#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

#define NITER 1000000

int cnt = 0;

sem\_t mutex;

void \* Count(void \* a)

{

int i, tmp;

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

{

//secciòn critica

// sem\_wait sirve para trarar de ingresar al codigo despues del semaforo

sem\_wait(&mutex);// 1 -> 0 ò 0 wait

tmp = cnt; /\* copy the global cnt locally \*/

tmp = tmp+1; /\* increment the local copy \*/

cnt = tmp; /\* store the local value into the global cnt \*/

//podrìa usarse cnt++;

sem\_post(&mutex);//mutex de 0 -> 1

}

}

int main(int argc, char \* argv[])

{

//mutex siempre es 1

sem\_init(&mutex,0,1);

pthread\_t tid1, tid2;

if(pthread\_create(&tid1, NULL, Count, NULL))

{

printf("\n ERROR creating thread 1");

exit(1);

}

if(pthread\_create(&tid2, NULL, Count, NULL))

{

printf("\n ERROR creating thread 2");

exit(1);

}

if(pthread\_join(tid1, NULL)) /\* wait for the thread 1 to finish \*/

{

printf("\n ERROR joining thread");

exit(1);

}

if(pthread\_join(tid2, NULL)) /\* wait for the thread 2 to finish \*/

{

printf("\n ERROR joining thread");

exit(1);

}

if (cnt < 2 \* NITER)

printf("\n BOOM! cnt is [%d], should be %d\n", cnt, 2\*NITER);

else

printf("\n OK! cnt is [%d]\n", cnt);

pthread\_exit(NULL);

}

Este es el segundo codigo sem2\_0

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

/\*

File originally obtained from http://www.csc.villanova.edu/~mdamian/threads/posixsem.html, modified afterwards.

This exercise involves solving the classic problem of producers and consumers using semaphores, also known as the bounded buffer problem. The goal is to complete the provided code in order to manage the race conditions of producers and consumers: a producer only produces if there is a free space in the buffer, a consumer only consumer if there is something in the buffer, and only one of them at a time can modify the buffer

\*/

#define BUFF\_SIZE 5 /\* total number of slots \*/

#define NP 3 /\* total number of producers \*/

#define NC 3 /\* total number of consumers \*/

#define NITERS 4 /\* number of items produced/consumed per producer/consumer\*/

typedef struct {

int buf[BUFF\_SIZE]; /\* shared var \*/

int in; /\* buf[in%BUFF\_SIZE] is the first empty slot \*/

int out; /\* buf[out%BUFF\_SIZE] is the first full slot \*/

sem\_t full; /\* keep track of the number of full spots \*/

sem\_t empty; /\* keep track of the number of empty spots \*/

sem\_t mutex; /\* enforce mutual exclusion to shared data \*///llevar aacabo la secciòn critica en la exclusion mutua

} sbuf\_t;

sbuf\_t shared;

void \*Producer(void \*arg)

{

;

int index, item, i;

int \*dir = (int \*) arg;

index = (int) arg;

for (i=0; i < NITERS; i++) {

/\* Produce item \*/

item = i;

/\* Prepare to write item to buf \*/

/\* If there are no empty slots, wait \*/

sem\_wait(&shared.empty);

/\* If another thread uses the buffer, wait \*/

sem\_wait(&shared.mutex);

shared.buf[shared.in] = item;

// printf("shared.in %d\n", shared.in);

shared.in = (shared.in+1)%BUFF\_SIZE;//modulo es porque solo hay 4 productos y si son 5 o màs el modulo ayuda a que sea circula que regreses a 0

printf("[P%d] Producing %d ...\n", index, item); fflush(stdout);

/\* Release the buffer \*/ //libero el semaforo de mutex

sem\_post(&shared.mutex);

/\* Increment the number of full slots \*/

sem\_post(&shared.full);

/\* Interleave producer and consumer execution \*/

if (i % 2 == 1) sleep(1);

}

return NULL;

}

void \*Consumer(void \*arg)

{

/\* Fill in the code here \*/

// The goal is to complete the provided code in order to manage the race conditions

// of producers and consumers: a producer only produces

// if there is a free space in the buffer, a consumer only consumer if there is something in

// the buffer, and only one of them at a time can modify the buffer

int index, item, i;

index = (int) arg;

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

{

/\* Interleave producer and consumer execution to show not consumed items are

consumed next time it runs.

\*/

if (i % 2 == 1) sleep(1);

// Wait if no items

//Esperar a un espacio lleno

sem\_wait(&shared.full);

/\* LOCK buffer \*/

sem\_wait(&shared.mutex);

// Get item

item = shared.buf[shared.out];

shared.out = (shared.out+1)%BUFF\_SIZE; // Update next to consume

printf("------> [C%d] Consuming %d ...\n", index, item); fflush(stdout);

// Unlock

//Regreso el valor de mutex de 0 -> 1

sem\_post(&shared.mutex);

//Incrementa el numero de espacios vacios

sem\_post(&shared.empty);

}

return NULL;

}

int main()

{

pthread\_t idP, idC;

int index;

sem\_init(&shared.full, 0, 0);

sem\_init(&shared.empty, 0, BUFF\_SIZE);

/\* Insert code here to initialize mutex\*/

sem\_init(&shared.mutex,0,1);

for (index = 0; index < NP; index++)

{

/\* Create a new producer \*/

pthread\_create(&idP, NULL, Producer, (void\*)&index);

}

/\* Insert code here to create NC consumers\*/

for (index = 0; index < NC; index++)

{

/\* Create a new producer \*/

pthread\_create(&idC, NULL, Consumer, (void\*)&index);

}

pthread\_exit(NULL);//This line turns the main thread into a zombie, remove it

/\* Insert code here to wait for the threads to finish \*/

}