

Ejemplo adicional de PThreads: Búsqueda del máximo de un vector

Este documento explica el ejemplo incluido en el archivo “maxVector.c”, que pretende complementar el seminario de PThreads. Concretamente, en “maxVector.c” se ha implementado la búsqueda del máximo de un vector usando PThreads.

Estas son las librerías que vamos a referenciar:

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <time.h>
```

Con ellas vamos a poder, en este orden:

- i) Crear y gestionar hilos PThreads
- ii) Usar funciones de entrada/salida
- iii) Usar las funciones estándar de C
- iv) Disponer de funciones de temporización

Vamos entonces al punto de entrada del programa, el “main”:

```
47  int main(int argc, char* argv){
48      if(argc!=3){
49          printf("Uso: ./programa tamVector numHilos\n");
50      }else{
```

Esperamos un tamaño de vector, que se creará aleatoriamente, y el total de hilos a desplegar. Si disponemos de esos parámetros, los leemos y preparamos tanto el vector, como el resultado, que se inicializa a un valor inválido. Luego comentaremos la función “inicializarVector”.

```
51      int tamVector = atoi(argv[1]);
52      int nHilos = atoi(argv[2]);
53      int* vector = 0;
54      inicializarVector(&vector, tamVector);
55      int resultado = -1;
```

Siguiendo en el “main”, creamos un vector de hilos, un único MUTEX, y un vector de tareas de hilo:

```
57      pthread_t* hilos = malloc(sizeof(pthread_t)*nHilos);
58      pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // Un mutex para "gobernarlos" a todos
59      tarea* deberes = malloc(sizeof(tarea)*nHilos);
```

Habrà tantos hilos como se haya indicado. En consecuencia, habrá también una tarea para cada hilo. Sin embargo, habrá un único MUTEX. Esto es importante, porque tiene que ser algo “común” a todos y que permita proteger una misma

sección crítica. Dicho de otra forma, aquí sería un grave error pensar que cada hilo necesitase un MUTEX propio, pues cada uno tendría plena disposición del suyo y no serviría para sincronizar los distintos accesos. Es decir, no se protegería nada.

El “*main*” continúa con la inicialización de tareas y lanzamiento de hilos:

```
61     for(int i = 0; i<nHilos; i++){
62         deberes[i].idHilo = i;
63         deberes[i].numHilos = nHilos;
64         deberes[i].vector = vector;
65         deberes[i].tamVector = tamVector;
66         deberes[i].mutex = &mutex;
67         deberes[i].resultadoGlobal = &resultado;
68         pthread_create(&(hilos[i]), 0, cuerpoHilo, &(deberes[i]));
69     }
```

Vemos que, de entrada, para cada hilo guardamos su ID y el total de hilos (haremos un balanceo de carga cíclico). Vemos también que guardamos el puntero al vector y su tamaño. Esto choca un poco con ejemplos del seminario, y los que se pueden ver normalmente en libros y tutoriales: lo más típico en ejemplos es usar variables globales. Sin embargo, ese enfoque es poco escalable, realista y aplicable a proyectos complejos. Aquí vemos entonces cómo podemos crear una especie de “contexto común” a los hilos a base de apuntarlos a las mismas zonas de memoria bajo nuestro control. Eso incluye tanto al mutex como a donde queremos guardar el resultado (“*resultadoGlobal*”).

El “*main*” se cierra de la siguiente forma:

```
70     //-----
71     for(int i = 0; i<nHilos; i++){
72         pthread_join(hilos[i], 0);
73     }
74     printf("Resultado: %d\n", resultado);
75
76     free(hilos);
77     free(deberes);
78     free(vector);
79 }
80 return 0;
81 }
```

Vemos simplemente un bucle para recoger cada hilo, el mostrado del resultado, y la liberación de la memoria dinámica reservada. No obstante, es importante destacar un detalle: El bucle de recogida es distinto e independiente al de creación. Es otro bucle, como debe ser. Lo resalto porque a veces he visto a alumnos poner el “*pthread_join*” inmediatamente después de su “*pthread_create*” previo, en un mismo bucle... Y esto, aunque pueda dar el resultado correcto, NO paraleliza el problema, ya que cada hilo lanzado es esperado inmediatamente después (no olvidemos que “*pthread_join*” es una llamada bloqueante).

Pasamos ahora a otras partes del ejemplo. Estaba pendiente la función “*inicializaVector*”, que reserva espacio para “*vector*” y escribe los contenidos del mismo. Lo más destacable es que, como queremos reservar el vector de forma “remota” (puede verse como “por referencia”), lo que necesita la función es “un puntero a nuestro puntero original”, que es el que queremos asignar realmente. Además, es interesante destacar que ponemos, en una posición aleatoria (en [0,

tamaño)) el máximo. Como los valores aleatorios estarán todos entre 0 y 99, poner un único 100 nos garantiza que ese será el resultado correcto, lo que puede verse como una estrategia inteligente de depuración implícita.

```
35 void inicializarVector(int** pointToVec, int tam){
36     *pointToVec = malloc(sizeof(int)*tam);
37     int* vec = *pointToVec;//Esto es como hacer un "alias", por comodidad
38
39     srand(time(0));
40     for(int i = 0; i<tam; i++){
41         vec[i] = rand() % 100; //Enteros de 0 a 99
42     }
43
44     vec[rand() % tam] = 100; // Ponemos un maximo al azar
45 }
```

Vamos a ver ahora la estructura que hemos definido para dar información a los hilos:

```
8 typedef struct{
9     int idHilo;
10    int numHilos;
11    int* vector;
12    int tamVector;
13    pthread_mutex_t* mutex;
14    int* resultadoGlobal;
15 } tarea;
```

Tiene los 6 campos que ya asignamos. Sólo queda insistir en que idHilo puede verse como la parte “específica” de cada hilo, mientras que los demás campos son contexto común. De hecho, podríamos haber creado una estructura “contextoComun” que aglutinara todos esos valores y direcciones de memoria compartidas, en lugar de replicarlas en cada hilo. Cada tarea específica tendría entonces un puntero a dicha estructura de contexto. Sin embargo, no lo he hecho así por sencillez, y porque, al fin y al cabo, estamos replicando unas pocas direcciones solamente.

Ya sólo nos queda el diseño de la función hebrada:

```
17 void* cuerpoHilo(void* arg){
18     tarea misDeberes = *((tarea*) arg);//Copiamos nuestra asignacion
19     int maximoLocal = -1;
20     int candidato = -1;
21     for(int i = misDeberes.idHilo; i<misDeberes.tamVector; i+=misDeberes.numHilos){
22         candidato = misDeberes.vector[i];
23         if(candidato>maximoLocal){
24             maximoLocal = candidato;
25         }
26     }
27     pthread_mutex_lock(misDeberes.mutex);
28     if(maximoLocal > *(misDeberes.resultadoGlobal) ){
29         *(misDeberes.resultadoGlobal) = maximoLocal;
30     }
31     pthread_mutex_unlock(misDeberes.mutex);
32     return 0;
33 }
```

Cada hilo empieza por copiarse su estructura “tarea” propia de-referenciando el puntero que recibe. Así, además de facilitar los accesos (evitamos los “->”), guardamos lo que necesitamos en nuestro stack directamente.

Seguidamente, se prepara una variable para guardar el resultado local que encuentre cada hilo tras su exploración particular (y otra, “candidato”, vinculada a ella para ahorrar lecturas de memoria).

Llegamos entonces al bucle principal de exploración. Como se anticipó, se hace un balanceo de carga cíclico, por lo que cada hilo empieza en la posición igual a su ID, y va saltando el total de hilos (sin exceder la longitud del vector). Dentro del bucle vemos cómo leemos por dónde vamos (como hilo), y actualizamos nuestro resultado local en caso de ser necesario. Todo este trabajo es independiente y paralelizable, pues sólo leemos valores compartidos (y de sólo lectura para todos), mientras que las actualizaciones son de una variable local y propia de cada hilo.

Acabado nuestro recorrido como hilo por el vector, nos encontramos con la sección crítica. Tenemos que bloquear el MUTEX, y comparar nuestro resultado local con el que hay en global (y que convenientemente inicializamos a -1, para que siempre fuera susceptible de actualizarse en primera instancia). Si nuestro resultado local es mayor que el registrado en el global, actualizamos, y desbloqueamos el MUTEX antes de acabar.

Es fundamental destacar aquí dos ideas. Para empezar, la sección crítica es lo más pequeña posible, y es accedida también el menor tiempo posible (una vez por hilo antes de acabar). Dicho de otra forma, sería muy mala idea meter toda esa comparación/actualización en el bucle de recorrido directamente en lugar de usar una variable local del hilo. Igualmente, sería aún peor enmarcar todo el bucle de recorrido dentro de la sección crítica, pues sólo podría haber un hilo explorando por instante. La segunda es que la protección del MUTEX debe incluir la propia lectura del valor global (el “if” de la línea 28). Esto es importante porque, aunque un hilo “sólo” lea ese valor, podría intentar hacerlo mientras otro estuviera escribiendo (en su línea 29). Es decir, es un valor compartido para lectura y escritura. Lo destaco porque es fácil olvidar que puede haber otros hilos escribiendo lo que otros puedan querer leer, y es un error que he visto algunas veces.

Estas serían las ideas clave del ejemplo. Para cualquier cuestión sobre el mismo, podemos verlo en clase.