



Trabajo Práctico 1: **Inter Process Communication**

Grupo 2:

Della Sala, Rocío 56507
Giorgi, M. Florencia 56239
Rodríguez, Ariel Andrés 56030
Santoflaminio, Alejandro 57042

Introducción

El trabajo práctico consistió en desarrollar un sistema que calcula de forma distribuida los hashes md5 de un grupo de archivos recibidos por parámetro. Para realizar esto fue necesario tener tres procesos diferentes (aplicación, esclavo y vista) y comunicarlos a través de distintos mecanismos de IPC (pipes, shared memory y semáforos) presentes en un sistema POSIX.

Composición del sistema

Los procesos principales que componen el sistema son:

Aplicación:

Este proceso es el encargado de comunicarse tanto con los esclavos (asignándoles archivos a procesar) como con la vista (compartiéndole lo que debe mostrar por pantalla).

NOTA: Se definió una estructura del tipo *slaves_o* para cada esclavo. La misma está compuesta de dos file descriptors para la comunicación bidireccional entre aplicación y esclavo y un valor booleano *isWorking* que nos ayuda a saber si el esclavo se encuentra liberado para asignarle más tareas.

En relación con los esclavos, lo que hace la aplicación es:

- Crearlos. Se crearon dos pipes usando los file descriptors de la estructura. Luego se realizó un fork con el objetivo de ejecutar a los esclavos. Antes de ejecutarlos, los pipes de cada esclavo fueron manipulados con la ayuda de las funciones *dup2* y *close* para establecer el funcionamiento que necesitábamos (por ejemplo, redireccionamos la entrada estándar del esclavo al descriptor de archivo de lectura que comunicaba aplicación -> esclavo).
- Almacenarlos a través de un vector punteros a la estructura *slave_o*.
- Distribuirles una cantidad predefinida de tareas a cada esclavo sólo si este se encuentra libre.
- Recibir y almacenar los resultados para luego escribirlo a un archivo en disco o compartírselos a la vista.

En relación con la vista, lo que hace la aplicación es:

- Crear (y luego destruir) un segmento en memoria común para ambos procesos (aplicación y vista), el cual será usado para compartir los resultados de los archivos procesados.
- Crear (y luego destruir) un semáforo para administrar la memoria compartida, ya que tenemos dos procesos accediendo a un recurso en común.
- Darle la posibilidad al usuario de ejecutar la vista, a través de un menú y un manual.

Esclavo:

Este proceso es el encargado de procesar los archivos y enviar a su padre los hashes md5. Para esto, optamos por reemplazar la entrada y salida estándar de estos

procesos por dos pipes distintos generados por el proceso aplicación. Además, para calcular cada hash md5 y recolectar el resultado del comando md5sum, realizamos un fork de cada proceso esclavo y creamos un pipe en el cual se ejecuta el comando, redireccionando su salida a este pipe.

Vista:

Este proceso representa la parte visual del sistema. En este proceso se toman los datos guardados por el proceso aplicación en la memoria compartida y se imprimen de manera secuencial. Además, se utilizaron semáforos para la lectura de la memoria compartida para evitar que se produzcan colisiones si la aplicación está escribiendo simultáneamente.

Por otro lado, creamos un TAD de una cola, la cual usamos tanto en la aplicación para almacenar los archivos a procesar como en los esclavos para manejar la cola de pedidos. Internamente es una lista simplemente encadenada con header con referencias al primer y último elemento de ésta.

Instrucciones de compilación

Para compilar el programa es necesario dirigirse a la carpeta donde se encuentra el trabajo práctico y ejecutar el comando `make all` en la terminal.

Instrucciones de ejecución

Para ejecutar el programa es necesario dirigirse a la carpeta donde se encuentra el trabajo práctico y ejecutar el comando `./application hash files` en la terminal, siendo `files` el directorio que contiene los archivos a procesar. Luego el programa nos avisará que si queremos ejecutar el proceso vista será necesario ejecutarlo en otra terminal con el PID que se nos indica haciendo `./view PID`.

Para ejecutar los test realizados es necesario dirigirse a la carpeta donde se encuentra el trabajo práctico y ejecutar el comando `./application test` o `./slave test`. **Aclaración:** Los test se encuentran realizados en conjunto con el programa ya que por una cuestión de tiempo y de uso de ciertas funciones, no se pudieron separar. En el caso del test de la aplicación este se encuentra en `applicationTest.c` y en el caso del test de esclavo se encuentra en `slave.c` en conjunto con el resto del programa. La carpeta test del proyecto es aquella que contiene los files a procesar.

Decisiones tomadas

- **Comunicación aplicación-esclavo:** Para establecer la comunicación entre aplicación y esclavo se usaron dos pipes por cada esclavo que tenga mi programa. De esta forma la comunicación es única para cada esclavo. Como en este caso la comunicación entre ambos estaba dada por pasaje de mensajes de forma bidireccional, optamos por este IPC.
- **Comunicación aplicación-vista:** Para establecer la comunicación entre la aplicación y la vista se hizo uso del concepto de memoria compartida. También se añadió el uso de semáforos para su sincronización. Como la vista depende de si el usuario la quiere ejecutar o no, no sería posible comunicar la vista y la aplicación a través de pipes ya que estos solo son útiles si hacemos un fork. Una solución a esto eran los

named pipes pero se optó por memoria compartida porque creemos que esta cumple mas con el concepto de lo que se quería lograr (compartir el buffer de llegada). En este caso se decidió que el tipo de dato a compartir sea del tipo char* para facilitar la escritura y la lectura de los hashes de los archivos. De esta forma se escribe la información necesaria una detrás de otra en un mismo string separando cada una con una barra vertical. El uso de semáforos fue necesario para evitar que dos procesos hagan uso de la memoria compartida en forma simultánea (uno escribiendo y el otro leyendo).

- Espacio de memoria compartida: Se decidió poner un límite de espacio en memoria compartida de 30KB. Esto permite procesar miles de archivos sin generar problemas de espacio de memoria.

Consideraciones

- La cantidad de archivos a procesar y de esclavos del programa está definida en los .h correspondientes mediante el uso de define. Definimos que la cantidad de archivos a procesar es 2 y la cantidad de esclavos es 3 pero eso se puede cambiar si el número de archivos totales cambia y se quiere mejorar la eficiencia.
- La aplicación termina los procesos esclavos enviándoles un carácter especial que nosotros definimos que sea '+'. **Para el correcto funcionamiento del trabajo, se pide no utilizar directorios o archivos con este carácter especial en el nombre.**
- Los resultados se envían a la vista separados por un '|' y se utiliza el carácter especial '?' para indicarle a la vista que ya no habrán más archivos para imprimir. **Para el correcto funcionamiento del trabajo, se pide no utilizar directorios o archivos con este último carácter especial en el nombre.**

Limitaciones

- Nos pareció que en el caso que la vista no se ejecute, sería deseable evitar la creación de la memoria compartida y del semáforo. Por limitaciones de tiempo en la entrega del trabajo y por la falta de una idea de cómo desarrollar una solución, no se pudo llevar a cabo.
- En principio lo que se quería hacer era crear una estructura char ** en donde guardábamos los hashes y esta misma estructura iba a ser aquella que compartimos con la vista. Por cuestiones de que teníamos poco conocimiento a la hora de programar shared memory, se optó por usar un char * que nos resultaba más fácil. De todos modos, se dejó el char ** para manejarnos con este a la hora de volcar el resultado a un archivo, aunque también podíamos usar esto en la memoria compartida.
- La cantidad de esclavos del programa se estableció en tiempo de compilación, a partir de define. Hubiera sido deseable, calcular la cantidad de esclavos en tiempo de ejecución a partir de la variable files. Por otro lado, se buscó una relación que fuera óptima en cuanto a minimizar el tiempo de ejecución de la aplicación. Dado que tener poca cantidad de archivos no modifica tanto el tiempo de ejecución de la misma, decidimos dejarlo establecido en un define para ahorrar molestias en el código.

Problemas encontrados y soluciones

- Si bien el programa en si funcionaba bien, al correr la herramienta Valgrind, sobre la cual se desarrollará más adelante, se detectaron diversos leaks de memoria y errores. Los mismos fueron corregidos gracias a la información que proporciona esta herramienta.
- Uno de los problemas graves que se presentaron a la hora de implementar la conexión entre el proceso de aplicación y los procesos esclavos fue la comunicación bidireccional, más específicamente la comunicación de esclavos a aplicación. El problema fue que por más que habíamos hecho una conexión bidireccional que parecía funcionar correctamente, a la hora de hacer que los esclavos no finalicen hasta que la aplicación les diga, no mandaban la información por el pipe, por lo que la aplicación quedaba colgada cuando hacía un read. Luego de mirar mucho el código y probar diferentes herramientas, nos encontramos con una herramienta que permite ver las lecturas y escrituras que hace un proceso y sus hijos (la herramienta utilizada fue "strace"). Gracias a esta herramienta concluimos que el problema en la conexión era que el esclavo para mandar información a la aplicación lo hacía a través de un printf, ya que habíamos redirigido la salida estándar del hijo al pipe de escritura. El problema fue que, por alguna razón, el printf no mandaba bien la información al padre. La solución fue bastante simple, cambiar el printf por la función write, indicándole que escriba a la salida estándar.

Herramientas de análisis

En el trabajo se realizó un análisis de memoria mediante Valgrind, una herramienta de software libre que ayuda en la depuración de problemas de memoria y rendimiento de programas. Utilizamos esta herramienta para los ejecutables *application*, *slave* y *view* y redujimos los "leaks" de memoria de estos hasta llegar a cero.

La siguiente imagen muestra el resultado al ejecutar el comando `valgrind --leak-check=full ./application hash ./test` en la terminal, siendo test el directorio que contiene los archivos a hashear. En este caso la vista no fue llamada.

```
Sending ./test/hola to slave number 1
Sending ./test/prueba to slave number 2
Sending ./test/Directorio2/archivo2 to slave number 2
Sending ./test/Directorio2/archivo6 to slave number 0
Sending ./test/Directorio2/archivo3 to slave number 0
Sending ./test/Directorio2/archivo5 to slave number 1
Sending ./test/Directorio2/archivo7 to slave number 1
Sending ./test/Directorio2/archivo4 to slave number 2
Sending ./test/doc1 to slave number 2
Stopping slaves from working..
Process 2683 finished
Process 2684 finished
Process 2685 finished
Finishing application process...
==2682==
==2682== HEAP SUMMARY:
==2682==    in use at exit: 0 bytes in 0 blocks
==2682==   total heap usage: 49 allocs, 49 frees, 107,208 bytes allocated
==2682==
==2682== All heap blocks were freed -- no leaks are possible
==2682==
==2682== For counts of detected and suppressed errors, rerun with: -v
==2682== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Como se observa en la imagen, al ejecutarse el programa no se detectaron “leaks” de memoria ni tampoco hay errores. Esto lo logramos luego de aplicar varias liberaciones de memoria en los lugares donde era correspondiente (donde se había alocado memoria dinámicamente mediante *malloc* o alguna otra función) y corregir si se detectaron zonas de memoria mal reservadas.

Para el caso en el cual la vista es llamada, cuando la aplicación me da la opción de ejecutarla, se ingresó el comando `valgrind --leak-check=full ./view PID` en una nueva terminal. El resultado fue el mismo que en el caso anterior tanto para la aplicación como para la vista.

Para poder analizar el ejecutable llamado *slave* también se utilizó Valgrind. Durante el uso de *application* se ejecuta a *slave* pero Valgrind no da una devolución sobre el mismo a menos que hagamos un análisis específico sobre este. Tampoco podemos usar Valgrind directamente sobre el ejecutable *slave*, porque este funciona dependiendo de *application*.

Para poder analizarlo se aprovecharon los tests realizados en *slave.c*. Se ejecutó el comando `valgrind --leak-check=full ./slave test` en la terminal, siendo `test` en este caso, no el directorio que contiene los archivos a procesar, sino un parámetro que indica que deseamos que se ejecute el test de *slave*. Como se observa en la imagen, al ejecutarse el programa no se detectaron “leaks” de memoria ni tampoco hay errores.

```
ab86a10a81603c6efd11ac0275753023 test/Directorio2/archivo5
94723b2180cca6cb84ed7abd63d24497 test/Directorio2/archivo7
7aac136b8b5473694fb0c2f1bffa4bfaf test/Directorio2/archivo4
b166315a180d3682d8a074aa874b3bf9 test/doc1

If all hashes are equal, then the test is succesful

Select an option from 1 to 3 to start a test
1 - Process a single hash
2 - Process all hashes from directory
3 - Exit

Option selected: 3

==3384==
==3384== HEAP SUMMARY:
==3384==   in use at exit: 0 bytes in 0 blocks
==3384== total heap usage: 63 allocs, 63 frees, 200,128 bytes allocated
==3384==
==3384== All heap blocks were freed -- no leaks are possible
==3384==
==3384== For counts of detected and suppressed errors, rerun with: -v
==3384== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Tests y demostraciones:

En cuanto a los tests, decidimos que todo el programa en conjunto (aplicación, esclavos y vista) tiene 5 fases “cores” que deberían tener su adecuado testeo:

1. Búsqueda recursiva de los archivos a los que tenemos que calcular el hash.
2. Conexión bidireccional entre un esclavo y la aplicación.
3. Cálculo del hash de un archivo mediante la función md5sum.
4. Redistribución de tareas a esclavos que ya terminaron sus pedidos.
5. Correcto funcionamiento de la vista, específicamente el uso de shared memory y semáforos.

Para el cálculo del hash se pide el path un archivo, se calcula el hash md5 a través del esclavo y además se ejecuta el comando md5sum con el path. Si ambos hashes son iguales, entonces el test es satisfactorio ya que el hash calculado por el esclavo es igual al calculado por el comando md5sum.

Para la búsqueda recursiva de archivos se diseñó un test parecido al anterior, con la sutil diferencia de que, en vez de recibir el path de un archivo, recibe un directorio. Entonces ahora el esclavo debe buscar primero todos los archivos y calcular el hash. Finalmente se ejecuta el comando md5sum sobre todos los archivos. Si todos los hashes calculados por el esclavo son iguales a los calculados por el comando, entonces el test es satisfactorio.

Para la conexión bidireccional entre un esclavo y la aplicación, se pide una frase cualquiera la cual se almacena en un vector, luego esta frase es enviada al esclavo y éste la devuelve. Si la frase enviada por el esclavo es igual a la guardada en el vector, el test es satisfactorio.

Para los últimos 2 testeos, no encontramos una manera óptima de probar su funcionamiento, por lo que decidimos mostrarlo/explicarlo en este informe a través de la terminal para mostrar su correcto funcionamiento:

Para la redistribución de tareas a esclavos que ya terminaron sus pedidos fue necesario utilizar un boolean *isWorking* el cual comienza siendo false, una vez que se le asignan tareas al esclavo se hace true y cuando se lee del pipe del esclavo una respuesta vuelve a ser false.

En la aplicación se cicla mientras queden tareas sin finalizar, se recorre el vector esclavos y se fija si en algún pipe hay algo para leer. Si no hay nada para leer de ese pipe, intentará leer el pipe siguiente.

La siguiente imagen muestra un caso donde el esclavo 2 termina sus tareas antes que los esclavos 0 y 1 y por ende las siguientes tareas a asignar son enviadas a este

```
Starting application process...Press ENTER to continue!
Creating order queue...
Fetching files...
12 files were fetched!
Creating 3 slaves..
Sending ./test/Directorio1/archivo1 to slave number 0
Sending ./test/doc2 to slave number 0
Sending ./test/chau to slave number 1
Sending ./test/hola to slave number 1
Sending ./test/prueba to slave number 2
Sending ./test/Directorio2/archivo2 to slave number 2
Sending ./test/Directorio2/archivo6 to slave number 2
Sending ./test/Directorio2/archivo3 to slave number 2
Sending ./test/Directorio2/archivo5 to slave number 0
Sending ./test/Directorio2/archivo7 to slave number 0
Sending ./test/Directorio2/archivo4 to slave number 1
Sending ./test/doc1 to slave number 1
Stopping slaves from working..
Process 2532 finished
Process 2533 finished
Process 2534 finished
Finishing application process...
```

esclavo.

Se decidió que el proceso aplicación muestre información acerca de la asignación de tareas para poder hacer más evidente (de forma visual) el hecho de que los distintos esclavos tardan diferentes cantidades de tiempo en procesar los archivos.

Para comprobar el correcto funcionamiento de los semáforos se optó por una comprobación simple. Primero se colocaron tanto en *application.c* como en *view.c* dos puts y en el caso de *application.c* se introdujo un while(1) como muestran las siguientes imágenes:

```
hashes[pointer] = malloc(100 * sizeof(char));
strcpy(hashes[pointer], buff);
modifySemaphore(-1,id_sem);
puts("Entro el padre, el hijo espera");
while(1);
strcat(shm, buff);
strcat(shm, "|");
modifySemaphore(1,id_sem);
puts("Salio el padre");
curr = buff;
```



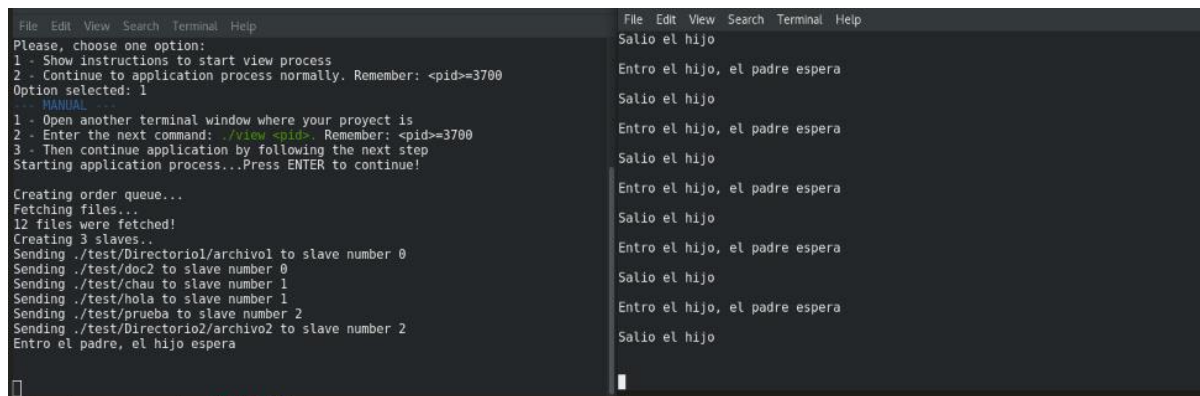
```
while(!done){
    modifySemaphore(-1,id_sem);
    puts("Entro el hijo, el padre espera");

    if(*s == '?')
        done = 1;
    else if(*s != '\0'){
        while(*s != '|'){
            printf("%c", *s);
            s++;
        }
        puts("");
        s++;
    }

    modifySemaphore(1,id_sem);
    puts("Salio el hijo");
}
```

Cuando la vista empezó a correr, imprimía esas cadenas continuamente hasta que la aplicación cambió el valor del semáforo.

Para asegurarnos de que si la aplicación accedió a la memoria compartida, la vista no puede hacerlo simultáneamente incluimos un loop infinito para poder hacerlo más evidente de forma visual. La siguiente imagen muestra dos terminales, en el lado izquierdo se encuentra la aplicación colgada en el loop infinito, en el lado derecho se encuentra la vista esperando que el semáforo modifique su valor para poder acceder a la memoria



```
File Edit View Search Terminal Help
Please, choose one option:
1 - Show instructions to start view process
2 - Continue to application process normally. Remember: <pid>=3700
Option selected: 1
... MANUAL ...
1 - Open another terminal window where your proyect is
2 - Enter the next command: ./view <pid>. Remember: <pid>=3700
3 - Then continue application by following the next step
Starting application process...Press ENTER to continue!

Creating order queue...
Fetching files...
12 files were fetched!
Creating 3 slaves..
Sending ./test/Directorio1/archivo1 to slave number 0
Sending ./test/doc2 to slave number 0
Sending ./test/chau to slave number 1
Sending ./test/hola to slave number 1
Sending ./test/prueba to slave number 2
Sending ./test/Directorio2/archivo2 to slave number 2
Entro el padre, el hijo espera
```

```
File Edit View Search Terminal Help
Salio el hijo
Entro el hijo, el padre espera
Salio el hijo
Entro el hijo, el padre espera
Salio el hijo
Entro el hijo, el padre espera
Salio el hijo
Entro el hijo, el padre espera
Salio el hijo
Entro el hijo, el padre espera
Salio el hijo
```

compartida.

De esta manera se comprobaba si se iban intercalando las impresiones de ambos procesos o no. Al comprobarse que no habia intercalación entonces se demostró que el semáforo funcionaba de forma correcta.

Bibliografía

- **Memoria compartida en C para Linux**
http://www.chuidiang.org/clinix/ipcs/mem_comp.php
- **Memoria compartida en Linux** <https://snatverk.blogspot.com.ar/2010/09/memoria-compartida-en-linux.html>
- **Semáforos**
<http://www.chuidiang.org/clinix/ipcs/semaforo.php>