



Universidad Tecnológica Centroamericana

Sede: Campus Unitec San Pedro Sula, Honduras

Sistemas Operativos I

*Actividad: **Proyecto #2: Sincronización de Procesos***

Ing. Román Arturo Pineda

Sección: 1853

Estudiante:

Angella Fernanda Falck Duran – 21941027

22 de junio de 2022

Problema #1: El problema de la leche

Los hilos deben garantizar la consistencia. De lo contrario, los subprocesos pueden dar lugar a una condición de carrera que conduzca a un resultado no determinista. Una carrera es cuando el resultado depende del orden en que terminan los hilos.

Por ejemplo:

El hilo A dice $x = 1$, y si $(x == 2)$, entonces imprime '¿Qué?'

El hilo B dice $x = 2$.

¿Qué es x ?

Los hilos requieren operaciones de sincronización para que esto sea respondido.

Explicación del algoritmo:

El problema Demasiadas botellas de leche modela a dos compañeros de alojamiento que comparten un refrigerador y que, como buenos compañeros, se aseguran de que el refrigerador este siempre bien abastecido de leche. Con compañeros de habitación tan responsables, el siguiente escenario es posible:

	Person A	Person B
3:00	Look in fridge: no milk	
3:05	Leave for store	
3:10	Arrive at store	Look in fridge: no milk
3:15	Leave store	Leave home
3:20	Arrive home, put milk away	Arrive at store
3:25		Leave store
3:30		Arrive home: too much milk!

El problema surge porque se deben cumplir las siguientes propiedades:

- Seguridad. Nunca más de una persona compra leche.
- Vivacidad. Si se necesita leche, eventualmente alguien la compra.

Código

```
#define THREADS 1

sem_t OKToBuyMilk;
int milkAvailable;

void* buyer(void *arg)
{
    sem_wait(&OKToBuyMilk); //lock thread

    if(milkAvailable>1){
        printf("There is too much milk\n");
    }
    else if(milkAvailable==1) //enough milk
    {
        printf("There is already enough milk\n");
    }else if(milkAvailable<1)
    {
        printf("There is not enough milk, buy milk\n");
    }else if(milkAvailable<2)
    {
        // Buy some milk
        milkAvailable++;
    }

    sem_post(&OKToBuyMilk); //unlock thread

    return NULL;
}
```

```
int main(int argc, char **argv)
{
    pthread_t threads[THREADS];

    milkAvailable = 3;

    /*Inicializar el semaforo en 1, semaforo between threads
    y no procesos*/
    if(sem_init(&OKToBuyMilk, 0, 1))
    {
        //printf("Could not initialize a semaphore\n");
        return -1;
    }

    //Crear threads
    for(int i = 0; i < THREADS; ++i)
    {
        if(pthread_create(&threads[i], NULL, &buyer, NULL))
        {
            //printf("Could not create thread %d\n", i);
            return -1;
        }
    }

    //Join threads
    for(int i = 0; i < THREADS; ++i)
    {
        if(pthread_join(threads[i], NULL))
        {
            //printf("Could not join thread %d\n", i);
            return -1;
        }
    }

    sem_destroy(&OKToBuyMilk);

    // Make sure we don't have too much milk.
    printf("Total milk: %d\n", milkAvailable);
}
```

Explicación código:

En este algoritmo podemos observar la manera en la que funciona el concepto de concurrencia:

Dentro del código, por medio de un enum podemos definir la cantidad de hilos/threads disponibles para nuestro buffer, el cual estaría trabajando en el problema, que en este caso es tener suficiente leche. "Too much milk" o demasiada leche, en español, se refiere a tener más que la necesaria (1 cartón de leche).

Dentro de este ejemplo podemos observar como el hecho de tener más trabajo que hilos puede crear un cuello de botella por falta de hilos para resolver el problema, y viceversa, si tenemos más hilos que trabajo estos intentarían trabajar en el mismo problema. Se imprime varias veces por esta misma razón.

Simulación para 1 thread

Caso#1: Hay Suficiente Leche

```
There is already enough milk
Total milk: 1

...Program finished with exit code 0
```

Caso#2: Hay Mucha Leche

```
There is too much milk
Total milk: 4

...Program finished with exit code 0
```

Caso #3: No Hay Leche

```
There is not enough milk, buy milk  
Total milk: 0  
  
...Program finished with exit code 0
```

Simulación para 2 threads

Caso#1: Prueba

Threads: 2, milkAvailable:1

```
There is already enough milk  
There is already enough milk  
Total milk: 1
```

Simulación para n threads - Utilizando 5

Caso #2: Prueba

Threads: 4, milkAvailable:3

```
There is too much milk  
There is too much milk  
There is too much milk  
There is too much milk  
Total milk: 3
```

Problema #2: Productores y Consumidores

En computación, el problema del productor-consumidor es un ejemplo clásico de problema de sincronización de multiprocesos.

Explicación del algoritmo:

El programa describe dos procesos, productor y consumidor, ambos comparten un buffer de tamaño finito. La tarea del productor es generar un producto, almacenarlo y comenzar nuevamente; mientras que el consumidor toma (simultáneamente) productos uno a uno. El problema consiste en que el productor no añada más productos que la capacidad del buffer y que el consumidor no intente tomar un producto si el buffer está vacío.

Solución:

La idea para la solución es la siguiente, ambos procesos (productor y consumidor) se ejecutan simultáneamente y se “despiertan” o “duermen” según el estado del buffer. Concretamente, el productor agrega productos mientras quede espacio y en el momento en que se llene el buffer se pone a “dormir”. Cuando el consumidor toma un producto notifica al productor que puede comenzar a trabajar nuevamente. En caso contrario, si el buffer se vacía, el consumidor se pone a dormir y en el momento en que el productor agrega un producto crea una señal para despertarlo. Se puede encontrar una solución usando mecanismos de comunicación de interprocesos, generalmente se usan semáforos. Una inadecuada implementación del problema puede terminar en un deadlock, donde ambos procesos queden en espera de ser despertados.

Código

```
16 int buffer = 0; //variable global contador del buffer
17 int buffersize = 7; //tamaño del buffer
18
19 void * productor()
20 {
21     int countdown = 10;
22
23     while(countdown > 0)
24     {
25         countdown--;
26
27         // Producir
28         printf("Producir -> Buffer en %d\n", buffer);
29
30         sem_wait(&c); //lock semaphore buffer (produzco)
31         sem_wait(&a); //lock semaphore
32
33         // Agregar
34         buffer++; //sumo al buffer
35         printf("Agregar -> Buffer en %d\n", buffer);
36         sem_post(&a); //unlock semaphore
37         sem_post(&b); //unlock semaphore
38
39         sleep(3);
40     }
41 }
```

```
void * consumidor()
{
    int m;
    int countdown = 10;

    while(countdown > 0)
    {
        sem_wait(&b); //lock semaphore
        sem_wait(&a); //lock semaphore

        // Agarrar
        buffer--;
        printf("Agarrar -> Buffer en %d\n", buffer);
        sem_post(&a); //unlock semaphore
        sem_post(&c); //unlock semaphore
        sleep(4);

        // Consumir
        printf("Consumir -> Buffer en %d\n", buffer);
        sleep(3);
    }
}
```

```

~ int main()
{
    //Inicializar ambos threads
    pthread_t thread1;
    pthread_t thread2;
    ~ //pthread_t thread3;
    ~ //pthread_t thread4;

    ~ //Solución realizada con semaforos
    //0 por que es compartido entre procesos
    sem_init(&a,0,1); //binary semaphore
    sem_init(&b,0,0); //binary semaphore
    sem_init(&c,0,bufferSize); //counting semaphore (contador, bufferSize)

    //Crear los threads de productor y consumidor
    pthread_create(&thread1,NULL,productor,NULL);
    //pthread_create(&thread3,NULL,productor,NULL);
    pthread_create(&thread2,NULL,consumidor,NULL);
    //pthread_create(&thread4,NULL,consumidor,NULL);

    sleep(60);
}

```

Explicación código:

Hay uno o más productores que generan datos y estos los colocan en un búfer. Solo existe un solo consumidor que un elemento a la vez del búfer. Para la solución de este problema, el sistema es limitado para prevenir la superposición de operaciones del búfer. En otras palabras, solo un productor o consumidor puede acceder al búfer al mismo tiempo. En este problema se está previendo tener un conflicto debido a tener múltiples productores, de esta manera evitamos que los 2 intenten escribir al mismo tiempo o insertar el objeto en el mismo espacio.

Simulación para 1 thread

Si se elimina 1 thread, por ejemplo, el de productores, el programa solo produciría y agregaría datos al buffer, esto no haría que se implementara el algoritmo de consumidores y productores correctamente, así como lo establece el problema.


```
afalck@Angellas-MacBook-Air Proyecto2_Siso1_AF % ./Algoritmo_PC
Producir -> Buffer en 0
Agregar -> Buffer en 1
Producir -> Buffer en 1
Agregar -> Buffer en 2
Producir -> Buffer en 2
Agregar -> Buffer en 3
Producir -> Buffer en 3
Agregar -> Buffer en 4
Producir -> Buffer en 4
Agregar -> Buffer en 5
Producir -> Buffer en 5
Agregar -> Buffer en 6
Producir -> Buffer en 6
Agregar -> Buffer en 7
Producir -> Buffer en 7
Agregar -> Buffer en 8
Producir -> Buffer en 8
Agregar -> Buffer en 9
Producir -> Buffer en 9
Agregar -> Buffer en 10
```

Lo mismo pasaría si solo trabajamos con el thread de consumidores, el programa solo agarraría y consumiría datos.

```
afalck@Angellas-MacBook-Air Proyecto2_Siso1_AF % ./Algoritmo_PC
Agarrar -> Buffer en -1
Consumir -> Buffer en -1
Agarrar -> Buffer en -2
Consumir -> Buffer en -2
Agarrar -> Buffer en -3
Consumir -> Buffer en -3
Agarrar -> Buffer en -4
Consumir -> Buffer en -4
```

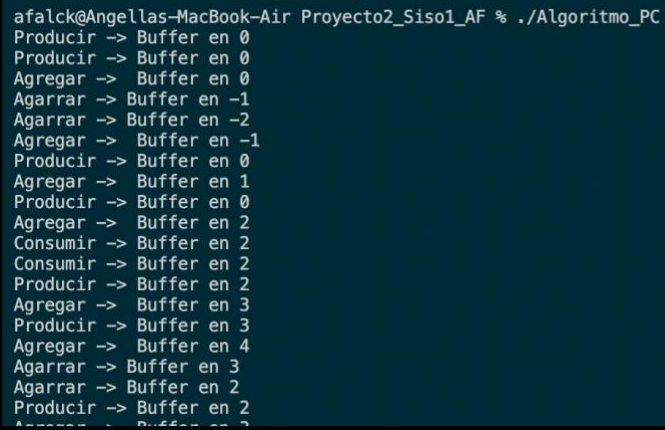
Simulación para 2 threads

```

Agregar -> Buffer en 1
Agarrar -> Buffer en 0
Producir -> Buffer en 0
Agregar -> Buffer en 1
Consumir -> Buffer en 1
Producir -> Buffer en 1
Agregar -> Buffer en 2
Agarrar -> Buffer en 1
Producir -> Buffer en 1
Agregar -> Buffer en 2
Consumir -> Buffer en 2
Producir -> Buffer en 2
Agregar -> Buffer en 3
Agarrar -> Buffer en 2
Producir -> Buffer en 2
Agregar -> Buffer en 3
Producir -> Buffer en 3
Agregar -> Buffer en 4
Consumir -> Buffer en 4
Producir -> Buffer en 4
Agregar -> Buffer en 5
Agarrar -> Buffer en 4
Producir -> Buffer en 4
Agregar -> Buffer en 5
Consumir -> Buffer en 5
Producir -> Buffer en 5
Agregar -> Buffer en 6
Agarrar -> Buffer en 5
Consumir -> Buffer en 5
Agarrar -> Buffer en 4
Consumir -> Buffer en 4
Agarrar -> Buffer en 3
Consumir -> Buffer en 3
Agarrar -> Buffer en 2
Consumir -> Buffer en 2
Agarrar -> Buffer en 1

```

Simulación para n threads



```
afalck@Angellas-MacBook-Air Proyecto2_Siso1_AF % ./Algoritmo_PC
Producir -> Buffer en 0
Producir -> Buffer en 0
Agregar -> Buffer en 0
Agarrar -> Buffer en -1
Agarrar -> Buffer en -2
Agregar -> Buffer en -1
Producir -> Buffer en 0
Agregar -> Buffer en 1
Producir -> Buffer en 0
Agregar -> Buffer en 2
Consumir -> Buffer en 2
Consumir -> Buffer en 2
Producir -> Buffer en 2
Agregar -> Buffer en 3
Producir -> Buffer en 3
Agregar -> Buffer en 4
Agarrar -> Buffer en 3
Agarrar -> Buffer en 2
Producir -> Buffer en 2
Agregar -> Buffer en 2
```

La solución anterior funciona perfectamente cuando hay uno o más productores y un consumidor. Cuando múltiples consumidores comparten el mismo espacio de memoria para almacenar el buffer la solución anterior puede llevar a resultados donde dos o más procesos lean o escriban la misma región al mismo tiempo. Por ende, la simulación es apta solamente para 2 threads.

Cada semáforo funciona como un administrador de cierta función que es posible ejercer sobre nuestro buffer.

Tenemos un administrador que suma (crea espacio en el buffer o espacio al sacar un objeto), y tenemos un administrador que resta (que indica cuantos espacios disponibles quedan dentro de nuestro buffer o espacio).

Nuestro tercer semáforo es un administrador, pero no de nuestro buffer o espacio, sino de nuestros tipos semáforos o administradores, y la manera en la que funciona es que este se asegura de que en caso de que tengamos dos administradores de suma o de resta, estos no intenten actuar sobre el mismo lugar dentro de nuestro buffer ya sea para sumar o restar, así no intentan ejercer sobre el mismo lugar simultáneamente

Cuando hay múltiples productores:

```
Producir -> Buffer en 0
Agregar -> Buffer en 1
Producir -> Buffer en 1
Agregar -> Buffer en 2
Agarrar -> Buffer en 1
Producir -> Buffer en 1
Producir -> Buffer en 1
Agregar -> Buffer en 2
Agregar -> Buffer en 3
Consumir -> Buffer en 3
Producir -> Buffer en 3
Agregar -> Buffer en 4
Producir -> Buffer en 4
Agregar -> Buffer en 5
Agarrar -> Buffer en 4
Producir -> Buffer en 4
Agregar -> Buffer en 5
Producir -> Buffer en 5
Agregar -> Buffer en 6
Consumir -> Buffer en 6
Producir -> Buffer en 6
Agregar -> Buffer en 7
Producir -> Buffer en 6
Agarrar -> Buffer en 6
Agregar -> Buffer en 7
Producir -> Buffer en 7
```

Problema #3: Algoritmo de Peterson

El algoritmo de Peterson, también conocido como solución de Peterson,¹ es un algoritmo de programación concurrente para exclusión mutua, que permite a dos o más procesos o hilos de ejecución compartir un recurso sin conflictos, utilizando sólo memoria compartida para la comunicación.

Explicación del algoritmo:

Mutex, o exclusión mutua, es un objeto en un programa que impide simultáneamente que varias personas accedan al mismo recurso.

- Durante la programación concurrente, las secciones críticas se utilizan para acceder a un recurso compartido por procesos o subprocesos.
- Solo un subproceso puede poseer el mutex durante el inicio del programa, por lo que se le asigna un nombre único.
- Si un subproceso tiene un recurso, debe bloquear el mutex de otros subprocesos para que otros subprocesos no puedan acceder a él simultáneamente. El subproceso desbloquea el mutex después de liberar el recurso.
- Los subprocesos múltiples entran en juego cuando dos subprocesos trabajan en los mismos datos simultáneamente. Como herramienta de sincronización, actúa como un candado.
- Si hay un mutex disponible, un subproceso puede adquirirlo; de lo contrario, se pone a dormir.

El algoritmo de Peterson se limita a dos procesos que se ejecutan alternativamente entre secciones críticas. Llamaremos a estos procesos P_i y P_j .

Solución:

La solución de Peterson necesita que se compartan dos elementos de datos entre los dos procesos:

- turno int: Indica su turno para entrar en su tramo crítico.
- Flag booleana[2]: Indica si un proceso está listo para entrar en su sección crítica. Da resultados como verdadero o falso.

Código

```
#define VALUE 100 //valor que desee el usuario
int flag[2];
int turn, val = 0;

//Funcion para saber que thread entro
/*
Aqui se reserva el valor para el cual cada subproceso intentara bloquear
el algoritmo, al incrementar el valor ingresado por el usuario y luego
desbloquear el subproceso.

Se ejecutara el doble de veces por que es independiente del tiempo
y tomara algunas iteraciones para solucionar el problema.
*/
void *work(void *s)
{
    int i = 0;
    int other = (int *)s;

    printf("Thread : %d\n", other);
    lock(other);

    for (i = 0; i < VALUE; i++) //
        val++;

    unlock(other);
}

void lock_init()
{
    // Se resetean, reseteando su preferencia a locks adquiridos.
    //esto se realiza dandoles un turno
    flag[0] = flag[1] = 0;
    turn = 0;
}
```

```
//Funcion para lock el valor de la bandera como 1
void lock(int other)
{
    //Esperar a que se ejecute el subproceso y cambiar el valor de la bandera
    flag[other] = 1; //Thread quiere entrar a la seccion critica
    turn = 1 - other; // Que otro thread tiene prioridad
    while (flag[1 - other] == 1 && turn == 1 - other);
}

//Funcion para desbloquear el valor de la bandera
//Marcar el hilo que ya no se tiene que ejecutar en la sección critica
void unlock(int other)
{
    flag[other] = 0; // Marking that this thread is no longer wants to enter the critical section
}

//Inicilizar valor entero en 0
//Crear los subprocesos y se unan para imprimir el valor final
void main()
{
    pthread_t t1, t2;
    val = 0; // valor compartido

    lock_init();

    //Crear los threads
    pthread_create(&t1, NULL, work, (void *)0);
    pthread_create(&t2, NULL, work, (void *)1);

    //Join threads
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    // Printing the result
    printf("Valor final -> %d\n", val);
}
```

Explicación código:

Lo que representa el algoritmo de Peterson es que desbloquear el subproceso, ejecutar la acción, y volver a bloquear el subproceso siempre va a tardar el doble de tiempo que tomaría contar hasta x (realizar un proceso cualquiera).

Se crea una clase para implementar el algoritmo de Peterson. Dentro de la clase, existen dos variables: flag y turn. Después de eso, implementé dos funciones, una para bloquear el valor de la bandera (lock) como 1 y la otra para desbloquear (unlock) el valor de la bandera.

En la función de bloqueo (lock), cambie la prioridad del subproceso, espere a que se ejecute el subproceso y cambie el valor de la bandera. En la función de desbloqueo (unlock), marque el hilo que ya no tiene que ejecutarse en la sección crítica.

Ahora cree una función que reserve el valor para el cual cada subproceso intentará bloquear el algoritmo de Peterson, incrementar un valor entero ingresado por el usuario y luego desbloquear el subproceso. El proceso anterior se ejecutará el doble de x (dependiendo el valor) porque es muy independiente del tiempo y puede tomar algunas iteraciones para solucionar este problema.

El proceso de asegurar la exclusión mutua puede ser realizado por dos procesos simultáneamente. Ambos procesos crean e inicializan variables compartidas antes de comenzar. Ninguno de los procesos está actualmente interesado en la sección crítica, por lo que los indicadores [0] y [1] se establecen en FALSO.

Simulación para 1 thread

```
afalck@Angellas-MacBook-Air Proyecto2_Siso1_AF % ./Algoritmo_Peterson
Thread : 0
Valor final -> 100
afalck@Angellas-MacBook-Air Proyecto2_Siso1_AF % █
```

El valor final será igual al valor inicial, ya que solo hay un thread, por ende no se bloqueara porque solo existe 1 subproceso.

Simulación para 2 threads

```
afalck@Angellas-MacBook-Air Proyecto2_Siso1_AF % ./Algoritmo_Peterson
Thread : 0
Thread : 1
Valor final -> 200
```

Simulación para n threads - Utilizando 3 (no funciona correctamente)

```
afalck@Angellas-MacBook-Air Proyecto2_Siso1_AF % ./Algoritmo_Peterson
Thread : 0
Thread : 1
Thread : 2
Valor final -> 300
afalck@Angellas-MacBook-Air Proyecto2_Siso1_AF % █
```

Problema #4: Algoritmo de Dekker

El algoritmo de Dekker es un algoritmo de programación concurrente para exclusión mutua, que permite a dos procesos o hilos de ejecución compartir un recurso sin conflictos. Fue uno de los primeros algoritmos de exclusión mutua inventados, implementado por Edsger Dijkstra.

Explicación del algoritmo:

El algoritmo de Dekker fue la primera solución probablemente correcta al problema de la sección crítica. Permite que dos subprocesos compartan un recurso de un solo uso sin conflicto, utilizando solo la memoria compartida para la comunicación. Evita la alternancia estricta de un ingenuo algoritmo de toma de turnos y fue uno de los primeros algoritmos de exclusión mutua que se inventaron.

Código

```
//las flags son coordenadas, turn indica el turno
int turn=0, flag[2]={0,0}, balance=0;

void test(int i){
    int j,k,m,p,q,r,c;
    j=1-i;

    for(k=0;k<3;k++){

        flag[i]=1;

        while(flag[j]){
            if (turn==j){
                flag[i]=0;
                printf(" Esperando: %d",i);
                while(turn==j);
                flag[i]=1;
            }
        }

        printf("\nEntrando a sección crítica %d",i);
        c=balance;
        /*critical section*/
        printf("\n proceso: %d",i);
        printf(" -> flag[%d]=%d, flag[%d]=%d",i,flag[i],j,flag[j]);
        printf(" turno de= %d\n",turn);
        balance=c+1000; //tiempo
        printf(" balance= %d",balance);
        turn=j;
        flag[i]=0;
        printf("\nSaliendo.. %d",i);
    }
}
```



```

int main(){

    pthread_t t1,t2;
    pthread_create(&t1,NULL,(void*)&test,(void*)0);
    printf("Crear el primer thread..\n");
    pthread_create(&t2,NULL,(void*)&test,(void*)1);
    printf("Crear el segundo thread..\n");
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);

    printf("\nTerminado: %d",balance);

    return(0);
}

```

Explicación código:

Más que mostrar la ejecución de la acción con las banderas (flags), lo que el algoritmo nos quiere demostrar es la concurrencia del programa, porque a pesar de que nuestra única variable inicializada es *j*, la inicialización completa de la misma depende del valor de *i*, que es nuestro proceso más pequeño si nos guiamos de la jerarquía de nuestros ciclos *for*.

El ejercicio nos quiere comprobar que a pesar de ser dependiente el uno del otro, es que la declaración de *j* siempre va a tomar *x* cantidad de tiempo, y todo esto es posible debido a el uso de memoria compartida, la cual asegura que nuestro proceso, a pesar de tardarse un tiempo '*x*', será realizado dentro de un margen controlado que no comprometerá la ejecución de nuestro proceso o programa. Aquí si aplica la exclusión mutua, todavía está en juego podemos decir por el hecho de que el tiempo que vamos a esperar sigue estando dentro de un margen aceptable. Sin embargo, este algoritmo solo puede manejar un máximo de 2 procesos.

Simulación para 1 thread

```
Crear el primer thread..

Entrando a sección critica 0
  proceso: 0 -> flag[0]=1, flag[1]=0 turno de= 0
  balance= 1000
Saliendo.. 0
Entrando a sección critica 0
  proceso: 0 -> flag[0]=1, flag[1]=0 turno de= 1
  balance= 2000
Saliendo.. 0
Entrando a sección critica 0
  proceso: 0 -> flag[0]=1, flag[1]=0 turno de= 1
  balance= 3000
Saliendo.. 0
Terminado: 3000
```

Simulación para 2 threads

```
Crear el primer thread..

Entrando a sección critica 0
  proceso: 0 -> flag[0]=1, flag[1]=0 turno de= 0
  balance= 1000
Saliendo.. 0
Entrando a sección critica 0
  proceso: 0 -> flag[0]=1, flag[1]=0 turno de= 1
  balance= 2000
Saliendo.. 0
Entrando a sección critica 0
  proceso: 0 -> flag[0]=1, flag[1]=0 turno de= 1
  balance= 3000
Saliendo.. 0
Crear el segundo thread..

Entrando a sección critica 1
  proceso: 1 -> flag[1]=1, flag[0]=0 turno de= 1
  balance= 4000
Saliendo.. 1
Entrando a sección critica 1
  proceso: 1 -> flag[1]=1, flag[0]=0 turno de= 0
  balance= 5000
Saliendo.. 1
Entrando a sección critica 1
  proceso: 1 -> flag[1]=1, flag[0]=0 turno de= 0
  balance= 6000
Saliendo.. 1
Terminado: 6000
```

Simulación para n threads

```
Crear el primer thread..  
Crear el segundo thread..  
Crear el tercer thread..  
  
Entrando a sección critica 1  
  proceso: 1 -> flag[1]=1, flag[0]=0 turno de= 0  
  balance= 1000  
Saliendo.. 1  
Entrando a sección critica 1  
  proceso: 1 -> flag[1]=1, flag[0]=0 turno de= 0  
  balance= 2000  
Saliendo.. 1  
Entrando a sección critica 1  
  proceso: 1 -> flag[1]=1, flag[0]=0 turno de= 0  
  balance= 3000  
Saliendo.. 1  
Entrando a sección critica 1  
  proceso: 1 -> flag[1]=1, flag[0]=0 turno de= 0  
  balance= 4000  
Saliendo.. 1  
Entrando a sección critica 1  
  proceso: 1 -> flag[1]=1, flag[0]=0 turno de= 0  
  balance= 5000  
Saliendo.. 1
```

No se maneja correctamente

Problema #5: Algoritmo de la Panadería

El algoritmo de la panadería de Lamport es un algoritmo de computación creado por el científico en computación Lord Leslie Lamport, para implementar la exclusión mutua de N procesos o hilos de ejecución.

Explicación del algoritmo:

Todos los subprocesos del proceso deben tomar un número y esperar su turno para usar un recurso informático compartido o para ingresar a su sección crítica. El número puede ser cualquiera de las variables globales, y los procesos con el número más bajo se procesarán primero.

Solución:

En la vida real, el sistema de los boletos funciona perfectamente, pero en un sistema informático la obtención del boleto es problemática: varios hilos pueden obtener el mismo número de turno.

En el algoritmo de Lamport se permite que varios hilos obtengan el mismo número. En este caso, se aplica un algoritmo de desempate, que garantiza que sólo un hilo entra en sección crítica. El desempate se realiza así: si dos o más hilos tienen el mismo número de turno, tiene más prioridad el hilo que tenga el identificador con un número más bajo.

Como no puede haber dos hilos con el mismo identificador, nunca se da el caso de que dos hilos evalúen al mismo tiempo que tienen derecho a ejecutar su sección crítica.

Código

```
#include "pthread.h" //Libreria pthread
#include "stdio.h"
#include "string.h"

#define Q_THREADS 1 //cantidad de threads, dependiendo el valor a considerar 1,2 o n threads
//std::thread thread_object(callable);
int num[Q_THREADS];
int selecting[Q_THREADS];
int resource;

void lock_thread(int threads)
{
    //Antes de agarrar el ticket, selecting es true (1)
    selecting[threads] = 1;
    int tkt_max = 0;
    // Encontrar el valor maximo de tickets en los threads
    for (int i = 0; i < Q_THREADS; i++)
    {
        int ticket = num[i];
        tkt_max = ticket > tkt_max ? ticket : tkt_max;
    }
    // Asignando nuevos valores como maximo+1 (valor maximo encontrado)
    num[threads] = tkt_max + 1;
    selecting[threads] = 0;

    //Comienza la sección de entrada
    for (int j = 0; j < Q_THREADS; j++)
    {
        // Aplicar las condiciones del algoritmo
        while (selecting[j])
        {
        }
        while (num[j] != 0 && (num[j] < num[threads] || (num[j] == num[threads] && j < threads)))
        {
        }
    }
}
```

```
//Termina la sección
void unlock_thread(int threads)
{
    num[threads] = 0; //setear a 0
}

// Sección critica (validando los recursos habilitados) que una persona no colisione teniendo el mismo ticket
void use_resource(int threads)
{
    if (resource != 0)
    {
        printf("El recurso lo tiene %d, pero esta en uso por: %d!\n", threads, resource);
    }
    resource = threads;
    printf("El thread: %d está usando el recurso\n", threads);

    resource = 0;
}

//Simplified function to show the implementation
void *thread_body(void *arg)
{
    //Pasos, se bloquea, se usa el recurso y se desbloquea el thread
    long thread = (long)arg;
    lock_thread(thread);
    use_resource(thread);
    unlock_thread(thread);
    return NULL;
}
```

```

int main(int argc, char **argv)
{
    //Setear memoria a 0
    memset((void *)num, 0, sizeof(num));
    memset((void *)selecting, 0, sizeof(selecting));
    resource = 0;

    // Variable thread
    pthread_t threads[Q_THREADS];

    for (int i = 0; i < Q_THREADS; ++i)
    {
        // Se crea un nuevo thread para utilizar la funcion
        pthread_create(&threads[i], NULL, &thread_body, (void *)((long)i));
        // "thread_body" como la rutina del thread (los pasos que se realizaran)
    }
    for (int i = 0; i < Q_THREADS; ++i)
    {
        // los threads se unen una vez el task haya terminado
        pthread_join(threads[i], NULL);
    }

    return 0;
}

```

Explicación código:

Para comenzar, en la función de lock_thread, el proceso establece su variable de "selección" en verdadero, lo que significa su intención de ingresar a la sección crítica.

Luego se le asigna el número de ticket más alto relacionado con otros procesos. La variable "seleccionar" se establece entonces en falso, lo que indica que ahora tiene un nuevo número de boleto. Con esto se propone cambiar el valor del boleto, pero no se debe permitir que otro proceso verifique el valor del boleto anterior, ya que este ya no está disponible. Por eso antes de ver el valor del ticket dentro del ciclo for, se asegura que todos los procesos tengan la variable "selección" en falso.

Después, se examinan los valores del ticket de los procesos, siempre asegurándose que el proceso con el num de ticket de proceso mas bajo este incluido en la sección critica como se explicaba en el algoritmo anteriormente. El valor del boleto al final se asignará como 0 en la sección de salida.

Simulación para 1 thread

```
afalck@Angellas-MacBook-Air Proyecto2_Siso1_AF % ./Algoritmo_Panaderia
El thread: 0 está usando el recurso
afalck@Angellas-MacBook-Air Proyecto2_Siso1_AF % █
```

Solo hay un thread simulando.

Simulación para 2 threads

```
afalck@Angellas-MacBook-Air Proyecto2_Siso1_AF % ./Algoritmo_Panaderia
El thread: 0 está usando el recurso
El thread: 1 está usando el recurso
afalck@Angellas-MacBook-Air Proyecto2_Siso1_AF % █
```

Solo hay dos thread simulando.

Simulación para n threads - Utilizando 10

```
afalck@Angellas-MacBook-Air Proyecto2_Siso1_AF % ./Algoritmo_Panaderia
El thread: 0 está usando el recurso
El thread: 1 está usando el recurso
El thread: 2 está usando el recurso
El thread: 4 está usando el recurso
El thread: 3 está usando el recurso
El thread: 5 está usando el recurso
El thread: 6 está usando el recurso
El thread: 7 está usando el recurso
El thread: 8 está usando el recurso
El thread: 9 está usando el recurso
```