su izquierda, exceptuamos al filósofo cero que le forzamos a que tome primero el de la derecha.

```
void Filosofo( int id, int nprocesos )
{
    int izq = (id+1) % nprocesos;
    int der = ((id+nprocesos)-1) % nprocesos;

    while(1)
    {
        //El primer filosofo tiene que coger los tenedores al revés para que no se produzca interbloqueo
        if(id == 0)
        {
            //Solicita tenedor derecho
            cout << GREEN << "Filosofo "<<id<< " coge tenedor der ..." << der << BLACK << endl << flush;
            MPI_Ssend(NULL, 0, MPI_INT, der, coger, MPI_COMM_WORLD);

            //solicita tenedor izquierdo
            cout<< RED <<"Filosofo "<<id<< " solicita tenedor izq ..." << izq << BLACK << endl << flush;
            MPI_Ssend(NULL, 0, MPI_INT, izq, coger, MPI_COMM_WORLD);
        }else//resto de filosofos [
        {
            //solicita tenedor izquierdo
            cout << RED <<"Filosofo "<<id<< " solicita tenedor izq ..." << izq << BLACK << endl << flush;
            MPI_Ssend(NULL, 0, MPI_INT, izq, coger, MPI_COMM_WORLD);

            //Solicita tenedor derecho
            cout<< GREEN << "Filosofo "<<id<< " coge tenedor der ..." << der << BLACK << endl << flush;
            MPI_Ssend(NULL, 0, MPI_INT, der, coger, MPI_COMM_WORLD);
        }
    }
}
```

Nótese que el filósofo solo hace llamadas (para ver el código completo revisar el archivo adj.), no tiene que esperar la respuesta de ningún otro proceso.

En donde si se reciben llamadas es en el proceso Tenedor, en este proceso se esperan dos llamadas, una para tomar el tenedor y otra para soltarlo.

Otro punto importante a tener en cuenta, es definir los tags apropiados, para poder saber que acción nos corresponde a la hora de enviar y recibir un mensaje.

```
//Atributos de control
#define soltar 0
#define coger 1
```

Otro cosa a tener en cuenta es en la llamada. Hay que indicarle bien que tenedor quiere llamar, según la posición del filósofo (o lo que es lo mismo su ID), sabemos que los tenedores que tiene que coger en todo momento son los que tiene contiguos a su misma ID.

Aquí podemos ver una línea de la función Filosofo que realiza una llamada.

```
... MPI_Ssend(NULL, 0, MPI_INT, der, coger, MPI_COMM_WORLD);
```

Analicemos:

NULL : Como no vamos a enviar ningún tipo de información , lo dejamos a null. No hace falta pasarle ninguna variable.

0 : Numero de elementos a enviar; Como no enviamos ninguno pues lo ponemos a cero

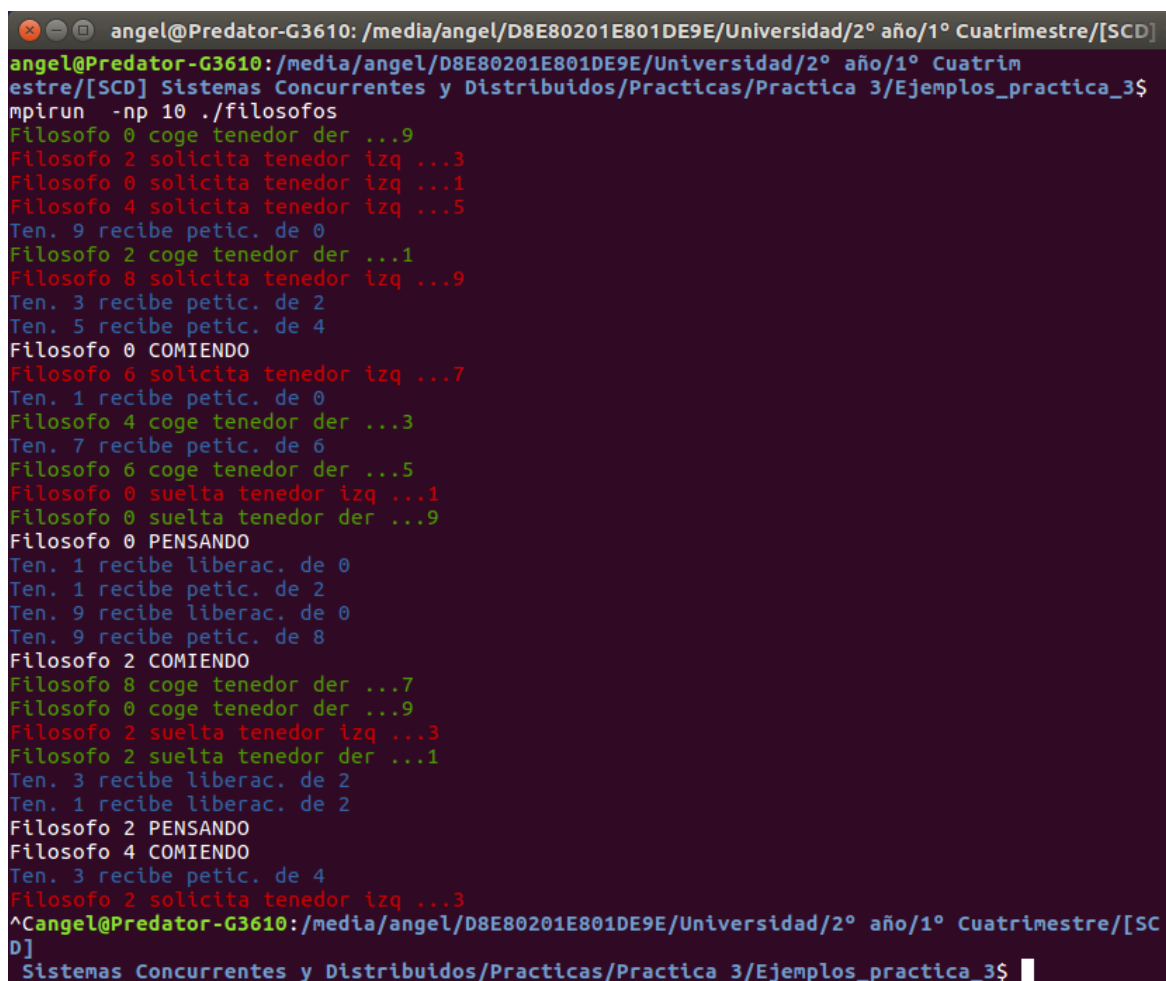
MPI_INT : tipo de dato va a ser entero

der: Significa ‘derecho’ y es un valor entero, este valor corresponde al ID de un proceso Tenedor. Ponemos este campo para llamar al proceso Tenedor correspondiente.

coger: Es el tag que hemos definido para saber como tratar y ubicar esta llamada.

MPI_COMM_WORLD : Comunicador

Captura de ejecución del programa:



```
angel@Predator-G3610: /media/angel/D8E80201E801DE9E/Universidad/2º año/1º Cuatrimestre/[SCD]
angel@Predator-G3610:/media/angel/D8E80201E801DE9E/Universidad/2º año/1º Cuatrimestre/[SCD] Sistemas Concurrentes y Distribuidos/Practicas/Practica 3/Ejemplos_practica_3$
mpirun -np 10 ./filosofos
Filosofo 0 coge tenedor der ...9
Filosofo 2 solicita tenedor izq ...3
Filosofo 0 solicita tenedor izq ...1
Filosofo 4 solicita tenedor izq ...5
Ten. 9 recibe petic. de 0
Filosofo 2 coge tenedor der ...1
Filosofo 8 solicita tenedor izq ...9
Ten. 3 recibe petic. de 2
Ten. 5 recibe petic. de 4
Filosofo 0 COMIENDO
Filosofo 6 solicita tenedor izq ...7
Ten. 1 recibe petic. de 0
Filosofo 4 coge tenedor der ...3
Ten. 7 recibe petic. de 6
Filosofo 6 coge tenedor der ...5
Filosofo 0 suelta tenedor izq ...1
Filosofo 0 suelta tenedor der ...9
Filosofo 0 PENSANDO
Ten. 1 recibe liberac. de 0
Ten. 1 recibe petic. de 2
Ten. 9 recibe liberac. de 0
Ten. 9 recibe petic. de 8
Filosofo 2 COMIENDO
Filosofo 8 coge tenedor der ...7
Filosofo 0 coge tenedor der ...9
Filosofo 2 suelta tenedor izq ...3
Filosofo 2 suelta tenedor der ...1
Ten. 3 recibe liberac. de 2
Ten. 1 recibe liberac. de 2
Filosofo 2 PENSANDO
Filosofo 4 COMIENDO
Ten. 3 recibe petic. de 4
Filosofo 2 solicita tenedor izq ...3
^Cangel@Predator-G3610:/media/angel/D8E80201E801DE9E/Universidad/2º año/1º Cuatrimestre/[SCD]
Sistemas Concurrentes y Distribuidos/Practicas/Practica 3/Ejemplos_practica_3$
```

Filósofos con Camarero central

Este problema es muy similar al anteriormente visto de los filósofos. En este caso se ha implementado un camarero central al que le llegan todas las peticiones de cada uno de los filósofos, 'preguntando' al camarero si pueden sentarse y/o levantarse.

En este problema nos va a hacer falta declarar más tags para indicar en las llamadas cuando van **levantarse** o **sentarse**. También he declarado una variable **camarero** con el número 10 que se usará como ID de hebra y no como tag.

```
//Atributos de control(MPI_TAG)
#define camarero 10 //ID del camarero(NO MPI_TAG)
#define soltar 0
#define coger 1
#define sentarse 2
#define levantarse 3
```

Cambios Main

Después en el main, cambiamos la validación para el número total de hebras poniéndolo a 11 en vez de 10. También tenemos que añadir la inicialización de la hebra del camarero. Quedaría así:

```
if( size!=11)
{
    if( rank == 0)
        cout<<"El numero de procesos debe ser 11" << endl << flush ;
    MPI_Finalize( );
    return 0;
}

if(rank==10)
    Camarero(rank,size); //finalmente llamamos al camarero
else if ((rank%2) == 0)
    Filosofo(rank,size); // Los pares son Filósofos
else
    Tenedor(rank,size); // Los impares son Tenedores
```

Fijarse en el detalle de que cuando el rank sea 10 se inicializa solo la hebra del camarero con ese mismo ID.

Cambios función Filósofo

Los cambios en el filósofo se han producido en el número y orden de llamadas que tiene que realizar . En vez de llamar directamente a un tenedor, ahora lo primero llama al camarero y una vez interactuó con este, seguirá o esperará si no hay ‘sillas disponibles’.

Lo primero el filósofo llamara al camarero pasándole el tag de ‘sentarse’

(***camarero valida la petición***)

Mientras filósofo espera respuesta del camarero. Si el camarero responde significa que hay hueco libre para sentarse y ya puede proceder el filósofo a tomar tenedores, comer , etc.

Veamos la parte en donde interactúa con el camarero:

```
// El filósofo pide sentarse
cout << YELLOW << "Filósofo " << id << " pide sentarse " << BLACK << endl;
MPI_Ssend(NULL, 0, MPI_INT, camarero, sentarse, MPI_COMM_WORLD);

// El filósofo espera a que le digan que puede sentarse
MPI_Recv(NULL, 0, MPI_INT, camarero, sentarse, MPI_COMM_WORLD, &status);
cout << YELLOW << "Filósofo " << id << " se sienta " << BLACK << endl;
// El filósofo se sienta
```

Lo primero enviamos una petición al camarero con el tag sentarse.

Lo segundo espera una llamada procedente del camarero con el tag de sentarse.

Si se recibe esta última petición entonces ya podrá seguir con la ejecución típica del filósofo.

Finalmente cuando ha terminado de comer y pensar tiene que ‘levantarse’ de su asiento por lo que hacemos otra llamada al camarero indicando que nos vamos.

Lo hacemos de la siguiente forma :

```
// el filósofo se levanta
cout << YELLOW << "Filósofo " << id << " se levanta " << BLACK << endl;
MPI_Ssend(NULL, 0, MPI_INT, camarero, levantarse, MPI_COMM_WORLD );
```

Le enviamos una señal al camarero con el tag de levantarse.

El resto del código no hay que modificarlo , tampoco la función tenedor, ya que no tiene nada que ver con el camarero solo con los filósofos.

Una detalle a tener muy en cuenta , es que en este problema podemos quitar la asignación invertida del tenedor a uno de los filósofos , ya que como van a haber solo 4 filósofos en la mesa , ya no se produce el problema del posible interbloqueo al intentar coger los tenedores, porque hay tenedores para todos !!!

Implementación función Camarero

Esta función es de nueva implementación. Su funcionamiento es simple lo primero lo tenemos que meter en un bucle inf. Para que siempre se este ejecutando. Dentro de este bucle haremos todas las validaciones correspondientes.

Lo primero que debemos hacer es que cada vez que seamos llamados debemos de comprobar cuantos huecos están libres para eso tendremos una variable entera que la iremos incrementado o decrementando según la petición recibida.

```
if(asientosDisponibles < 4)//4 Es el maximo de filosofos que estaran comiendo
    MPI_Probe(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status); //Puede sentarse o levantarse
else
    MPI_Probe(MPI_ANY_SOURCE,levantarse,MPI_COMM_WORLD,&status); //Solo puede levantarse
```

Comprobamos que las plazas libres que disponemos no sobrepasen a 4 que sera el tope de filósofos que estarán comiendo .

Si entra en el if quiere decir que tenemos plazas libres , por lo que nos da igual la fuente que nos este llamando y el tag que aporte .

Pero si entra en el else significa que no nos quedan plazas libres , entonces esperamos solo llamadas entrantes que contengan el tag de ‘levantarse’ para ir dejando hueco a otros filósofos.

Hacemos los MPI_Probe para ‘sondear’ las llamadas que tenemos disponibles y así también podemos obtener en la var. status el tag de las hebras que nos llaman para proceder como es debido.

```
//Accedemos a status y consultamos el MPI_TAG
if(status.MPI_TAG == sentarse)//se sienta
{
    ID_filo=status.MPI_SOURCE;
    MPI_Recv(NULL,0,MPI_INT,buf,sentarse,MPI_COMM_WORLD,&status);
    asientosDisponibles++;

    MPI_Send(NULL,0,MPI_INT,ID_filo,sentarse,MPI_COMM_WORLD);
    cout << "Filosofo " << ID_filo << " se sienta. Hay " << asientosDisponibles << " filosofos sent
}

if(status.MPI_TAG==levantarse)//se levanta
{
    ID_filo=status.MPI_SOURCE;
    MPI_Recv(NULL,0,MPI_INT,ID_filo ,levantarse,MPI_COMM_WORLD,&status);
    asientosDisponibles--;
    cout << "Filosofo " << ID_filo << " se levanta. Hay " << asientosDisponibles << " filosofos sen
```

En el primer if se cumplirá la condición si el tag recibido en el MPI_Probe es ‘sentarse’ . Guardamos el ID también obtenido en la variable status gracias a MPI_Probe. Finalmente recibimos la llamada , aumentamos el valor de asientos disponibles y enviamos un aviso al filosofo correspondiente indicándole que se puede sentar.

En el segundo if veremos que el tag sea levantarse, en este caso recibimos la llamada , de un filosofo en concreto y decrementamos el valor de asientos disponibles.

Muestra de ejecución

```
angel@Predador-G3610:/media/angel/D8E80201E801DE9E/Universidad/2º año/1º Cuatrín  
estre/[SCD] Sistemas Concurrentes y Distribuidos/Practicas/Practica 3/Ejemplos_practica_3$ mpirun -np 11 ./filosofos_canarero  
Filosofo 2 pide sentarse  
Filosofo 0 pide sentarse  
Filosofo 4 pide sentarse  
Filosofo 6 se sienta. Hay 1 filósofos sentados.  
Filosofo 4 se sienta. Hay 2 filósofos sentados.  
Filosofo 2 se sienta. Hay 3 filósofos sentados.  
Filosofo 2 se sienta  
Filosofo 2 solicita tenedor izq ...3  
Filosofo 8 pide sentarse  
Filosofo 8 se sienta  
Filosofo 8 solicita tenedor izq ...9  
Filosofo 0 pide sentarse  
Filosofo 0 se sienta  
Filosofo 0 solicita tenedor izq ...7  
Filosofo 6 coge tenedor der ...5  
Filosofo 8 se sienta. Hay 4 filósofos sentados.  
Men. 1 recibe petic. de 2  
Men. 3 recibe petic. de 2  
Men. 7 recibe petic. de 6  
Filosofo 2 coge tenedor der ...1  
Filosofo 2 COMIENDO  
Filosofo 4 se sienta  
Filosofo 4 solicita tenedor izq ...5  
Men. 9 recibe petic. de 8  
Filosofo 8 coge tenedor der ...7  
Filosofo 6 COMIENDO  
Men. 5 recibe petic. de 6  
Filosofo 2 suelta tenedor izq ...3  
Filosofo 2 suelta tenedor der ...1  
Men. 3 recibe liberac. de 2  
Filosofo 2 se levanta  
Filosofo 2 PENSANDO  
Filosofo 2 se levanta. Hay 3 filósofos sentados.  
Filosofo 0 se sienta. Hay 4 filósofos sentados.  
Filosofo 0 se sienta  
Filosofo 0 solicita tenedor izq ...1  
Filosofo 6 se levanta. Hay 3 filósofos sentados.  
Filosofo 0 suelta tenedor izq ...7  
Filosofo 0 suelta tenedor der ...5  
Filosofo 6 se levanta  
Filosofo 6 PENSANDO  
Filosofo 8 COMIENDO  
Men. 5 recibe petic. de 4  
Men. 7 recibe liberac. de 6  
Men. 7 recibe petic. de 8  
Filosofo 4 coge tenedor der ...3  
Filosofo 4 COMIENDO  
Men. 5 recibe liberac. de 6  
Men. 5 recibe petic. de 4  
Men. 1 recibe liberac. de 2  
Filosofo 0 coge tenedor der ...9  
Men. 1 recibe petic. de 0  
Filosofo 2 pide sentarse  
Filosofo 2 se sienta  
Filosofo 2 solicita tenedor izq ...3  
Filosofo 2 se sienta. Hay 4 filósofos sentados.  
Filosofo 0 pide sentarse  
Filosofo 8 suelta tenedor izq ...9  
Filosofo 8 suelta tenedor der ...7  
Filosofo 8 se levanta
```