Instituto Tecnológico de Buenos Aires

Informe

72.11 - Sistemas Operativos - TP1

Integrantes:

Farbiarz, Bruno Ilan.
62644

Fernandez Dinardo, Juan Ignacio.
62466

Martinez, Tomas.
62878

1. Decisiones tomadas durante el desarrollo

1.1 Comunicación entre Procesos (IPC)

Nos resulta adecuado empezar comentando sobre el flujo de comunicación entre los procesos que conforman al programa para que de esta forma se pueda realizar una observación más íntegra de la implementación. El primer desafío presentado en el transcurso del trabajo práctico fue el de como conectar el proceso master a los procesos slaves, cuyo propósito sería el de hashear los archivos. Optamos por usar dos pipes unidireccionales por relación master-slave: uno para pasar el filepath desde el master para que el slave pueda leer, y el otro pipe para que el slave pueda escribir el filepath ya hasheado y que, de esta forma, el master pueda leerlo. Esto hace la comunicación ágil y libre de posibles race conditions ya que estamos leyendo y escribiendo en pipes diferentes. Además, mediante el uso de la función select, logramos una sincronización en la relación master-slave tal que el master pueda saber qué pipes poseen el resultado listo para leer, evitando así que al intentar leer el proceso principal se bloquee, generando una demora y demás ineficiencias, o que lea un resultado a medias, es decir, que el master intente leer cuando justo el slave se encuentra escribiendo.

Por otro lado, dentro de cada slave decidimos que inicien un nuevo proceso que se encargue de ejecutar el programa md5sum a través de execv. Este último recibe el filepath por pasaje de argumentos, mientras que el resultado es devuelto al slave mediante un nuevo pipe.

Finalmente, para conectar el proceso master con el proceso view optamos por armar una memoria compartida que es escrita y leída con la ayuda de un semáforo. Esto es debido a que una vez que los dos procesos están conectados a la memoria compartida es muy fácil que se intente escribir y leer sobre ella al mismo tiempo, y con la ayuda del semáforo podemos avisarle al otro proceso si hay algo escrito para leer o si ya se leyó y puede escribir devuelta para que sea leído por el proceso vista.

1.2 Creación de Slaves

La creación de los procesos slaves se basa en los siguientes casos:

- Se crean un máximo de 10 slaves;
- Si la cantidad de archivos a procesar (tareas) es menor a 10, entonces se crean tantos slaves como tareas;
- Si la cantidad de tareas es 10 o más, entonces a cada slave se le asigna una carga inicial de dos tareas (si es que hay suficientes archivos para todos los slaves);
- Si la cantidad de tareas es menor a 10, entonces se asigna una única tarea por slave;

- Una vez que cada slave termine con su carga inicial, en el caso de que aún haya más tareas por procesar, estas se van asignando a los slaves a medida que estos se vayan liberando;

Cabe destacar que una vez se hayan procesado todos los archivos, el master es el encargado de terminar con la ejecución de cada slave.

1.3 Implementación de la Memoria Compartida

Como se mencionó anteriormente para conectar al proceso master con el view se implementó una memoria compartida mediante el uso de POSIX en la cual el master participa con el rol de escritor, mientras que el view hace de lector. La implementación de esta memoria está diseñada de forma tal de que el escritor escriba en memorias diferentes a través del uso de un offset. Es decir, los resultados no se sobreescriben. Dado que cada resultado está separado de un \0, la lectura se va realizando de a un resultado y a medida que se va recibiendo la información.

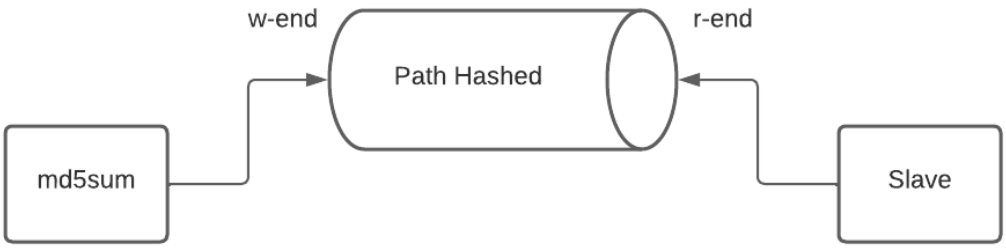
1.4 Mecanismos de Sincronización

Para sincronizar la lectura de la memoria junto a la escritura y así evitar condiciones de carrera o demás problemas de ejecución, se decidió implementar un semáforo. Este último representa la cantidad de resultados que se pueden leer, es decir, arranca inicializado en cero por lo que el proceso vista queda bloqueado hasta que el master escriba un resultado. Esto es porque inmediatamente después de que el master termina de escribir un resultado en memoria, incrementa el semáforo dando la posibilidad de que el proceso vista pueda leer. Además, complementando lo explicado en el apartado anterior, dado que el proceso master no sobrescribe los resultados, view maneja su propio offset por lo que puede ir leyendo los resultados a medida que estos estén disponibles.

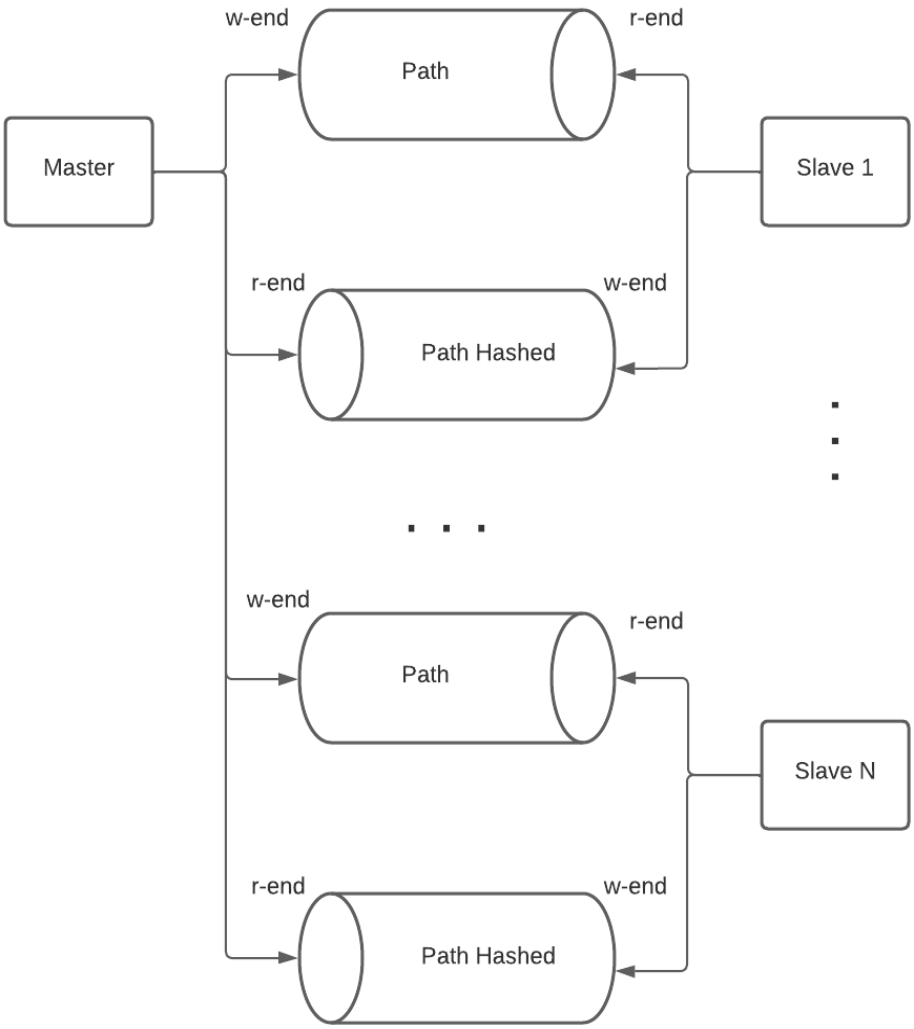
1.5 Manejo de Errores

En cuanto al manejo de errores se verificaron los retornos de la mayor cantidad de funciones posibles o, por lo menos, de todas aquellas que consideramos que tienen mayores posibilidades de fallar. Esto se hizo a través de la función perror() del manual la cual, previamente a abortar, imprime por STDOUT un mensaje con una descripción de la posible causa de falla.

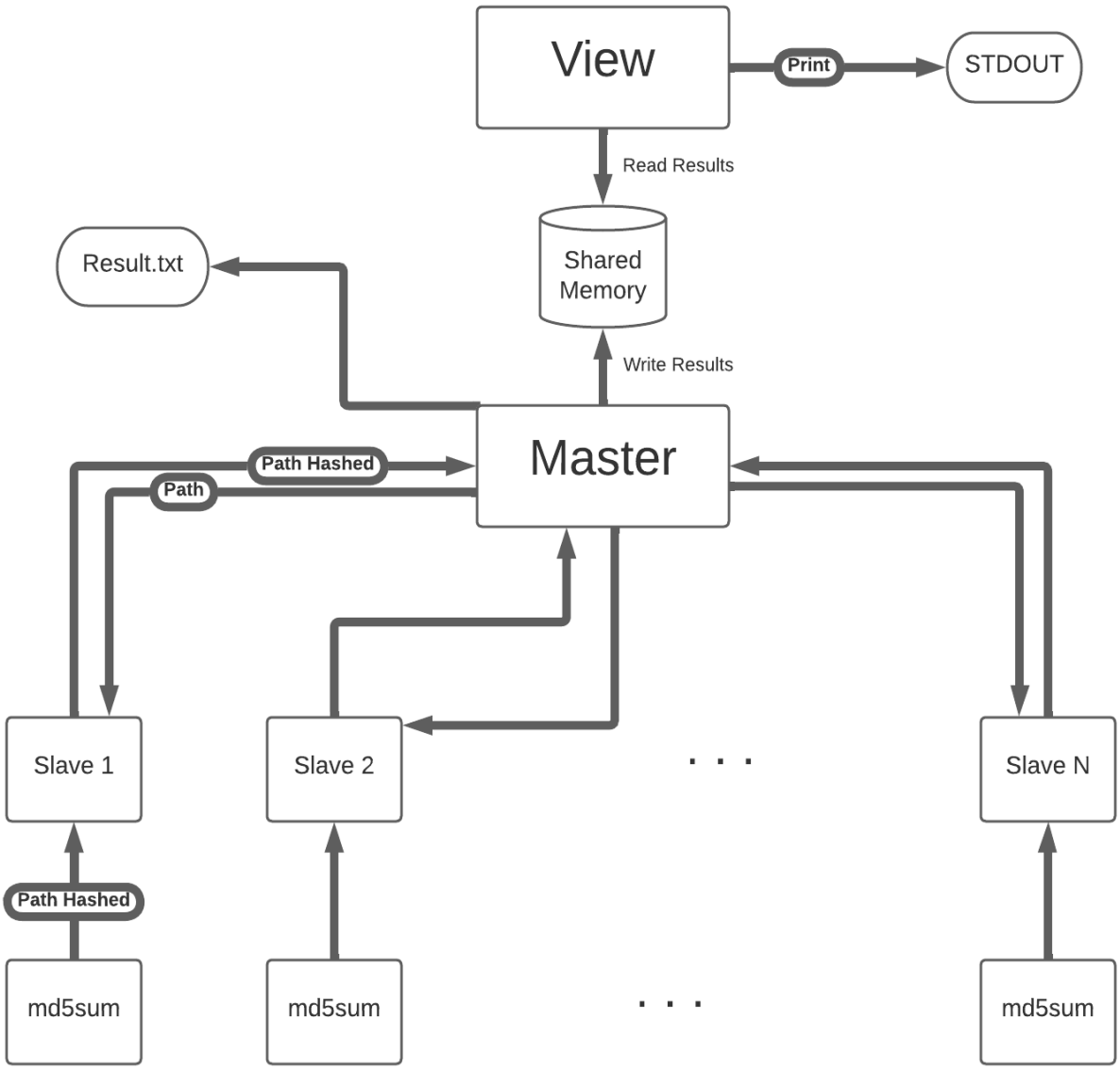
2. Diagramas



Pipe que comunica al proceso slave con el md5sum.



Comunicación entre master y cada proceso slave. Utilización de dos pipes por relación.



Flujo general del programa. Se muestra la comunicación entre los Slaves y el md5sum, las relaciones master-slave y la comunicación entre master y view mediante el uso de la memoria compartida.

3. Instrucciones de Compilación

Dentro del directorio en el cual se encuentran los archivos bajados, ejecutar el siguiente comando:

```
$ make
```

De esta forma se obtendrán todos los archivos listos para la ejecución.

4. Instrucciones de Ejecución

Por un lado podemos ejecutar el programa master junto a los paths a los archivos que deseamos hashear:

```
$ ./master <INSERT FILE PATH>
```

Es importante mencionar que pueden pasarse como argumentos la cantidad de paths a archivos que se necesite. El resultado con las claves generadas para cada archivo se encontrarán en un archivo de texto titulado como “resultados” ubicado en el directorio del programa.

Por otro lado, paralelamente al proceso master, se puede ejecutar el programa view, que se encargará de imprimir por salida estándar los resultados:

```
$ ./view <INSERT_MEM_SIZE>
```

En cuanto al valor de tamaño de memoria a pasar como argumento, una vez ejecutado el master, este imprimirá por STDOUT el valor de memoria que se utilizará, por lo que tendremos 5 segundos para copiarlo y pegarlo como argumento para view.

Para iniciar ambos programas a la vez, se deberá ejecutar lo siguiente:

```
$ ./master <INSERT FILE PATH> | ./view
```

5. Limitaciones

Por un lado, en cuanto al procesamiento de los archivos, como no se sabe en qué sistema se puede llegar a correr este código, se le puso un tope de procesos esclavos relativamente bajo (10), para que no se presenten inconvenientes. Si se pusiera un tope más alto, podrían surgir problemas donde el procesador no pueda manejar fluidamente todos los procesos y terminará exigiendo mucho a la computadora que lo corra. También, el trabajo no hace chequeos de validación sobre todo el INPUT que recibe, por lo que problemas de ejecución podrían surgir de ahí. Luego, hay una limitación sobre la longitud de los file paths que puede leer el programa, ya que se arma un segmento de memoria fijo que si el path es más largo que lo que entra, el programa no puede guardar el path entero y genera un error. Aun así, esto último es fácilmente modificable. Finalmente, el programa está hecho para que los slaves tengan una carga inicial de dos paths y no se encuentra lo suficientemente modularizado como para poder modificar este valor.

Por otro lado, en caso de que el programa tenga alguna falla durante la ejecución y necesite abortar, puede que haya recursos como la memoria compartida y semáforos que no cierren correctamente, pudiendo provocar problemas en las siguientes ejecuciones.

Finalmente, dado que en la memoria compartida se guarda la información de todos los resultados, es decir, no se sobreescribe nada, se necesitará siempre tener una cantidad de memoria disponible para que toda esta información pueda almacenarse.

6. Problemas encontrados durante el desarrollo y cómo se solucionaron

A lo largo de la implementación, nos fuimos encontrando con una variedad de problemas de los cuales a la gran mayoría los consideramos como evidentes o esperables. Esto es parte de la cuestión lógica de trabajar por primera vez con ciertos recursos como lo son la memoria compartida y los semáforos. Generalmente mediante lecturas del manual de Linux y consultas a internet, todos aquellos problemas que surgieron terminamos solucionandolos sin tanto problema.

Sin embargo, cabe destacar que hubo algunas cuestiones que realmente nos complicaron y generaron demasiada demanda de nuestra parte para poder hacer funcionar el trabajo. Principalmente porque tuvimos que realizar dos implementaciones del proceso vista y su sincronización con el máster a través de la memoria compartida y los semáforos. La razón de esto fue porque originalmente nuestra primera implementación fue mediante System V. Particularmente fue realizada por uno de los integrantes del grupo y, al momento de testear por

parte del resto de los integrantes, resultó que casi nunca les funcionaba. En la mayoría de los casos, al correr el programa en docker, este funcionaba la primera vez y luego dejaba de hacerlo. Luego de una ardua tarea de investigación, concluimos que podía ser problema de compatibilidad con la librería la cual es más antigua que POSIX y que en ningún momento era mencionada por la cátedra. De esta forma, se migró el código a POSIX, produciendo un retraso inesperado en el desarrollo del trabajo práctico. Finalmente, una vez terminada la implementación, nuevamente ocurre que no le andaba a todos los integrantes del grupo. Después de más horas de investigación, a uno de los chicos le empezó a andar luego de ejecutar el siguiente comando "rm /dev/shm/*", esto fue un fuerte indicador de que siempre estuvimos haciendo mal la liberación de recursos y teníamos problemas cuando queríamos volver a ejecutar el programa. Particularmente, recordamos que la primera implementación con System V en ningún momento hacía uso de la función sem_unlink().

7. Citas de fragmentos de código reutilizado de otras fuentes

En general no se citó código para este trabajo, sino que en general se realizaron diferentes consultas en las distintas etapas de la implementación. Particularmente más en sobre cómo crear y conectar tanto la memoria compartida como el semáforo, al igual que como cerrar correctamente cada uno, así como también la sincronización de los procesos slave con el master mediante select. Las fuentes a las que mayor cantidad de veces recurrimos fueron el manual de Linux y chatGPT.