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 este comportamiento se realiza sin gastar un solo ciclo de procesamiento del proceso maestro. Las *message queues* resultan ineficaces en situaciones en las que se envían datos sin cota superior de tamaño, en cual situación sería más adecuado usar un *pipe*. Pero al ser datos acotados ($hash = 32$ bytes, nombre de archivo maximo y path maximo definidos por POSIX), los *message queues* son óptimos para el problema.

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:

```
/SO–TP1$ ./Binaries/view (PID)
```

Donde (PID) es el identificador del proceso maestro al que se quiere conectar. El PID es mostrado en pantalla cuando se ejecuta el proceso maestro.

Para facilitar el uso del programa vista, el programa maestro posee una opción *-w* la cual espera hasta 60 segundos para que el usuario tenga tiempo de conectar la aplicación vista. Ejemplo de uso:

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

2.3. Testeo

El programa maestro posee la opción *-t* para ejecutar los testeos unitarios del sistema. Ejemplo de uso:

```
/SO–TP1$ ./Binaries/run –t
```

Capítulo 3

Problemas y Resoluciones

3.1. Proceso Maestro

3.1.1. Problema de espera activa

El *message queue* que devuelve los *hashes* se crea con la opción de bloqueo de POSIX activada, esto hace que el proceso se bloquee por parte del sistema operativo cuando la cola esta vacía y luego el sistema operativo desbloquea el proceso cuando la cola pasa a no vacía, así evitando el problema de espera activa.

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

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. 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 simple 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 ./*

==21728==
==21728== HEAP SUMMARY:
==21728==     in use at exit: 0 bytes in 0 blocks
==21728==   total heap usage: 11 allocs, 11 frees, 14,520 bytes allocated
==21728==
==21728== All heap blocks were freed -- no leaks are possible
==21728==
==21728== For counts of detected and suppressed errors, rerun with: -v
==21728== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
>> valgrind ./Binaries/view 28143

==28234==
==28234== HEAP SUMMARY:
==28234==     in use at exit: 144 bytes in 4 blocks
==28234==   total heap usage: 5 allocs, 1 frees, 1,168 bytes allocated
==28234==
==28234== LEAK SUMMARY:
==28234==    definitely lost: 0 bytes in 0 blocks
==28234==    indirectly lost: 0 bytes in 0 blocks
==28234==      possibly lost: 0 bytes in 0 blocks
==28234==    still reachable: 144 bytes in 4 blocks
==28234==         suppressed: 0 bytes in 0 blocks
==28234== Rerun with --leak-check=full to see details of leaked memory
==28234==
==28234== For counts of detected and suppressed errors, rerun with: -v
==28234== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

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

==13485== HEAP SUMMARY:
==13485==     in use at exit: 0 bytes in 0 blocks
==13485==   total heap usage: 4 allocs, 4 frees, 8,704 bytes allocated
==13485==
==13485== All heap blocks were freed -- no leaks are possible
==13485==
==13485== For counts of detected and suppressed errors, rerun with: -v
==13485== 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 excepto en la ejecución de la vista, analicemos dicho caso con mayor profundidad:

```
>> valgrind --leak-check=full --show-leak-kinds=all ./Binaries/view 10671

==10758==
==10758== HEAP SUMMARY:
==10758==     in use at exit: 144 bytes in 4 blocks
==10758==   total heap usage: 5 allocs, 1 frees, 1,168 bytes allocated
==10758==
==10758== 24 bytes in 1 blocks are still reachable in loss record 1 of 4
==10758==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd6
==10758==    by 0x537354E: tsearch (tsearch.c:338)
==10758==    by 0x50547A9: check_add_mapping (sem_open.c:113)
==10758==    by 0x505499D: sem_open (sem_open.c:181)
==10758==    by 0x1090C4: openSemaphores (in /home/juangod/ITBA/SO/SO-TP1/Bina
```

```
==10758==    by 0x108EEF: main (in /home/juangod/ITBA/SO/SO-TP1/Binaries/view)
==10758==
==10758== 24 bytes in 1 blocks are still reachable in loss record 2 of 4
==10758==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd6
==10758==    by 0x537354E: tsearch (tsearch.c:338)
==10758==    by 0x50547A9: check_add_mapping (sem_open.c:113)
==10758==    by 0x505499D: sem_open (sem_open.c:181)
==10758==    by 0x109110: openSemaphores (in /home/juangod/ITBA/SO/SO-TP1/Bina
==10758==    by 0x108EEF: main (in /home/juangod/ITBA/SO/SO-TP1/Binaries/view)
==10758==
==10758== 48 bytes in 1 blocks are still reachable in loss record 3 of 4
==10758==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd6
==10758==    by 0x5054759: check_add_mapping (sem_open.c:96)
==10758==    by 0x505499D: sem_open (sem_open.c:181)
==10758==    by 0x1090C4: openSemaphores (in /home/juangod/ITBA/SO/SO-TP1/Bina
==10758==    by 0x108EEF: main (in /home/juangod/ITBA/SO/SO-TP1/Binaries/view)
==10758==
==10758== 48 bytes in 1 blocks are still reachable in loss record 4 of 4
==10758==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd6
==10758==    by 0x5054759: check_add_mapping (sem_open.c:96)
==10758==    by 0x505499D: sem_open (sem_open.c:181)
==10758==    by 0x109110: openSemaphores (in /home/juangod/ITBA/SO/SO-TP1/Bina
==10758==    by 0x108EEF: main (in /home/juangod/ITBA/SO/SO-TP1/Binaries/view)
==10758==
==10758== LEAK SUMMARY:
==10758==    definitely lost: 0 bytes in 0 blocks
==10758==    indirectly lost: 0 bytes in 0 blocks
==10758==      possibly lost: 0 bytes in 0 blocks
==10758==    still reachable: 144 bytes in 4 blocks
==10758==         suppressed: 0 bytes in 0 blocks
==10758==
==10758== For counts of detected and suppressed errors, rerun with: -v
==10758== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Los *leaks* surgen de los semáforos que son utilizados tanto por el proceso vista como por el proceso maestro, pero solo aparecen como fugas en el proceso vista. Esto se debe a que los semáforos son cerrados por el proceso maestro, por lo tanto valgrind afirma que son fugas de memoria ya que exceden al proceso vista.

Capítulo 5

Limitaciones

5.1. Limitaciones del Sistema e Implementación

- Los sistemas POSIX poseen un techo sobre la máxima cantidad de mensajes que puede tener una message queue, dicho máximo está dado por el menor de los valores de

  `/proc/sys/fs/mqueue/msg_max`

  y

  `/proc/sys/fs/mqueue/msg_default`

  Afortunadamente dichos archivos son fáciles de modificar, si se envían más archivos que el máximo, un mensaje de alerta se enviará explicando el error y como modificar los archivos.

- El sistema no cuenta con una búsqueda recursiva de archivos. Por ejemplo, pasar por parámetro ./* solo adjunta los archivos pertenecientes al directorio actual, no aquellos que pertenezcan a los subdirectorios del mismo.

- 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*.

- El sistema no puede calcular el *hash* de directorios, por una limitación de la herramienta de md5sum.

- Si existe una zona de memoria compartida asociada con la misma key que la que utiliza la aplicación(muy poco probable) previo a la ejecución, puede generar problemas.

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