

INFORME

S.O. - SISTEMAS OPERATIVOS

Trabajo Práctico 1

Inter Process Communication

- Grupo 10: Matías Heimann (57503), Johnathan Katan (56653), Lóránt Mikolás (57347) y Gabriel Silvatici(53122).
- Vencimiento de la entrega: 04/04/2018 (23:59).

Introducción

El objetivo de este trabajo práctico fue implementar diferentes tipos de IPC en un sistema POSIX. Concretamente la tarea realizada consiste en repartir un grupo de archivos por parte de un proceso llamado aplicación (`applicationProcess`) a procesos esclavos (`slaveProcess`) que deben retornar los nombres de los archivos acompañados de los hashes de los mismos. El hash es obtenido mediante el comando `md5sum`. Por último, el proceso aplicación guarda los datos recibidos en un buffer compartido con un proceso vista (`viewProcess`). El proceso vista cuando es ejecutado muestra en salida estándar los datos guardados en el buffer.

En este informe se explicarán las estructuras e IPCs utilizados, problemas encontrados en la implementación y sus respectivas soluciones. También se detallarán la bibliografía utilizada, fragmentos de código utilizados como referencia y las instrucciones de instalación y ejecución.

Implementación (IPCs, funciones y estructuras utilizadas)

Creación de los procesos esclavos:

El objetivo de crear procesos esclavos a partir del proceso aplicación es distribuir la carga de calcular los hashes de los archivos. De esta manera, se tienen múltiples procesos esclavos procesando los archivos y calculando sus hashes.

La cantidad de procesos esclavos creados se decide en proporción a la cantidad de archivos a procesar, se crea una cantidad de procesos esclavos igual a un quinto de la cantidad total de archivos. Es decir, se busca que haya menos esclavos que archivos, de manera que cada esclavo pueda procesar múltiples archivos. Esta proporción se decidió arbitrariamente.

También, para un mejor manejo de los procesos esclavos a partir del proceso *aplicación*, se decidió crear una estructura llamada *slaveADT*, en la cual se guarda la información necesaria para poder comunicarse con ese proceso. Aprovechando esta estructura, se pudo manejar prolijamente la comunicación con cada proceso esclavo.

Comunicación entre el proceso aplicación y los procesos esclavos:

Con respecto a la comunicación entre el proceso aplicación y los procesos esclavos se usaron dos pipes por esclavo. Se tiene un pipe (unidireccional) de la aplicación a un esclavo y otro (también unidireccional) desde ese esclavo a la aplicación.

En un primer momento, se encontró un desafío al intentar comunicarse con el proceso aplicación desde el proceso esclavo a través de los pipes que se crearon. La explicación y resolución de este desafío están explicados en la sección “*Problemas encontrados y Soluciones*”, bajo el título “*Comunicándose con el proceso aplicación desde un proceso esclavo*”.

Cálculo de los hashes en los procesos esclavos:

Cada proceso esclavo es responsable de calcular los hashes de los archivos que fueron recibidos, y luego enviárselos al proceso aplicación. Para resolver el cálculo del hash de cada archivo recibido, se optó por utilizar el comando *md5sum* de la terminal.

Para ejecutar el comando *md5sum* desde el proceso esclavo, se utilizaron las funciones *fork* y *exec*, pasándole a *exec* como argumento el programa a ejecutar y su respectivo argumento (nombre del archivo).

Se encontró un desafío al intentar obtener el resultado del comando *md5sum* y guardarlo en una variable en el proceso esclavo de forma prolija. La explicación y resolución de este desafío están explicados en la sección “*Problemas encontrados y Soluciones*” del informe, bajo el título “*Cómo obtener el resultado del comando md5sum correctamente desde un proceso esclavo*”.

Memoria compartida entre proceso vista y proceso aplicación:

En cuanto al buffer compartido entre el proceso vista y el proceso aplicación se usaron las funciones *ftok(...)*, *shmget(...)* y *shmat(...)*, *shmdt(...)* y *shmctl(...)* (de System V IPCs).

En primer lugar con *ftok(...)* se obtiene el identificador del IPC a partir de un *path*. En segundo lugar, se retorna un segmento de memoria con su respectivo identificador con *shmget(...)*. En tercer lugar, *shmat(...)* permite realizar el “mapeo” de la memoria al espacio del proceso que la llama.

Por último los comandos *shmdt(...)* y *shmctl(...)* se utilizaron para liberar el segmento de memoria pedido.

Las fuentes que utilizamos para entender este tipo de procedimiento (además de man y el material proveído por la cátedra) se encuentran en la bibliografía. Además se debe aclarar que en la primera posición de memoria se guardaron la cantidad de direcciones ocupadas.

Para finalizar, notamos que hay una limitación en la implementación que se realizó debido a que hay una cantidad de memoria fija en el buffer, por lo que con un número suficientemente grande de archivos la memoria no sería suficiente.

Semáforo para acceder la memoria compartida:

De manera análoga el semáforo se obtuvo con la misma *key* que la memoria compartida utilizando *semget(...)*. Después se inicializó el semáforo con *semctl(...)* (que requiere de la union *semun* para pasarle los argumentos). Se trata de un semáforo que sólo deja el pasaje de 1 proceso a la zona crítica. En este caso será el proceso vista (para leer) o el proceso aplicación (para leer y escribir).

Guardando los Hashes calculados y nombres de archivos a un archivo en disco al finalizar:

El proceso aplicación, al finalizar, debe escribir en disco los resultados obtenidos. Para realizar esto usamos la función *fwrite*, creando un nuevo archivo llamado “*SavedHashes.txt*”, en el cual se guarda la información en el siguiente formato:
<nombre de archivo> : <Hash del archivo>

Problemas encontrados y Soluciones

Cómo obtener el resultado del comando *md5sum* correctamente desde un proceso esclavo:

Un desafío encontrado fue que el comando *md5sum* imprime en salida estándar el resultado calculado, pero se deseaba guardar ese resultado en alguna variable del proceso esclavo. Para resolver este problema, se ejecutó el comando *md5sum* en un proceso distinto (el cual se generó haciendo *fork* desde el proceso esclavo), y luego, mediante un pipe, se conectó la salida estándar del proceso donde se ejecutó *md5sum* con la entrada estándar del proceso esclavo. De esta manera, el proceso esclavo y el proceso que ejecutó *md5sum* quedaron conectados mediante un pipe, por donde el proceso esclavo recibe el hash calculado y enviado por *md5sum*.

Comunicándose con el proceso aplicación desde un proceso esclavo:

Una vez ejecutado un proceso esclavo, es importante poder comunicarse con el proceso aplicación para enviarle los hashes calculados. Esto se hizo mediante uno de los pipes creados desde el proceso aplicación.

El problema encontrado inicialmente fue que al ejecutar el proceso esclavo, no se tenían los file descriptors necesarios para comunicarse mediante los pipes. Para resolver esto, al ejecutar los procesos esclavos, se les envió por argumentos del main los file descriptors que deberían usar para comunicarse mediante los pipes. De esta manera, se pudo leer los archivos recibidos y enviar los hashes calculados correctamente.

Cómo saber la cantidad de archivos a procesar y si son realmentes files regulares:

Para resolver este problema decidimos encolar los archivos en una cola (únicamente si se trataba de un *file regular*)

Aunque no es la manera más eficiente priorizamos la prolijidad del código en este caso. Además se tiene conocimiento de la cantidad exacta de archivos a procesar.

Para finalizar para determinar si el file era regular utilizamos la función `stat` y la macro `S_ISREG`.

Cómo indicarle al proceso esclavo que debe finalizar:

Cuando todos los archivos son procesados y el programa debe terminar, se debe finalizar primero los procesos esclavos, ya que estos fueron creados por el proceso aplicación mediante la función `fork()`. De esta manera se liberan apropiadamente los recursos del sistema, sin que queden procesos huérfanos. Una primera solución a este problema fue enviar una señal `SIGKILL` a los procesos esclavos mediante la función `kill`, para interrumpir al proceso esclavo y terminarlo. El problema de esto es que no se estaban liberando correctamente los recursos del proceso esclavo, como las estructuras utilizadas, ya que se lo interrumpe abruptamente. Para resolver este problema, se optó por enviar una señal `SIGTERM` con la función `kill`, que le permite al proceso esclavo atrapar esa señal, ejecutar un handler para esa señal, y liberar los recursos del proceso antes de terminar.

Para asignarle al proceso esclavo un handler que maneje esta señal, se consultó el link que está en el inciso 9. en la sección “*Bibliografía*” del informe.

Cómo leer el nombre del archivo desde el pipe de lectura cuando el proceso aplicación se comunica con el esclavo para pasarle los nombres de los archivos:

Al no tener la posibilidad de pasar la longitud como parámetro y la función `read` utiliza la longitud que se leerá en esa operación de lectura, en el momento de la escritura en el pipe, decidimos poner un ‘\n’ al final de cada nombre de archivo escrito. Las lecturas se hacen de a un carácter hasta que se llega al salto de línea.

Cómo indicarle al proceso vista que debe finalizar:

El proceso vista finaliza su ejecución cuando el proceso aplicación le “indica” que debe terminar. Esto se hace mediante la memoria compartida. En el segundo *int* se dejó un flag que se inicializa en cero y que el proceso aplicación modifica poniéndolo en 1 para indicar que el proceso vista debe finalizar.

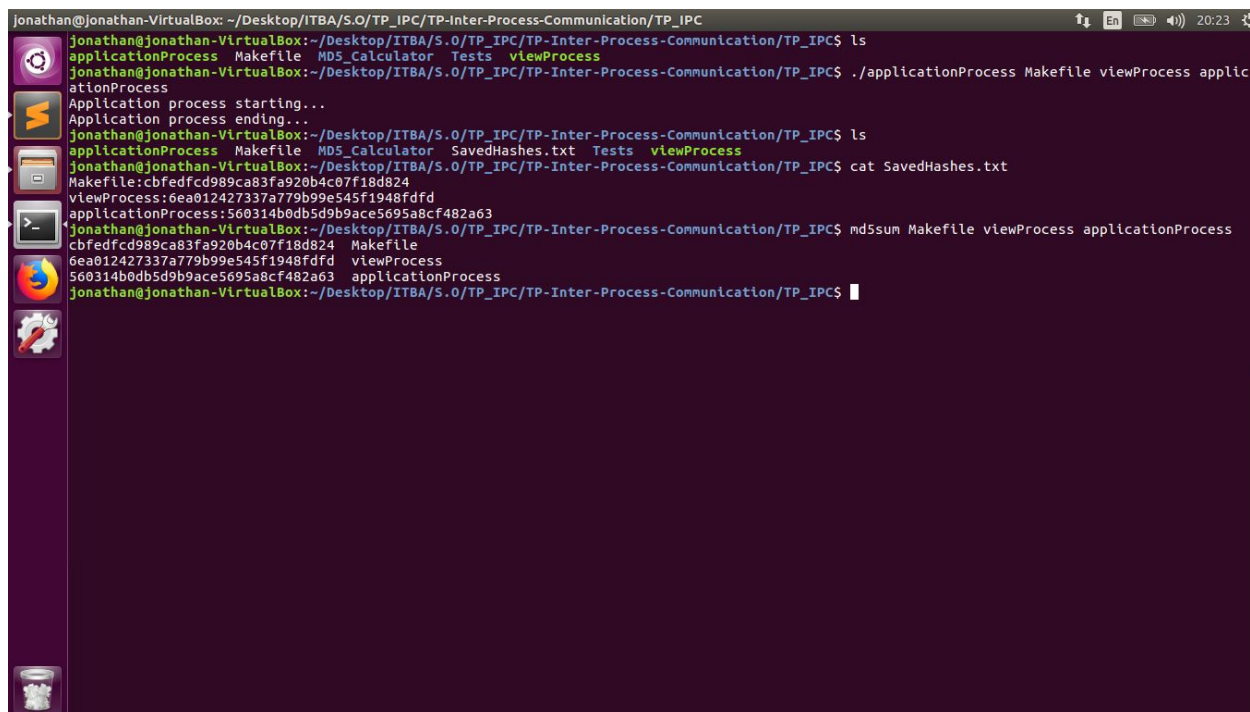
Resultados

Valgrind:

Pudimos correr valgrind sin errores tanto en el proceso vista como en el proceso aplicación.

Hashes obtenidos al ejecutar el proceso aplicación:

En la imagen 1 (abajo) se ven una serie de comandos que demuestran el correcto funcionamiento del proceso aplicación. Primero, se ejecuta pasándole tres archivos, *Makefile*, *viewProcess* y *applicationProcess*. Luego, se ve que se genera el archivo *SavedHashes.txt*. Al abrirlo, se ve que están listados los nombres de los archivos con sus respectivos hashes. Luego, para verificar que son los hashes correctos, se ejecuta el comando *md5sum* en cada uno de los archivos, y se obtienen los mismos hashes obtenidos en el archivo *savedHashes.txt*.



```

jonathan@jonathan-VirtualBox: ~/Desktop/ITBA/S.O/TP_IPC/TP-Inter-Process-Communication/TP_IPC
jonathan@jonathan-VirtualBox:~/Desktop/ITBA/S.O/TP_IPC/TP-Inter-Process-Communication/TP_IPC$ ls
applicationProcess Makefile MD5_Calculator Tests viewProcess
jonathan@jonathan-VirtualBox:~/Desktop/ITBA/S.O/TP_IPC/TP-Inter-Process-Communication/TP_IPC$ ./applicationProcess Makefile viewProcess applic
ationProcess
Application process starting...
Application process ending...
jonathan@jonathan-VirtualBox:~/Desktop/ITBA/S.O/TP_IPC/TP-Inter-Process-Communication/TP_IPC$ ls
applicationProcess Makefile MD5_Calculator SavedHashes.txt Tests viewProcess
jonathan@jonathan-VirtualBox:~/Desktop/ITBA/S.O/TP_IPC/TP-Inter-Process-Communication/TP_IPC$ cat SavedHashes.txt
Makefile:cbfedfcd989ca83fa920b4c07f18d824
viewProcess:6ea012427337a779b99e545f1948dfd
applicationProcess:560314b0db5d9b9ace5695a8cf482a63
jonathan@jonathan-VirtualBox:~/Desktop/ITBA/S.O/TP_IPC/TP-Inter-Process-Communication/TP_IPC$ md5sum Makefile viewProcess applicationProcess
cbfedfcd989ca83fa920b4c07f18d824 Makefile
6ea012427337a779b99e545f1948dfd viewProcess
560314b0db5d9b9ace5695a8cf482a63 applicationProcess
jonathan@jonathan-VirtualBox:~/Desktop/ITBA/S.O/TP_IPC/TP-Inter-Process-Communication/TP_IPC$

```

Imagen 1

Tiempos calculados al ejecutar el proceso aplicación:

Se midió el tiempo transcurrido a lo largo de la vida del proceso en microsegundos, en función de la cantidad de archivos a procesar, se observa que a medida que se agregan archivos a procesar el tiempo no avanza linealmente (como se esperaba), sino avanza logarítmicamente, aprovechando la concurrencia de procesos.

Considerando que un archivo tiene un nombre con longitud no mayor que 30 caracteres, el programa aceptará hasta 250 archivos, debido a la memoria compartida.

(Promedio de 5 ejecuciones, en microsegundos)

<u>Cantidad de archivos</u>	<u>Tiempo en microsegundos</u>
1	850
2	982
3	1161
4	1236
5	1405
10	2052
20	2888
50	4150
100	6806

Conclusión

En conclusión, el trabajo práctico nos sirvió para entender el funcionamiento de diferentes formas de IPC, los beneficios de TDD y valgrind.

Además notamos las implicancias del uso correcto de semáforos y de una memoria compartida, al igual que la importancia de distribuir la carga de procesamiento entre diferentes procesos. Esta importancia se ve fuertemente reflejada al observar la tabla de comparación de tiempos en la sección “*Resultados*” del informe, donde el tiempo de ejecución, a medida que el programa se ejecuta con más archivos, crece logarítmicamente. Esto se debe a la concurrencia de procesos, como se mencionó previamente. Si se hubieran calculado los hashes de muchos archivos con un solo proceso, este tiempo hubiera crecido linealmente.

Una mejora posible al trabajo sería agregar memoria compartida que se expanda a medida que se necesite más, y también realizar más tests para hacer más robusto el trabajo.

Por último vale la pena aclarar que se incorporaron a nuestra rutina de programación buenos hábitos como el uso de man (no google) y valgrind.

Instrucciones para ejecución

Se debe acceder al directorio TP_IPC, el cual está dentro de la carpeta TP-Inter-Process-Communication. Una vez allí, se deben ejecutar en la terminal primero el comando `make clean` y luego el comando `make all`. Se debe tener en cuenta que los ejecutables son creados en la carpeta TP_IPC por el Makefile mientras que los ejecutables de los tests estarán dentro del directorio Tests.

Para ejecutar el proceso aplicación se debe ejecutar el comando `./applicationProcess`. Luego en caso de querer pasar parámetros se deben pasar espaciados y con un espacio previo al comando de ejecución del proceso aplicación. Un ejemplo sería `./applicationProcess file1 file2 file3`. En el ejemplo mencionado se pasan como parámetros `file1`, `file2` y `file3`, es decir, estos serán los archivos a procesar.

Para ejecutar un proceso vista se debe ejecutar desde una terminal distinta el comando `./viewProcess` y luego se especifica el *pid* del proceso aplicación. Por ejemplo `./viewProcess 615`. Se ejecutará el proceso vista y mostrará los hashes de los archivos que fueron procesados por el proceso aplicación que tiene el identificador 615.

Observación: en caso de pasar como parámetro la ruta relativa de un archivo que no existe o de algo que no es archivo regular este no será procesado.

Bibliografía

A continuación se encuentra la bibliografía utilizada para realizar el trabajo práctico. Se debe tener en cuenta que también fueron usados elementos teóricos y prácticos enseñados en clase y proveídos por la cátedra como parte del material que se encuentra en el campus.

1. <https://users.cs.cf.ac.uk/Dave.Marshall/C/node27.html>
2. <http://www.cse.psu.edu/~deh25/cmpsc473/notes/OSC/Processes/shm.html>
3. https://www.ibm.com/support/knowledgecenter/en/ssw_i5_54/apiref/apiexusmem.htm
4. <https://www.tldp.org/LDP/lpg/node53.html>
5. https://www.tutorialspoint.com/cprogramming/c_unions.htm
6. <http://pubs.opengroup.org/onlinepubs/007904875/functions/semop.html>
7. <https://stackoverflow.com/questions/39207480/process-communication-with-semaphore-in-c> (no consideramos que stackoverflow sea una fuente confiable de información pero nos ayudó a entender la dinámica de los semáforos)
8. <https://cse.yeditepe.edu.tr/~sbaydere/fall2010/cse331/files/SystemVIPC.pdf>
9. <https://airtower.wordpress.com/2010/06/16/catch-sigterm-exit-gracefully/>