Sistemas Operativos y Distribuidos

Anexo práctico 4

Casos de estudio

GNU/Linux

La realización del anexo GNU/Linux es obligatorio. Es importante la comprensión de estos contenidos dado que serán necesarios para la resolución de los próximos anexos.

A.L.4.1. Dado el siguiente programa implementado en *POSIX Threads*, en el cual se pretende que se calcule el doble de TOTAL:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define TOTAL 10000000

int sum = 0;

void *sumamucho(void *param) {
    int i;
    for (i = 0; i < TOTAL; i++) {
        sum += 1;
    }
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, sumamucho, NULL);
    pthread_create(&tid2, NULL, sumamucho, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("Total es %d\n", sum);
    return EXIT_SUCCESS;
}
```

a) Ejecutar varias veces el programa y analizar los resultados obtenidos. ¿Por qué no se obtiene el valor deseado? Explicar la respuesta.
b) Un programador inexperto intenta corregir el programa anterior añadiendo la función `sleep(5)` luego de la creación del primer hilo. ¿Logra solucionarse el problema al añadir esta función? ¿Podrías contradecir de alguna manera la solución propuesta por el programador?
c) Dar al menos tres soluciones posibles para el programa anterior con funciones de la librería de *POSIX Threads*.
d) De las soluciones anteriores, ¿cuál es la más lenta y cuál es la que ofrece mejor *performance*?

A.L.4.2. Ejecutar y explicar el siguiente programa implementado en *POSIX Threads*.

Nota: este programa es una versión modificada de un ejemplo del libro *Pthreads Programming* (Nichols, B., D. Buttlar & J. Proulx, 1996. *Pthreads Programming*. O'Reilly & Associates, USA. ISBN: 1-5692-115-1).

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define WHITE "\033[m"
#define RED "\033[31m"
#define GREEN "\033[32m"
#define TCOUNT 10
#define COUNT_LIMIT 12
int count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t){
    int i;
    long my_id = (long)t;
    for (i=0; i<TCOUNT; i++) {
        pthread_mutex_lock(&count_mutex);
        count++;
        if (count == COUNT_LIMIT) {
            pthread_cond_signal(&count_threshold_cv);
            printf(RED"inc_count(): hilo %ld, count = %d  Umbral alcanzado.\n"WHITE, my_id, count);
        }
        printf(GREEN"inc_count(): hilo %ld, count = %d, unlocking mutex\n"WHITE, my_id, count);
        pthread_mutex_unlock(&count_mutex);
        sleep(1);
    }
    pthread_exit(NULL);
}

void *watch_count(void *t){
    long my_id = (long)t;
    printf("watch_count(): hilo %ld\n", my_id);
    pthread_mutex_lock(&count_mutex);
    while (count<COUNT_LIMIT) {
        pthread_cond_wait(&count_threshold_cv, &count_mutex);
        printf("watch_count(): hilo %ld señal de condición recibida.\n", my_id);
        count += 100;
        printf("watch_count(): hilo %ld count = %d.\n", my_id, count);
    }
    pthread_mutex_unlock(&count_mutex);
    pthread_exit(NULL);
}

int main (int argc, char *argv[]){
    int i; long t1=1, t2=2, t3=3;
    pthread_t threads[3];
    pthread_attr_t attr;
    pthread_mutex_init(&count_mutex, NULL);
    pthread_cond_init (&count_threshold_cv, NULL);
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    pthread_create(&threads[0], &attr, watch_count, (void *)t1);
    pthread_create(&threads[1], &attr, inc_count, (void *)t2);
    pthread_create(&threads[2], &attr, inc_count, (void *)t3);
    for (i=0; i<3; i++) {
        pthread_join(threads[i], NULL);
    }
    pthread_attr_destroy(&attr);
    pthread_mutex_destroy(&count_mutex);
    pthread_cond_destroy(&count_threshold_cv);
    return EXIT_SUCCESS;
}
```

A.L.4.3. Implementar el problema del **productor-consumidor** utilizando únicamente la *API POSIX Threads*.

A.L.4.4. Implementar el **primer y segundo problema de los lectores y escritores** utilizando únicamente la *API POSIX Threads*.

A.L.4.5. Implementar cada uno de los incisos del ejercicio **4.13** utilizando *POSIX Threads* y la librería `semaphore.h` para la sincronización.

A.L.4.6. Implementar el ejercicio **4.14** utilizando *POSIX Threads* y la librería `semaphore.h` para la sincronización.

A.L.4.7. Implementar el problema del **productor-consumidor** utilizando *POSIX Threads* y la librería `semaphore.h` para la sincronización.

A.L.4.8. Implementar **el problema de los filósofos cenando** utilizando *POSIX Threads* y la librería `semaphore.h` para la sincronización.

A.L.4.9. Implementar **el problema del barbero dormilón** utilizando *POSIX Threads* y la librería `semaphore.h` para la sincronización.

A.L.4.10. Implementar dos programas que compartan un espacio de memoria compartida *POSIX*, en donde uno de los programas escriba mensajes en la memoria compartida y una vez finalizado permita al otro programa leer los mensajes escritos por el primero. Antes de finalizar, el segundo programa debe eliminar el segmento creado por el primero.

A.L.4.11. Implementar el problema del **productor-consumidor** utilizando procesos, la librería `semaphore.h` y un segmento de memoria compartida en *POSIX*.

A.L.4.12. Resolver el ejercicio **A.L.2.12.** haciendo uso de un segmento de memoria *POSIX* compartido entre el proceso padre y el proceso hijo. En particular, el proceso padre debe crear el segmento de memoria compartida y el proceso hijo debe computar la secuencia de Fibonacci y almacenar los resultados en el segmento compartido. El proceso padre debe mostrar la secuencia por la salida estándar y eliminar el segmento creado antes de finalizar.

A.L.4.13. Implementar una cola de mensajes *POSIX* tal que un proceso envie 10 mensajes con el siguiente formato "`Mensaje nro: <n>`", donde `<n>` se corresponde con el número de mensaje. Al mismo tiempo se debe recibir el mensaje e imprimirlo con el siguiente formato: "`Recibí Mensaje nro: <n>`".

A.L.4.14. Implementar el problema del **productor-consumidor** utilizando procesos y una cola de mensajes *POSIX*.

A.L.4.15. Considerar un servidor que recibe mensajes de un cliente hasta que el cliente envía el mensaje "Fin", momento en el cual los dos finalizan su ejecución. Implementar el servidor y cliente utilizando procesos y una cola de mensajes *POSIX*.

A.L.4.16. Monitorear la memoria compartida y la cola en `/dev`.