Texto

Descripción generada automáticamente con confianza media

**Instituto Tecnológico de Costa Rica**

**Ingeniería en Computación**

**Principios de Sistemas Operativos**

IC6600

**Proyecto 1**

**Equipo de Trabajo**

Ericka Yu Min Guo Chen – 2021061073

Tomás Coto Quesada – 2021040456

**Profesor**

Armando Arce Orozco

**Fecha**

24/09/2023

Tabla de Contenidos

[Introducción 3](#_Toc362448264)

[Descripción del problema 4](#_Toc229204180)

[Definición de estructuras de datos 6](#_Toc928933546)

[Descripción detallada y explicación de los componentes principales 7](#_Toc1326902356)

[Mecanismos de creación y comunicación de procesos 9](#_Toc335522201)

[Pruebas de rendimiento 10](#_Toc1653768054)

[Conclusiones: 13](#_Toc1761939354)

# **Introducción**

En el siguiente trabajo se detalla las estructuras de datos que se usaron para el desarrollo del proyecto, así como los componentes principales y la forma en que los procesos se comunican entre ellos. Además, se adjuntan los resultados del tiempo que duró el programa en ejecutarse dependiendo de la cantidad de procesos que use. Esto con el fin de obtener con qué cantidad es lo más optimo.

El problema por solucionar es recrear el comando grep y ejecutarlo en archivos grandes utilizando múltiples procesos. El fin de hacer esto es analizar con cuantos procesos el programa se vuelve más eficiente. Para esto se deben de sincronizar los procesos hijos usando al padre como mediador. La salida del programa debe de ser la línea en la que la palabra se encuentre.

# **Descripción del problema**

El objetivo de este proyecto es comparar el rendimiento de realizar una tarea colaborativa utilizando múltiples procesos. Para ello se deberá desarrollar una versión multiprocesos del programa llamado grep que permite buscar múltiples palabras sobre un archivo e imprimir el párrafo en dónde aparecen dichas palabras.

En este proyecto se deben realizar múltiples pruebas para determinar la cantidad óptima de procesos para ejecutar este tipo de tarea sobre un único archivo muy grande.

**Uso del programa**

El programa grep recibe una expresión regular y una lista de archivos:

grep 'pattern1|pattern2|pattern3|...' archivo

Por omisión el programa debe mostrar la línea del archivo en donde se encuentra la expresión regular.

**Manejo de expresiones regulares**

El lenguaje C por sí mismo no maneja expresiones regulares, sin embargo, se puede usar la librería "regex.h" que cuenta con las funciones regcomp y regexec que permiten realizar esta tarea. La función regcomp se debe ejecutar inicialmente y recibe la expresión regular ingresada por comando, luego la función regexec se debe ejecutar sobre cada una de las líneas del archivo.

**Manejo de múltiples procesos**

El programa debe contar con múltiples procesos que se encarguen de inspeccionar el archivo. Debido a que las búsquedas se realizan sobre archivos de gran tamaño no sería eficiente crear un único proceso. Es por ello por lo que es necesario crear un conjunto compartido de procesos (pool de procesos) que incluya solo una cantidad limitada de procesos, y en donde cada proceso trabajará sobre un buffer propio en donde se debe leer una porción del archivo. Como parte del proyecto se deben realizar diferentes pruebas (con diferentes archivos grandes) para determinar un tamaño "óptimo" del pool de procesos. Se utilizará el tiempo de ejecución para determinar dicho tamaño óptimo.

En este caso se utilizará un pool de procesos estático, es decir, el programa creará la totalidad de los procesos al inicio del programa y únicamente utilizará estos para realizar todo el trabajo. Tal como usted debe suponer, a los procesos que terminen de buscar en su porción del archivo deberán leer otra parte del archivo para realizar la búsqueda hasta terminar con todo el archivo. Se debe tener especial cuidado de que dos o más procesos lean la misma porción del archivo, o bien, que traten de escribir al mismo tiempo en la salida.

Note que al leer una porción del archivo puede suceder que una línea no sea leída completamente. Por eso cada proceso debe revisar si alguna línea se cortó, y actualizar el puntero de lectura del archivo (fseek) hasta el final de la última línea que se leyó completa.

**Sincronización entre procesos**

Los procesos se deberán sincronizar entre ellos mediante mensajes (NO sockets). La sincronización incluye determinar la posición de la última lectura al archivo y escribir a la salida. Para ello puede utilizar al padre como un proceso coordinador al cual los otros procesos realizan estas peticiones.

# **Definición de estructuras de datos**

Las estructuras de datos para que se usan son dos. Se crea una para el manejo de mensajes llamada Message, esta tiene los atributos:

* Long type: para saber qué tipo es el mensaje.
* Int process: para saber con qué proceso se relaciona
* Long linePosition: esto para saber desde qué punto el proceso debe leer
* Int childStatus: esto para que saber si el hijo está ocupado o libre
* Char text[MSGSZ]: este se usa para mandar la línea en que se encuentre la palabra

La otra estructura que se usa es Child\_status, la función de esta es mantener el estado del hijo con el fin de saber si se encuentra ocupado o libre para ver si puede leer. Consta de los atributos:

* Long type: para saber que tipo es
* Long pid: para saber a que proceso se relaciona
* Int status: para saber si está libre u ocupado

# **Descripción detallada y explicación de los componentes principales**

Los componentes principales del programa constan de 4 funciones y el main. Dos de estas funciones se encargan de revisar el estado de los procesos y las otros dos son para la escritura y lectura de los procesos.

La función verificarFinalizacionProcesos tiene como argumento un array del struct child\_status. El objetivo de esta función es revisar si no hay ningún proceso leyendo ni en busca de la palabra. Esto se hace para saber cuándo el programa puede finalizar sin dejar a medias un proceso. Para esto recorre el array revisando el estado de los hijos, si encuentra todos los hijos libres entonces retorna 1.

La función buscarProcesoDesocupado tiene como parámetros un array del struct child\_status y un integer “posicion”. La idea de estos parámetros es saber si el hijo “i” en la posición “posicion” se encuentra libre. Para esto se pregunta si el estatus de ese hijo es igual a 0 y en caso de serlo lo cambia a 1. Con esto retorna la posición en la que se encuentre.

La función process tiene como parámetros un buffer en el que busca una palabra, la palabra a buscar y el id de la cola del padre. El objetivo de esta función es enviar en la cola de mensajes la línea, dentro del buffer, donde el patrón se encuentre. Para esto primero mediante la librería regex crea la expresión regular a buscar, para luego iterar la cantidad de veces que la palabra este presente en el buffer.

Dentro de este ciclo crea la línea a enviar al padre para ser impresa. Para esto obtiene donde empieza la palabra y donde termina, hace un ciclo desde el inicio del patrón hasta el primer salto de línea que encuentre y hace lo mismo, pero del final de la palabra hasta el siguiente salto de línea. Cuando recorre todo el buffer o ya no hay más patrones es cuando se sale del ciclo.

La función que se encarga de la lectura del archivo se llama readFile tiene como parámetros un char con el nombre del archivo, el desplazamiento de donde empieza a leer, el id de la cola padre, el número del hijo que está trabajando y un char con el patrón a buscar. Para esto primero usa fseek para moverse al char donde debe empezar la lectura, de ahí mediante fgets llena el buffer y con ftell obtiene la nueva línea donde debe leer.

Al hacer esto revisa si debe de avisar que sigan leyendo o que ya se acabó el archivo. En caso de tener que seguir leyendo avisa al padre que terminó de leer, en que posición terminó y que puede poner al siguiente a leer. Seguido de esto se pone a procesar lo leído. Cuando termina de procesar avisa al padre que ya está libre. En caso de que ya sea el final del documento sigue la misma estructura solo que con un tipo distinto de mensaje indicando que ya no hay que leer solo procesar y cuando termina de procesar avisa que está libre.

El main es el que se encarga de crear y sincronizar los procesos. Este se explicará en la siguiente sección. También es en este momento donde se crean las colas de los mensajes y el array con la cantidad de hijos que sean.

# **Mecanismos de creación y comunicación de procesos**

El mecanismo de creación es simple. Al inicio del programa mediante un for crea la cantidad de proceso solicitados para después enviar el mensaje al padre de que el hijo existe para que el padre les asigne el estado libre a dichos procesos hijos. Luego de esto el hijo se cicla esperando un mensaje que le indique que posición leer.

Terminado de crear los hijos, el padre se cicla hasta que todos sus hijos lean y procesen el archivo. Es en este momento donde se da la sincronización entre procesos. Para esto utiliza la cola de mensajes del padre y la del hijo. Se usan dos colas pues la comunicación entre estos es unilateral. Para comenzar con la lectura y procesamiento del archivo, al finalizar la creación de los hijos, el padre envía por defecto que la primera lectura comenzará desde la línea 0 y a partir de ahí se espera por un mensaje por parte del hijo para saber en qué posición quedó de la lectura para poder asignarle la tarea al próximo proceso libre.

Cada que el padre recibe un mensaje revisa que tipo es y dependiendo de lo que sea, ejecuta una función distinta. Si el mensaje es de tipo dos busca que proceso se encuentra desocupado y comienza el procedimiento de lectura. En este mensaje también va incluido en que posición debe iniciar. Si es de tipo tres revisa si tiene que finalizar el programa. Si es de tipo cuatro cambia el estado del hijo a desocupado. Por último, si el mensaje es de tipo cinco, el padre imprime la línea donde se encuentre la palabra.

# **Pruebas de rendimiento**

Para las pruebas de rendimiento lo que se hizo fue correr el programa 5 veces con tres, seis, siete, ocho y 15 procesos y registrar sus tiempos. Con estos tiempos se hace un promedio del tiempo de las corridas por cantidad de procesos. Las pruebas se realizaron en un procesador Ryzen 5 5600g. Además, cada que se completaban las pruebas se reinicia la máquina para que el estado sea el mismo en cada prueba. Las pruebas son buscar la palabra en los libros:

* Quijote en Don Quijote de la Mancha
* Dios en la Divina Comedia
* Anna en Anna Karenina

La tabla con los resultados es:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Palabra** | **Cantidad de Procesos:** | | | | | **Promedios** | | | | |
| 3 | 6 | 7 | 8 | 15 | 0.6814 | 0.7 | 0.6742 | 0.6778 | 0.6862 |
| **Quijote** | 0.675 | 0.676 | 0.662 | 0.7 | 0.675 |
| 0.686 | 0.664 | 0.662 | 0.653 | 0.695 |
| 0.668 | 0.817 | 0.671 | 0.693 | 0.696 |
| 0.681 | 0.675 | 0.691 | 0.679 | 0.669 |
| 0.697 | 0.668 | 0.685 | 0.664 | 0.696 |
| **Dios** | 0.249 | 0.267 | 0.259 | 0.273 | 0.273 | 0.252 | 0.2644 | 0.2614 | 0.2716 | 0.2736 |
| 0.249 | 0.261 | 0.264 | 0.272 | 0.27 |
| 0.258 | 0.268 | 0.26 | 0.269 | 0.273 |
| 0.251 | 0.262 | 0.259 | 0.272 | 0.278 |
| 0.253 | 0.264 | 0.265 | 0.272 | 0.274 |
| **Anna** | 0.832 | 0.703 | 0.684 | 0.71 | 0.72 | 0.8318 | 0.7072 | 0.6946 | 0.714 | 0.724 |
| 0.849 | 0.697 | 0.715 | 0.712 | 0.726 |
| 0.839 | 0.701 | 0.703 | 0.694 | 0.712 |
| 0.821 | 0.722 | 0.685 | 0.741 | 0.722 |
| 0.818 | 0.713 | 0.686 | 0.713 | 0.74 |

La forma de leer esta tabla es que los números que se encuentra son el tiempo en segundos que el programa dura por proceso. Después el contenido de la parte derecha de la tabla es el promedio por corrida de cada proceso.

Los gráficos con los resultados de cada prueba:

# **Conclusiones:**

De los resultados obtenidos se puede hacer la observación que de estos casos la cantidad optima es de 7 procesos. Sin embargo, la diferencia no es tan abismal salvo en ciertas ocasiones. También se logra ver como entre las corridas con la misma cantidad de procesos en algunos casos la diferencia en el tiempo es muy grande, como es en el caso de 6 procesos.

Otra de las conclusiones es que al tener menos de 3 procesos el programa se vuelve lo suficientemente lento como para que pasado 3 minutos siga sin haberse impreso una sola vez. También se observa como la cantidad de procesos depende de la palabra y longitud del archivo. Esto como se observa en el segundo gráfico y en el último grafico donde para la segunda prueba, buscar Dios en la Divina Comedia, lo más optimo son 3 procesos mientras que para todas las otras pruebas lo óptimo son 7 procesos.

También se entiende que no siempre entre más procesos haya mejor pues como se observa en los datos tanto 8 como 6 procesos es más lento que usar 7 procesos. De aquí se entiende que al tener que el programa sincronizar más procesos de la cuenta, pierde el tiempo haciéndolo más lento. Es un caso muy parecido a tener una cantidad de procesos menor. Es en esta situación donde al estar el proceso leyendo su sección y este termine, el padre se le hace imposible asignarle uno nuevo. Así mismo se observa que la diferencia con 15 procesos en promedio no es muy grande salvo cuando el archivo es pequeño.