**Exercise 1**

What is this programa doing?:

|  |  |
| --- | --- |
| #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;  } | 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;  } |

***SOLUTION***

The main process will create 10 lightweight processes. Each of these lightweight processes 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 one (n> N) then the process wakes up all dormant light processes.

The main process waits at the end for the light processes.

**Exercise 2**

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);

}

**EJERCICIO 3**

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 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);

}

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);

}

**EJERCICIO 4**

Make a program that creates 10 "threads", the first "thread" will add the numbers from 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);

// 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;

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);

}