Trabajo Práctico 1
Sistemas Operativos (72.11)

Juan Godfrid - 56609
Pablo Radnic - 57013
Joaquín Ormachea -
Francisco Delgado -

4 de abril de 2018

Índice general

Capítulo 1

Diseño

1.1. Diseño del Sistema

La aplicación fue diseñada para el sistema operativo Linux, pero debería funcionar para cualquier sistema operativo que respete la norma POSIX. El sistema consiste en tres tipos de procesos, un proceso maestro, un proceso vista y S procesos esclavos, S se calcula en base a la cantidad de archivos a procesar mediante la fórmula:

Sea N el numero de archivos a procesar, P el indice de proceso ponderado

$$S = \left\lceil \frac{N}{P} \right\rceil$$

El índice de proceso ponderado actualmente está definido como 4. Encontrar el índice de proceso ponderado más eficiente(o una fórmula que lo represente) excede los objetivos del trabajo práctico por lo que no se discutirá fuera de esta aclaración.

La aplicación comunica los distintos procesos entre sí mediante semáforos, *message queues* y zonas de memoria compartida. El funcionamiento de la aplicación es el siguiente: El proceso maestro es ejecutado y encola los N archivos a procesar a una cola de mensajes (a la cual tienen acceso los S procesos esclavos) y crea los procesos esclavos. El proceso esclavo recibe el archivo a procesar leyendo de dicha cola, lo procesa, y envía el resultado mediante otra cola al proceso maestro. Luego dicho proceso lee otro archivo de la cola de archivos y repite el procedimiento, y si está vacía, termina su ejecución. El proceso maestro lee de la cola de archivos y escribe el resultado a disco, luego revisa el semáforo para escribir en el buffer compartido, si no es su turno y el proceso vista está conectado, duerme hasta que el semáforo cambie de estado antes de poder realizar la escritura en el buffer. Cuando el proceso maestro termina de escribir en el buffer, cambia de estado el semáforo. Si el proceso vista no está conectado el semáforo cambia de estado instantáneamente. El proceso vista duerme hasta que el semáforo indique su turno, cuando lo es, lee del buffer compartido, escribe en pantalla la información deseada y cambia de estado el semáforo.

1.2. IPCs

1.2.1. Message Queue

El sistema utiliza dos *message queues* del tipo POSIX (son más modernas que sus contrapartes SYSTEM V) en la intercomunicación de los procesos maestro y esclavo. El primero se utiliza para enviar los archivos que se requieren *hashear* hacia los esclavos, mientras que el segundo se utiliza para enviar los *hashes* ya procesados devuelta al proceso maestro. La utilidad de dicho IPC es una multicausalidad. Al ser una cola, el *message queue* asegura el comportamiento FIFO, por lo que es ideal para enviar los *hashes* de vuelta al proceso maestro, ya que asegura que el orden de llegada corresponde al orden de procesamiento. El *message queue* también resulta adecuado para enviar los nombres de los archivos a los esclavos ya que los mensajes sirven como instrucciones atómicas: Los esclavos leen las instrucciones de la cola, cuando terminan, leen otra instrucción, y cuando está vacía la cola significa que el procesamiento concluyó. Todo esto se realiza sin gastar un solo ciclo de procesador del proceso maestro.

1.2.2. Semáforos

Con respecto a los métodos de intercomunicación de procesos para respetar sus "turnos" de actividad sobre la memoria compartida, optamos por dejar 3 bytes delante del buffer denominados Safety Code, Visual Semaphore y Status Semaphore, respectivamente. Safety Code tiene un valor determinado por el proceso maestro, esto nos permite verificar que la aplicacion vista no conecte a un espacio de memoria compartida incorrecto desde el comienzo de su ejecucion. Visual Semaphore es un semáforo que informa si el proceso vista está corriendo, es iniciado por defecto en 0 y cuando se conecta la aplicación, se cambia a 1. Para finalizar de forma correcta ambas aplicaciones, el proceso maestro desconecta a la aplicación vista poniendo en 0 este semáforo. Por último, Status Semaphore es un indicador del turno de cada proceso, 0 para el proceso maestro y 1 para el proceso vista. A partir de esto, los procesos maestro y vista evitan colisiones entre ellas.

Para evitar el *busy waiting* utilizamos la librería de semáforos de POSIX. Decidimos hacer un híbrido entre los bytes reservados y los semaphores del API de POSIX. Guardamos los valores dentro de los bytes reservados y utilizamos los semáforos POSIX para bloquear o liberar los procesos cuando necesitan manejar una espera. Este híbrido fue implementado debido a problemas encontrados en la API de POSIX de semáforos. *Léase: Sección 3.1.1.*

1.2.3. Memoria Compartida

Capítulo 2

Utilización

2.1. Compilación y Ejecución

El sistema posee un archivo de tipo *GNU make* que discrimina en la compilación del código fuente en base a qué proceso debería pertenecer, de este modo logra compilar separadamente los archivos binarios de manera autónoma. El programa debe recibir como argumento por línea de comandos una lista de archivos a procesar, también puede recibir una expresión *bash* para los archivos (por ejemplo: ./*.c). Un ejemplo de uso del programa es el siguiente:

```
/SO–TP1$ make
/SO–TP1$ ./Binaries/run main.c
```

2.2. Proceso Vista

El *make* compila también de manera separada el proceso vista, para ejecutarlo se debe enviar el siguiente comando:

Capítulo 3

Problemas y Resoluciones

3.1. Proceso Maestro

3.1.1. Problema de espera activo

msq recieve con block asi no hace espera activa sino que el SO lo despierta cuando la queue cambia de empty a nonempty

3.2. Proceso Esclavo

3.2.1. *Hasheo* de archivos

Nos vimos ante la necesidad de utilizar la función de Linux md5sum para conseguir el hash de los archivos. Sin embargo este función era únicamente accesible a través de línea de comandos de Linux y no en el entorno de biblioteca de funciones de c. Al no tener las herramientas para crear una función propia debimos encontrar otra solución para el problema.

Luego de investigación en Internet se propuso utilizar la función popen() con parámetro cmd que ejecuta un proceso de línea de comandos de Linux y a través de él ejecutar la función md5sum con los parámetros correspondientes. Luego la salida de la ejecución de las funciones es traída al entorno de la aplicación con la función fgets.

3.3. Proceso Vista

ssss

3.3.1. Problema de espera activa

Con la librería POSIX de semáforos inicialmente intentamos eliminar el uso de los bytes reservados en el buffer compartido. Para esto utilizamos la función *sem_getvalue*, la cual nos presentó problemas en su ejecución dentro del ciclo principal del master. Siempre que la función era llamada dentro del ciclo, se generaba un *segmentation fault*. Debido a esto tuvimos que manejarnos sin ver los valores del semaphore por lo que finalmente decidimos hacer un híbrido

entre los bytes reservados y los semáforos del API, así evitando la espera activa mediante la utilización de los semáforos de la API y utilizando los valores de los semáforos en el buffer de memoria compartida.

Capítulo 4

Testing

4.1. Unit testing

Se decidió implementar unit tests para cubrir toda funcionalidad implementada por el TP que sea, por lo menos, más de una mascara o algun agregado simlpe a una funcion de la librería estándar o POSIX. Esto se debe a que las estructuras de POSIX, tal como el sistema de queues y el sistema de semáforos que se proveen, resultan redundantes para testear debido a que estas funciones son de uso general del sistema operativo y ya estan testeadas por fuera de nuestro trabajo práctico. Por eso consideramos que todos los tests que meramente se resumían en un test de una función de POSIX no eran necesarios.

4.2. Fugas de Memoria

Era posible la presencia de fugas de memoria en cualquiera de los tres procesos, las fugas de memoria son errores de software que ocurren cuando los bloques de memoria reservados no son liberados antes de la finalización del programa. Para asegurar la ausencia de fugas de memoria se tomo el recaudo de utilizar la herramienta *Valgrind*. A continuación se encuentran los registros del programa para distintos parámetros.

Registros de Valgrind:

```
>>valgrind ./Binaries/run ./*

==7295==
==7295== HEAP SUMMARY:
==7295==     in use at exit: 0 bytes in 0 blocks
==7295==   total heap usage: 11 allocs, 11 frees, 17,528 bytes allocated
==7295==
==7295== All heap blocks were freed -- no leaks are possible
==7295==
==7295== For counts of detected and suppressed errors, rerun with: -v
==7295== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
>>valgrind ./Binaries/view 1234

==7550==
==7550== HEAP SUMMARY:
==7550==     in use at exit: 0 bytes in 0 blocks
==7550==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==7550==
==7550== All heap blocks were freed -- no leaks are possible
==7550==
==7550== For counts of detected and suppressed errors, rerun with: -v
==7550== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
>>valgrind ./Binaries/slave a a

==8156==
==8156== HEAP SUMMARY:
==8156==     in use at exit: 0 bytes in 0 blocks
==8156==   total heap usage: 4 allocs, 4 frees, 8,704 bytes allocated
==8156==
==8156== All heap blocks were freed -- no leaks are possible
==8156==
==8156== For counts of detected and suppressed errors, rerun with: -v
==8156== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Podemos afirmar que para los casos vistos nuestros programas no presentan fugas de memoria.

Capítulo 5

Limitaciones

5.1. Limitaciones del Sistema

max message

5.2. Limitaciones de la Implementación

Debido a que el programa reserva los argumentos -t y -w para la ejecución de modo de testeo y modo de espera, no se puede tener un archivo con este nombre como primer parámetro ya que interpretará el modo de ejecución y no un archivo para *hashear*.

PRIVACIDAD DE LOS IPC

Capítulo 6

Bibliografía

6.1. Fuentes

1. https://stackoverflow.com/questions/3395690/md5sum-of-file-in-linux-c
2. https://stackoverflow.com/questions/4553012/checking-if-a-file-is-a-directory-or-just-a-file
3. https://users.cs.cf.ac.uk/Dave.Marshall/C/node27.html