P4. PROYECTO DE PROGRAMACIÓN CONCURRENTE

Andrés Río Nogués y Francesc Navarro Vázquez y Aleix Falgosa Campoy

21 noviembre, 2024

UNIVERSITAT DE BARCELONA

1 Introducción

En esta práctica, se realiza un programa para procesar las temperaturas registradas en distintos países, generando una tabla que recopila las temperaturas máximas y mínimas, junto con las fechas en las que estas ocurrieron. Inicialmente, se implementará un enfoque secuencial, sin aplicar técnicas de optimización. Posteriormente, se incorporará el uso de hilos y concurrencia, lo que permitirá analizar las ventajas que estas técnicas ofrecen en términos de eficiencia y rendimiento. Veremos un código multihilos y el paradigma productor consumidor.

2 Implementación del código secuencial

En esta parte nos centraremos en un código sin hilos para lograr nuestro objetivo de ver las temperaturas máximas y mínimas.

P1. ¿Cuánto tiempo tarda en ejecutarse la aplicación? ¿Cómo lo puedes calcular?

```
oslabadmin@oslab:/media/sf_PRACTIQUES/P4/Part 1/EarthSurfaceTemperatureData-with_support_code$ time ./secuencial GlobalLandTemperat
uresByCityIDs.csv GlobalLandTemperaturesByCityData.csv

real    0m11.045s
user    0m6.570s
sys     0m1.707s
```

Figura 1: Tiempo que tarda en ejecutarse la aplicación

Para obtener el tiempo que tarda en ejecutarse la aplicación podemos añadir el comando "time" al inicio del comando que usamos para ejecutar la aplicación. Esto nos devolverá 3 tiempos: real, user y sync. real nos da el tiempo total de ejecución del programa; user nos da el tiempo que la CPU ha pasado ejecutando el programa; y sys el tiempo que la CPU ha pasado ejecutando llamadas al sistema en nombre de tu programa. En nuestro caso, la aplicación tarda unos 11 segundos en ejecutarse.

P2. ¿A qué países corresponden el top-10 de las temperaturas más altas?

```
035 39.65
008 39.16
060 38.28
013 38.05
027 37.94
107 37.81
016 37.52
132 37.45
018 36.37
034 36.22
```

Figura 2: Top 10 países con las temperaturas más altas

Para obtener el top de 10 de temperaturas más altas hemos usado la siguiente instrucción

```
awk 'NR>1 {print $1, $5}' salida.txt | sort -k2,2nr | head -10
```

Con **awk NR¿1 {print $1, $5}**, extraemos la primera (ID del país) y la quinta columna (temperatura máxima) del archivo salida.txt, omitiendo el encabezado (NR¿1). Usando la tubería **sort -k2,2nr** ordenamos las filas en función de la temperatura máxima, de forma numérica (n) y en orden descendente (r). Para acabar, mediante la tubería **head -10** se seleccionan únicamente las 10 primeras líneas (top 10 de IDs de países con mayor temperatura máxima). Los paises obtenidos en orden de mayor T a menor T són : Algeria (35), Iran(8) , Iraq (60) , Arabia Saudi (13) , Pakistan (27) , Qatar (107), India (16) , Bahrain (132)

, Emiratos arabes (18) y Egipto (34)

P3. ¿Qué comandos de la terminal puedes utilizar para calcular el número de datos registrados dentro de cada fichero CSV?

El comando que hemos usado es

```
wc -l GlobalLandTemperaturesByCityIDs.csv
GlobalLandTemperaturesByCityDataSmall.csv
GlobalLandTemperaturesByCityData.csv | head -n -1
```

wc -l nos permite contar las líneas que tienen los ficheros .csv, mientras que **head -n -1** oculta la impresión de una cuarta línea que muestra el total de líneas que hay, es decir, suma todas las líneas de los diferentes ficheros. Con lo que el archivo csv de los ids contiene 159 datos, el de datos reducidos tiene 15901 datos y el de datos grande contiene, 8599213 datos.

Figura 3: Número de datos dentro de los csv (Resultado del comando)

P4. ¿Cuáles son las funciones dentro de tu aplicación que más tardan en ejecutarse? ¿Cuál es el porcentaje de tiempo que ¿ocupa la ejecución de cada una de las funciones?

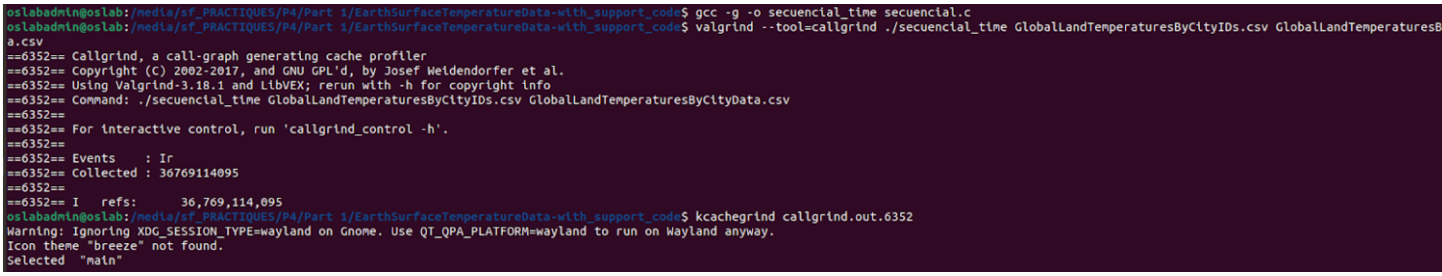

Figura 4: Compilación y ejecución para mostrar el timepo en kcachegrind

Ir	Ir per call	Count	Callee
100.00	36 768 047 770	1	readTemperatureData (secuencial_time: secuencial.c)
0.00	775 724	1	printMatrix (secuencial_time: secuencial.c)
0.00	129 338	1	readCountryIDs (secuencial_time: secuencial.c)
0.00	9 554	1	initializeData (secuencial_time: secuencial.c)
0.00	357	2	0x0000000000109190
0.00	687	1	0x0000000000109110
0.00	465	1	0x0000000000109160

Figura 5: Tiempo de ejecución de cada función

La función que más tarda en ejecutarse es **readTemperatureData**, ya que se encarga de leer y procesar todos los datos del archivo .csv, lo cual es una operación muy costosa. Esta ocupa el 100% del tiempo de ejecución, ya que está tanto tiempo leyendo datos, que hace que el tiempo que se están ejecutando las otras funciones sea prácticamente 0%.

P5. ¿Qué errores se observaron a través de la salida del Valgrind? ¿Cómo los has corregido?

```
oslabadmin@oslab:/media/sf_PRACTIQUES/P4/Part 1/EarthSurfaceTemperatureData-with_support_code$ valgrind --tool=memcheck --leak-check=full --track-origins=yes .
s.csv GlobalLandTemperaturesByCityData.csv
==6648== Memcheck, a memory error detector
==6648== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6648== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==6648== Command: ./secuencial_time GlobalLandTemperaturesByCityIDs.csv GlobalLandTemperaturesByCityData.csv
==6648==
==6648==
==6648== HEAP SUMMARY:
==6648==     in use at exit: 0 bytes in 0 blocks
==6648==   total heap usage: 6 allocs, 6 frees, 25,992 bytes allocated
==6648==
==6648== All heap blocks were freed -- no leaks are possible
==6648==
==6648== For lists of detected and suppressed errors, rerun with: -s
==6648== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 6: Detección de errores usando Valgrind

La salida obtenida al ejecutar el código con Valgrind, nos dice que no hay ninguna fuga de memoria ni otros errores relacionados con la memoria. Lo que indica que nuestro programa gestiona de manera correcta la memoria dinámica.

3 Código multihilos con funciones de bloqueo

En esta parte del código veremos la misma implementación anterior pero esta vez usando un código paralelo mediante hilos. También añadiremos variables "mutex" para proteger las secciones críticas y lograr la programación multihilos. Para lograr esto crearemos una función threadFunc, que se encargara de leer una parte del archivo de forma protegida y actualizará la matriz de datos.

P6. ¿Qué variables debería pasar por argumento el hilo principal a los secundarios?

Las variables que deben pasarse como argumentos a los hilos secundarios incluyen:

- **data_filename**: El vector de char de archivo (`char *data_filename`) que representa el nombre del archivo de datos compartido entre los hilos.
- **Array de identificadores de países**: Para mapear las temperaturas con los países (`int *ids`).
- **Matriz de fechas y temperaturas mínimas**: Para almacenar las fechas y valores mínimos por país (`char (*dtTmin)[DATE_LENGTH]` y `float *Tmin`).
- **Matriz de fechas y temperaturas máximas**: Para almacenar las fechas y valores máximos por país (`char (*dtTmax)[DATE_LENGTH]` y `float *Tmax`).
- **Cantidad de países**: Para verificar y limitar las operaciones dentro de los índices válidos (`int num_countries`).
- **current_line**: Nos indica la linea actual del archivo, así los hilos leen la siguiente zona del archivo.
- **Mutex para sincronización**: Necesario para proteger las secciones críticas que actualizan datos compartidos (`pthread_mutex_t *mutex`). Tenemos 2, uno para actualizar matriz de datos y otro para leer y guardar el archivo de datos.

P7. ¿Qué partes del código (secciones críticas) has protegido para evitar interferencias? Justifica tu respuesta.

En el código, las secciones críticas protegidas son las actualizaciones de las matrices compartidas (**Tmin**, **Tmax**, **dtTmin** y **dtTmax**) y la lectura del archivo de datos. Las matrices se actualizan cuando un hilo encuentra nuevas temperaturas mínima o máxima para un país, lo que requiere exclusión mutua para evitar condiciones de carrera. Del mismo modo, la lectura del archivo está protegida para garantizar que cada hilo procese líneas únicas sin interferencias. Estas protecciones mediante **pthread_mutex_lock** y **pthread_mutex_unlock** aseguran la integridad de los datos y evitan conflictos durante la ejecución concurrente.

P7. Muestra una tabla mostrando los tiempos de ejecución para diferente número de hilos (H = 1, 2, 3, 4, 6 y 8), así como diferentes líneas por bloque (N = 1, 10, 100, 1.000, 10.000, 100.000). Comenta los resultados.

Hilos	BLOCK_SIZE 1	BLOCK_SIZE 10	BLOCK_SIZE 100	BLOCK_SIZE 1.000	BLOCK_SIZE 10.000	BLOCK_SIZE 100
1	11.916s	11.663s	11.506s	11.819s	11.942s	12.246s
2	6.570s	6.393s	6.247s	6.451s	6.555s	7.437s
3	4.763s	4.649s	4.654s	4.706s	4.795s	5.698s
4	3.905s	3.869s	3.717s	3.765s	3.952s	4.683s
6	3.083s	3.009s	2.838s	2.978s	3.101s	3.893s
8	2.558s	2.521s	2.504s	2.532s	2.807s	3.663s

Table 1: Tiempos de ejecución para diferentes valores de BLOCK_SIZE y número de hilos.

Observando la tabla, podemos notar que a medida que se incrementa el número de hilos, el tiempo de ejecución disminuye debido a que el trabajo se divide entre más procesos concurrentes, lo que permite aprovechar mejor los recursos del sistema. Además, se observa que conforme aumenta el tamaño del bloque, el tiempo también disminuye inicialmente, ya que se reduce la sobrecarga de gestión de bloques más pequeños. Sin embargo, a partir de cierto tamaño de bloque, el tiempo de ejecución comienza a aumentar nuevamente, ya que los hilos deben esperar más tiempo a que otros hilos completen la lectura o procesamiento del bloque, lo que introduce demoras adicionales y afecta el rendimiento.

4 Paradigmas y sincronización de hilos (semanas 12-14)

En esta última parte de la práctica utilizaremos el paradigma de productor-consumidor para aumentar la eficiencia de nuestro código anterior. Ahora lo que haremos será un productor (el hilo principal), que leera el archivo y guardará su información en un buffer y varios consumidores (hilos secundarios) que procesaran esta información.

Nosotros para implementar nuestro código hemos optado por una solución diferente : En el main en vez de usar el hilo principal como producer, creamos un hilo que llame a la función *producer*. De esta manera tenemos el código más legible y modular y nos facilita diferenciar entre el código producer y consumer. La razón por la que solo se crea 1 hilo de producer es que la lectura es secuencial y no tiene sentido que se encarguen 2 hilos o más para este trabajo. Luego mientras el producer produce, el main crea los hilos de consumer que llaman a la función *consumer* para procesar los bloques de manera paralela.

Cada sección crítica (vistas en el punto 9) tiene su propio mutex y ya una vez todos los bloques han sido leidos, liberamos memoria y guardamos resultados en el archivo salida_hilos.txt

P8. Nuestro buffer tendrá un tamaño para poder almacenar B bloques y se recomienda que B >= H (hilos secundarios) ¿Cuál es la motivación de esta recomendación? Realiza un experimento para diferentes valores de $B < H$ y $B \geq H$ y comenta los resultados.

Si el búfer tiene un tamaño menor que el número de hilos secundarios, es probable que los hilos consumidores se encuentren frecuentemente en una situación de espera (bloqueados) porque el búfer se llena rápidamente, lo que lleva a una subutilización de los hilos. Esto puede causar que los consumidores tengan que esperar a que el productor procese y libere espacio en el búfer, disminuyendo la eficiencia general. Un búfer suficientemente grande garantiza que siempre haya bloques disponibles para que los consumidores procesen, lo que maximiza la utilización de los recursos del sistema (CPU y memoria) y mejora el rendimiento general del sistema.

Para ver la diferencia ejecutaremos diversas pruebas para ver los resultados. Las pruebas han sido realizadas con el fichero grande:

```
real    0m3.079s
user    0m3.553s
sys     0m1.063s
```

Figura 7: B=1 H=2

```
real    0m2.360s
user    0m4.019s
sys     0m0.567s
```

Figura 8: B=2 H=2

```
real    0m2.307s
user    0m4.875s
sys     0m1.698s
```

Figura 9: B=2 H=4

Figura 10: Comparación de diferentes configuraciones de búfer y hilos

Una vez tenemos los resultados podemos ver de forma clara como la recomendación ayuda a la eficiencia del código, reduciendo el tiempo. El hecho de que los hilos siempre tengan bloques para consumir ayuda al flujo de ejecución para que de la ilusión de que se estan procesando varios bloques a la vez.

Hay que tener cuidado con el número de bloques e hilos porque más hilos no significa más velocidad, al hacer que los hilos esten más tiempos bloqueados por los mutexes y tener que gastar más tiempo en su sincronización. No solo eso si no que muchos bloques són una cantidad de memoria considerable, por eso hay que buscar el equilibrio entre ambos valores.

P9. ¿Cuál es la/s sección/es crítica/s del código que deben protegerse y por qué? Justifica tu respuesta.

En esta parte de la práctica tenemos 2 secciones críticas diferentes que tenemos que vigilar. La primera seria igual a la anterior parte, la sección donde editamos la matriz de temperaturas. Es importante que haya un mutex que bloqueé la zona cada vez que se editen los datos para que cuando se actualizen no haya

otro hilo editando la información recién añadida. La segunda zona es propia del paradigma consumidor productor que seria el buffer. Tanto el hilo principal como todos los hilos secundarios acceden al buffer y editan sus valores constantemente/s, ya sea el producer añadiendo contenido y cambiando su puntero como los consumidores que también necesitan acceder al puntero y al contenido de manera ordenada.

Al ser dos secciones críticas totalmente diferentes con recursos independientes entre si, podemos usar 2 mutexes diferentes. Se podría hacer con 1 perfectamente para menor complejidad, pero ya que a la hora de procesar bloques no se tiene en cuenta nada del buffer, hacer que los hilos esperen para editar la matriz por culpa de otro que esta dentro del buffer le reduce la eficiencia al código. Con cuidado podemos aprovechar esta mejora, teniendo cuidado para no provocar ningún deadlock. Es por eso que optamos por usar 2 diferentes en las diferentes secciones del código para mejorar la velocidad de nuestra implementación del paradigma.

P10. ¿Por qué el uso de strcpy es ineficiente en este caso? Justifica tu respuesta.

El uso de **strcpy** para transferir bloques de datos del productor al buffer es ineficiente porque copia carácter por carácter desde una ubicación en memoria a otra, lo que resulta costoso en términos de tiempo y recursos, especialmente para bloques grandes de texto como los de los archivos CSV. Esto no solo ralentiza el procesamiento, sino que también implica una asignación redundante de memoria, ya que los datos ya existen en el productor.

Es por eso que utilizar punteros mejora considerablemente nuestro código. Usar punteros en lugar de strcpy para transferir datos entre el producer y el consumer es más eficiente porque evita copiar realmente el contenido de las cadenas de caracteres, que puede ser caro en términos de tiempo y recursos, especialmente cuando se trabaja con grandes bloques de datos. En lugar de realizar una copia de los datos, el uso de punteros permite simplemente intercambiar las referencias a los bloques de datos, lo que es una operación mucho más rápida y que consume menos memoria. Esto significa que, en lugar de duplicar el contenido en la memoria, solo se modifican las direcciones de memoria a las que apuntan los punteros, lo que resulta en un uso más eficiente de la memoria y un rendimiento general mejorado en aplicaciones multihilo donde la velocidad de acceso y procesamiento es el objetivo principal.

P11. ¿Cómo sabrán los consumidores que el productor ya ha leído todos los bloques de datos?

Los consumidores pueden determinar que el productor ha finalizado la lectura de bloques utilizando una bandera compartida, como **producer_done**, que se establece en true cuando el productor completa su tarea. Cuando el buffer está vacío, los consumidores verifican esta bandera; si está activada y no quedan datos por procesar, los consumidores terminan su ejecución de manera controlada. Este proceso se gestiona utilizando mutexes y variables condicionales para sincronizar el acceso al buffer, asegurando que los consumidores trabajen únicamente cuando haya datos disponibles o cuando el productor haya terminado.

5 Conclusiones y valoraciones personales

En esta práctica para acabar la asignatura hemos visto como funcionan los hilos y varias de sus funciones para optimizar código. Ver la diferencia entre usar un código secuencial y un código multihilos nos ha aclarado en que contextos los hilos nos pueden ayudar a ejecutar el código de forma más rápida. También a como usarlos bien, identificando y protegiendo las secciones críticas de nuestro código mediante los mutex. Además del paradigma productor consumidor, que aprovecha los hilos para combinar acciones diferentes que también ayuda a la eficiencia, no solo de velocidad de ejecución si no también de espacio de memória. Ha sido una buena práctica para acabar con la asignatura de sistemas operativos 2, profundizando más el temario que vimos en Sistemas operativos 1.

6 Contribución de cada uno de los miembros del grupo

Hemos realizado de manera conjunta el trabajo, poniendo en común las diferentes preguntas, y la forma de realizar el código. Andrés 33,3%, Quico 33,3% y Aleix 33,3%.