

Torres Ramos Juan Luis
Portafolios SCD

INDICE

Tema1: Programación multihebra y semáforos

Tema2: Monitores

Tema3: Paso por mensajes MPI

ANTES DE LEER EL PORFOLIO

Este portafolio contiene las traza de código, en forma de captura, más importantes de las actividades realizadas de practicas durante todo el cuatrimestre.

Si hay algo que no cuadra, o que me falta algo, o algún fallo que haya en este pdf con respecto al portafolio, en la carpeta de código se encuentran TODOS los ejercicios resueltos, con su correspondiente Makefile para compilar para ejecutar el código.

#include <mpi.h> de los ejercicios del tema 3, lo tengo con otra dirección, depende del ordenador hay que cambiarlo dicha ruta para que compile.

TODOS los ejercicios me compilan y se ejecutan correctamente.

Si hay algun problema con el portafolio coméntemelo a mi gmail

Tema 1

Ejercicio 1: Calculo numérico de integrales

- Objetivo: Calculo del valor I de la integral de una función f de variable real (entre 0 y 1, por ejemplo) y valores reales positivos.



```
1 // -----
2 // Funciones Hebras
3
4 // funcion que ejecuta cada hebra
5 double funcion_hebra( long i){
6     double SumaParcial = 0.0;
7
8     for ( int j = i; j < m ; j++){
9         const double z = double(i+0.5)/m;
10        SumaParcial += f(z);
11    }
12
13    return SumaParcial;
14 }
15
16
17 // aqui uso vector de futures para calcular como en la integral
18 // n = 4
19 double calcular_integral_concurrente(){
20
21     // declaramos
22     future<double> futuros[n];
23
24     // inicializamos
25     for (int i = 0; i < n; i++)
26         futuros[i] = async(launch::async, funcion_hebra, i);
27
28     double resultado;
29     // hacemos get aacumulandolo en resultado
30
31     for (int i = 0; i
32         < n; i++)
33         resultado += futuros[i].get();
34
35     return resultado;
36
37
38 }
```

Ejercicio 2: Sincronización de hebras con semáforos

Ejercicio 2.1: Problema del productor-consumidor

```
● ● ●
1 // -----
2 // Funciones para las hebras
3
4 void funcion_hebra_productora(){
5     for (unsigned i = 0; i < num_items; i++){
6         unsigned dato = producir_dato();
7
8         libres.sem_wait();
9         //SC
10
11        //OPCIÓN FIFO
12        buffer[plibre]=dato;
13        plibre++;
14
15
16        //OPCIÓN LIFO
17        //buffer[primera_libre] = dato;
18        //primera_libre++;
19        //primera_libre = primera_libre % tam_vec;
20
21        ocupadas.sem_signal();
22
23    }
24 }
25
26 void funcion_hebra_consumidora(){
27     for (unsigned i = 0; i < num_items; i++){
28         unsigned dato;
29
30         ocupadas.sem_wait();
31         //SC
32
33
34         //OPCIÓN FIFO
35         dato = buffer[plibre -1 ];
36         plibre--;
37
38         //OPCIÓN LIFO
39         //dato = buffer[primera_ocupada];
40         //primera_ocupada++;
41         //primera_ocupada = primera_ocupada % tam_vec;
42
43
44         cout << "Extraigo: " << dato << endl;
45
46         libres.sem_signal();
47
48         consumir_dato(dato);
49     }
50 }
```

Ejercicio 2.2: Fumadores

```
1 // -----
2 // Funciones para las hebras
3
4 void funcion_hebra_estanquero(){
5
6     while (true){
7         int i = producir_ingrediente();
8         // Como hemos producido un ingrediente, el mostrador no esta vacio -> pasa a 0, hacemos un wait
9
10        mostrador_vacio.sem_wait();
11        // Como hemos producido un ingrediente hacemos un cout de lo que hemos puesto
12        cout << "Hemos puesto el ingrediente: " << i << endl;
13
14        // ingrediente i corresponde al fumador i para que fume
15        // en consecuencia, hay un ingrediente disponible para fumador[i] -> lo ponemos a 1
16
17        fumadores[i].sem_signal();
18
19        //SC
20    }
21 }
22
23 void funcion_hebra_fumadores(int num_fumador){
24
25     while (true){
26
27         // Aqui se retira un ingrediente, por lo que deja de haber el ingrediente i para
28         // el fumador i, en consecuencia lo pasamos a 0
29         int i = num_fumador;
30         fumadores[i].sem_wait();
31
32         // hago un cout de que se ha gastado el ingrediente
33         cout << "Se ha retirado el ingrediente: " << i << " por el fumador " << i << endl;
34
35         // como se ha retirado el ingrediente del mostrador, el mostrador queda vacio lo ponemos a 1
36         mostrador_vacio.sem_signal();
37
38
39         // Ahora el fumador procede a fumar
40         fumar(num_fumador);
41     }
42 }
```

TEMA 2

Ejercicio 1:

- Compila y ejecuta monitor_em.cpp. Veras que ejecuta las funciones test_1,test_2 y test_3, cada una de ellas usa uno de los tres monitores descritos.

- Verifica que el valor obtenido es distinto del esperado en el caso del monitor sin exclusión mutua. Verifica que los otros dos monitores (con EM), el valor obtenido coincide con el esperado.

- Prueba a quitar el unlock de MContador2::incrementa. Describe razonadamente que ocurre.

Test 1

Valor obtenido: 16529

Valor esperado: 20000

Test 2

Valor obtenido: 20000

Valor esperado: 20000

Test 3

Valor obtenido: 20000

Valor esperado: 20000

Cuando quito el unlock de MContador2::incrementa, lo que ocurre es que realiza el test 1 y luego ,cuando realiza dicha funcion, se queda esperando al cierre, que hemos borrado, por lo que se queda en stand by. Cuando abrimos un cerrojo siempre hemos de cerrarlo.

Ejercicio 2: Razona

1. La hebra que entra la ultima al metodo cita (hebra señaladora, es siempre la primera en salir de dicho metodo).

-> Cuando hacemos un notify_one, o un notify_all para despertar a las hebras, la hebra señaladora, la hebra que ha sido llamada, la ultima, estaa continuara la ejecucion, y las otras estaran en espera para ejecutarse tmb el resto.

2. El orden en el que las hebras señaladas logran entrar de nuevo al monitor no siempre coincide con el orden de salida de wait (se observa porque los numeros de orden de entrada no aparecen ordenados a la salida).

-> Pierde el orden ya que tras depus de ser despertadas, se ponen en espera en la cola sin ningun orden y se ejecuta prierio la señalada.

3. El constructor de la clase no necesita ejecutarse en exclusión mutua.

-> El constructor solo inicializa las variables, por lo que no hay problemas de interfoliacion. Con notify_one y su bulce y un notify_all ocurre lo mismo.

Ejercicio 3: ProdConsMFIFO, monitorSC

```
1 // -----
2 // Monitor
3
4 class MonitorPC{
5 private:
6     int buffer[tam_vec]; // array con los datos insertados pendientes de extraer
7     int primera_libre; // Sol LIFO
8
9     // la condicion variable sustituye a mi funcion de semaforo en un monitor
10    condition_variable libres; // cola de espera hasta que nº hebras < tam_vec (prod)
11    condition_variable ocupadas; // cola de espera hasta nº hebras > 0 (cons.)
12
13    // creo mi guarda
14    mutex cerrojo;
15
16 public:
17
18     MonitorPC();
19     void insertar(int dato); // escribe un valor (E) CONSUMIDOR
20     int extraer(); // lee un valor (L) PRODUCTOR
21 };
```

```
1 // -----
2 // Funciones Hebras
3
4 void funcion_hebra_productora(MonitorPC *monitor){
5     for (int i = 0; i < num_items; i++){
6         int dato = producir_dato();
7         monitor->insertar(dato);
8     }
9 }
10
11 void funcion_hebra_consumidora(MonitorPC *monitor){
12     for (int i= 0; i < num_items; i++){
13         int dato = monitor->extraer();
14         consumir_dato(dato);
15     }
16 }
```

```
● ● ●
1 // -----
2 // Funcion Monitor
3
4 void MonitorPC::insertar(int dato){
5
6     // escribe un valor (E) CONSUMIDOR
7
8     //0. guarda
9     unique_lock<mutex> guarda(cerrojo);
10
11    // estamos en la hebra productora, insertamos
12    // condicion impuesta en el apartado, si esta llena
13    if (primera_libre == num_items) // si vemos que plibres, nuestro contador es igual al tamaño ya no quedan libres, los ponemos a espera
14        libres.wait(guarda);
15
16    assert(primer_libre < num_items); // TIENE QUE SER ASSERT PARA LOGRAR LA EM
17
18    // si no esta lleno mi buffer, introduzco uno en la cola
19    // incremento primera libre y luego notifico a la cola
20    // de ocupada que se ha llenado una
21
22    buffer[primera_libre] = dato;
23    ++primera_libre;
24
25    ocupadas.notify_one();
26
27
28
29
30 }
31
32 int MonitorPC::extraer(){
33     //0. guarda
34     unique_lock<mutex> guarda(cerrojo);
35     int valor;
36
37     if (primera_libre == 0)
38         ocupadas.wait(guarda);
39     assert(0 < primera_libre);
40     // si primera_libre es mayor que 0 puedo extraer
41     --primera_libre;
42     valor = buffer[primera_libre];
43
44     // he liberado una pos libre
45     libres.notify_one();
46
47
48     return valor;
49 }
50 }
```

Ejercicio 4: Prod Cons FIFO multiple

```
1 // -----
2 // Funciones Hebras
3
4 void funcion_hebra_productora(MonitorPC *monitor,int valor){
5     while (true){
6         //for (int i = 0; i < num_items; i++){
7             int dato = producir_dato(valor);
8             monitor->insertar(dato);
9         }
10    }
11
12 void funcion_hebra_consumidora(MonitorPC *monitor){
13     while (true){
14         //for (int i= 0; i < num_items; i++){
15             int dato = monitor->extraer();
16             consumir_dato(dato);
17         }
18    }
}
```

```
1 // -----
2 // Funciones
3
4
5 unsigned producir_dato(int valor) // produce mas de lo que pide error, jode el contador, hay que actualizarlo
6 {
7     static int contador = valor * (num_items/tam_productoras) ;//
8
9     this_thread::sleep_for( chrono::milliseconds( aleatorio<20,100>() ) );
10
11     mtx.lock();
12     cout << "producido: " << contador << endl << flush ;
13     mtx.unlock();
14
15     cont_prod[contador] ++ ; // array compartido
16     return contador++ ;
17 }
```

```
● ● ●
```

```
1 int MonitorPC::extraer(){
2     //0. guarda
3     unique_lock<mutex> guarda(cerrojo);
4     int valor;
5
6     while (primera_libre == 0)
7         ocupadas.wait(guarda);
8     assert(0 < primera_libre);
9     // si primera_libre es mayor que 0 puedo extraer
10    --primera_libre;
11    valor = buffer[primera_libre];
12
13    // he liberado una pos libre
14    libres.notify_one();
15
16
17    return valor;
18
19 }
```

```
● ● ●
```

```
1 void MonitorPC::insertar(int dato){
2
3     // escribe un valor (E) CONSUMIDOR
4
5     //0. guarda
6     unique_lock<mutex> guarda(cerrojo);
7
8     // estamos en la hebra productora, insertamos
9     // condicion impuesta en el apartado, si esta llena
10    while (primera_libre == num_items) // si vemos que plibres, nuestro contador es igual al tamaño ya no quedan libres, los ponemos a espera
11        libres.wait(guarda);
12
13    assert(primera_libre < num_items); // TIENE QUE SER ASSERT PARA LOGRAR LA EM
14
15    // si no esta lleno mi buffer, introduzco uno en la cola
16    // incremento primera libre y luego notifico a la cola
17    // de ocupada que se ha llenado una
18
19    buffer[primera_libre] = dato;
20    ++primera_libre;
21
22    ocupadas.notify_one();
23
24
25
26
27 }
28
```

Ejercicio 5: Lectores

```
● ○ ●
1 // -----
2 // Monitor
3 class MonitorLE : public HoareMonitor {
4 private:
5     bool escrib; // true si un escritor esta escribiendo, false si no hay
6                 // escritores escribiendo
7     int n_lec;  // numero de lectores que estan escribiendo en un momento dado
8
9     CondVar lectura; // se usa por los lectores (ini_lectura) para esperar cuando
10                // hay un escritor escribiendo (escrib == true)
11     CondVar escritura; // se usa por los escritores (ini_escritura) para esperar
12                // cuando hay otro escritor escribiendo (escrib == true) o
13                // bien hay lectores leyendo n_lec > 0
14
15    // estas condiciones cumple que not escrib or n_lec == 0
16    // si se termina de llenar -> signal
17
18 public:
19     MonitorLE();
20     void ini_lectura();
21     void fin_lectura();
22     void ini_escritura();
23     void fin_escritura();
24 };
```

```
● ○ ●
1 // -----
2 // Funciones hebras
3
4 void funcion_hebra_lectores(MRef<MonitorLE> monitor, int num_lect) {
5     while (true) {
6         Espera();
7         monitor->ini_lectura();
8         leer(num_lect);
9         monitor->fin_lectura();
10    };
11 }
12
13 void funcion_hebra_escritores(MRef<MonitorLE> monitor, int num_escrit) {
14     while (true) {
15         Espera();
16         monitor->ini_escritura();
17         escribir(num_escrit);
18         monitor->fin_escritura();
19    }
20 }
21
```

```
1 // -----
2 // Funciones Monitor
3 void MonitorLE::ini_lectura() {
4     if (escrib)
5         lectura.wait(); // si se esta escribiendo, no se pede hacer otra cosa
6
7     n_lec += 1; // incremento el nº lectores
8
9     // como he incrementado, tmbm llamo
10    lectura.signal();
11 }
12
13 void MonitorLE::fin_lectura() {
14     // terminamos una lectura, decrementamos
15     n_lec -= 1;
16
17     // si vemos que es el ultimo lector, desbloqueamos un escritor y que escriba
18     if (n_lec == 0)
19         escritura.signal();
20 }
21
22 void MonitorLE::ini_escritura() {
23     // si vemos que hay otros procesos esperamos la escritura se hace uno a uno
24     if (escrib || n_lec > 0)
25         escritura.wait();
26
27     escrib = true; // podemos escribir
28 }
29
30 void MonitorLE::fin_escritura() {
31     escrib = false; // hemos terminado de leer
32
33     // si hay lectores, despertamos a uno, si no despertamos a un escritor
34     if (!lectura.empty()) {
35         lectura.signal();
36     } else {
37         escritura.signal();
38     }
39 }
```

Ejercicio 6: Fumadores

```
● ● ●  
1 // -----  
2 // MOnitor SU  
3  
4 class Estanco : public HoareMonitor {  
5  
6 private:  
7     CondVar mostrador, clientes[num_fumadores]; // mis hebras  
8     int ingrediente_mostrador;  
9  
10 public:  
11     Estanco();  
12     void ponerIngrediente(int x);  
13     void ObtenerIngrediente(int i);  
14     void esperaRecogerIngrediente();  
15 };  
16
```

```
● ● ●  
1 // -----  
2 // constructor  
3  
4 Estanco::Estanco() {  
5     for (int i = 0; i < num_fumadores; i++) {  
6         clientes[i] = newCondVar();  
7     }  
8     mostrador = newCondVar();  
9     ingrediente_mostrador = -1;  
10 }
```

```
● ● ●
1 // -----
2 // Funciones Monitor
3 void Estanco::ponerIngrediente(int x) {
4     ingrediente_mostrador = x;
5     cout << "\nIngrediente " << x << " en el mostrador" << endl;
6     //cout << "\nMostrador lleno";
7     clientes[x].signal();
8 }
9
10 void Estanco::ObtenerIngrediente(int i) {
11     if (ingrediente_mostrador != i) {
12         clientes[i].wait();
13     }
14
15     cout << "\nSe retira el ingrediente " << i << " del mostrador." << endl;
16     //cout << "\nMostrador Vacio" << endl;
17
18     ingrediente_mostrador = -1;
19
20     mostrador.signal();
21 }
22
23 void Estanco::esperaRecogerIngrediente() {
24     if (ingrediente_mostrador != -1)
25         mostrador.wait();
26 }
27
```

```
● ● ●
1 // -----
2 // FUnciones Hebras
3
4 void funcion_hebra_estanquero(MRef<Estanco> estanco) {
5     //for (int i = 0; i < 10; i++){
6     while (true) {
7         int ingrediente = producir_ingrediente();
8         estanco->ponerIngrediente(ingrediente);
9         // cout <<"\nEn el mostrador esta el ingrediente: "<<ingrediente;
10        // ingrediente_mostrador=ingrediente;
11        // Se le hace un signal al fumador que necesite ese ingrediente
12        // sem_signal(ingr_disp[ingrediente_mostrador]);
13        estanco->esperaRecogerIngrediente();
14    }
15 }
16
17 void funcion_hebra_fumador(MRef<Estanco> estanco, int num_fum) {
18     //for (int i = 0; i < 10; i++){
19     while (true) {
20         estanco->ObtenerIngrediente(num_fum);
21         fumar(num_fum);
22     }
23 }
```

Tema 3:

Ejercicio 1: Productores Consumidores Múltiples

```
● ● ●  
1 int producir(int id)  
2 {  
3     int k = num_items/num_productores;  
4     static int contador = id*k ;  
5     sleep_for( milliseconds( aleatorio<10,100>() ) );  
6     contador++ ;  
7     cout << "Productor ha producido valor " << contador << endl << flush;  
8     return contador ;  
9 }
```

```
● ● ●  
1 void funcion_productor(int id)  
2 {  
3     for ( unsigned int i= 0 ; i < num_items/num_productores ; i++ )  
4     {  
5         // producir valor  
6         int valor_prod = producir(id);  
7         // enviar valor  
8         cout << "Productor va a enviar valor " << valor_prod << endl << flush;  
9  
10        // AÑADO LA ETIQUETA CORRESPONDIENTE  
11        MPI_Ssend( &valor_prod, 1, MPI_INT, id_buffer, etiqueta_productor, MPI_COMM_WORLD );  
12    }  
13 }
```

```

1 void funcion_buffer()
2 {
3     int      buffer[tam_vector],      // buffer con celdas ocupadas y vacías
4         valor,                      // valor recibido o enviado
5         primera_libre    = 0, // índice de primera celda libre
6         primera_ocupada   = 0, // índice de primera celda ocupada
7         num_celdas_ocupadas = 0, // número de celdas ocupadas
8
9         //COMPARO LA ETIQUETA A VER SI ES ACEPTABLE
10        etiqueta_emisor_aceptable ; // identificador de emisor aceptable
11
12
13    MPI_Status estado ;           // metadatos del mensaje recibido
14
15    for( unsigned int i=0 ; i < num_items*2 ; i++ )
16    {
17        // 1. determinar si puede enviar solo prod., solo cons, o todos
18
19        if ( num_celdas_ocupadas == 0 )           // si buffer vacío
20            etiqueta_emisor_aceptable = etiqueta_productor ; // $~~$ solo prod.
21        else if ( num_celdas_ocupadas == tam_vector ) // si buffer lleno
22            etiqueta_emisor_aceptable = etiqueta_consumidor ; // $~~$ solo cons.
23        else
24            // si no vacío ni lleno
25            etiqueta_emisor_aceptable = MPI_ANY_TAG ; // $~~$ cualquiera
26
27        // 2. recibir un mensaje del emisor o emisores aceptables
28
29        MPI_Recv( &valor, 1, MPI_INT, MPI_ANY_SOURCE,etiqueta_emisor_aceptable, MPI_COMM_WORLD, &estado );
30
31        // 3. procesar el mensaje recibido
32
33        switch( estado.MPI_TAG ) // leer emisor del mensaje en metadatos
34        {
35            case etiqueta_productor: // si ha sido el productor: insertar en buffer
36                buffer[primera_libre] = valor ;
37                primera_libre = (primera_libre+1) % tam_vector ;
38                num_celdas_ocupadas++ ;
39                cout << "Buffer ha recibido valor " << valor << endl ;
40                break;
41
42            case etiqueta_consumidor: // si ha sido el consumidor: extraer y enviarle
43                valor = buffer[primera_ocupada] ;
44                primera_ocupada = (primera_ocupada+1) % tam_vector ;
45                num_celdas_ocupadas-- ;
46                cout << "Buffer va a enviar valor " << valor << endl ;
47                MPI_Ssend( &valor, 1, MPI_INT, estado.MPI_SOURCE, etiqueta_consumidor, MPI_COMM_WORLD);
48                break;
49        }
50    }

```

```
1 void funcion_consumidor(int id)
2 {
3     int         peticion,
4         valor_rec = 1 ;
5     MPI_Status  estado ;
6
7     for( unsigned int i=0 ; i < num_items/num_consumidores; i++ )
8     {
9         MPI_Ssend( &peticion, 1, MPI_INT, id_buffer, etiqueta_consumidor, MPI_COMM_WORLD);
10        MPI_Recv ( &valor_rec, 1, MPI_INT, id_buffer, etiqueta_consumidor, MPI_COMM_WORLD,&estado );
11        cout << "Consumidor " << id << " ha recibido valor " << valor_rec << endl << flush ;
12        consumir( valor_rec,id );
13    }
14 }
```

```
1 int main( int argc, char *argv[] )
2 {
3     int id_propio, num_procesos_actual;
4
5     // inicializar MPI, leer identif. de proceso y número de procesos
6     MPI_Init( &argc, &argv );
7     MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
8     MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );
9
10    if ( num_procesos Esperado == num_procesos_actual )
11    {
12        // ejecutar la operación apropiada a 'id_propio'
13        if ( id_propio < num_productores )
14            funcion_productor(id_propio);
15        else if ( id_propio == id_buffer )
16            funcion_buffer();
17        else
18            funcion_consumidor(id_propio);
19    }
20    else
21    {
22        if ( id_propio == 0 ) // solo el primero escribe error, indep. del rol
23        { cout << "el número de procesos esperados es: " << num_procesos Esperado << endl
24          << "el número de procesos en ejecución es: " << num_procesos_actual << endl
25          << "(programa abortado)" << endl ;
26    }
27 }
28
29 // al terminar el proceso, finalizar MPI
30 MPI_Finalize( );
31 return 0;
32 }
```

Ejercicio 2: Filosofos

```
1 void funcion_filosofos(int id_propio){
2
3     // Determino cual es mi tenedor de la derecha y el de la izquierda
4     int id_tenedor_izquierda = (id_propio+1) % num_tenedores; // % es por que hay varios, no solo uno
5     int id_tenedor_derecha = (id_propio+num_tenedores-1) % num_tenedores;
6     int valor;
7
8     while (true){
9
10         //1. Toma los tenedores ( primero el izquierdo, luego el derecho)
11
12         // solucion al interbloqueo
13         if (id_propio == 0) {
14             cout <<"Filósofo " <<id_propio <<" solicita ten. der." <<id_tenedor_derecha <<endl;
15             MPI_Ssend(&valor, 1, MPI_INT, id_tenedor_derecha, 0, MPI_COMM_WORLD);
16
17             cout <<"Filósofo " <<id_propio << " solicita ten. izq." <<id_tenedor_izquierda <<endl;
18             MPI_Ssend(&valor, 1, MPI_INT, id_tenedor_izquierda, 0, MPI_COMM_WORLD);
19         }
20
21
22         cout <<"Filósofo " <<id_propio <<" solicita ten. izquierda " <<id_tenedor_izquierda <<endl;
23         MPI_Ssend(&valor, 1, MPI_INT, id_tenedor_izquierda, 0, MPI_COMM_WORLD);
24
25         cout <<"Filósofo " <<id_propio << " solicita ten. derecha" <<id_tenedor_derecha <<endl;
26         MPI_Ssend(&valor, 1, MPI_INT, id_tenedor_derecha, 0, MPI_COMM_WORLD);
27
28
29         //2. Come
30         come(id_propio);
31
32
33         //3. Soltar los tenedores
34         cout <<"Filósofo " <<id_propio <<" suelta ten. izquierda " <<id_tenedor_izquierda <<endl;
35         MPI_Ssend(&valor, 1, MPI_INT, id_tenedor_izquierda, 0, MPI_COMM_WORLD);
36
37         cout <<"Filósofo " <<id_propio <<" suelta ten. derecha " <<id_tenedor_derecha <<endl;
38         MPI_Ssend(&valor, 1, MPI_INT, id_tenedor_derecha, 0, MPI_COMM_WORLD);
39
40         //4. piensa
41         piensa(id_propio);
42     }
43 }
```

```
1 void funcion_tenedor(int id_propio){
2
3     int valor, id_filosofo;
4     MPI_Status estado;
5
6     while (true){
7
8         // 1. Espera hasta recibir un mensaje de un filosofo (mensaje petición)
9         // ANY_SOURCE -> DE CUALQUIERA
10        MPI_Recv(&valor,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&estado);
11        id_filosofo = estado.MPI_SOURCE;
12        cout << "El filosofo " << id_filosofo << "coge el tenedor " << id_propio << endl;
13
14        // 2. Espera hasta recibir un mensaje del mismo filosofo emisor del anterior (liberación)
15        MPI_Recv(&valor,1,MPI_INT,id_filosofo,0,MPI_COMM_WORLD,&estado);
16        cout << "El filosofo " << id_filosofo << "coge el tenedor " << id_propio << endl;
17    }
18 }
```

Ejercicio 3: Filosofo con camarero

```
1 void funcion_filosofos(int id_propio){
2
3     // Determino cual es mi tenedor de la derecha y el de la izquierda
4     int id_tenedor_izquierda = (id_propio+1) % num_tenedores; // % es por que hay varios, no solo uno
5     int id_tenedor_derecha = (id_propio+num_tenedores-1) % num_tenedores;
6     int valor;
7
8     while (true){
9
10         // 1. Se sienta
11         cout << "El filosofo " << id_propio << " llama al camarero para sentarse " << endl;
12         MPI_Ssend(&valor, 1, MPI_INT, id_camarero, etiqueta_sentarse, MPI_COMM_WORLD);
13
14         // 2. Toma tenedores
15         cout <<"Filósofo " <<id_propio <<" solicita ten. izquierda " <<id_tenedor_izquierda <<endl;
16         MPI_Ssend(&valor, 1, MPI_INT, id_tenedor_izquierda, 0, MPI_COMM_WORLD);
17
18         cout <<"Filósofo " <<id_propio << " solicita ten. derecha" <<id_tenedor_derecha <<endl;
19         MPI_Ssend(&valor, 1, MPI_INT, id_tenedor_derecha, 0, MPI_COMM_WORLD);
20
21         // 3. Come
22         come(id_propio);
23
24         // 4. Suelta tenedores
25         cout <<"Filósofo " <<id_propio <<" suelta ten. izquierda " <<id_tenedor_izquierda <<endl;
26         MPI_Ssend(&valor, 1, MPI_INT, id_tenedor_izquierda, 0, MPI_COMM_WORLD);
27
28         cout<< "Filósofo " <<id_propio <<" suelta ten. derecha " <<id_tenedor_derecha <<endl;
29         MPI_Ssend(&valor, 1, MPI_INT, id_tenedor_derecha, 0, MPI_COMM_WORLD);
30
31         // 5. Se levanta
32         cout << "El filosofo " << id_propio << " llama al camarero para levantarse " << endl;
33         MPI_Ssend(&valor, 1, MPI_INT, id_camarero, etiqueta_levantarse, MPI_COMM_WORLD);
34
35         // 6. Piensa
36         piensa(id_propio);
37     }
38 }
```

```
1 void funcion_tenedor(int id_propio){
2
3     int valor, id_filosofo;
4     MPI_Status estado;
5
6     while (true){
7
8         // 1. Espera hasta recibir un mensaje de un filosofo (mensaje peticion)
9         // ANY_SOURCE -> DE CUALQUIERA
10        MPI_Recv(&valor,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&estado);
11        id_filosofo = estado.MPI_SOURCE;
12        cout << "El filosofo " << id_filosofo << "coge el tenedor " << id_propio << endl;
13
14         // 2.Espera hasta recibir un mensaje del mismo filosofo emisor del anterior (liberacion)
15         MPI_Recv(&valor,1,MPI_INT,id_filosofo,0,MPI_COMM_WORLD,&estado);
16         cout << "El filosofo " << id_filosofo << "coge el tenedor " << id_propio << endl;
17     }
18 }
```

```
1 void funcion_camarero(){
2
3     const int maximo_filosofos_sentador= num_filosofos -1 ;// no puede haber 5 sentados
4     int num_sentados = 0; // contador
5
6     int valor;
7     int id_filosofo;
8     MPI_Status estado;
9
10    while(true){
11        if ( num_sentados < maximo_filosofos_sentador){
12
13            // Solo recibe mensajes aquellos que tengan el tag sentarse "esten sentados"
14            // y cuando num_sentados no sea 5
15            MPI_Recv(&valor, 1, MPI_INT, MPI_ANY_SOURCE, etiqueta_sentarse, MPI_COMM_WORLD, &estado);
16            id_filosofo = estado.MPI_SOURCE;
17
18            cout << "El filosofo " << id_filosofo << "se ha sentado " << endl;
19            num_sentados++;
20        }
21
22        // si no se sientan se levantan
23        // Solo pueden recibir mensajes que tengan el tag de levantarse y cuando puedan siempre
24        MPI_Recv(&valor, 1, MPI_INT, MPI_ANY_SOURCE, etiqueta_levantarse, MPI_COMM_WORLD, &estado);
25        cout << "El filosofo " << id_filosofo << " se ha levantado" << endl;
26        num_sentados--;
27    }
28
29 }
```

```
1 // -----
2 // main
3
4 int main( int argc, char ** argv){
5
6     int id_propio, num_procesos_actual;
7
8     // inicializar MPI, leer identif. de proceso y n mero de procesos
9     MPI_Init( &argc, &argv );
10    MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
11    MPI_Comm_size( MPI_COMM_WORLD, &num_procesos_actual );
12
13    if (num_procesos == num_procesos_actual){ // tiene que ser 10
14        // pares son filosofos
15
16        if (id_propio % 2 == 0){
17            // si es par
18            if (id_propio == id_camarero)
19                funcion_camarero();
20            else
21                funcion_filosofos(id_propio);
22        } else {
23            // si es impar
24            funcion_tenedor(id_propio);
25        }
26
27
28
29    } else if (id_propio == 0)
30        cerr << "Error: Se esperan 11 procesos " << endl;
31
32    MPI_Finalize();
33    return 0;
34 }
```