Algoritmos de Dekker y Peterson

# Algoritmo de Dekker

#include <stdio.h>

#include <pthread.h>

#define TRUE 1

#define FALSE 0

volatile \_Bool threadAWant, threadBWant;

volatile int turn = 0;

volatile int count = 0;

volatile int limit = 0; //número de veces máximo que cada thread suma 1 a count,

//N = 100, 1000, 10000....

//funcion del primer thread

void\* functionThreadA(void \*arg) {

int iteration = 0;

pthread\_t tid = pthread\_self();

printf("Thread %lu\n", tid);

while (1) {

//el thread "quiere" entrar a la sección crítica

threadAWant = TRUE;

\_\_sync\_synchronize();

while (threadBWant == TRUE) {

if (turn == 1) {

threadAWant = FALSE;

while (turn != 0) {

}

threadAWant = TRUE;

}

\_\_sync\_synchronize();

}

//inicia seccion crítica

++count;

printf("In thread #%ld -> %d;\n", tid, count);

++iteration;

//fin de la sección crítica

turn = 1;

threadAWant = FALSE;

\_\_sync\_synchronize();

if (iteration == limit)

return 0;

}

}

//funcion del segundo thread

void\* functionThreadB(void \*arg) {

long iteration = 0;

pthread\_t tid = pthread\_self();

printf("Thread %lu\n", tid);

while (1) {

//el thread "quiere" entrar a la sección crítica

threadBWant = TRUE;

\_\_sync\_synchronize();

while (threadAWant == TRUE) {

if (turn != 1) {

threadBWant = FALSE;

while (turn != 1) {

}

threadBWant = TRUE;

}

\_\_sync\_synchronize();

}

//inicia seccion crítica

++count;

printf("tid #%ld -> %d;\n", tid, count);

++iteration;

//termina sección crítica

turn = 0;

threadBWant = FALSE;

\_\_sync\_synchronize();

if (iteration == limit)

return 0;

}

}

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

threadAWant = 0;

threadBWant = 0;

pthread\_t threadA, threadB;

printf("\nnumero de veces de suma (N): ");

scanf("%d", &limit);

pthread\_create(&threadA, 0, functionThreadA, 0);

pthread\_create(&threadB, 0, functionThreadB, 0);

pthread\_join(threadA, 0);

pthread\_join(threadB, 0);

printf("\nValor final de count: %d\n", count);

return 0;

}

# Algoritmo de Peterson

// Filename: peterson\_yieldlock\_memoryfence.c

// Use below command to compile:

// gcc -pthread peterson\_yieldlock\_memoryfence.c -o peterson\_yieldlock\_memoryfence

#include<stdio.h>

#include<pthread.h>

*/\* #include "mythreads.h" \**/

int flag[2];

int turn;

const int MAX = 1e9;

int ans = 0;

void lock\_init()

{

// Initialize lock by reseting the desire of

// both the threads to acquire the locks.

// And, giving turn to one of them.

flag[0] = flag[1] = 0;

turn = 0;

}

// Executed before entering critical section

void lock(int self)

{

// Set flag[self] = 1 saying you want

// to acquire lock

flag[self]=1;

// But, first give the other thread the

// chance to acquire lock

turn = 1-self;

// Memory fence to prevent the reordering

// of instructions beyond this barrier.

\_\_sync\_synchronize();

// Wait untill the other thread looses the

// desire to acquire lock or it is your

// turn to get the lock.

while (flag[1-self]==1 && turn==1-self)

// Yield to avoid wastage of resources.

sched\_yield();

}

// Executed after leaving critical section

void unlock(int self)

{

// You do not desire to acquire lock in future.

// This will allow the other thread to acquire

// the lock.

flag[self]=0;

}

// A Sample function run by two threads created

// in main()

void\* func(void \*s)

{

int i = 0;

int self = (int \*)s;

printf("Thread Entered: %d\n",self);

lock(self);

// Critical section (Only one thread

// can enter here at a time)

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

ans++;

unlock(self);

}

// Driver code

int main() {

pthread\_t p1, p2;

// Initialize the lock

lock\_init();

// Create two threads (both run func)

Pthread\_create(&p1, NULL, func, (void\*)0);

Pthread\_create(&p2, NULL, func, (void\*)1);

// Wait for the threads to end.

Pthread\_join(p1, NULL);

Pthread\_join(p2, NULL);

printf("Actual Count: %d | Expected Count:"

" %d\n",ans,MAX\*2);

return 0;

}

// mythread.h (A wrapper header file with assert

// statements)

#ifndef \_\_MYTHREADS\_h\_\_

#define \_\_MYTHREADS\_h\_\_

#include <pthread.h>

#include <assert.h>

#include <sched.h>

void Pthread\_mutex\_lock(pthread\_mutex\_t \*m)

{

int rc = pthread\_mutex\_lock(m);

assert(rc == 0);

}

void Pthread\_mutex\_unlock(pthread\_mutex\_t \*m)

{

int rc = pthread\_mutex\_unlock(m);

assert(rc == 0);

}

void Pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr,

void \*(\*start\_routine)(void\*), void \*arg)

{

int rc = pthread\_create(thread, attr, start\_routine, arg);

assert(rc == 0);

}

void Pthread\_join(pthread\_t thread, void \*\*value\_ptr)

{

int rc = pthread\_join(thread, value\_ptr);

assert(rc == 0);

}

#endif // \_\_MYTHREADS\_h\_\_

REFERENCIA(S)

https://www.geeksforgeeks.org/petersons-algorithm-for-mutual-exclusion-set-1/

https://www.geeksforgeeks.org/petersons-algorithm-for-mutual-exclusion-set-2-cpu-cycles-and-memory-fence/?ref=rp