P4. PROYECTO DE PROGRAMACIÓN CONCURRENTE

Sistemas Operativos 2
Noviembre-Diciembre 2024

1 INTRODUCCIÓN

La programación secuencial, paralela y concurrente es esencial en la informática moderna, permitiendo la ejecución simultánea de múltiples tareas y procesos en sistemas multiprocesador y multihilo.

A diferencia de la programación secuencial, donde las tareas se ejecutan de forma lineal una después de otra, la programación concurrente mejora el rendimiento y la eficiencia de las aplicaciones al distribuir la carga de trabajo entre varios núcleos de CPU o computadoras, acelerando cálculos intensivos y procesamiento masivo de datos. Sus beneficios incluyen una mejor respuesta del usuario, escalabilidad para manejar cargas de trabajo crecientes, optimización de recursos y aplicaciones más rápidas en campos como análisis de *big data*, simulaciones científicas, desarrollo de videojuegos y servidores web, entre otros. La programación concurrente permite a las aplicaciones aprovechar al máximo los recursos de hardware disponibles, ofreciendo oportunidades para el desarrollo de software más poderoso y eficiente en diversos campos tecnológicos.

El objetivo de este proyecto es familiarizarse con los datos climáticos proporcionados y ser capaz de implementar en C un código que analice dichos datos y permita calcular las temperaturas mínimas y máximas de cada país. Esta temática es importante ya que se incluye dentro de las acciones relacionadas con el análisis del cambio climático y son parte de los Objetivos de Desarrollo Sostenible de las Naciones Unidas (ODS 13: Acción por el clima).

Esta implementación se realizará de manera secuencial y después se modificará durante las sucesivas semanas para hacerla concurrente, utilizando funciones de creación y bloqueo de hilos, así como mediante la implementación de algún paradigma y/o técnica de sincronización de hilos.

2 FAMILIARIZACIÓN CON LOS DATOS E IMPLEMENTACIÓN DE CÓDIGO SECUENCIAL (SEMANA 9)

El conjunto de datos utilizado en esta práctica es una recopilación empleada en un reto computacional disponible en *Kaggle* ("Climate Change: Earth Surface Temperature Data"), generado a partir de una compilación realizada por Berkeley Earth. El Estudio de Temperaturas Superficiales de Berkeley Earth combina 1.600 millones de registros de temperatura provenientes de 16 archivos preexistentes. Está bien estructurado y permite dividir los datos en subconjuntos interesantes (por ejemplo, por país). Hemos preprocesado el archivo CSV utilizado en esta práctica para generar los identificadores de países, como se describe a continuación.

En esta práctica se proporcionan varios ficheros en formato CSV (valores separados por comas) que contienen información relativa a temperaturas medias registradas en ciudades de todo el mundo. Cada línea de estos ficheros pertenece a datos registrados en una fecha concreta, en una ciudad de un país concreto. El contenido de cada fichero se detalla a continuación:

- *GlobalLandTemperaturesByCityIDs.csv*: Lista de identificadores numéricos (IDs) correspondiente a los países.
- *GlobalLandTemperaturesByCityData.csv*: Tabla con datos que contienen fechas (*dt*), temperaturas (*AverageTemperature*), países (*Country*), ID del país (*CountryID*) entre otros datos.
- *GlobalLandTemperaturesByCityData_reduced.csv*: Similar al CSV anterior, pero con un número reducido de líneas para facilitar el testeo del código.

El objetivo de esta semana es familiarizarse con los datos e implementar en código C una aplicación (`secuencial.c`) que lea los ficheros CSV (*IDs* y *Data*) de la siguiente forma.

```
./secuencial IDs.csv Data.csv
```

El objetivo de la aplicación es comprobar cada una de las temperaturas registradas para cada país dentro del fichero CSV Data y actualizar una matriz (previamente inicializada a cero) que muestre, para cada país, la temperatura mínima y máxima registradas, así como su fecha. A continuación, se muestra un ejemplo de dicha matriz (después de la inicialización).

CountryID	dtTmin	Tmin	dtTmax	Tmax
001	0	0	0	0
002	0	0	0	0
003	0	0	0	0
...	0	0	0	0

A medida que vayas analizando los ficheros CSV e implementando el código responde a las siguientes preguntas:

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

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

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

Callgrind es una herramienta de análisis de rendimiento que se incluye con Valgrind. A diferencia del análisis de memoria que realiza Valgrind, Callgrind se utiliza para realizar el análisis de rendimiento de un programa. Su principal objetivo es proporcionar información detallada sobre cómo se está utilizando la CPU durante la ejecución del programa. Callgrind realiza un

seguimiento de todas las funciones que se llaman, cuántas veces se llaman, cuánto tiempo se gasta en cada función y cómo se relacionan unas con otras.

La utilidad de Callgrind radica en su capacidad para identificar cuellos de botella en el código. Al analizar la información generada por Callgrind, los desarrolladores pueden encontrar las partes del código que consumen la mayor cantidad de tiempo de CPU y optimizar esas secciones para mejorar el rendimiento del programa. También puede ayudar a identificar funciones que se llaman frecuentemente, lo que puede ser útil para optimizar el flujo del programa.

Para utilizar el Callgrind previamente se debe haber compilado el ejecutable con la opción *debugger* y a continuación se utiliza la estructura `valgrind --tool=callgrind ./ejecutable` (más argumentos si los hubiera). Esto genera un archivo del estilo `callgrind.out.<pid>` que se abre con la aplicación `kcachegrind` (se ha de instalar la primera vez en la máquina virtual mediante `apt install` con `oslabadmin`). Una vez acabado el análisis, se recomienda eliminar el fichero `callgrind.out.<pid>` para evitar que `kcachegrind` lea varios ficheros a la vez. Se pide ejecutar el Callgrind, familiarizándote con los diferentes aspectos de la aplicación. Se recomienda buscar algún tutorial. **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? Se recomienda utilizar el fichero CSV pequeño para estas pruebas (*GlobalLandTemperaturesByCityDataSmall.csv*).

Como hemos visto, **Valgrind** es una herramienta muy interesante y ampliamente utilizada para el análisis de código en C y otros lenguajes de programación. Su principal función es detectar errores de memoria, como el uso incorrecto de punteros, fugas de memoria (*memory leaks*), y otros problemas relacionados con la gestión de la memoria dinámica en programas escritos en C y C++. Analiza tu código `secuencial.c` con el Valgrind (previamente compilado con la opción correspondiente), para analizar posibles errores y corregirlos. **P5**. ¿Qué errores se observaron a través de la salida del Valgrind? ¿Cómo los has corregido?

3 IMPLEMENTACIÓN DE FUNCIONES DE BLOQUEO (SEMANAS 10 Y 11)

El objetivo principal para estas dos semanas es optimizar el código secuencial anterior y reducir así su tiempo de ejecución, utilizando múltiples hilos para gestionar los datos climáticos. Para ello, deberemos tener en cuenta las secciones críticas, y protegerlas adecuadamente utilizando monitores. Como no se han visto aún monitores en teoría aún y para ayudar en el desarrollo de la práctica, se provee de dos manuales con ejemplos de programación con hilos que encontrareis en el campus virtual (**[LAB] P4 - Documentos soporte a la programación multihilo**). Se recomienda comenzar a familiarizarse con ellos antes de continuar con la práctica.

Al empezar a ejecutar el programa `secuencial.c`, el `main` solo tendrá un único hilo (principal). Este hilo principal leerá los códigos de los países (quizás mediante una función `read_contry_ids`). Luego se llama a una función que es la que lee los datos de temperatura (por ejemplo, `read_temperature_data`). El objetivo aquí es aumentar la eficiencia del procesamiento de los datos de temperatura utilizando múltiples hilos, repartiendo la extracción de los datos del archivo CSV entre múltiples hilos (secundarios).

A continuación, se describe el esquema (recomendado) a implementar con mayor detalle:

1. El hilo principal llama a la función `read_temperature_data`, abre el archivo y lee sólo la cabecera.

2. Antes de empezar a procesar los datos, el hilo principal creará, con la función `pthread_create`, 2 hilos secundarios (`H`) que accederán de forma concurrente al archivo de datos para extraer la información que contiene. Al crear los hilos secundarios el hilo principal pasará a los hilos secundarios, preferentemente por argumento (y NO por variables globales), todas las variables necesarias que los hilos secundarios necesitan para procesar los datos del archivo. **Es recomendable comenzar la implementación con solo 1 hilo secundario** para evitar problemas de *race conditions* y verificar que la programación del programa es correcta antes de implementar dos (o más) hilos.
3. Los **2 hilos secundarios** procesarán concurrentemente el archivo de datos como se describe a continuación:
    a. El hilo lee un bloque de `N` líneas seguidas del archivo (`N` podrá tener valores de 1, 10, 100, o 10.000, …) y las almacena en una **variable propia del hilo**.
    b. Una vez leído el bloque de `N` líneas, el hilo pasará a extraer la información de cada línea del bloque para actualizar la información de la matriz compartida que muestra el Id del país, temperaturas máxima y mínima, así como la fecha de estas (p.ej., `num_countries`).
    c. Una vez que el hilo (secundario) ha actualizado la información de `num_countries`, el hilo vuelve al paso anterior hasta que no haya más datos a leer. Cuando no haya más datos a leer los hilos secundarios saldrán del bucle y dejarán de ejecutarse.
4. Mientras los hilos secundarios procesan el archivo, el hilo principal se quedará esperando, mediante la función `pthread_join`, a que los hilos secundarios acaben su trabajo. Observar que los 2 hilos secundarios se crean al empezar a procesar el archivo, y se destruyen una vez terminan de procesarlo.
5. Una vez hayan finalizado todos los hilos secundarios, el hilo principal se despertará, cerrará el archivo y saldrá de la aplicación imprimiendo por pantalla los resultados de la matriz de temperatura.

Como sugerencia de buenas prácticas al programar con hilos, se recomienda pasar información del hilo principal a los secundarios mediante parámetros. Dado el esquema de implementación anterior, **P6**. ¿Qué variables debería pasar por argumento el hilo principal a los secundarios?

Como se ha visto en teoría, se requiere de una buena planificación y sincronización entre hilos para evitar interferencia entre ellos, y evitar así resultados inesperados. Para ello se utilizarán las funciones de bloqueo `pthread_mutex_lock` y `pthread_mutex_unlock` para asegurar exclusión mutua. **P7**. ¿Qué partes del código (secciones críticas) has protegido para evitar interferencias? Justifica tu respuesta.

Realiza las comprobaciones correspondientes utilizando los ficheros CSVs y comprueba que los resultados son los mismos que la ejecución del programa secuencial sin hilos (semana 9). **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. Se recomienda tomar los tiempos varias veces para mostrar los valores promedios y su desviación estándar (valor promedio ± stdev) en cada uno de los casos.

4 PARADIGMAS Y SINCRONIZACIÓN DE HILOS (SEMANAS 12-14)

El objetivo para estas últimas semanas de semestre es utilizar el **paradigma del productor-consumidor** para aumentar la eficiencia del procesamiento de los datos de las temperaturas. Al empezar a ejecutar el `main` sólo habrá un hilo (hilo principal). Este hilo será el **productor** y se encargará de leer los códigos de los países y las líneas del archivo CSV en bloques de N líneas a través de la función `read_temperature_data` (por ejemplo). Por otro lado, se crearán

varios hilos secundarios (**consumidores**) que serán los encargados de extraer la información de los bloques de datos que ha leído el productor y actualizar la matriz de temperaturas.

El esquema a implementar es el siguiente:

1. El hilo principal llama a la función `read_temperature_data` (u otra función similar). El hilo principal, antes de crear los hilos secundarios, tendrá que crear el **búfer de comunicación** entre el hilo principal y los hilos secundarios. El búfer de comunicación almacenará los bloques de N líneas que el hilo principal lee para que los hilos secundarios puedan acceder a ellos y procesarlos. El búfer de comunicación tendrá un tamaño fijo, es decir, podrá almacenar **B bloques de N líneas**.
2. A continuación, el hilo principal, antes de abrir el archivo a procesar, creará, con la función `pthread_create`, H = 2 hilos secundarios. Al crear los hilos secundarios el hilo principal pasará a hilos secundarios, **preferentemente por argumento** y no por variables globales, todas las **variables** necesarias que los hilos secundarios necesitan para procesar los datos del archivo CSV.
3. El hilo principal abrirá el archivo CSV a procesar y leerá el archivo de datos en bloques de N líneas. Cada bloque de N líneas será "transferido" al búfer de comunicación. Si el búfer de comunicación se llena con B bloques, el hilo principal deberá dormirse en una **variable condicional** para esperar que los hilos secundarios comiencen a procesar datos. Cuando un consumidor coja un bloque del búfer despertará al productor para que pueda introducir un nuevo bloque en el buffer.
4. Los H = 2 hilos secundarios serán los encargados de procesar los datos que se hayan introducido en el búfer. Si hay bloques a procesar en el búfer de comunicación, el hilo podrá tomar uno (que todavía no se haya procesado). Una vez cogido, el hilo secundario extraerá los datos del bloque y actualizará la matriz de temperaturas. Una vez se haya procesado el bloque, el hilo secundario volverá a tomar otro bloque del búfer de comunicación. En caso de que no haya ningún bloque a procesar, el/los hilo/s consumidor/es se dormirá/an a una variable condicional para esperar a que el productor inserte un nuevo bloque en el búfer de comunicación y los despierte.
5. Cuando el hilo principal haya leído todo el archivo y haya "transferido" los bloques al búfer de comunicación, cerrará el archivo y se quedará esperando, con la función `pthread_join`, que los hilos secundarios acaben de procesar los bloques que puedan existir en el búfer de intercomunicación.
6. Una vez hayan finalizado todos los hilos secundarios con su trabajo, el hilo principal se despertará del `pthread_join`, saldrá de la aplicación imprimiendo por pantalla los resultados del análisis del archivo.

Observar que, en la implementación anterior, el productor es el único hilo encargado de leer el archivo de datos. Éste leerá el archivo de datos en bloques de N líneas y "transferirá" cada bloque al búfer que comparten productor y consumidores. El búfer tendrá un tamaño para poder almacenar B bloques y se recomienda que B ≥ H (hilos secundarios). **P8.** ¿Cuál es la motivación de esta recomendación? Realiza un experimento para diferentes valores de B<H y B ≥ H y comenta los resultados.

Por otro lado, los consumidores tomarán del búfer los bloques de N líneas, y extraerán de cada línea la información necesaria para actualizar la matriz compartida de temperaturas. Observar que sólo los hilos consumidores acceden a esta matriz. Los hilos secundarios acceden a recursos (variables) compartidos entre ellos. **P9**. ¿Cuál es la/s sección/es crítica/s del código que deben protegerse y por qué? Justifica tu respuesta.

Para entender correctamente el funcionamiento de los hilos dentro del esquema del productor-consumidor, se proponen las siguientes **recomendaciones**:

1. Repasar las fichas del campus que habla de los monitores y las variables condicionales. Esto permitirá implementar el esquema básico del productor-consumidor con un productor y un consumidor, para un tamaño de búfer B = 1. El consumidor deberá salir de su función de entrada ejecutando el `return` al final de la función. No está permitido utilizar funciones que maten los hilos para finalizarlos.
2. A continuación, modifica el código para utilizar H = 2 hilos secundarios y un búfer en el que se puedan almacenar B bloques. Se deberá implementar el algoritmo de comunicación para múltiples consumidores y productores. Una de las complicaciones se encuentra en saber cuándo terminan los consumidores ya que los consumidores deben terminar de forma "limpia".
3. Asegurar de que los resultados que obtienen al procesar el archivo con múltiples hilos son los mismos que los obtenidos al ejecutar el código de las semanas anteriores.

A continuación, se describe una **propuesta de implementación para "transferir" la información del productor al búfer y del búfer al consumidor**. Implementar un algoritmo poco eficiente será penalizado. Supongamos la siguiente declaración de una celda (que almacena un bloque de datos):

```
typedef struct cell {
int nelems;
char **lines;
} cell;
```

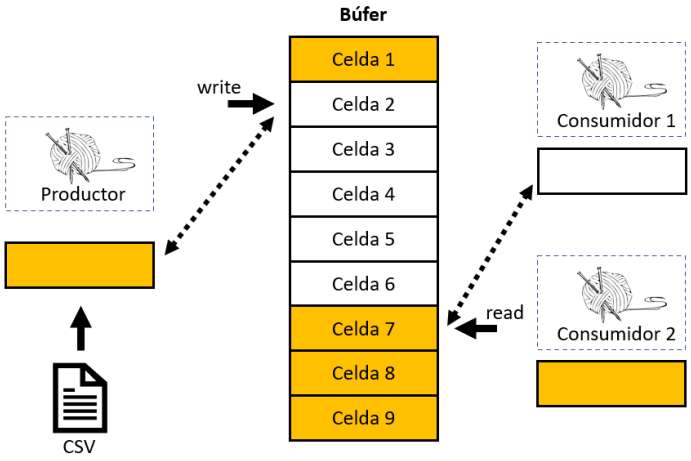

Figura 1. Esquema productor-consumidor.

En la figura 1 el **búfer de comunicación** tiene capacidad para B = 9 celdas. Además, tanto en el productor como en los consumidores hay también una celda. Las celdas de color amarillo son celdas que contienen información para procesar (productor, algunas celdas del búfer) o que está siendo procesada en ese momento (consumidor 2). Las celdas de color blanco son celdas que no contienen información o celdas en las que la información ya ha sido procesada (consumidor 1 y algunas celdas del búfer). A continuación, se describe una forma de proceder para conseguir una implementación eficiente:

1. Al reservar memoria la memoria dinámica, se reserva memoria para todas las celdas que hay en el dibujo. Cada bloque, representado con la variable líneas en la estructura, tendrá un tamaño para almacenar N líneas. La memoria asociada a estos bloques no se liberará hasta que se hayan procesado todos los bloques del archivo. Hay que evitar, en la medida de lo posible, reservar y liberar memoria mientras se procesan los archivos ya que estas funciones son costosas.
2. El productor leerá el archivo en bloques de N líneas. La variable `nelems` sirve para indicar cuántas líneas se han leído del archivo. El valor de esta variable será N, a excepción del último bloque de archivo, por el que se podrán haber leído menos líneas. El valor de `nelems` indicará el número de líneas que se han leído del archivo.
3. Una forma de proceder para realizar la transferencia de la información es "copiar" la información entre las celdas. En particular, para el caso del productor, se trata de copiar el bloque de N líneas del productor en la posición que corresponde dentro del búfer. Para ello será necesario utilizar funciones como `strcpy` que permiten copiar cadenas de caracteres. Sin embargo, esto implicaría una solución bastante ineficiente. **P10**. ¿Por qué el uso de `strcpy` es ineficiente en este caso? Justifica tu respuesta. Otra forma de proceder es "jugar" con los punteros en C. El objetivo aquí es "intercambiar" la celda que tiene el productor con la celda que hay en la posición correspondiente al búfer. A continuación, se muestra un ejemplo donde se intercambian dos celdas:

```
cell cell_producer, cell_buffer, tmp;
// Supongamos se ha reservado memoria para cell_producer
y cell_buffer
// Supongamos que ahora se leen datos y se llena el
cell_producer
// Ahora intercambiamos las dos celdas
tmp = cell_buffer;
cell_buffer = cell_producer;
cell_producer = tmp;
```

Lo que estamos haciendo es "transferir" un bloque de datos del productor al consumidor, y coger uno que esté libre del búfer. Los consumidores procederán de forma similar: supongamos que uno consumidor ha terminado de procesar sus datos y quiere coger el siguiente bloque a procesar. La idea es "intercambiar" el bloque del que dispone el consumidor con un bloque para procesar del buffer.

Observar que al realizar este último intercambio (entre el consumidor y el búfer) estamos poniendo una celda con datos ya procesados en el búfer. No pasa nada, dado que en algún momento el productor tomará la celda con datos ya procesados y la sobrescribirá con los datos que lea del archivo. Una vez el productor haya sobrescrito el búfer con los datos del archivo, volverá a realizar la "transferencia" como se ha comentado antes.

El esquema que se ha presentado aquí es el caso particular en el que el buffer tiene tamaño B = 1. En la práctica se ha de ampliar esta idea para que el buffer almacene B celdas, tal y como se muestra en la figura 1. **P11**. ¿Cómo sabrán los consumidores que el productor ya ha leído todos los bloques de datos?

5 INFORME

Se requiere la entrega de un informe sobre el trabajo realizado. **El informe se debe entregar en formato PDF** (NO se admiten informes en formato doc, docx, odt, etc). Este informe debe incluir una **introducción**, una sección donde se muestran y comentan las **pruebas realizadas y respuestas a las preguntas**, así como una sección con las **conclusiones y valoraciones personales de la práctica.** Importante: **se ha de indicar en una sección la contribución de cada uno de los miembros del grupo, indicando el porcentaje de la nota que merecería cada uno (p.ej., 25%-25%-50%; 50%-50%; 30%-70%).**

El documento ha de tener un **máximo de 8 páginas** (sin contar la portada o índice).

La nota del informe corresponderá a las pruebas realizadas y respuestas a las preguntas (70%), escritura y expresión (10%), así como la estructura y presentación del informe (20%).

6 CÓDIGO FUENTE

Se requiere la entrega de una carpeta con nombre `src` que contenga el código fuente de la práctica. Se debe incluir todos los ficheros necesarios para ser compilado (sin incluir los datos de temperatura) con `make` y ejecutado en la máquina virtual Linux de la asignatura. Es conveniente incluir el fichero `Makefile` correspondiente con instrucciones para compilar el código con opciones de optimización `-O`.

Se pide que el código este comentado indicando los cambios realizados, el código sea modular y limpio, y se utilice un buen estilo de programación.

7 ENTREGA Y EVALUACIÓN

Cada grupo debe subir un **único archivo ZIP** a la tarea asociada a esta práctica en el campus virtual con el nombre **P4_GrupoXX_nombre1_apellido1_nombre2_apellido2.zip**, y debe incluir el **informe** solicitado con todas las secciones especificadas en el apartado anterior, así como una carpeta llamada `src` con el **código fuente** de la práctica.

En esta práctica el **código fuente** entregado tiene un peso del **70%** de la nota de la práctica, mientras que el **informe** tiene un peso de un **30%.**