Instituto Tecnológico de Buenos Aires

72.11 Sistemas Operativos

TP1

INTEGRANTES:

(n° 62028) Nicolás Matías Margenat - nmargenat@itba.edu.ar

(n° 62057) Manuel Esteban Dithurbide - mdithurbide@itba.edu.ar

(n° 62067) Marcos Gronda - mgronda@itba.edu.ar

Índice

Introducción

En el presente informe se busca detallar cómo fue el proceso de desarrollo del Trabajo Práctico 1 de la materia Sistemas Operativos, cuyo objetivo era aprender a utilizar los distintos tipos de IPCs presentes en un sistema POSIX. Para ello se implementó un sistema que distribuía el cómputo del md5 de M archivos entre varios procesos. En la *Figura 1* se puede ver un diagrama de cómo están conectados los distintos procesos.

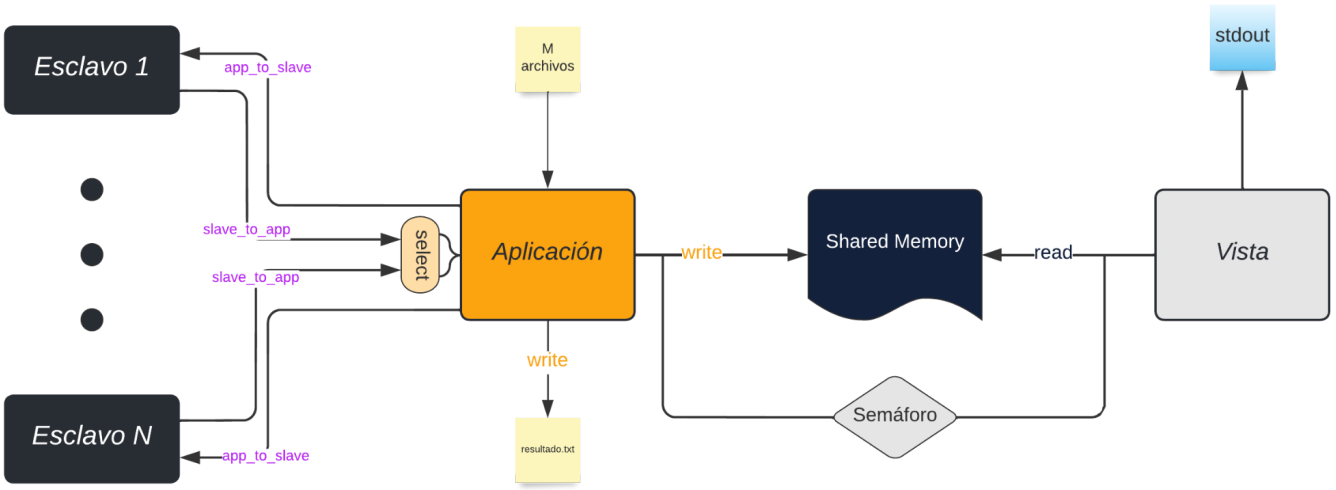

Figura 1. Diagrama de conexión de los procesos.

A continuación se detallarán los contenidos del presente informe. En una primera instancia, se darán las instrucciones de compilación y ejecución. Luego, se detallarán las decisiones importantes que se tomaron durante el desarrollo, seguido de los problemas que surgieron durante el mismo. Finalmente, se precisarán las limitaciones que se cree que tiene el trabajo.

Instrucciones de compilación y ejecución

Requerimientos: Para compilar los programas es necesario estar dentro del contenedor de docker provisto por la cátedra.

Compilación:

1. Para abrir el contenedor de docker ejecutar:

```
user@linux:~$ docker pull agodio/itba-so:1.0
user@linux:~$ cd tp1-so
user@linux:~$ docker run -v "${PWD}:/root" --security-opt
seccomp:unconfined -ti agodio/itba-so:1.0
user@docker:~$ cd root
```

2. Para compilar el programa, ejecutar dentro del contenedor docker:

```
user@docker:~/root$ make all
```

Ejecución: Hay 3 posibles maneras para ejecutar el programa:

- *Caso 1*: Simplemente cálculo de los hash

```
./md5 <files>
```

- *Caso 2*: Calcular los hash y ver los resultados mientras se procesa

```
./md5 <files> | ./vista
```

- *Caso 3*: Equivalente al caso 2 pero en dos terminales en simultáneo:

    - *Terminal 1*:

    ```
    ./md5 <files>
    ```

    El programa va a imprimir 3 nombres. Estos tienen que ser usados como argumentos para el proceso *vista* en el mismo orden que se imprimen

    - *Terminal 2*:

    ```
    ./vista <arg1> <arg2> <arg3>
    ```

Decisiones tomadas durante el desarrollo

En primer lugar, se detallarán los mecanismos de IPC que se utilizaron:

- Dado que se especifica en el enunciado, se utilizaron pipes anónimos para la comunicación entre el proceso aplicación y los procesos esclavos.
- Para obtener el resultado del comando md5sum de los procesos subesclavos se decidió redireccionar la salida estándar de estos a la entrada estándar del proceso esclavo, ya que nos pareció la manera más natural de hacerlo sin crear archivos intermedios.
- Para la comunicación entre el proceso vista y el app se utilizó un bloque de memoria compartida en conjunto con dos semáforos.

Por otro lado, en una de las clases se discutió acerca de las distintas maneras de lograr la sincronización de los procesos Vista y Aplicación. Según el profesor, la manera más elegante de lograr esto era utilizando un semáforo que incrementase a medida que se escribían archivos, y que decrementase cuando se leían. Luego de discutirlo, el equipo llegó a la conclusión de que este modo de realizar la sincronización era simple y eficaz, por lo que se decidió implementarlo.

En lo que respecta al proceso Vista, se tenían dos formas de invocarlo. El primero es aquel en el que se lo invoca con un pipe de la siguiente manera: *'./md5 <files> | ./vista'*; para el cual se decidió alocar espacio en el heap y utilizar la función *fgets* ya que es segura, pues permite poner un límite a la cantidad de bytes leídos, y permitía leer fácilmente la entrada estándar. La otra alternativa era invocarlos en terminales distintas, caso en el cual debería ser capaz de recibir argumentos.

A la memoria compartida se le escribe desde el proceso App una estructura la cual contiene el hash calculado, el nombre de archivo, el pid del esclavo que se encargó de calcularlo y además, se decidió guardar la cantidad de archivos que faltan por calcular. Se agregó este dato para que el proceso Vista esté al tanto de cuando finalizan los cálculos de md5 y por lo tanto, ser capaz de terminar su ejecución.

Un problema no menor fue decidir cómo liberar la memoria compartida, ya que podía suceder que se desvinculase cuando todavía el proceso Vista no había terminado de leer. Se tomó la decisión de usar un semáforo para indicar cuando finaliza el proceso Vista. Sabiendo esto, el proceso Aplicación puede liberar los recursos utilizados de una manera segura.

Relacionado al semáforo anterior, surgió la pregunta de cómo debería afectar al proceso App si el proceso Vista nunca es ejecutado. Se llegó a la conclusión que el semáforo debía empezar abierto; de este modo, si el proceso Vista no lo cierra, se podrían liberar automáticamente los recursos usados desde el proceso Aplicación. Se tomó esta decisión de esperar al Vista ya que se considera que el encargado de liberar los recursos es el proceso principal pues es quien los creó.

Finalmente, en relación a la liberación de recursos en caso de error, se definieron 3 casos:

1) Ocurre un error en el proceso Vista
2) Ocurre un error en el proceso Slave
3) Ocurre un error en el proceso Aplicación

En el primer caso, se llegó a la conclusión que si ocurre un error, esto no debería afectar al proceso App. A este fin, se decidió que si ocurriese un error, se le señalaría que se dejó de leer la memoria compartida, así el App puede liberarlo eventualmente.

En el segundo, al igual que el proceso Vista, no se afectará al proceso Aplicación y este podrá liberar los datos eventualmente. Hay que tener en cuenta que al no señalizar al proceso app que ocurrió un error, este asumirá que sigue existiendo, por lo que genera ciertos problemas que se esbozan en la *sección de limitaciones*.

En el último caso, utilizando la función *on_exit()*, se garantiza que en cualquier caso, ya sea de error o de finalización normal, se liberan los recursos compartidos.

Problemas durante el desarrollo

Durante el desarrollo, se encontraron con algunos problemas que requirieron una cantidad considerable de tiempo para resolver. Estos fueron:

- La manera de recibir los files. En una primera instancia, se creía que había que realizar un parseo de la expresión que se pasase como argumento, lo cual no era trivial de realizar manualmente. Para ello, se comenzó a investigar la función *glob()*, y comprenderla le llevó al equipo un tiempo. Sin embargo, para la desgracia o fortuna del grupo, un tiempo más tarde se llegó a la conclusión de que no era necesario utilizarla ya que esto lo hacía por defecto Linux.
- El manejo de los file descriptors a lo largo de todo el código. Este inconveniente vino de la mano de cómo guardar la información de los slaves. La solución por la cual se optó fue armar una estructura, llamada *slave_info,* en la cual se almacenan ordenadamente a la hora de crear los slaves y pipes. El equipo tuvo que estar atento con los *dup2()* y *dup()* a lo largo del código.
- La función *select()*. Para poder hacerlo funcionar, se necesitó investigar en diferentes documentaciones/foros. Finalmente, gracias a ejemplos y el uso de fd_set, FD_ZERO y FD_SET se logró su correcto funcionamiento.
- Un problema menor fue el uso de *execv()* para ejecutar el md5, ya que al momento de ejecutar las funciones de la familia *exec* se borra la memoria y esto era un problema. Se solucionó esto mediante investigación, ejemplos del manual, y la realización de algunos programas con menor complejidad.
- Un posible caso de uso es que el proceso App al ejecutarse, no solo se le envíen archivos, si no también directorios. Para solucionar este posible problema, se agregó una verificación de archivos utilizando la función *stat()* de la dependencia *<sys/stat.h>*. De esta manera al comienzo del proceso App, se arma un vector de files dejando de lado los directorios.
- La liberación de recursos en caso de error fue un gran problema. Se decidió finalmente utilizar la función *on_exit()* y setear en esta funciones que hacen *unlink* de la *shared memory* y de los semáforos. De esta manera, sin importar la razón por la que se termina la ejecución del programa, se liberan los recursos.
- Correcta modularización. La correcta modularización de las funciones fue otro tema que requirió una gran inversión de tiempo. Se decidió crear un *resource_manager* que administrase los recursos que usaban nuestros procesos, es decir, la memoria compartida, los semáforos, los pipes, la creación de files, así como el *forkeo*. De esta manera, si ocurría algún error en la ejecución de ellos, se cortaba la ejecución del programa y se enviaba un mensaje de error explicando qué fue lo que falló.

Limitaciones

El programa debido a su diseño contiene algunas limitaciones funcionales:

- Una limitación encontrada fue en el envío de los argumentos entre el proceso Vista y el App. Esto se debe a que, en el caso del pipe (*./md5 <archivos>| ./vista*), el proceso App escribe los argumentos (shm y sem) en salida estándar, la cual es redireccionada a la entrada estándar del proceso Vista. Por lo tanto, en el segundo caso de ejecución (correr *./md5* y *./vista* en diferentes consolas) el proceso App imprime por salida estándar (en esta ocasión la consola) los nombres de la shm y el sem. Igualmente, con el objetivo de darle una funcionalidad, decidimos que estos sean los argumentos que se le deben enviar al proceso Vista. Como dicen los expertos en el tema: *"It's not a bug, it's a feature"*.
- Otra limitación que se encontró está relacionada con el manejo de errores. Esta se presenta cuando ocurre un error en uno de los Slaves, por lo que corta la ejecución. En este caso, por un lado, no se le señala al proceso App que deje de mandar nombres de archivos para procesar al Slave, y por el otro, el nombre que se estaba procesando mientras ocurrió el error no se procesa de manera correcta; consecuentemente perdiendo el cálculo de un archivo.
- Por otro lado, debido al tamaño de la shm, el programa tiene una limitación en la cantidad de archivos que puede procesar. En este caso la máxima cantidad es de 50 archivos.
- Dado que no se captura la señal de interrupción de ejecución (*CTRL + C*), en caso de que el programa se vea interrumpido por ella, no se liberan los recursos creados.

Citas de código

- La función *is_regular_file()*, encontrada en el archivo *app.c* se extrajo de una publicación de Stack Overflow:

Jookia, et al. “Checking If a File Is a Directory or Just a File.” *Stack Overflow*, 13 Marzo 2015, https://stackoverflow.com/questions/4553012/checking-if-a-file-is-a-directory-or-just-a-file.