9

Practica 6 : MUTEX
Sistemas Operativos

Universidad Autónoma Metropolitana – Azcapotzalco
9 de julio de 2018

Índice

1. Introducción

El uso de procesos o hilos tiene grandes ventajas cuando un programa tiene que realizar varias tareas simultáneamente pues la división de tareas puede ayudar a que su desempeño sea mejor, por su puesto esto varia dependiendo del sistema operativo que se use y del hardware con el que se disponga, aunque al tener varios procesos trabajando de forma concurrente se pueden tener ciertos riesgos di estos usan recursos compartidos los cuales se pueden dar si dos procesos tienen que usar un mismo dispositivo de E/S, o si usan una memoria compartida para comunicarse entre ellos; en ambos casos si los procesos tratasen de acceder al recurso simultáneamente terminarían dándose errores en la ejecución del programa. A esto ultimo se le conoce como condiciones de carrera y a la parte del código de los programas donde ocurren estas condiciones se nombran como región critica, existen varias soluciones para este problema basadas en software y/o hardware las cuales deben satisfaces ciertos requisitos[1] como lo son la exclusión mutua (no puede haber mas de dos procesos en sus regiones críticas), progreso (solo los procesos que estén ejecutando su región critica pueden decidir quien sera el siguiente en ocupar el recurso) y espera limitada (ningún proceso puede ocupar el recurso por un tiempo indefinido).

Algunas soluciones para este problema son: los semáforos los cuales se implementa en hardware y en software, de la misma forma se hacen los mutex aunque estos son una versión simplificada de los semáforos y la solución de Petterson la cual se implementa completamente en software.

2. Desarrollo

Se hizo un programa que simulara el problema del productor consumidor el cual consiste en tener en un programa dos procesos (en este caso la implementación se hará con hilos con el uso de la API PTHREAD) que compartirán un mismo arreglo de datos y una variable que indique el tamaño de este, uno de los procesos agregara un producto en la localidad de memoria que sera indicada con el tamaño del arreglo en ese momento y aumentara en una unidad esta variable (lo que significara poner una 'P' en dicha posición del arreglo) esto hasta que el arreglo alcance su capacidad máxima, por otra parte, de manera similar el otro procesos quitara un producto del arreglo a la vez (asignando un ' ' a la localidad de memoria correspondiente) hasta que el tamaño del arreglo sea cero. En el siguiente código se muestra la implementación del problema antes mencionado, en las funciones de productor y consumidor se remarcan las regiones criticas del programa que se dan cuando estos compiten por el recurso, en este caso escribir en la localidad de memoria correspondiente, para resolver el problema que se da por las condiciones de carrera se implemento un mutex en donde para asegurar la exclusión mutua el primer proceso en abrir el mutex podrá acceder al recurso mientras el otro tendrá que esperar a que se desbloquee el mutex para acceder a dicho recurso.

Código 1: Programa Principal.

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

#define N 10
#define MAX 50

char buffer[N] = {'_'};
pthread_mutex_t mutex;
pthread_cond_t condP, condC;
int pos = 0;

void imprimeBuffer(const char *buf){
    for(int i = 0; i < N; ++i){
        printf("%c", buffer[i]);
    }
    printf("\n");
}

void *productor(void *arg){
    for(int i = 0; i < MAX; ++i){
        pthread_mutex_lock(&mutex);
        if(pos >= 5) pthread_cond_wait(&condP, &mutex);
        buffer[pos++] = 'P';//Region Critica
        imprimeBuffer(buffer);
        if(pos >= 5) pthread_cond_signal(&condC);
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(0);
}

void *consumidor(void *arg){
    int rc;
    struct timespec ts;
    struct timeval tp;
```

```
    gettimeofday(&tp, NULL);

    ts.tv_sec  = tp.tv_sec;
    ts.tv_nsec = tp.tv_usec * 1000;
    ts.tv_sec += 2;

    for(int i = 0; i < MAX; ++i){
        pthread_mutex_lock(&mutex);
        if(pos == 0) pthread_cond_wait(&condC, &mutex);
        pthread_cond_timedwait(&condC, &mutex, &ts);
        buffer[--pos] = '_';//Region Critica
        imprimeBuffer(buffer);
        if(pos == 0) pthread_cond_signal(&condP);
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(0);
}

int main(int argc, char const *argv[])
{
    pthread_t prod, cons;
    pthread_mutex_init(&mutex, 0);
    pthread_cond_init(&condC, 0);
    pthread_cond_init(&condP, 0);
    pthread_create(&cons, NULL, consumidor, NULL);
    pthread_create(&prod, NULL, productor, NULL);
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);
    pthread_cond_destroy(&condC);
    pthread_cond_destroy(&condP);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

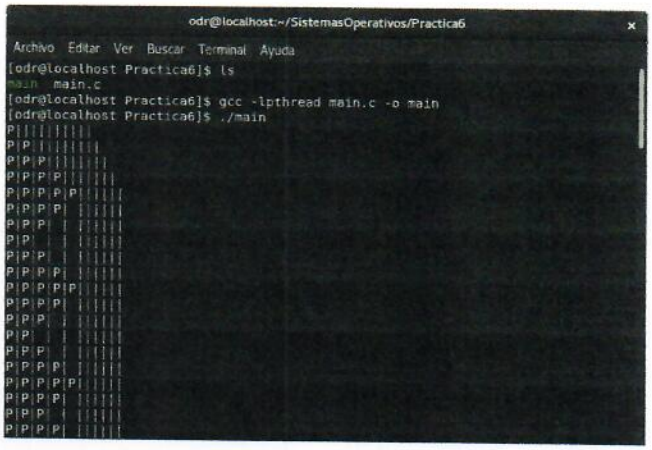

Figura 1: Primera Versión.

En la segunda versión el productor tendrá que esperar a que el consumidor vacié el arreglo para poder agregar mas productos por lo que ahora el consumidor desbloqueara al productor una vez que el tamaño del arreglo sea cero y este se bloqueara hasta que el productor haya llenado el arreglo con la cantidad de productos que se requiera, esto se puede observar de una mejor manera en la Figura 2. La razón por la que la región critica se encuentra entre el bloqueo y desbloqueo de los hilos es para asegurar la espera limitada ya que si se incluye en esta parte la región no critica de la función (tanto del productor como del consumidor) este podría tardar mas de lo que necesita y estaría apartando el recurso de los demás sin hacer uso de este.

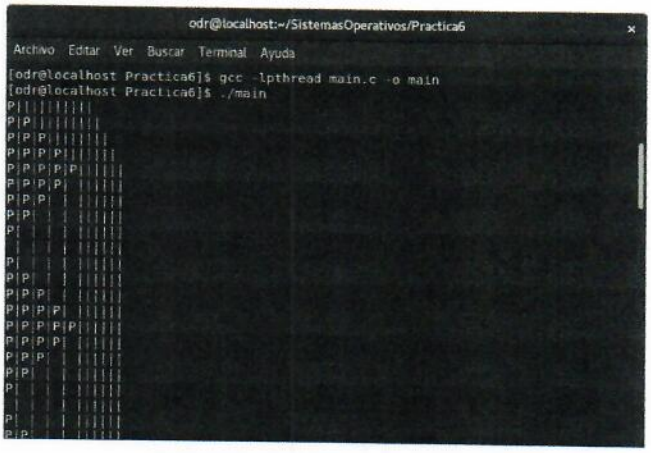

Figura 2: Segunda Versión.

Ahora se expondrán dos versiones del anterior código, en ambas versiones se usaran la funciones "pthread_cond_waitz "pthread_cond_signal"la primera bloqueara al hilo que hace uso de esta función hasta que se mande una señal que lo desbloquee la cual se manda con la segunda función. En la primera versión el productor se bloqueara hasta que logre completar cinco productos y quedara en espera de que el consumidor quite algún producto para ser desbloqueado. Debido a que en este caso prácticamente no hay una parte del código de la función que no sea critica el tiempo que tarda en completarse la transición entre el consumidor y el productor es instantáneo por lo que no se nota que el productor puede estar agregando productos independientemente de si el arreglo no se ha vaciado. Para poder hacer mas notorio este comportamiento se implemento la función "pthread_cond_timedwait"para que el consumidor tarde unos milisegundos en desbloquear al productor y así es como en la Figura 1 se muestra el comportamiento antes mencionado, también cabe mencionar que la condición que esta antes de la función "pthread_cond_signal.[en] el consumidor no se hace uso en esta versión.

3. Conclusión

Si bien el uso de varios procesos o hilos podrían optimizan la ejecución de un programa es importante tener una sincronización entre estos pues como se mostró al compartir varios recursos se pueden dar condiciones de carrera con estos recursos generando

así errores dentro del programa. La solución por mutex mostrada aquí es mas óptima que otras como la solución de Petterson ya que los mutex al tener implementación conjunta en hardware y software cuando un hilo no este en su región critica puede bloquear se ejecución mientras que la segunda solución al estar completamente implentada en software la forma de bloquearlos es mantenerlos en un ciclo por lo que todos los hilos siempre se estan ejecutando.

Referencias

1. . G. G. Silberschatz A., Galvin P., *Fundamentos de Sistemas Operativos*, 7.ª ed. McGraw-Hill, 2006, pp. 172 - 174.