Trabajo Práctico 1: Informe

Descripción: Programa que calcula el hash MD5 de múltiples archivos de forma paralela utilizando múltiples procesos.

Integrantes:

- José Martín Torreguitar
- Thomas Beck
- Clara Guzzetti
- Ignacio Matías Vidaurreta

Instrucciones de Compilación y Ejecución

(Suponemos que el receptor del programa está utilizando un Sistema Operativo basado en Unix)
Para compilar nuestro proyecto basta con utilizar el makefile utilizando el siguiente comando desde la terminal en la carpeta del proyecto:

make all

Una vez compilado, basta con ejecutar:

./application [OPTION] … [FILES] …

-w indica la cantidad de workers involucrados
-f filename indica el nombre del archivo donde guardar los resultados en lugar de el default que es *results.res*

Para correr los tests creados utilizar el siguiente comando:

make test

Donde FILES es una serie de archivos separados por un espacio.
El programa se corrió en los siguientes sistemas:

- Docker dado por la cátedra
- Ubuntu 16.04 LTS
- Arch Linux 4.17.11

Decisiones Tomadas

1- Shared Memory

Se utilizó *shared memory* para el paso de información entre *application* y *viewer* porque así lo requería el trabajo y en sí presenta múltiples ventajas. La ventaja principal de utilizar *shared memory* por sobre otras formas de IPCs como por ejemplo *message passing* es la velocidad de comunicación, dado que una vez implementada la zona de memoria, escribir en esta es simplemente hacer una escritura en memoria normal.

Una restricción importante es que actualmente se reservan 1000 bytes de shared memory, por lo que eso es lo máximo que puede almacenar. En caso de que se necesitase trabajar con una cantidad de datos mayor, se puede modificar la macro *DFLT_SIZE* para agrandar la memoria de la shared memory.

2- Pipes

Para la comunicación entre application y slave se utilizaron pipes. Se tomó esta decisión para aprovechar la relación de parentezco entre slave y application, porque al crear un pipe desde un proceso padre y utilizar *execv* para darle el control al hijo, éste hereda los pipes del padre, de esta manera pudimos mapear pipes a la salida o entrada estandar y esta conexión continúa teniendo efecto en cada proceso hijo por ser estructuras primitivas de carácter local.
Para el trabajo utilizamos 2 pipes. Un pipe lo utilizabamos para mandarle los *paths* de los archivos a los hijos y el otro lo utilizabamos para que los slaves enviaran el cálculo de hash del archivo a application. En ambos casos se utilizó un único pipe, **lo cual puede presentar un problema bastante grave**. Como se puede comprobar al hace *man 7 pipe* bajo el título *Pipe Capacity* si bien éste varía según la versión de Linux y es susceptible a cambios, a partir de Linux 2.6.35 la capacidad default del pipe es 65536 bytes, por lo cual no podremos garantizar la atomicidad del pipe (y con ello el correcto funcionamiento de nuestro programa) si se trata de intercambiar esa cantidad de información en un mismo instante. En caso de que el usuario necesite un pipe más grande, se puede redefinir la capacidad del pipe con *F_SETPIPE_SZ* de fcntl.

3- Semaphores

Para evitar problemas de sincronización a la hora de recibir los hashes calculados utilizamos semáforos. Se tomó esta decisión teniendo en cuenta que se quería evitar *busy waiting* por parte de viewer. De ésta manera no se ocuparon recursos de procesamiento en la espera de los cálculos que enviaba *application* y *viewer* pudo acceder al *shared memory* de una manera atómica.

4 - Limitaciones

Como mencionamos antes, por la decisión tomada antes respecto de un sólo pipe, es limitada la cantidad de archivos que podemos hashear con nuestro programa.

Problemas Encontrados

User Defined Signal 1

En una primera versión del TP se utilizó la función *signal*() para definir el handler de *SIGUSR1* pero éste al recibir la señal en el *pause()* interrumpía el read que leía del file descriptor (en ese momento era un read y luego se pasó a readline, ver problema: Programa no Finaliza) el cual devolvía un error que al imprimir el errno obteníamos:

Interrupted system call

y finalmente, antes de terminar el programa imprimía *User Defined Signal 1*.
Ambos problemas se solucionaron al dejar de utilizar la función *signal()* como handler y utilizar la función que recomienda el man de signal() (pues avisa que ésta última no es para nada portable y que devuelve cosas distintas dependiendo la versión de linux) la cual es *sigaction()* que presenta una estructura mucho más robusta que *signal()*.
Al utilizar *sigaction()* como handler solucionamos la impresión de *User Defined Signal 1* y el error de la interrupción del read se solucionó utilizando el flag de sigaction *SA_RESTART* el cual es un flag que controla qué sucede cuando se envía una señal durante ciertas primitivas (como en nuestro caso, read) y el handler no devuelve errores, una de las opciones es hacer que la primitiva continúe desde donde estaba antes de ser interrumpido, lo cual es lo que se necesitaba para solucionar el problema[1].

Programa no Finaliza

El problema en este caso era que el programa imprimía sin problemas los hashes pero nunca terminaba. Esto sucedía porque en ningún momento se

[1] https://www.gnu.org/software/libc/manual/html_node/Flags-for-Sigaction.html

cerraba el pipe de lectura del slave, entonces se quedaba esperando a leer más cuando ya no había nada más para enviar, quedando de esta manera en un *deadlock.* Para solucionar esto se agregaron 2 variables: Una llamada **total**, la cual guarda la cantidad total de archivos enviados por argumento y otra llamada **printed_files**, la cual es un acumulador inicializado en 0 que cada vez que se guarda un archivo en la shared memory se aumenta en 1. Cuando se cumple la condición '*printed files == total'* quiere decir que no hay más archivos que leer entonces se cierra el pipe de lectura del slave, lo cual hace que envíe un EOF a los slaves indicandoles a éstos que dejen de leer.