EXERCISE 1

Use mutex and condition variables to avoid race conditions.
Write a program that creates 10 threads. Each one of them calculates the value of the PI number using the Monte Carlo method and stores it in its corresponding position in an array. When all the threads have finished the main program calculates the average of the PI values stored in the array
Mutex and condition variables must be used so that there are no Race problems.

SOLUTION

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <math.h>

#define RADIO 5000
#define PUNTOS 1000000

//Variable global compartida por todos los threads, incluido el
int main
float valoresPIthreads[10];
pthread_mutex_t mtx;
pthread_cond_t varcond;
int yacopiada=0;

void *calcula_pi (void *kk);

int main() {
        pthread_attr_t attr;
        pthread_t thread[10];
        int i;
        float *valorpi=0, suma=0, media=0;

        pthread_cond_init (&varcond, NULL);
        pthread_mutex_init (&mtx, NULL);

        pthread_attr_init(&attr);
        for (i=0;i<10;i++) {
          pthread_create(&thread[i],&attr,calcula_pi,&i);
          //Cambiamos el sleep de un ejemplo anterior por la
espera
          pthread_mutex_lock(&mtx);
            while (yacopiada==0) pthread_cond_wait (&varcond,
&mtx);
            yacopiada=0;
          pthread_mutex_unlock(&mtx);
          printf ("Creado thread %d\n",i);
        }
        for (i=0;i<10;i++) {
                pthread_join(thread[i],NULL);
```

```
        }
        for (i=0;i<10;i++) {
                printf("Valor        del        thread        %d:
%f\n",i,valoresPIthreads[i]);
                suma=suma+valoresPIthreads[i];
        }
        media=suma/10.0;
        printf("El valor medio de Pi obtenido es: %f\n",media);

}


void *calcula_pi (void *idthread)
{
        int j, y=0, x=0, cont=0,numthread;
        float pi=0, h=0;

        pthread_mutex_lock (&mtx);
        numthread=*((int *)idthread);
        yacopiada=1;
        pthread_cond_signal (&varcond);
        pthread_mutex_unlock (&mtx);
        printf ("Inicio th %d\n", numthread);
        srandom((unsigned)pthread_self());
        for (j=0;j<PUNTOS;j++) {
                y=(random()%((2*RADIO)+1)-RADIO);
                x=(random()%((2*RADIO)+1)-RADIO);
                h=sqrt((x*x)+(y*y));
                if ( h<=RADIO ) cont++;
        }
        valoresPIthreads[numthread]=(cont*4)/(float)PUNTOS;
        pthread_exit(&pi);

}
```

EXERCISE 2

The following code implements an application with two threads: one prints the even numbers on the screen and the other prints the odd numbers on the screen.

```
#include <pthread.h>
#include <stdio.h>

int dato_compartido = 0;


void pares(void)
{   int i;
    for(i=0; i < 100; i++ )
```

```
        printf("Thread1 = %d \n", dato_compartido++);
}
void impares(void)
{   int i;
    for(i=0; i < 100; i++ )
        printf("Thread2 = %d \n", dato_compartido++);
}

int main(void)
{
     pthread_t th1, th2;
     pthread_create(&th1, NULL, pares, NULL);
     pthread_create(&th2, NULL, impares, NULL);
     pthread_join(th1, NULL);
     pthread_join(th2, NULL);
}
```

The program must have the following output:

```
Thread1 = 0
Thread2 = 1
Thread1 = 2
Thread2 = 3
Thread1 = 4
Thread2 = 5
Thread1 = 6
.....
```

A first program execution shows the following:

```
Thread1 = 0
Thread1 = 1
Thread1 = 2
Thread1 = 3
Thread2 = 3
Thread2 = 4
Thread2 = 5
.....
```

You are requested to resolve the following sections:

1. Indicate what problems are generated by using a shared variable to send the data to print from the even thread to the odd thread.
2. Implement a version of the previous program that solves the previous problems using some of the concurrency management techniques.

1.

SOLUTION

1.- Race problems: Occurs when two processes access shared variables (simultaneously or in the wrong order) in such a way that the value of the variables is no longer consistent with the program logic. In this example, both the even Thread and the odd Thread only have a shared variable that each accesses on a single line (the even Thread prints the variable and the odd Thread also). If each of these lines were atomic (once passed to machine code) and there is only one CPU, then there would be no danger of running (first one would be printed and the shared_data variable would be increased and then the other would be printed), otherwise , that is, in the one shown in the example it could happen that a value of the variable was repeated on the screen:

Thread1 = 3
Thread2 = 3

Since a context switch occurred before the variable was incremented. To avoid this problem, it is better to convert these lines into mutual exclusion zones with some of the techniques seen (semaphores or mutex).

Progress problems: The proposed code shows two threads of execution. But there is nothing to ensure in what order they will be executed. If the scheduler follows a batch policy, it would first execute one and then another, so the latter could not progress until the first finishes (which could not happen if it were an infinite loop). This problem should be avoided by explicitly indicating in both processes when they should leave the CPU so that the other process can progress (using semaphores or conditions).

<u>Espera acotada</u>: It is not enough for the proposed code to ensure that no process is going to stop, but it requires an iteration of the even Thread to alternate with another iteration of odd Thread. (Each process must have a limited wait to achieve an iteration of the other process). To achieve this, it is necessary to use one of the techniques seen (using semaphores or conditions).

2.-Solution with semaphores

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>


int dato_compartido = 0;
sem_t par,impar;


void pares(void)
{   int i;
    for(i=0; i < 100; i++ ) {
         sem_wait(&par);
         printf("Thread1 = %d \n", dato_compartido++);
         sem_post(&impar);
```

```
    }
}

void impares(void)
{   int i;
    for(i=0; i < 100; i++ ) {
        sem_wait(&impar);
            printf("Thread2 = %d \n", dato_compartido++);
            sem_post(&par);
    }
}

int main(void) {
     pthread_t th1, th2;
     sem_init(&par,0,1);
     sem_init(&impar,0,0);
     pthread_create(&th1, NULL, pares, NULL);
     pthread_create(&th2, NULL, impares, NULL);
     pthread_join(th1, NULL);
     pthread_join(th2, NULL);
     sem_destroy(&par);
     sem_destroy(&impar);
}
```

Solution with mutex and conditions:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

int dato_compartido = 0;
int es_par = 0;
pthread_mutex_t m;
pthread_cond_t cL, cV;

void pares(void)
{
        int i;
        for(i=0; i < 100; i++ )
        {
          pthread_mutex_lock(&m);
          while (es_par==0)
          {
                  pthread_cond_wait(&cL,&m);
          }
          printf("Thread1 = %d \n", dato_compartido++);
          es_par=0;
          pthread_cond_signal(&cV);
          pthread_mutex_unlock(&m);
        }
}


void impares(void)
```

```
    {
            int i;
            for(i=0; i < 100; i++ )
            {
              pthread_mutex_lock(&m);
              while (es_par==1)
              {
                 pthread_cond_wait(&cV,&m);
              }
              printf("Thread2 = %d \n", dato_compartido++);
              es_par=1;
              pthread_cond_signal(&cL);
              pthread_mutex_unlock(&m);
            }
    }


int main(void)
    {
        pthread_t th1, th2;
        pthread_mutex_init(&m, NULL);
        pthread_cond_init(&cL, NULL);
        pthread_cond_init(&cV, NULL);
        pthread_create(&th1, NULL, pares, NULL);
        pthread_create(&th2, NULL, impares, NULL);
        pthread_join(th1, NULL);
        pthread_join(th2, NULL);
        pthread_mutex_destroy(&m);
        pthread_cond_destroy(&cL);
        pthread_cond_destroy(&cV);

    }
```

EXERCISE 3

The Operating Systems students of the Carlos III University of Madrid are asked to solve the problem of the producer / consumer with unlimited buffer (that is, there is no limitation on the number of elements that a producer can generate since the storage buffer is considered infinite). Students are asked to implement the producer function and the consumer function using semaphores, and avoiding concurrency problems. One of the students delivers the following solution:

```
int n;
semaphore s=1;
semaphore esperar=0;

void productor(void)
{
  while (1)
  {
    producir();
```

```
    wait(mutex);
    añadir(buffer);
    n++;
    if (n==1) signal(esperar);
    signal(mutex);
  }
}

void consumidor(void)
{
  while (1)
  {
    wait(mutex);
    coger(buffer);
    n--;
    if (n==0) wait(esperar);
    signal(mutex);
    consumir();

  }
}
```

This solution is not correct. It asks:

a) Find a counterexample that assumes the failure of this solution.

b) Correct the code so that the problem found is solved, or implement a new code that works.

SOLUTION

a) It can happen that the producer generates elements and the consumer consumes them, at a given moment there is only one element, the consumer executes the consume function, which leaves the buffer with 0 elements. The value of the variable n is queried and since it is 0 the consumer must fall asleep before releasing the mutex, which causes a deadlock. The consumer is locked out and prevents the consumer from executing by not releasing the mutex semaphore before falling asleep.

Action	N	Wait
Inicialmente	0	0
Productor: sección crítica	1	1
Consumidor: wait(esperar)	1	0
Consumidor: sección crítica	0	0
Consumidor: if n==0 wait(esperar)	0	0

b)

Solution 1: protect the statement that checks the value of n in the consumer with the mutex semaphore

```
int n;
semaphore s=1;
semaphore esperar=0;

void productor(void)
{
  while (1)
  {
    producir();
    wait(mutex);
    añadir(buffer);
    n++;
    if (n==1) signal(esperar);
    signal(mutex);
  }
}



void consumidor(void)
{
  while (1)   {
    wait(mutex);
    coger(buffer);
    n--;
    signal(mutex);
    consumir();
   wait(mutex);
    if (n==0) wait(esperar);
    signal(mutex);
  }
}
```

Solution 2: Add a local variable to the consumer procedure and evaluate this variable to put it asleep instead of evaluating the global variable n.

```
int n;
semaphore s=1;
semaphore esperar=0;

void productor(void)
{
  while (1)   {
    producir();
```

```
      wait(mutex);
      añadir(buffer);
      n++;
      if (n==1) signal(esperar);
      signal(mutex);
    }
}

void consumidor(void)
{  int m;       //variable local
   while (1)   {
      wait(mutex);
      coger(buffer);
      n--;
      m=n;
      signal(mutex);
      consumir();
      if (m==0) wait(esperar);
   }
}
```

EXERCISE 4

Given the following schema:

Process P1	Process P2
...	…
action1()	action2()

Ensure that action1 () is always executed before action2 (), by using:

a) Semaphores.
b) Mutex and conditional variables.

SOLUTION

a) Semaphores (semaphore s initializaed to 0)

Process P1	Process P2
...	…
	wait (s)
action1 ( )	action2 ( )
signal (s)	

b) Mutex and conditional variables.

Process P1	Process P2
...	…
action1 ( )	*lock (mutex);*
lock (mutex);	*continuar = true;*
cond_signal(var_cond);	*while(continuar != true) {*
unlock (mutex);	
cond_wait(mutex, var_cond);	

}
unlock (mutex);
action2 ( )

EXERCISE 5

Readers-Writers with semaphores.

a) Giving priority to readers (a writer cannot access the modification of the resource if I have readers who want to consult it).
b) Giving priority to the writers (a new reader cannot access the reading of the resource if there are writers who wish to modify it).

SOLUTION

a) Readers have priority.

sem_lectores = 1; sem_recurso = 1;

```
Lector() {
    wait(sem_lectores);
    num_lectores = num_lectores + 1;
    if(num_lectores == 1)
RECURSO
         wait(sem_recurso);
    signal(sem_lectores);
    ...
    //ACCESO A RECURSO
    ...
    wait(sem_lectores);
    num_lectores = num_lectores - 1;
    if(num_lectores == 0)
         signal(sem_recurso);
    signal(sem_lectores);
}
```

```
Escritor(){
   wait(sem_recurso);
   ...
   //MODIFICACION         DEL

   ...
   signal(sem_recurso);
```

b) Writers have priority.

sem_lectores = 1; sem_recurso = 1; sem_escritores = 1; lectores = 1;

```
Lector() {
    wait(lectores);
    wait(sem_lectores);
  num_escritores + 1
    num_lectores = num_lectores + 1;
    if(num_lectores == 1)
         wait(sem_recurso);
    signal(sem_lectores);
    signal(lectores);
```

```
Escritor(){
   wait(sem_escritores);
   num_escritores            =

   if(num_escritores == 1)
        wait(lectores);
   signal(sem_escritores);
   wait(sem_recurso);
   ...
```

```
    ...
   RECURSO
    //ACCESO A RECURSO
    ...
    wait(sem_lectores);
    num_lectores = num_lectores - 1;
   num_escritores - 1
    if(num_lectores == 0)
          signal(sem_recurso);
    signal(sem_lectores);
 }
```

```
    //MODIFICACION         DEL

    ...
    signal(sem_recurso);
    wait(sem_escritores);
    num_escritores           =

    if(num_escritores == 0)
          signal(lectores);
    signal(sem_escritores);
 }
```

EXERCISE 6

A famous autograph signs in a shop. The celebrity can only sign one autograph at a time. The signing room has a limited capacity of 20 seats. The famous man says that he will only go out to sign autographs if there are more than 5 people in the room. If there are not at least 5 people in the room, it will sleep until there are (when there are 4 people or less it will go to sleep). People who want to sign and cannot enter the room due to exceeding the allowed capacity will leave without being able to receive the autograph, those who receive the autograph will leave the room.
The famous represents a lightweight Process of a type that always remains in the system and executes the famous function. People represent Threads running the *fan* function.

void famoso()
{
while(1)
{
//Código del Process famoso
}
}
	void fan()
	{
	// Código del fan
	}

Given the different definitions shared by all Processes:

```
#define AFORO_MAX 20
int ocupacion=0;    //Almacena la ocupación de la sala
int firmado=0;      //Indica si el famoso ya ha hecho la firma solicitada por el fan
pthread_mutex_t m;           //Mutex para región crítica
pthread_cond_t famoso_durmiendo; //variable condicional para que el famoso espere dormido hasta que entren 5 personas
pthread_cont_t autografo;      //variable condicional para que las personas esperen hasta haber recibido su autógrafo
void Firmar();      //Función a la que debe llamar el famoso para hacer una firma
```

REQUESTED:

You are requested to encode the 'famous' and 'fan' functions using the mutex and the given conditional variables.
NOTE: It is not necessary to initialize the mutexes or the conditional variables, they are assumed already initialized.

SOLUTION

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define AFOROMAX         20      /* Numero máximo de fans*/
#define FANSMIN         5      /* Numero mínimo de fans*/
#define TRUE        1
#define FALSE       0

pthread_mutex_t mutex;     /* mutex para controlar el acceso al
                           buffer compartido */
pthread_cond_t famoso_durmiendo;   /* controla la espera del famoso*/
pthread_cond_t autografo;   /* controla la espera de los fans*/
int ocupacion=1;
int firmado=0;

void  *famoso(void *kk)  {
   int i;

   while (1){
     pthread_mutex_lock(&mutex);        /* acceder al contador */
     while (ocupacion < FANSMIN)
       pthread_cond_wait(&famoso_durmiendo, &mutex); /* se bloquea */
     printf ("FAMOSO FIRMA: %d\n", ocupacion);
     firmado++;
     ocupacion--;
     pthread_cond_signal(&autografo);
     pthread_mutex_unlock(&mutex);
     sleep(random()%2);
   }
   pthread_exit(0);
}
void  *fan(void *kk)  {
   int i;

   pthread_mutex_lock(&mutex);        /* acceder al contador */
   if (ocupacion != AFOROMAX)  {
     ocupacion++;
     printf ("fan espera: %u\n", pthread_self());
     pthread_cond_signal(&famoso_durmiendo);
     while (firmado==0)
         pthread_cond_wait(&autografo, &mutex); /* se bloquea */
     firmado-- ;
     printf ("FIN fan atendido: %u\n", pthread_self());
   }
```

```
   else
     printf ("FIN fan sin atender: %u\n", pthread_self());

   pthread_mutex_unlock(&mutex);
   pthread_exit(0);
}

main(int argc, char *argv[]){
    int i;
    pthread_t th1, th2;
    pthread_attr_t attrfan;

    pthread_attr_init(&attrfan);
    pthread_attr_setdetachstate( &attrfan, PTHREAD_CREATE_DETACHED);

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&famoso_durmiendo, NULL);
    pthread_cond_init(&autografo, NULL);

    pthread_create(&th1, NULL, famoso, NULL);
    for (i=0;i<60;i++){
      pthread_create(&th2, &attrfan, fan, NULL);
    }

    pthread_join(th1, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&famoso_durmiendo);
    pthread_cond_destroy(&autografo);

    exit(0);
}
```

EXERCISE 7

Carry out a program in C that serves to control an irrigation system with 5 irrigation valves and 3 water inlets.
The program will have one thread for each irrigation valve and one thread for each input.
A menu will be shown to the user who will be the one to decide whether to open or close a water inlet.
When the user decides to open a water input one of the input threads will be placed as open and when he decides to close an input one of the open input threads will go to its closed state.
The number of open irrigation valves must be equal to or less than the number of open inlets. When an inlet opens, an irrigation valve must also open. The operation of the irrigation valves should be as follows:

1. the valve thread must wait for the number of inlets to be greater than the number of open valves,
2. when this is the case, the valve thread will try to take the right to be the one that goes to the open state,
3. in this state it will be 3 seconds,
4. then it will give way to another thread and
5. It will be 1 second until it tries to go back to the open state if the situation at that moment allows it.

During the 3 seconds of waiting with the valve open, it will not be necessary to check if the number of open inlets is greater than or equal to the number of valves.

SOLUTION

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

#define MAX_VALVULAS  5
#define MAX_ENTRADAS  3

int entradasAbtas=0;
int valvulasAbtas=0;
int abrirentrada=0;
int cerrarentrada=0;

pthread_mutex_t me;
pthread_mutex_t mv;
pthread_cond_t abrirE;
pthread_cond_t cerrarE;
pthread_cond_t vValvula;


void * valvula (void *n){
  while(1) {
    pthread_mutex_lock(&mv);
```

```
//Válvula cerrada
      while( entradasAbtas<=valvulasAbtas ) {
        pthread_cond_wait(&vValvula,&mv);
      }
      valvulasAbtas++;
//Válvula abierta
    pthread_mutex_unlock(&mv);
    sleep(3);
    pthread_mutex_lock(&mv);
      valvulasAbtas--;
//Válvula cerrada
      pthread_cond_signal(&vValvula);
    pthread_mutex_unlock(&mv);
    sleep(1);
  }
}

void * entrada (void *n){
  while(1) {
//Entrada cerrada
    pthread_mutex_lock(&me);
      while(abrirentrada==0 ) {
        pthread_cond_wait(&abrirE,&me);
      }
      abrirentrada=0;
      pthread_mutex_lock(&mv);
        entradasAbtas++;
//Entrada Abierta
        pthread_cond_signal(&vValvula);
      pthread_mutex_unlock(&mv);
    pthread_mutex_unlock(&me);
  // esperamos a que se ordene el cierre
    pthread_mutex_lock(&me);
      while(cerrarentrada==0 ) {
        pthread_cond_wait(&cerrarE,&me);
      }
      cerrarentrada=0;
      entradasAbtas--;
    pthread_mutex_unlock(&me);
  }
  pthread_exit(NULL);

}

int main() {
      int i;
        char resp[10];
        pthread_t identrada[MAX_ENTRADAS];
      pthread_t idvalvula[MAX_VALVULAS];

      pthread_mutex_init(&me,NULL);
      pthread_mutex_init(&mv,NULL);
      pthread_cond_init(&abrirE,NULL);
      pthread_cond_init(&cerrarE,NULL);
      pthread_cond_init(&vValvula,NULL);

  for (i=0; i< MAX_VALVULAS; i++)
    pthread_create(&idvalvula[i],NULL,valvula,NULL);
```

```
  for (i=0; i< MAX_ENTRADAS; i++)
    pthread_create(&identrada[i],NULL,entrada,NULL);
  while(1) {
    printf  ("Ahora  hay  %d  entradas  abiertas  y  %d  valvulas
abtas\n",entradasAbtas,valvulasAbtas);
    printf ("Si quieres abrir una entrada pulse A si quiere cerrar una
valvula pulse C:");
    scanf ("%s", resp);
    if  (resp[0] == 'A' ) {
      pthread_mutex_lock(&me);
        abrirentrada=1;
        pthread_cond_signal(&abrirE);
      pthread_mutex_unlock(&me);
    }
    if  (resp[0] == 'C' ) {
      pthread_mutex_lock(&me);
        cerrarentrada=1;
        pthread_cond_signal(&cerrarE);
      pthread_mutex_unlock(&me);
    }
  }

      pthread_mutex_destroy(&me);
      pthread_mutex_destroy(&mv);
}
```

EXERCISE 8

Describe what the following program does:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define N 10
#define TAMANIO 1024

void *trabajador(void *arg);

int vector[TAMANIO];
struct b_s {
  int n;
  pthread_mutex_t m;
  pthread_cond_t ll;
} b;


int main(void) {
  pthread_t hilo[N];
  int i;


  b.n = 0;
  pthread_mutex_init(&b.m,
NULL);
  pthread_cond_init(&b.ll,
NULL);
  par=0; impar=1;
```

```
void *trabajador(void *arg) {
       int inicio=0, fin=0, i;


       id  = *(int *)arg;
       inicio =(id)*(TAMANIO/N);

       fin = (id+1)*(TAMANIO/N);

       for(i=inicio; i<fin; i++) {
              vector[i] = id;
       }
       pthread_mutex_lock(&b.m);
       b.n++;
       if (N<=b.n) {
              pthread_cond_broadcast(&b.ll);
       } else {
              pthread_cond_wait(&b.ll,
       &b.m);
       }
       pthread_mutex_unlock(&b.m);

       return 0;
}
```

<pre> for(i=0; i<N; i++)
    pthread_create(&hilo[i],
                 NULL,
trabajador,
                 (void
*)&i);
 	
  for(i=0; i<N; i++)
    pthread_join(hilo[i],
NULL);
 	
pthread_cond_destroy(&b.ll);
 	
pthread_mutex_destroy(&b.m);
 	
  return 0;
}</pre>	

SOLUTION

The main Process will create 10 Threads. Each of these Threads establishes a range (start… end) in which to store values in the vector. When each worker finishes storing values, he increments b.n and asks if b.n is equal to N:

- If the Process is not the last: (n <= N) then the Process goes to sleep.
- If the Process is the last (n> N) then the Process wakes up all sleeping Threads.

The main Process waits for the Threads at the end.

EXERCISE 9

Add a local variable to the consumer procedure and evaluate this variable to put it to sleep instead of evaluating the global variable n.

SOLUTION

```
int n;
semaphore s=1;
semaphore esperar=0;

void productor(void)
{
  while (1)
  {
    producir();
    wait(mutex);
    añadir(buffer);
    n++;
    if (n==1) signal(esperar);
```

```
      signal(mutex);
    }
}

void consumidor(void)
{
  while (1)
  {
      wait(mutex);
      coger(buffer);
      n--;
      if (n==0)
     {  signal(mutex);
         wait(esperar);
     }
    else
       signal(mutex);
    consumir();
  }
}
```

EXERCISE 10

Write a program that runs the problem of the barbershop serving clients. Barbers can serve a maximum of 4 clients within the barbershop. If there is no work, the barbers sleep. The barbershop is modeled as a Process. Each customer who enters occupies an armchair. If everything is already occupied, clients try to enter and if they cannot, they leave.

SOLUTION

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define MAX_CLIENTES 4
#define SEG_LLEGADA_CLIENTE 10

int ocupacion=0;

pthread_mutex_t m;
pthread_cond_t barbero_durmiendo;
pthread_cond_t corte_pelo;


void CortarElPelo() {
  printf("Estoy cortando el pelo....ocupacion=%d\n",ocupacion);
```

```
  sleep(3);
  printf("He terminado de cortar el pelo!!!\n");
}

void * barbero ()
{
  while(1)
  {
     pthread_mutex_lock(&m);
     while(ocupacion==0 )
     {
       printf("Soy el barbero y duermo\n");
       pthread_cond_wait(&barbero_durmiendo,&m);
     }
     CortarElPelo();
     ocupacion--;
     pthread_cond_signal(&corte_pelo);
     pthread_mutex_unlock(&m);
  }

  pthread_exit(NULL);
}

void * cliente(void * p) {
  int n_cliente;

  n_cliente=(int)p;
  pthread_mutex_lock(&m);

  if(ocupacion != MAX_CLIENTES) {
    ocupacion++;
    printf("Soy   el   cliente   %d   y   acabo   de   llegar.
Ocupacion=%d\n",n_cliente,ocupacion);
    pthread_cond_signal(&barbero_durmiendo);
    pthread_cond_wait(&corte_pelo,&m);
  }
  else
  {
    printf("Soy el cliente %d y no hay sillas. Me voy!!\n",
n_cliente);
  }
  pthread_mutex_unlock(&m);
  pthread_exit(NULL);
}


int main()
{
        int num;
        pthread_t t_barbero;
        pthread_t * p_cliente;
```

```
        int contador=0;

        pthread_mutex_init(&m,NULL);
        pthread_cond_init(&barbero_durmiendo,NULL);
        pthread_cond_init(&corte_pelo,NULL);

        pthread_create(&t_barbero,NULL,barbero,NULL);

        srand(time(NULL));

        while(1) //simulacion de llegada de Processes al sistema
        {
                num=rand()%2;
                if(num==0)
                {
                        contador++;
                        p_cliente=malloc(sizeof(pthread_t));

pthread_create(p_cliente,NULL,cliente,(void*)&contador);
                }
                else
                {
                        sleep(2);
                }
        }
        pthread_mutex_destroy(&m);
}
```

EXERCISE 11

Write a program that meets the philosophers who eat program, for 5 philosophers using MUTEX. Five philosophers sit around a table and spend their lives dining and thinking. Each philosopher has a plate of noodles and a fork to the left of his plate. Two forks are necessary to eat the noodles, and each philosopher can only take the ones to his left and right.

SOLUTION

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define MAX          5      /* Numero máximo de veces que come
cada filosofo*/
#define OCUPADO        1
#define LIBRE       0

#define NUMFILOSOFOS 5
```

```
pthread_mutex_t mtx;     /* mutex para controlar el acceso a los
tenedores*/
pthread_cond_t  espera;      /*  controla  la  espera  de  los
filosofos*/
int tenedores[NUMFILOSOFOS];

pthread_mutex_t mtxIndice;     /* mutex para controlar el acceso
al indice del filosofo*/
pthread_cond_t  esperaIndice;     /*  controla  la  espera  en  el
índice */
int hiloespera=1;

void  *filosofo(void *indice)  {   /* codigo del que escribe los
pares */
   int i,j,tenedor1,tenedor2;

   srandom ((unsigned)pthread_self());

    pthread_mutex_lock(&mtxIndice);        /* acceder al indice
*/
       hiloespera=0;
       i=*((int *) indice);
       pthread_cond_signal(&esperaIndice);
    pthread_mutex_unlock(&mtxIndice);            /*  acceder  al
indice */

   tenedor1= i;
   tenedor2= i+1;
   if (tenedor2 == NUMFILOSOFOS) tenedor2=0;

 for(j=0; j <= MAX; j++ )  {
    pthread_mutex_lock(&mtx);
      while           (tenedores[tenedor1]==OCUPADO           ||
tenedores[tenedor2]==OCUPADO)
           pthread_cond_wait(&espera, &mtx);
      tenedores[tenedor1]=OCUPADO;
      tenedores[tenedor2]=OCUPADO;
printf("Filosofo %d va a comer\n",i);
    pthread_mutex_unlock(&mtx);

    sleep  (1+  random()%2);     //cogiendo  la  comida  con  los
tenedores


printf("Filosofo %d deja de comer\n",i);
    pthread_mutex_lock(&mtx);
      tenedores[tenedor1]=LIBRE;
      tenedores[tenedor2]=LIBRE;
      pthread_cond_broadcast(&espera);
    pthread_mutex_unlock(&mtx);
```

```
    sleep ( random()%3);   //espera un moemnto para masticar
  }
    printf ("FIN filosofo %d\n",i);
    pthread_exit(0);
}


int main(int argc, char *argv[]){
    pthread_t th[NUMFILOSOFOS];
    int i;

    pthread_mutex_init(&mtx, NULL);
    pthread_cond_init(&espera, NULL);
    pthread_mutex_init(&mtxIndice, NULL);
    pthread_cond_init(&esperaIndice, NULL);

   for (i=0; i<NUMFILOSOFOS; i++)
     tenedores[i]=LIBRE;

   for (i=0; i<NUMFILOSOFOS; i++){
    pthread_mutex_lock(&mtxIndice);         /* acceder al indice
*/
    pthread_create(&th[i], NULL, filosofo, &i);
      while (hiloespera==1)
            pthread_cond_wait(&esperaIndice, &mtxIndice ); /* se
espera */
      hiloespera=1;
    pthread_mutex_unlock(&mtxIndice);            /*  acceder  al
indice */
   }
   for (i=0; i<NUMFILOSOFOS; i++)
    pthread_join(th[i], NULL);

    pthread_mutex_destroy(&mtx);
    pthread_mutex_destroy(&mtxIndice);
    pthread_cond_destroy(&espera);
    pthread_cond_destroy(&esperaIndice);

    exit(0);
}
```

EXERCISE 12

Make a program that declares a print function and passes it as parameters e1 string to print.

Next, the main program must prepare the parameters with 2 strings "hello" and "world \ n" and launch 2 threads that try to print "hello world" in that order N times and finish.

SOLUTION

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

#define N 3

pthread_t thread1, thread2; /* Declaración de los threads */
pthread_attr_t attr; /*atributos de los threads*/
pthread_mutex_t impresor=PTHREAD_MUTEX_INITIALIZER;


/* Definición de las función imprimir */
void *imprimir (void *arg)
{
  char a[12];

  pthread_mutex_lock (&impresor);
  strcpy(a, (char*)arg);

  printf("%s ",a);
  pthread_mutex_unlock (&impresor);

  pthread_exit (NULL);
}


/*Función main*/
int main (void)
{
  char cadena_hola[]="Hello ";
  char cadena_mundo[]="world \n";
  int i;

  pthread_attr_init (&attr);

  for (i=1; i<=N; i++) {
     pthread_create(&thread1,     &attr,     imprimir,     (void
*)cadena_hola);
     pthread_create(&thread2,     &attr,     imprimir,     (void
*)cadena_mundo);
  }

  pthread_exit (NULL);
}
```

EXERCISE 13

Write a simple program to see how mutex works. The main program creates 4 threads and waits until they have all finished. The normal thing would be for the main to do a pthread_join, but it must be done with mutex to see how to use them.

SOLUTION

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define TRUE 1
#define FALSE 0
#define NUMTHREADS 4


pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;
pthread_attr_t attr;
int hijosVivos;

void *f( void *n){
        int n_local,*p;

        p=(int *)n;
        n_local=*p;
        printf ("Creado TH:n_local %d ((int)time:%d)\n",n_local,
(int)time(NULL));
        sleep (4);
        pthread_mutex_lock (&m);
        hijosVivos --;
        printf          ("FIN          TH:n_local           %d
((int)time:%d)\n",n_local,(int)time(NULL));
        pthread_mutex_unlock (&m);
        pthread_exit(NULL);
}

int main (){
        pthread_t thid;
        int n=33,i,fin;

      pthread_mutex_init(&m,  NULL);  //inicializo  el  mutex  con
los atributos por defecto
      //inicializo los atributos del thread
      pthread_attr_init (&attr);
      pthread_attr_setdetachstate                        (&attr,
PTHREAD_CREATE_DETACHED);
      hijosVivos=NUMTHREADS;
      for (i=1; i<=NUMTHREADS; i++){
        pthread_create (&thid, &attr, f, &i);
```

```
        sleep(1);
        //espero a quese cree el thread, aunque esta no es la forma adecuada lo normalsería usar mutex y varcondicionales
      }
      fin=FALSE;
      while (!fin){
        pthread_mutex_lock (&m);
          if (hijosVivos ==0) fin =TRUE;
        pthread_mutex_unlock (&m);
      }
      printf ("Han terminado todos los threads \n");
}
```

EXERCISE 14

Write a program to test POSIX barriers. The program must create six threads and one barrier. Each thread must sleep 3 seconds and wait to pass the barrier. The parent must wait for all threads to finish.

SOLUTION

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>


#define NUMTHREADS 6
//Primero pasan 3 y luego otros 3
#define THBARRERA 3

pthread_t th[NUMTHREADS];
pthread_barrier_t mi_barrera;

void * sync_carrera (void * data)
{
   int espera=random()%5;
   printf ("Espera %d thread %d\n", espera, (int)pthread_self());
   sleep(espera);

   pthread_barrier_wait(&mi_barrera);
   printf ("Paso la mi_barrera thread %d\n", (int) pthread_self());
   pthread_exit(NULL);
}

int main (int argc, char ** argv)
{
  int i;
```

```
  pthread_barrier_init(&mi_barrera, NULL, THBARRERA );

  for (i=0; i<NUMTHREADS; i++)
    pthread_create(&th[i], NULL, sync_carrera, NULL);

  printf ("THS creados\n");

  for (i=0; i<NUMTHREADS; i++) {
    pthread_join(th[i], NULL);
   // Espera por un thread concreto. Si el orden de finalización
no es el de creación
   // (ej. termina el 1 y luego el 0) espera por el 0 hasta que
termine y luego espera por el 1)
    printf ("Fin th:%d\n", (int)th[i]);
  }
  pthread_barrier_destroy(&mi_barrera);
  printf ("FIN\n");
}
```

EXERCISE 15

Implement a program that solves the producer-consumer problem with MUTEX. The program describes two threads, producer and consumer, that share a finite size buffer. The producer's task is to generate an integer, store it, and start over; while the consumer takes (simultaneously) numbers one to one. The problem is that the producer does not add more numbers than the buffer capacity and that the consumer does not try to take a number if the buffer is empty.

SOLUTION

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX_BUFFER          10     /* tamanio del buffer */
#define DATOS_A_PRODUCIR    1000    /* datos a producir */

pthread_mutex_t mutex;      /* mutex para controlar el acceso al
                            buffer compartido */
pthread_cond_t no_lleno;    /* controla el llenado del buffer */
pthread_cond_t no_vacio;    /* controla el vaciado del buffer */
int n_elementos;            /* numero de elementos en el buffer
*/

int buffer[MAX_BUFFER];     /* buffer comun */

void *Productor(void *kk)  {   /* codigo del productor */
   int dato, i ,pos = 0;
```

```
    for(i=0; i < DATOS_A_PRODUCIR; i++ )  {
        dato = i;       /* producir dato */
        pthread_mutex_lock(&mutex);        /* acceder al buffer
*/
        while (n_elementos == MAX_BUFFER)  /* si buffer lleno */
            pthread_cond_wait(&no_lleno, &mutex); /* se bloquea
*/
        buffer[pos] = i;
        printf("produce %d \n", buffer[pos]);     /* produce
dato */
        pos = (pos + 1) % MAX_BUFFER;
        n_elementos ++;
        pthread_cond_signal(&no_vacio);   /* buffer no vacio */
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(0);
}

void *Consumidor(void *kk) {   /* codigo del sonsumidor */
    int dato, i ,pos = 0;

    for(i=0; i < DATOS_A_PRODUCIR; i++ ) {
        pthread_mutex_lock(&mutex);    /* acceder al buffer */
        while (n_elementos == 0)       /* si buffer vacio */
            pthread_cond_wait(&no_vacio, &mutex); /* se bloquea
*/
        dato = buffer[pos];
        pos = (pos + 1) % MAX_BUFFER;
        n_elementos --;
        pthread_cond_signal(&no_lleno);   /* buffer no lleno */
        pthread_mutex_unlock(&mutex);
        printf("Consume %d \n", dato);     /* consume dato */
    }
    pthread_exit(0);
}


int main(int argc, char *argv[]){
    pthread_t th1, th2;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&no_lleno, NULL);
    pthread_cond_init(&no_vacio, NULL);
    pthread_create(&th1, NULL, Productor, NULL);
    pthread_create(&th2, NULL, Consumidor, NULL);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&no_lleno);
```

```
    pthread_cond_destroy(&no_vacio);

    exit(0);
}
```

EXERCISE 16

Implement a program that solves the producer-consumer problem with POSIX Semaphores. The program describes two threads, producer and consumer, that share a finite size buffer. The producer's task is to generate an integer, store it, and start over; while the consumer takes (simultaneously) numbers one to one. The problem is that the producer does not add more numbers than the buffer capacity and that the consumer does not try to take a number if the buffer is empty.

SOLUTION

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define MAX_BUFFER          1024    /* tamanio del buffer */
#define DATOS_A_PRODUCIR   10000    /* datos a producir */

sem_t elementos;                /* elementos en el buffer */
sem_t huecos;                   /* huecos en el buffer */
int buffer[MAX_BUFFER];         /* buffer comun */

int main(void)
{
   pthread_t th1, th2;  /* identificadores de threads */

   /* inicializar los semaforos */
   sem_init(&elementos, 0, 0);
   sem_init(&huecos, 0, MAX_BUFFER);
/* crear los Threads */
   pthread_create(&th1, NULL, Productor, NULL);
   pthread_create(&th2, NULL, Consumidor, NULL);

   /* esperar su finalizacion */
   pthread_join(th1, NULL);
   pthread_join(th2, NULL);

   sem_destroy(&huecos);
   sem_destroy(&elementos);
   exit(0);
}
```

```
void Productor(void)   /* codigo del productor */
{
   int pos = 0;  /* posicion dentro del buffer */
   int dato;     /* dato a producir */
   int i;

   for(i=0; i < DATOS_A_PRODUCIR; i++ )   {
      dato = i;            /* producir dato */
      sem_wait(&huecos);   /* un hueco menos */
      buffer[pos] = i;
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&elementos);  /* un elemento mas */
  }
  pthread_exit(0);
}


void Consumidor(void)  /* codigo del Consumidor */
{
   int pos = 0;
   int dato;
   int i;

   for(i=0; i < DATOS_A_PRODUCIR; i++ ) {
      sem_wait(&elementos);    /* un elemento menos */
      dato = buffer[pos];
      pos = (pos + 1) % MAX_BUFFER;
      sem_post(&huecos);    /* un hueco mas */
      /* cosumir dato */
   }
   pthread_exit(0);
}
```

EXERCISE 16

Make a program that creates 10 "threads", the first "thread" will add the numbers 001-100 of a file that contains 1000 numbers, and the following "threads" will successively add the numbers that correspond to them: 101-200, 201- 300, 301-400, 401-500, 601-700, 701-800, 801-900 and 901-1000 respectively. The children will return to the father the sum made, printing this the total sum.
Use MUTEX to ensure that there are no concurrency problems between the threads.

SOLUTION

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
void *suma(void *rango);

pthread_mutex_t mtx;
pthread_cond_t cond;
int obtenidoRango;

pthread_attr_t attr;
int f=0;
pthread_t thread[10];

int main() {
    int i=0, n=0, rango=0, *estado, pestado=0, nbytes=0, nreg=0;
    estado=&pestado;
    pthread_attr_init(&attr);

    if((f=open("numeros.dat", O_RDONLY))==-1) {
        fprintf(stderr,"Error en la apertura del fichero\n");
        return(-1);
    }
    nbytes=lseek(f,0,SEEK_END);
    nreg=nbytes/sizeof(int);
    for(i=0;i<10;i++) {
        obtenidoRango=0;
        pthread_mutex_lock(&mtx);
        pthread_create(&thread[i],&attr,suma,&rango);
        while (obtenidoRango==0)
          pthread_cond_wait(&cond, &mtx);
        pthread_mutex_unlock(&mtx);
        rango+=100;
    }
    for(i=0;i<10;i++) {
      pthread_join(thread[i],(void **)&estado);
      printf("Suma Parciales en Prog. Principal: %d\n",*estado);
      n+=*estado;
    }
    printf("Suma Total: %d\n",n);
    printf("Total numeros sumados: %d\n",nreg);
    close(f);
    return(0);
}

void *suma(void *rango) {
int j=0, valor, *suma, num=0;

//sleep(1);

    pthread_mutex_lock(&mtx);
      valor=*((int *)rango);
```

```
    obtenidoRango=1;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mtx);

    suma=(int *)malloc (sizeof (int));
   *suma=0;
    printf("Rango: %d a %d\n",valor+1,valor+100);
    lseek(f,valor * sizeof(int),SEEK_SET);
    for(j=0;j<100;j++) {
        read(f,&num,sizeof(int));
        *suma+=num;
    }
    printf("\tSuma Parcial: %d\n",*suma);
    pthread_exit(suma);
}
```

EXERCISE 17

There is an array of 100 elements in which you want to carry out several iterations in each of which the mean of the sum of the content of that box and its two adjacent cells must be placed in each box. In other words, for any cell in the array between 1 and 98 the new value of the cell will be v [i] = (v [i-1] + v [i] + v [i + 1]) / 3; for box 0 the same procedure will be applied but assuming that its left adjacent is box 99 and for box 99 it will be assumed that its right adjacent is box 0.
The procedure is applied for 10 iterations and to optimize it is desired that half of the array is processed by a thread and the other half by another thread.
The operations for calculating the new values will be performed in an auxiliary array and only when the two threads have finished their iteration will they dump the values of the real array to the auxiliary to continue with the next iteration.
Therefore, the procedure that each thread will follow is:

1.- Copy the new values in the auxiliary array

2.- When you have finished copying them you should wait for the other thread to finish before dumping the data from the auxiliary array into the real array. This way the data of one iteration is not modified before the other uses the box they have in common

3.- Once the data has been copied into the real array, you should wait for the other thread to also finish copying the auxiliary data onto the real array before proceeding to the next iteration of step 1. As in step 2, you have to wait for the common boxes to be updated.

Develop a C program with mutex and condition variables that solves this problem.

SOLUTION

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define TAM 10
#define NUMITER 20
pthread_attr_t attr;
pthread_t idth[2];
pthread_mutex_t mtx1,mtx2;
pthread_cond_t varcond1,varcond2;
int contfin1=0, contfin2=0;
float v[TAM];

void rellenarArray(){
  int i;
    for (i=0; i<TAM;i++)
      v[i]=i;
}
void mostrarArray(){
  int i;
    for (i=0; i<TAM;i++)
      printf (":%.2f", v[i]);
    printf ("\n");
}
void *hilo0(void *num) {
   int i,j;
   float vaux[TAM/2];
```

```
   int yoinicializo1=0, yoinicializo2=0;


printf ("Hilo 0\n");

for (j=0; j<NUMITER; j++) {

   vaux [0]= (v[TAM-1]+v[0]+v[1])/3;

   for (i=1; i<TAM/2 ; i++)

     vaux [i]= (v[i-1]+v[i]+v[i+1])/3;

   pthread_mutex_lock (&mtx1);

     contfin1++;

     //El que  va a esperar en el wait ser· el encargado de
inicializar la variable contfin1 para la siguiente iteracciÛn

     if (contfin1==1) yoinicializo1=1;

   //Esperar que termine el otro si no ha terminado

     while (contfin1!=2)

       pthread_cond_wait(&varcond1, &mtx1);

     for (i=0; i<TAM/2 ; i++)

       v[i]=vaux[i];

     pthread_cond_signal(&varcond1);

     if (yoinicializo1){

       contfin1=0;

       yoinicializo1=0;

     }

   pthread_mutex_unlock (&mtx1);


   pthread_mutex_lock (&mtx2);

     contfin2++;

     if (contfin2==1) yoinicializo2=1;
```

```
   //Esperar  a que el otro copie los datos
     while (contfin2!=2)
          pthread_cond_wait(&varcond2, &mtx2);
     pthread_cond_signal(&varcond2);
     if (yoinicializo2){
       contfin2=0;
       yoinicializo2=0;
       mostrarArray();
     }
   pthread_mutex_unlock (&mtx2);
}
pthread_exit(0);
}
void *hilo1(void *num) {
   int i,j;
   float vaux[TAM/2];
   int yoinicializo1=0, yoinicializo2=0;


printf ("Hilo 1\n");
for (j=0; j<NUMITER; j++) {
   for (i=TAM/2; i<TAM-1 ; i++)
     vaux [i-TAM/2]= (v[i-1]+v[i]+v[i+1])/3;
   vaux [TAM/2-1]= (v[TAM-2]+v[TAM-1]+v[0])/3;
   pthread_mutex_lock (&mtx1);
     contfin1++;
     //El que  va a esperar en el wait ser· el encargado de
inicializar la variable contfin1 para la siguiente itereacciÛn
```

```
    if (contfin1==1) yoinicializo1=1;
  //Esperar que termine el otro si no ha terminado
    while (contfin1!=2)
      pthread_cond_wait(&varcond1, &mtx1);
      for (i=TAM/2; i<TAM ; i++)
        v[i]=vaux[i-TAM/2];
    pthread_cond_signal(&varcond1);
    if (yoinicializo1){
      contfin1=0;
      yoinicializo1=0;
    }
  pthread_mutex_unlock (&mtx1);


  pthread_mutex_lock (&mtx2);
    contfin2++;
    if (contfin2==1) yoinicializo2=1;
  //Esperar  a que el otro copie los datos
    while (contfin2!=2)
          pthread_cond_wait(&varcond2, &mtx2);
    pthread_cond_signal(&varcond2);
    if (yoinicializo2){
      contfin2=0;
      yoinicializo2=0;
      mostrarArray();
    }
  pthread_mutex_unlock (&mtx2);
}
```

```
pthread_exit(0);

}


int main(){

   int i;


    rellenarArray();

    pthread_mutex_init (&mtx1, NULL);

    pthread_mutex_init (&mtx2, NULL);

    pthread_attr_init(&attr);

    pthread_create(&idth[0],&attr,hilo0,NULL);

    pthread_create(&idth[1],&attr,hilo1,NULL);

  // Espero la finalizaciÛn de los threads

    for (i=0; i<2; i++)

      pthread_join(idth[i],NULL);

    return(0);

}
```

EXERCISE 18

Make a program that creates 2 children using fork (heavy Processes). The first one must write the even numbers (from 2 to 10) and the other the odd numbers (from 1 to 9). The ordered numbers should appear on the screen, so the executions should be alternate. Named semaphores will be used as a synchronization mechanism between the Processes. Execution example:
Son 1: 1 Son 2: 2 Son 1: 3 Son 2: 4 Son 1: 5 Son 2: 6 Son 1: 7 Son 2: 8 Son 1: 9 Son 2:10.

SOLUTION

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <fcntl.h>
```

```
#include <sys/stat.h>
#include <semaphore.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sched.h>


int dato_compartido = 0;
sem_t *sem1, *sem2;

void  uno(sem_t *sem1, sem_t *sem2)
{
          int i;
               for (i=0; i<10; i++) {
                    sem_wait(sem1);
                    printf("Thread       1       %d       \n",
dato_compartido++);
                    sem_post(sem2);
            }
}

void  dos (sem_t *sem1, sem_t *sem2)
{
          int i;
               for (i=0; i<10; i++) {
                    sem_wait(sem2);
                    printf("Thread       2       %d       \n",
dato_compartido++);
                    sem_post(sem1);
            }
}


int main(void) {
         int status;

       sem1 = sem_open("mysem1", O_CREAT, O_RDWR, 1);
       if (sem1 == SEM_FAILED) {
               perror("Failed to open semphore for sem1");
               exit(-1);
       }
       sem2 = sem_open("mysem2", O_CREAT, O_RDWR , 0);
       if (sem2 == SEM_FAILED) {
               perror("Failed to open semphore for sem2");
               exit(-1);
       }

         if (fork() == 0) {
                     uno (sem1, sem2);
         } else {
            if (fork() == 0) {
                        dos (sem1, sem2);
```

```
            } else {
                        wait(&status);
                       wait(&status);
              sem_close(sem1);
              sem_close(sem2);
          }
       }
}
```

EXERCISE 19

Make a program that creates 2 Processes and allows them to be synchronized using a pipe. The first of them must create the pipe before creating the child Process.
Then each one of them must print "I am the father" "I am the son" synchronously and in this order, reading and writing in the pipe as a synchronization mechanism.

SOLUTION

```
// EXERCISE de las transparencias sobre el uso de las tuberÌas
//   ImplemntaciÛn de na secciÛn crÌtica con pipes

 #include <unistd.h>
 #include <stdio.h>
 #include <stdlib.h>

 int  main(void) {

   int fildes[2];   /* pipe para sincronizar */
  char c;  /* caracter para sincronizar */

 pipe(fildes);
   write(fildes[1], &c, 1);   /* necesario para entrar en la sección critica la primera vez
*/
   if (fork() == 0)  {   /* Process hijo */
           for(;;) {
                   read(fildes[0], &c, 1);  /* entrada seccion critica */
                     // Seccion critica
                         printf ("El hijo entra en seccion critica\n");
                   sleep (2);  // espero para que se vea que el padre no entra
                     printf ("El hijo sale de la seccion critica\n");
                    write(fildes[1], &c, 1);  /* salida seccion critica */
                   sleep (random()%2); // Espero para que no siempre entre el mismo
             }
             } else {    /* Process padre */
                         for(;;) {
```

```
                    read(fildes[0], &c, 1);   /* entrada seccion critica */
                            //Seccion critica
                    printf ("El padre entra en seccion critica\n");
                     sleep (2);  // espero para que se vea que el hijo no entra
                      printf ("El padre sale de la seccion critica\n");
                     write(fildes[1], &c, 1);  /* salida seccion critica */
                          sleep (random()%2);   //Espero para que no siempre entre el
mismo
                          }
                 }
         }
}
```