



Ejercicios de concurrencia

Ejercicio 1

Escriba un programa que cree 10 threads. Cada uno de ellos calcula el valor del número PI usando el método de Montecarlo y lo almacena en la posición que le corresponde de en un array. Cuando han terminado todos los threads el programa principal calcula la media de los valores de pi almacenados en el array

Se deben usar mutex y variables condición para que no haya problemas de Carrera.

Solución

```
#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 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]);
```



Ejercicios de concurrencia

```
        suma=suma+valoresPlthreads[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++;
    }
    valoresPlthreads[numthread]=(cont*4)/(float)PUNTOS;
    pthread_exit(&pi);
}
```

Ejercicio 2

El siguiente código implementa una aplicación con dos threads: uno imprime por pantalla los números pares y otro imprime por pantalla los números impares.

```
#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++ )
```



Ejercicios de concurrencia

```
        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);  
}
```

Se desea que la ejecución proporcione por pantalla la siguiente salida:

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

Se ha realizado una primera ejecución del código y aparece lo siguiente:

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

Se pide resolver los siguientes apartados:

1. Indicar que problemas genera el utilizar una variable compartida para enviar el dato a imprimir desde el thread pares al thread impares.
2. Implementar una versión del programa anterior que resuelva los problemas anteriores utilizando alguna de las técnicas de gestión de concurrencia.



Ejercicios de concurrencia

Solución

1.-

Problemas de carrera: Ocurre cuando dos procesos acceden a unas variables compartidas (simultáneamente o en un orden incorrecto) de forma que el valor de las variables deja de ser coherente con la lógica del programa. En este ejemplo tanto el Thread pares como el Thread impares solo disponen de una variable compartida a la que acceden cada uno en una sola línea (el Thread pares imprime la variable y el Thread impares también). Si cada una de estas líneas fuera atómica (una vez pasado a código máquina) y solo hay una CPU, entonces no habría peligro de carrera (primero se imprimiría una y se incrementaría la variable `dato_compartido` y luego se imprimiría la otra), en caso contrario, es decir, en el mostrado en el ejemplo podría ocurrir que se repitiera por pantalla un valor de la variable:

Thread1 = 3

Thread2 = 3

Ya que se ha producido un cambio de contexto antes de que la variable fuera incrementada. Para evitar este problema, es mejor convertir dichas líneas en zonas de exclusión mutua con algunas de las técnicas vistas (semáforos o mutex).

Problemas de progreso: El código propuesto muestra dos hilos de ejecución. Pero no hay nada que asegure en que orden se ejecutarán. Si el planificador sigue una política batch primero ejecutaría uno y luego otro con lo que este último no podría progresar hasta que termine el primero (cosa que podría no ocurrir si fuera un bucle infinito). Este problema debe ser evitado indicando explícitamente en ambos procesos cuando deben abandonar la CPU para que el otro proceso pueda progresar (usando semáforos o conditions).

Problemas de espera limitada: Al código propuesto no le es suficiente con asegurar que ningún proceso va a quedarse parado, sino que requiere que se alterne una iteración del Thread pares con otra iteración de Thread impares. (Cada proceso debe tener una espera limitada a la consecución de una iteración del otro proceso). Para conseguirlo es necesario utilizar alguna de las técnicas vistas (usando semáforos o conditions).

2.-

Solución con semáforos

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

```
int dato_compartido = 0;
sem_t par, impar;
```



Ejercicios de concurrencia

```
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);
}
```

Solución con mutex y condiciones:

```
#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++);
    }
}
```



Ejercicios de concurrencia

```

        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);
}

```

Ejercicio 3

Se plantea a los alumnos de Sistemas Operativos de la Universidad Carlos III de Madrid resolver el problema del productor/consumidor con buffer ilimitado (es decir, no existe limitación en el número de elementos que puede generar un productor ya que el buffer de almacenamiento se considera infinito). Se pide a los alumnos que implementen la función productor y la función consumidor usando semáforos, y evitando que se produzcan problemas de concurrencia. Uno de los alumnos entrega la siguiente solución:

```
int n1=0;
```



Ejercicios de concurrencia

```
semaphore smutex = 1;
semaphore esperar = 0;
```

```
void productor(void)
```

```
{
    while (1)
    {
        producir();
        wait(mutex); → lo pone a 0 y entra
        añadir(buffer);
        n++; n+1 = 1
        if (n==1) signal(esperar); libera esperar = 1
        signal(mutex); libera mutex = 1
    }
}
```

```
void consumidor(void)
```

```
{
    while (1)
    {
        wait(mutex); bloquea mutex = 0
        coger(buffer);
        n--; n-1 = 0
        if (n==0) wait(esperar); Si es n=1 o n=-1 no entra, y libera. esperar = 0 bloquea
        signal(mutex); libera mutex = 1
        consumir();
    }
}
```

Dicha solución no es correcta. Se pide:

- Encontrar un contraejemplo que suponga el fallo de esta solución.
- Corregir el código de manera que el problema encontrado sea solucionado, o implementar un nuevo código que funcione.

Solución

- Puede ocurrir que el productor genere elementos y el consumidor los consuma, en un momento dado hay un solo elemento, el consumidor ejecuta la función consumir con lo cual el buffer queda con 0 elementos. Se consulta el valor de la variable n y como es 0 el consumidor debe quedarse dormido antes de soltar el mutex, con lo cual se produce un interbloqueo. El consumidor está bloqueado e



Ejercicios de concurrencia

impide que el consumidor pueda ejecutarse al no soltar el semáforo mutex antes de quedarse dormido.

Acción	N	esperar
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)

Solución 1: proteger la instrucción que comprueba el valor de n en el consumidor con el semáforo mutex

```
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);
    }
}
```




Ejercicios de concurrencia

Solución 2: Añadir una variable local al procedimiento consumidor y evaluar esta variable para dejarle dormido en lugar de evaluar la variable global 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);  
    }  
}
```



Ejercicios de concurrencia

EJERCICIO 4

Dado el siguiente esquema:

```

Proceso P1          Proceso P2
...
accion1()           accion2()

```

Asegurar que la acción1() siempre se ejecuta antes que la acción2(), mediante:

- Semáforos.
- Mutex y variables condicionales.

SOLUCIÓN

- Semáforos (con un semáforo s inicializado a 0)

```

Proceso P1          Proceso P2
...
accion1()           wait(s)
                    s = 0 => s = -1 => s = 0
                    accion2()
                    signal(s) s = -1 => s = 0

```

Handwritten notes:
 Bloquear sem
 P2 espera
 P1 ejecuta y desbloquea P2
 P2 ejecuta.

- Mutex y variables condicionales.

```

Proceso P1          Proceso P2
...
accion1()           lock(mutex);
lock(mutex);        continuar = true;
cond_signal(var_cond);
unlock(mutex);       while(continuar != true) {
                    cond_wait(mutex, var_cond);
                    }
                    unlock(mutex);
                    accion2()

```

Handwritten notes:
 mutex_lock
 ...
 while (no 1
 cond_wait
 ...
 cond_signal
 mutex_unlock



Ejercicios de concurrencia

EJERCICIO 5

Lectores-Escritores con semáforos.

- Dando prioridad a los lectores (un escritor no puede acceder a la modificación del recurso si tengo lectores que quieren consultarlo).
- Dando prioridad a los escritores (un nuevo lector no puede acceder a la lectura del recurso si existen escritores que desean modificarlo).

SOLUCIÓN

- Dando prioridad a los lectores.

Valores iniciales de semáforos: 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);
```

- Dando prioridad a los escritores.

Valores iniciales de sems: sem_lectores = 1; sem_recurso = 1; sem_escritores = 1;
lectores = 1;

```
Lector() {
    wait(lectores);
    wait(sem_lectores);
    num_lectores = num_lectores + 1;
    if(num_lectores == 1)
        RECURSO
        //ACCESO A RECURSO
        ...
        wait(sem_lectores);
}

Escritor(){
    wait(sem_escritores);
    num_escritores = num_
    if(num_escritores == 1)
        wait(lectores);
    signal(sem_escritores);
    wait(sem_recurso);
    ...
    //MODIFICACION DEL
    ...
    signal(sem_recurso);
    wait(sem_escritores);
```



Ejercicios de concurrencia

```
num_lectores = num_lectores - 1;      num_escritores = num_  
escritores - 1  
if(num_lectores == 0)                 if(num_escritores == 0)  
    signal(sem_recurso);               signal(lectores);  
    signal(sem_lectores);             signal(sem_escritores);  
}
```



Ejercicios de concurrencia

EJERCICIO 6

Un famoso firma autógrafos en una tienda. El famoso solo puede firmar un autógrafo a la vez. La sala donde se firma tiene un aforo limitado de 20 plazas. El famoso dice que solo saldrá a firmar autógrafos si en la sala hay más de 5 personas. Si no hay al menos 5 personas en la sala, dormirá hasta que las haya (en el momento que haya 4 personas o menos se pondrá a dormir). Las personas que quieran firmar y no puedan entrar a la sala por rebasar el aforo permitido se irán sin poder recibir el autógrafo, las que reciban el autógrafo abandonarán la sala.

El famoso representa a un proceso ligero de un tipo que permanece siempre en el sistema y ejecuta la función famoso. Las personas representan a procesos ligeros que ejecutan la función fan.

<pre>void famoso() { while(1) { //Código del proceso famoso } }</pre>	<pre>void fan() { // Código del fan }</pre>
---	---

Dadas las siguientes definiciones compartidas por todos los procesos:

```
#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_cond_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
```

SE PIDE: Codificar las funciones ‘famoso’ y ‘fan’ utilizando el mutex y las variables condicionales dadas.

NOTA: No hay que inicializar los mutex ni las variables condicionales, se suponen ya inicializadas.

SOLUCIÓN

<pre>#include <pthread.h> #include <stdio.h> #include <stdlib.h></pre>
--



Ejercicios de concurrencia

```
#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);
}
```



Ejercicios de concurrencia

```
    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);
}
```



Universidad
Carlos III de Madrid

Departamento de Informática
Grado en Ingeniería Informática
Sistemas Operativos



Ejercicios de concurrencia



Ejercicios de concurrencia

EJERCICIO 7

Ejercicio de válvulas de riego.

Realizar un programa en C que sirva para controlar un sistema de riego de 5 válvulas de riego y 3 entradas de agua.

El programa tendrá un thread por cada válvula de riego y un thread por cada entrada.

Se mostrará un menú al usuario que será el que decida si se debe abrir o cerrar una entrada de agua.

Cuando el usuario decide abrir una entrada de agua uno de los threads de entrada se colocará como abierto y cuando decida cerrar una entrada uno de los threads de entrada abiertos pasará a su estado de cerrado

En número de válvulas de riego abiertas debe ser igual o menor que el número de entradas abiertas. Cuando una entrada se abra se debe abrir también una válvula de riego. El funcionamiento de las válvulas de riego debe ser el siguiente;

1. el thread de la válvula debe esperar a que el número de entradas sea mayor que el número de válvulas abiertas,
2. cuando así sea el thread de la válvula intentará tomar el derecho a ser él el que pase a estado de abierto,
3. en este estado estará 3 segundos,
4. después dará paso a otro thread y
5. estará 1 segundo hasta volver a intentar pasar de nuevo al estado de abierto si la situación en ese instante se lo permite.

Durante los 3 segundos de espera con la válvula abierta no será necesario controlar si el número de entradas abiertas es mayor o igual que el número de válvulas.

SOLUCIÓN

```
/*Este programa sirve para controlar un sistema de riego de 5 válvulas
de riego y 3 entradas de agua.
 * Se debe cumplir la restricción de que nunca debe de haber abiertas
un número de válvulas superior al número de entradas de agua abiertas
*/

#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;
```



Ejercicios de concurrencia

```
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);
}
```



Ejercicios de concurrencia

```
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);
}
```



Ejercicios de concurrencia

EJERCICIO 8

Indique que realiza el siguiente código:

<pre>#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; 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>	<pre>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>
--	--

SOLUCIÓN

El proceso principal creará 10 procesos ligeros. Cada uno de estos procesos ligeros establece un rango (inicio... fin) en el que guardar valores en el vector. Cuando cada trabajador termina de guardar valores, incrementa b.n y pregunta si b.n es igual a N:

Ejercicios de concurrencia

- Si el proceso no es el último: ($n \leq N$) entonces el proceso se duerme.
- Si el proceso es el último ($n > N$) entonces el proceso despierta a todos los procesos ligeros dormidos.

El proceso principal espera al final a los procesos ligeros.

EJERCICIO 9

Añadir una variable local al procedimiento consumidor y evaluar esta variable para dejarle dormido en lugar de evaluar la variable global n .

Solución

```
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();
    }
}
```



Ejercicios de concurrencia

}

EJERCICIO 9

Escribir un programa que ejecuta el problema de la barbería que atiende clientes. Los barberos pueden atender como máximo a 4 clientes dentro de la barbería. Si no hay trabajo los barberos duermen. La barbería se modela como un proceso. Cada cliente que entra ocupa un sillón. Si ya está todo ocupado, los clientes intentan entrar y si no pueden se van.

Solución

```
#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);
}
```



Ejercicios de concurrencia

```
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 procesos 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);
        }
    }
}
```



Ejercicios de concurrencia

```
}
    pthread_mutex_destroy(&m);
}
```

EJERCICIO 10

Escribir un programa que resuelva el programa de los filósofos que comen, para 5 filósofos usando MUTEX. Cinco filósofos se sientan alrededor de una mesa y pasan su vida cenando y pensando. Cada filósofo tiene un plato de fideos y un tenedor a la izquierda de su plato. Para comer los fideos son necesarios dos tenedores y cada filósofo sólo puede tomar los que están a su izquierda y derecha.

Solución

```
#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)
```




Ejercicios de concurrencia

```
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);
```



Ejercicios de concurrencia

```
    exit(0);  
}
```

EJERCICIO 11

Realizar un programa que declare una función imprimir y que le pase como parámetros 1 string a imprimir.

A continuación el programa principal debe preparar los parámetros con 2 string "hola" y "mundo \n" y lanzar 2 threads que intenten imprimir "hola mundo" en ese orden N veces y terminar

Solución

```
#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 la 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[]="Hola ";  
    char cadena_mundo[]="mundo \n";  
    int i;  
  
    pthread_attr_init (&attr);
```



Ejercicios de concurrencia

```
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);  
}
```

EJERCICIO 12

Escribir un programa sencillo para ver como funcionan los mutex. El programa principal crea 4 threads y espera hasta que han terminado todos. Lo normal sería que el main hiciera un pthread_join, pero se debe hacer con mutex para que se vea como utilizarlos.

Solución

```
#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
```



Ejercicios de concurrencia

```
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 que se cree el thread, aunque esta no es la forma adecuada lo
    normal sería 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");
}
```

EJERCICIO 13

Escribir un programa para probar las barreras de POSIX. El programa debe crear seis threads y una barrera. Cada thread debe dormir 3 segundos y esperar para poder pasar la barrera. El padre debe esperar a que terminen todos los threads.

Solución

```
#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)
```



Ejercicios de concurrencia

```
{
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");
}
```

EJERCICIO 14

Implementar un programa que resuelva el problema del productor-consumidor con MUTEX. El programa describe dos thread, productor y consumidor, que comparten un **buffer** de tamaño finito. La tarea del productor es generar un número entero, almacenarlo y comenzar nuevamente; mientras que el consumidor toma (simultáneamente) números uno a uno. El problema consiste en que el productor no añada más números que la capacidad del buffer y que el consumidor no intente tomar un número si el buffer está vacío.

Solución

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX_BUFFER 10 /* tamaño 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 */
```



Ejercicios de concurrencia

```
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 consumidor */
    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);
}
```



Ejercicios de concurrencia

```
    exit(0);  
}
```

EJERCICIO 15

Implementar un programa que resuelva el problema del productor-consumidor con Semáforos POSIX. El programa describe dos thread, productor y consumidor, que comparten un **buffer** de tamaño finito. La tarea del productor es generar un número entero, almacenarlo y comenzar nuevamente; mientras que el consumidor toma (simultáneamente) números uno a uno. El problema consiste en que el productor no añada más números que la capacidad del buffer y que el consumidor no intente tomar un número si el buffer está vacío.

Solución

```
#include <unistd.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <pthread.h>  
#include <semaphore.h>  
  
#define MAX_BUFFER      1024  /* tamaño 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 procesos ligeros */  
    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);  
}
```



Ejercicios de concurrencia

```
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 */
        /* consumir dato */
    }
    pthread_exit(0);
}
```

EJERCICIO 16

Realizar un programa que cree 10 "threads", el primer "thread" sumara los números del 001-100 de un fichero que contiene 1000 numeros, y los siguientes "threads" sumaran sucesivamente los numeros que les correspondan: 101-200, 201-300, 301-400, 401-500, 601-700, 701-800, 801-900 y 901-1000 respectivamente. Los hijos devolveran al padre la suma realizada, imprimiendo este la suma total.

Utilice MUTEX para asegurar que no hay problemas de concurrencia entre los threads.

Solución

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/stat.h>
```




Ejercicios de concurrencia

```
#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);
        // sleep (1);
        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;
```



Ejercicios de concurrencia

```
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);
}
```



Universidad
Carlos III de Madrid

Departamento de Informática
Grado en Ingeniería Informática
Sistemas Operativos



Ejercicios de concurrencia