Trabajo Práctico N°1

Grupo 16

Sistemas Operativos - 72.11

Arancibia Carabajal, Nicolás Guillermo	64481
Birsa, Juan Pablo	64500
Borda, Tomás	64517

Instituto Tecnológico de Buenos Aires

Fecha de entrega: 16/09/2024

Docentes: Ariel Godio

Alejo Ezequiel Aquili

Fernando Gleiser Flores

Guido Matias Mogni

1. Introducción

Durante el presente informe se detalla el desarrollo del trabajo práctico. El mismo abarca el uso de los diferentes mecanismos de IPC presentes en un sistema POSIX mediante la comunicación entre diversos procesos con el fin de obtener el md5 de múltiples archivos. Para eso se hizo uso de comunicación mediante pipes anónimos y memoria compartida, así como se utilizaron semáforos con nombre para sincronizar la información en esta última.

2. Desarrollo del proyecto

2.1 Programas desarrollados e IPCs utilizados

Se realizaron tres programas con funciones definidas:

- Un programa esclavo (slave) que se encarga de procesar los archivos recibidos por entrada estándar mediante el uso del programa *'md5sum'*.
- Un programa aplicación (md5) que se encarga de crear procesos hijos que van a ejecutar el programa esclavo y van a recibir por parte del proceso aplicación los archivos a procesar hasta agotarlos. Además, se encarga de escribir los resultados de dichos hijos por orden de llegada en un archivo llamado *'results.txt'* y en una zona de memoria compartida
- Un programa vista que se encarga de conectarse a la zona de memoria creada por el proceso aplicación y leer e imprimir en pantalla la información.

Para la comunicación entre el proceso aplicación y los hijos se hizo uso de *pipes anónimos*, de forma tal que el mismo tuviese un par de pipes por cada hijo, utilizando uno para envíar la información y otro para recibir los resultados. Este mecanismo de IPC permite que el proceso aplicación cree hijos y redireccione tanto su salida como entrada estándar a los pipes correspondientes, de forma tal que el proceso esclavo pueda ejecutar normalmente sin preocuparse por el piping, permitiendo también su ejecución de forma independiente. Se observa a continuación el funcionamiento del sistema de comunicación descrito anteriormente para un proceso esclavo cualquiera:

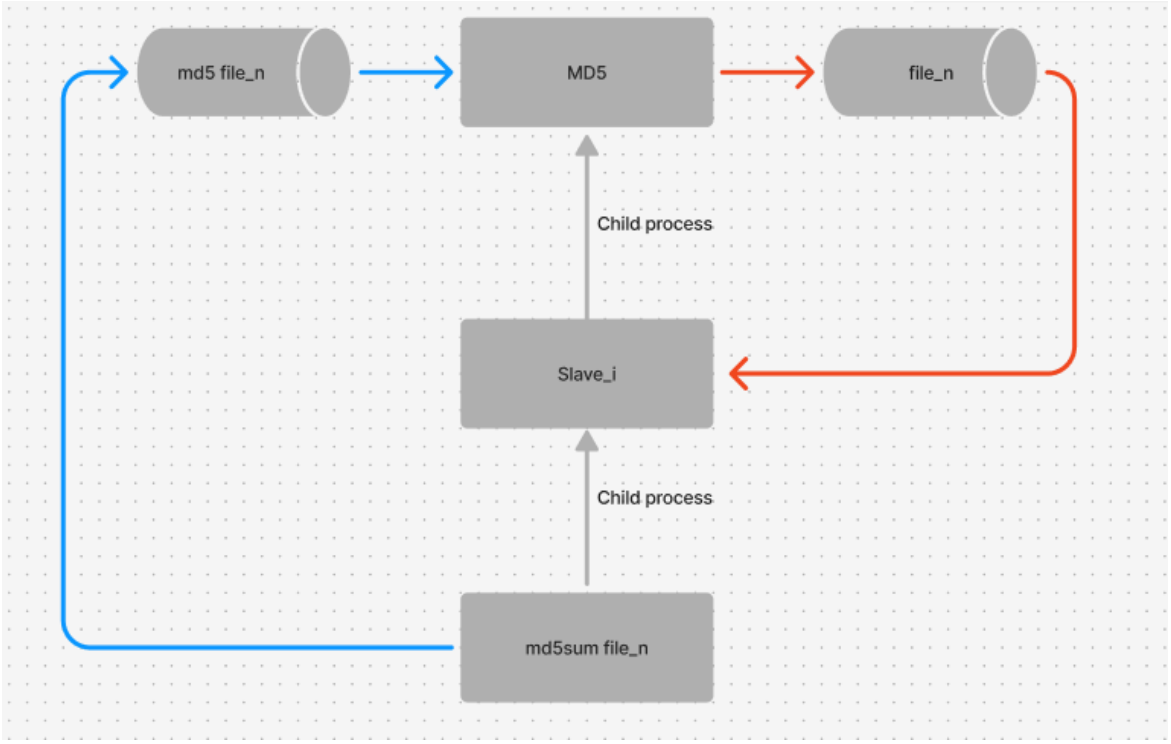

Figura 1: Funcionamiento del sistema de comunicación entre el proceso aplicación (MD5) y un esclavo cualquiera (slave_i). Se observa el envío de información al hijo mediante un pipe y la recepción de la respuesta mediante otro. Por claridad del diagrama, se eliminó el extremo w-end del slave i.

Por otro lado, la comunicación entre el proceso aplicación y el vista se realizó por medio de una *shared memory*, y mediante el uso de semáforos para sincronizar la lectura de los datos escritos en ella. Al ser el proceso aplicación quien la crea, el mismo imprime por salida estándar la información necesaria para que el proceso vista pueda conectarse. En el primer lugar de la shared memory se encuentra la cantidad de argumentos a leer para que el vista sepa cuándo finaliza el buffer, mientras que luego se encuentra un buffer de estructuras que poseen la información necesaria a imprimir: PID del esclavo que procesó el archivo, así como md5 y nombre de este último. Para esperar a que el proceso aplicación previamente escriba los datos en la memoria, el proceso vista debe esperar la indicación del mismo por medio de un semáforo. A continuación se grafica el proceso descrito anteriormente.

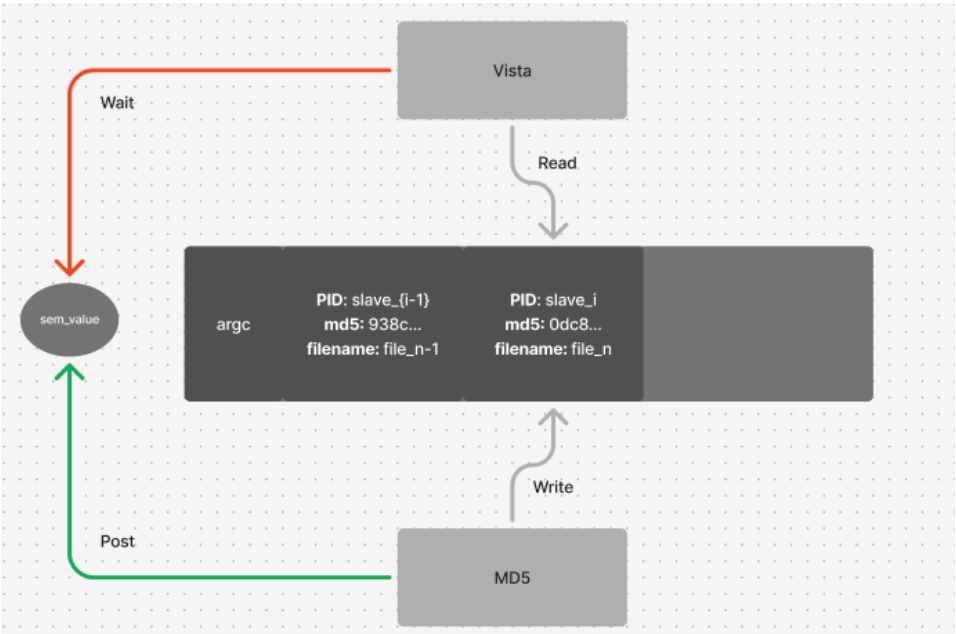

Figura 2: Se expone el funcionamiento de la comunicación entre el proceso vista y el aplicación (md5). El segmento del medio indica el espacio de memoria compartida, donde el proceso md5 escribe la información. Además, se observa el semáforo que se utiliza para sincronizar la lectura de la misma.

2.2 Inconvenientes durante el desarrollo

Durante el desarrollo del trabajo nos encontramos con diversos problemas. A continuación se enumeran los mismos, junto a una breve descripción y su resolución:

+ **Utilización de semáforo**: Se presentaron dos alternativas para sincronizar la información entre el proceso aplicación y el vista, estas fueron utilizar el mismo nombre que permitía conectarse a la shared memory para abrir un semáforo o que éste estuviera dentro de la memoria compartida, es decir, dentro de la memoria habría un segmento que sería de tipo *sem_t* y el puntero a esa zona de memoria (Obtenido mediante la función *mmap*) se inicializaría como semáforo. No obstante, se optó por la primera alternativa con el fin de no tener problemas al escribir la información en la memoria compartida, ya que podría accidentalmente pisar la zona asignada al semáforo y ocasionar que la lectura de los datos sea errónea.

+ **Envío de archivos por el pipe**: Cuando se cambió la cantidad inicial de archivos a por lo menos más de 1, se encontraba que el slave por algún motivo para ciertos archivos no recibía el path completo, dando lugar a la falla del programa. Con la ayuda de strace se comprobó que el path al archivo se enviaba correctamente desde md5 pero en el slave se podía terminar leyendo más de un path. El problema ocurría porque *read* intentaba leer la cantidad máxima de caracteres permitidos para un path, y si el mismo era más corto, seguía leyendo hasta completar el máximo de bytes, provocando que las rutas quedaran acortadas. Para solucionar este problema, en el proceso *md5* se completó el string de la ruta con caracteres nulos hasta alcanzar la

cantidad máxima de bytes que el *slave* esperaba leer. Esto evitó la lectura incompleta o accidental de varios paths en una sola operación de lectura.

+ **Tamaño del buffer desde el vista:** El proceso vista, para obtener el puntero a la zona de memoria compartida, debe hacer uso de la función mmap, la cual solicita el tamaño de memoria al que se va a quedar acceder con el puntero que la misma devuelve. Para esto, es necesario conocer de antemano la longitud del buffer. Es por ello que se decidió escribir en la primera posición de la memoria compartida este valor (Ver Figura 2), de tal forma que lo primero que hace el proceso md5 al abrir la zona de memoria es escribir este valor, permitiendo que el vista al conectarse pueda leerlo y almacenarlo para utilizarlo como parámetro en mmap y como condición de corte del proceso.

+ **Separación de índices de archivos:** Desde el proceso *'slave'* se reciben los nombres de los archivos a procesar por *'md5sum'*. Estos mismos pueden ser obtenidos a partir de un IPC, ya sea un pipe anónimo, o a su vez, a partir de la terminal, en el caso de ejecutar directamente el archivo compilado *'slave'*. Por esto mismo, dependiendo de la forma en la que el proceso fue ejecutado, debe "separar" los archivos de distinta forma, con el objetivo final de procesarlos uno a uno.

3. Instrucciones de compilación y ejecución

3.1 Compilación

Para la compilación del trabajo se requiere de un contenedor de docker. La imágen que utilizaremos podemos descargarla con ***docker pull agodio/itba-so-multi-platform:3.0***. Corriendo el comando *docker images* deberíamos poder ser capaces de ver la imágen una vez descargada. Para ejecutar el contenedor de docker utilizaremos el siguiente comando:

docker run -v "${PWD}:/root" --privileged -ti agodio/itba-so-multi-platform:3.0

Una vez dentro podemos acceder a los archivos del proyecto ejecutando ***cd root***.

Utilizaremos makefile para compilar los archivos fuente (Los cuales se encuentran en la carpeta src). Para ello podemos ejecutar ***make*** o ***make all***. Una vez compilados y linkeditados, deberíamos ser capaces de ver los ejecutables en el actual directorio. Para borrar los archivos ejecutables generados podemos utilizar ***make clean***.

3.2 Ejecución

Todos los archivos son ejecutables, tanto slave como vista y md5. Utilizaremos los mismos para computar el md5 de diferentes archivos:

Ejecución mediante slave

Únicamente corriendo ***./slave*** seremos capaces de ejecutar dicho programa. A continuación, se quedará esperando a que ingresemos por entrada estándar el path de los archivos a procesar. Podemos ingresar uno o varios separados entre sí por un espacio.
Al finalizar, se imprimirá en pantalla el resultado de md5sum por cada uno de los archivos.

Ejecución de md5 y vista mediante piping

Ejecutando ***./md5 <files> | ./vista***, donde en <*files*> debemos indicar el path de los archivos a procesar, podremos conectar ambos procesos y, a medida que cada uno de los archivos sea procesado, veremos en pantalla el resultado, junto al PID del proceso que se encargó de ello.
Además, al finalizar la ejecución, veremos en el directorio actual un archivo *'results.txt'* con la información previamente visualizada en pantalla.

Ejecución de md5 y vista separados

Para esto tenemos dos opciones, se puede abrir una nueva terminal y ejecutar ***docker ps***. Allí deberíamos ver un contenedor llamado SO que estamos ejecutando actualmente, junto al ID del mismo. Para ejecutar un nuevo contenedor conectado a éste corremos el siguiente comando:

docker exec -ti <container ID> bash

Donde en container ID indicaremos el ID de nuestro contenedor SO obtenido anteriormente con ***docker ps***. Luego podemos movernos a root y seguir con la ejecución.

Ahora desde una de las terminales activas ejecutaremos ***./md5 <files>*** con los archivos a procesar y veremos que sale un mensaje en pantalla que dice *sharedMemory*.
En la otra terminal ejecutamos ***./vista sharedMemory*** y, si se hace antes de que md5 termine de ejecutar, deberíamos ver que el programa vista se queda esperando hasta que comienza a imprimir la información que md5 va escribiendo en la memoria.

Adicionalmente, al finalizar la ejecución también encontraremos el archivo *'results.txt'* en nuestro directorio.

La otra opción es correr el md5 en background, para esto utilizar el comando ***./md5 <files> &*** y una vez que se imprimió el nombre de la shared memory se puede correr vista ejecutando ***./vista sharedMemory*** siempre y cuando el proceso md5 no haya terminado.

4. Limitaciones

+ La cantidad máxima de hijos, dada la forma en que son creados en md5 es 255. Como el select soporta files descriptor cuyo número esté dentro del intervalo [0,1024), dado que por cada hijo se usan 4 files descriptor, tener más de los hijos establecidos llevaría a una falla del select y por consecuencia del programa

+ Hay una cantidad máxima de archivos, esto dado que la línea de comandos tiene un tamaño máximo de bytes que soporta, dentro del contenedor de docker y con la imagen provista con la cátedra este límite es aproximadamente 2 MB, por ende si los archivos a mandar (el tamaño de cada string que representa al path) en su conjunto superan esa cantidad, directamente la shell avisa del problema puesto que no es propio del md5.

+ La cantidad máxima de chars del path de los archivos que se van a procesar no puede ser más de 100, pues para la creación de la shared memory, en específico para determinar su tamaño al no utilizarse un buffer circular y la cantidad de archivos a procesar en cada ejecución es dada por el usuario que ejecuta el programa, es la única manera de asegurarse suficiente lugar para todos los datos, puesto que lo único que podemos saber rápidamente es la cantidad de argumentos enviados no la longitud de cada uno.

+ El programa aplicación asume que todos los parámetros recibidos son archivos, NO explora ni procesa el contenido de directorios.

5. Observaciones

A continuación se listan ciertas observaciones respecto al proyecto:

- Al analizar el *slave* con PVS-Studio se obtiene una nota por utilizar el path de md5sum de forma estática. No obstante, esto fue ignorado para el desarrollo del trabajo, ya que el mismo se compila y ejecuta dentro del contenedor docker y la ubicación de md5sum dentro de él va a ser siempre la descrita en el string utilizado en el proceso *slave*.

- El programa aplicación está desarrollado de tal manera que cada 4 archivos se crea un hijo nuevo (por ejemplo, de 0 a 7 archivos crea un hijo, de 8 a 11 inclusive crea dos, y así) hasta un máximo de 5. Esto permite que no se desperdicien recursos ni se generen hijos innecesariamente cuando la cantidad que haría falta es menor al máximo. Estos valores pueden ser modificados por medio de las constantes **CHILDS_QTY** y **MIN_FILES_PER_SON** presentes en el código fuente de md5.

- La carga inicial de archivos por hijo es el 10% de la cantidad de archivos distribuido por la cantidad de hijos ($\frac{cantArchivos}{cantHijos}$. 10%) o un valor default fijado en 1 en caso de que la división resulte nula. Este funcionamiento se decidió para asegurarse de cumplir con la condición de la consigna que indica que cada hijo debe recibir una cantidad inicial de archivos *considerablemente menor* al total. En caso de querer modificarlo, se puede hacer mediante la constante **INITIAL_LOAD_PERCENTAGE** presente en el md5.

6. Conclusión

El trabajo práctico logró implementar un sistema que utiliza mecanismos de comunicación entre procesos (IPC), dentro de un entorno POSIX, para procesar archivos mediante la generación del proceso *'md5sum'*. No obstante, se emplearon diferentes técnicas, ya sean pipes anónimos, memoria compartida y semáforos con nombre, lo que permitió la correcta coordinación entre los procesos involucrados: el proceso principal *'md5'*, los procesos esclavos encargados de procesar los md5, *'slave'*, y por último, el proceso de visualización de resultados *'vista'*.

Por último, el trabajo requirió un buen manejo de los conceptos de IPC y sincronización de procesos en un entorno POSIX, logrando alcanzar los objetivos propuestos, y superando las limitaciones técnicas encontradas durante el desarrollo.

7. Bibliografía

- Apuntes de clase

- Páginas del manual (sem_overview(7), shm_overview(7), select(2), pipe(2))