Threads - Manejo de hilos. Uso de POSIX Threads

1) Compile el ejemplo que crea 5 hilos (threads.c) utilizando:

$ gcc -Wall -pthread -o ht ht.c

Notas cobre el ejemplo:

a) pthread_create crea el hilo, el primer argumento almacena el Id del Hilo, el segundo pasa parámetros iniciales, el tercero indica cual es la función que ejecuta el hilo, el cuarto es el *unico* argumento que podemos pasar al hilo.

b) Todos los hilos deben invocar pthread_exit al terminar.

2) Use 5 hilos para computar los 5 primeros términos de la serie de Fibonacci

0 1 1 2 3 5 8 13 ...

a_n = a_n-1 + a_n-2

Use una expresión no recursiva para la función de la serie.

Declare un arreglo común donde cada hilo debe almacenar el resultado.

Imprima el arreglo al finalizar

Piense de qué modo 'sincronizar' al hilo principal, recuerde que las variables globales son comunes.

Código en C con pthreads:

```
#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define NUM_THREADS 5


int fibonacci[NUM_THREADS]; // Arreglo compartido para almacenar resultados
```

```
// Función que cada hilo ejecutará
void *compute_fibonacci(void *param) {
    int index = *(int *)param;

    // Cálculo de Fibonacci sin recursión
    if (index == 0) {
        fibonacci[index] = 0;
    } else if (index == 1) {
        fibonacci[index] = 1;
    } else {
        fibonacci[index] = fibonacci[index - 1] + fibonacci[index - 2];
    }

    pthread_exit(0);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];

    // Crear hilos
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_args[i] = i;
        pthread_create(&threads[i], NULL, compute_fibonacci, &thread_args[i]);
```

```
    }

    // Esperar a que todos los hilos terminen
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    // Imprimir resultados
    printf("Fibonacci: ");
    for (int i = 0; i < NUM_THREADS; i++) {
        printf("%d ", fibonacci[i]);
    }
    printf("\n");

    return 0;
}
```

Explicación rápida:

1. **Cada hilo recibe un índice i**, que indica qué término calcular.
2. **Calcula el término i de Fibonacci** sin usar recursión.
3. **Almacena el resultado en el arreglo global fibonacci[]**.

4. **El hilo principal espera con pthread_join()** hasta que todos los hilos terminen.

5. **Imprime los resultados.**

Problema posible: Condición de carrera

Este código **podría fallar** si un hilo intenta calcular fibonacci[i] antes de que fibonacci[i-1] y fibonacci[i-2] estén listos. Para evitarlo, podemos **sincronizar** con pthread_barrier_t o usar pthread_mutex_t si fuera necesario.

¿Por qué usar un mutex?

Cada hilo debe esperar a que los valores previos de Fibonacci sean calculados antes de acceder al arreglo compartido. El mutex garantiza que un solo hilo acceda a la memoria compartida a la vez.

Código en C con pthread_mutex_t

```
#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>


#define NUM_THREADS 5


int fibonacci[NUM_THREADS];  // Arreglo compartido
```

```
pthread_mutex_t mutex;      // Mutex para sincronizar acceso

// Función que cada hilo ejecutará
void *compute_fibonacci(void *param) {
   int index = *(int *)param;

   // Bloquear la sección crítica
   pthread_mutex_lock(&mutex);

   if (index == 0) {
      fibonacci[index] = 0;
   } else if (index == 1) {
      fibonacci[index] = 1;
   } else {
      fibonacci[index] = fibonacci[index - 1] + fibonacci[index - 2];
   }

   // Desbloquear la sección crítica
   pthread_mutex_unlock(&mutex);

   pthread_exit(0);
```

```
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];

    // Inicializar el mutex
    pthread_mutex_init(&mutex, NULL);

    // Crear hilos
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_args[i] = i;
        pthread_create(&threads[i], NULL, compute_fibonacci, &thread_args[i]);
    }

    // Esperar a que todos los hilos terminen
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
```

```
    // Imprimir resultados
    printf("Fibonacci: ");
    for (int i = 0; i < NUM_THREADS; i++) {
        printf("%d ", fibonacci[i]);
    }
    printf("\n");

    // Destruir el mutex
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

Explicación del código con pthread_mutex_t

1. **Se declara un pthread_mutex_t mutex** para proteger la memoria compartida.

2. **Cada hilo bloquea (pthread_mutex_lock()) antes de modificar fibonacci[]**.

3. **Calcula el número de Fibonacci** y **luego desbloquea (pthread_mutex_unlock())**.

4. **El hilo principal espera con pthread_join()** hasta que todos los hilos terminen.

5. **Se destruye el mutex** con pthread_mutex_destroy(&mutex) al final.

Ventaja del Mutex:

Evita condiciones de carrera porque solo un hilo accede al arreglo a la vez.
Sincroniza correctamente el acceso a la memoria compartida.

Posible problema: Este mutex **no impone un orden estricto de ejecución**, por lo que el hilo i podría ejecutarse antes que i-1, provocando valores incorrectos.