Instituto Tecnológico de Buenos Aires

Trabajo Práctico Especial

Primera entrega

Perez de Gracia, Mateo
Quian Blanco, Francisco
Stanfield, Theo

Sistemas Operativos - 72.11

Segundo cuatrimestre 2023 - Grupo 7

Tabla de contenidos

1 Introducción

El trabajo práctico consiste en aprender a utilizar los distintos tipos de IPCs presentes en un sistema POSIX. Para ello se implementará un sistema que distribuirá el cómputo del md5 de múltiples archivos entre varios procesos.

En forma resumida, el sistema se encarga de recibir una serie de archivos y procesarlos para generar un **Hash MD5** por medio del aplicativo `md5sum`. Estos *hashes* generados serán escritos a un archivo de salida y a su vez a un buffer en un espacio de memoria compartida, de la cual otro proceso encargado leerá e imprimirá por pantalla los resultados.

El sistema a desarrollar cuenta de tres procesos que se comunican entre sí:

- **Proceso app:** Dicho proceso es el encargado de recibir los archivos a procesar, ejecutar los *workers* (llamados *slaves* que serán los procesos encargados de generar el resultado deseado de cada archivo recibido) y escribir los resultados tanto en un archivo de salida como en un buffer compartido.
- **Proceso slave:** Es el encargado de recibir los archivos que el **proceso app** le designó y ejecutar el proceso `md5sum` para obtener como resultado el *hash* deseado.
- **Proceso view:** Su única tarea es leer del buffer de memoria compartida con el **proceso app** e imprimir por salida estándar los resultados presentes.

2 Requerimientos

Para el trabajo en cuestión se estarán utilizando los siguientes componentes:

- gitflow plugin
- docker

El plugin de gitflow (explicado en GitFlow) permite manejar los branch de `git` de una forma más sencilla que la habitual, siguiendo la filosofía del modelado de ramas de gitflow.

2.1 GitFlow

Para instalar `gitflow` simplemente se necesita usar el *package manager* de la distribución en la que trabajes:

Para Ubuntu o Debian

```
$ apt install git-flow
```

Para Arch

```
$ yay -S gitflow-avh
```

`gitflow` se basa en la creación de ramas promoviendo una separación entre **desarrollo** (*develop*) y **lanzamientos** (*releases*), utilizando ramificaciones de *features* cortos que se mergean a la rama de desarrollo.

Para simplificar el manejo de dichas ramas, dicho plugin para git crea comandos que resumen una serie de pasos para asegurarse la mejor implementación de dicho modelado.

2.1.1 Inicializando un gitflow repo

Para inicializar el modelado de gitflow en el repositorio actual debemos correr alguno de los siguientes comandos:

```
$ git flow init
$ git flow init -d  # toma los valores por default
```

2.1.2 Manejando ramas

Ahora para el manejo de todas las ramas y del repo en general, transcribo del README de gitflow, el manual de como manejar las ramas.

- To list/start/finish feature branches, use:

  ```
  git flow feature
  git flow feature start <name> [<base>]
  git flow feature finish <name>
  ```

  For feature branches, the `<base>` arg must be a commit on `develop`.

- To push/pull a feature branch to the remote repository, use:

  ```
  git flow feature publish <name>
  git flow feature pull <remote> <name>
  ```

- To list/start/finish release branches, use:

```
git flow release
git flow release start <release> [<base>]
git flow release finish <release>
```

For release branches, the `<base>` arg must be a commit on `develop`.

- To list/start/finish hotfix branches, use:

```
git flow hotfix
git flow hotfix start <release> [<base>]
git flow hotfix finish <release>
```

For hotfix branches, the `<base>` arg must be a commit on `master`.

- To list/start support branches, use:

```
git flow support
git flow support start <release> <base>
```

For support branches, the `<base>` arg must be a commit on `master`.

3 Compilando y ejecutando

Para compilar el proyecto es tan sencillo como correr el script `compile.sh` que se encarga del manejo de la imágen de docker para compilar el proyecto igual para cualquiera que haya clonado el presente repositorio.

```
$ ./compile.sh
```

Para ejecutar el proyecto se debe correr el programa `app` enviando como parámetros la ubicación de los archivos sobre los que se desea trabajar.

```
$ ./build/app [FILES...]
```

Este creará un espacio de *shared memory* para comunicarse con un posible proceso vista. La ubicación de este espacio de memoria es el único mensaje que el proceso `app` envía por salida estándar y dicha ubicación es la que debe utilizar el proceso **vista** para conectarse al espacio de memoria compartida. Para esto puede realizarse por medio de un *pipe* o por parámetro.

```
$ ./build/app [FILES...] | ./build/view
$ ./build/view [PATH]
```

A modo de testeo, bajo la carpeta `assets/bin` se encuentra un *script* de **bash** para crear una serie de archivos en una carpeta `test` con contenidos variados para utilizar junto con los procesos del proyecto.

```
$ ./assets/bin/test.sh
$ ./build/app test/* | ./build/view
```

Los resultados a su vez también son escritos en un archivo llamado `tpe_so_output.txt`.

4 Decisiones tomadas

4.1 IPCs utilizados

Para el desarrollo del presente trabajo se utilizaron diferentes tipos de **IPCs** (*Inter Process Communication*) para comunicar los procesos entre sí:

- **Pipes:** Utilizamos los llamados *pipes* para la comunicación entre el **proceso app** y cada uno de los **procesos slave**. Para esto se utilizaban dos *pipes* por cada *slave* dado que se necesitaba una comunicación bidireccional (*master* envía los *files* al *slave* y el *slave* envía el output de `md5sum` al *master*). Los **procesos slave** reciben por `stdin` la ubicación de los archivos a procesar e imprimen por `stdout` el resultado del proceso `md5sum` por lo que el **proceso app** fue el encargado de cambiar los *FileDescriptors* de cada uno de los *slaves* ejecutados para sus correspondientes `stdin` y `stdout` sea algún extremo de los pipes utilizados.
- **Shared memory:** Para la comunicación entre el **proceso app** y el **proceso view**, utilizamos memoria compartida. Para esto designamos un espacio de memoria compartida que crea el **proceso app** y el **proceso view** se "conecta" a dicho espacio. En ese espacio se encuentra disponible, no solo la información del output de los **procesos slave**, sino que también variables que permitían un mejor manejo de dicho espacio, como por ejemplo, índices de lectura y escritura, para, que si el **proceso app** escribe más rápido en el buffer que lo que lee el **proceso view**, dicho último proceso no pierda información, sino que va actualizando su índice a medida que lee y el **proceso app** escribe un output al final del otro.

Sin embargo, necesitabamos sincronizar la comunicación entre el **proceso app** y el **proceso view**, pues era posible que el **proceso view** lea más rápido que lo que escribe el **proceso app** y por ende lea información no deseada.

Para dicha sincronización utilizamos los conocidos **semáforos** de manera que cuando el **proceso app** terminaba de escribir informaba que ya había información disponible para consumir. Mientras no hubiera información para consumir por el **proceso view**, dicho proceso se quedaría bloqueado hasta que dicha condición cambie.

4.2 TADs

Una de las decisiones más importantes que tomamos a principio del proyecto fue trabajar con dos TADs (Tipos Abstractos de Datos). Creamos dos TADs, uno para guardar la información de los esclavos y otro para la información de la memoria compartida. Al crear las librerías, tanto `slave_manager` como `shm_lib`, se facilita mucho la modularización del código y, al crear los `struct` que agrupen tantas variables, el código queda mucho más limpio y agradable a la vista. También se facilita el pasaje y recibo de parámetros en diversas funciones debido a que se puede enviar y recibir la estructura como parámetro y no estar pasando parámetros individualmente.

4.3 Organización

Como el trabajo practico estaba dividido muy estrictamente en partes (la aplicación, los esclavos y la vista), decidimos entre el grupo separar estas tareas para un avance más rápido y seguro del proyecto. Esto permitió a cada integrante del grupo realizar su parte en el proyecto sin mucha necesidad de comunicación entre nosotros. Ahora sí, llegando al final del proyecto nos comunicamos entre nosotros para resolver la comunicación entre cada una de las partes. De esta manera pudimos conectar cada una de las partes tomando muchas decisiones menores a lo largo del camino.

5 Diagrama

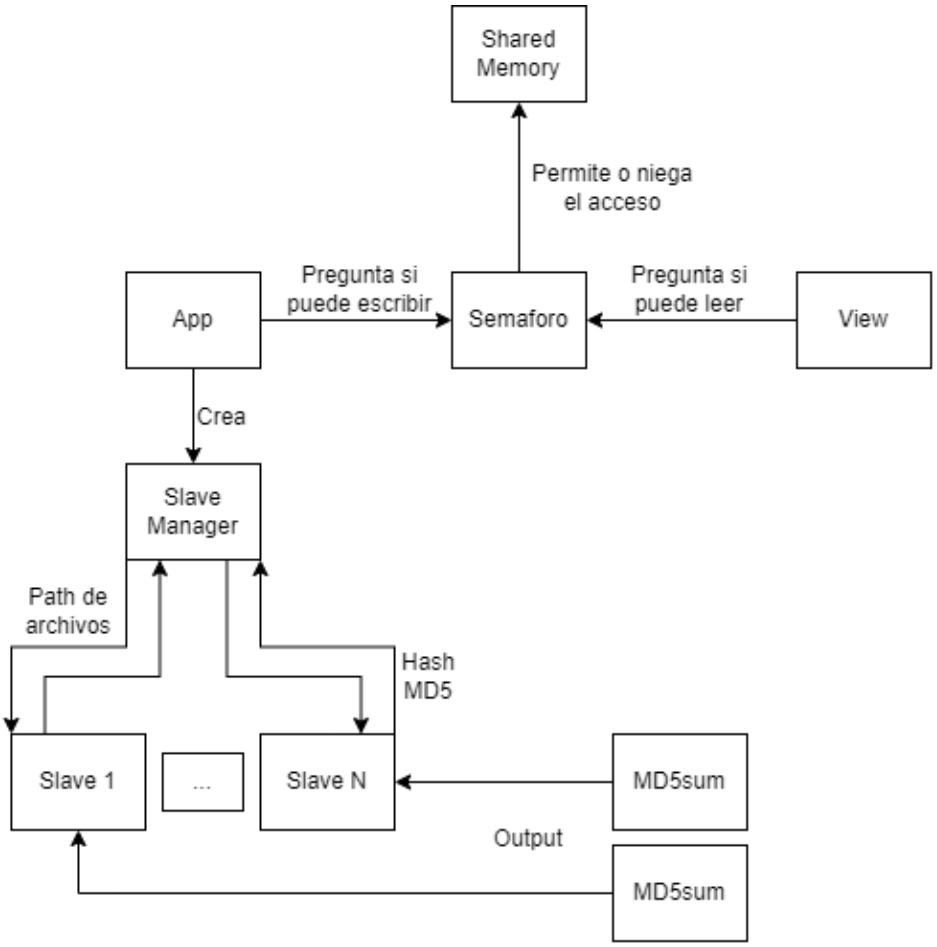

Figura 1: Diagrama de comunicación entre procesos

6 Problemas encontrados

6.1 Shared memory

Al estar investigando acerca del tema de "*shared memory*", pudimos entender varios aspectos del tema. Implementarlo en el proyecto no tuvo mucho problema más que utilizamos las syscalls que quedaron obsoletas con el paso del tiempo, como pueden ser `shmget`, `shmat` o `shmdt`. Por esta razón, tuvimos que cambiar las funciones creadas y usar syscalls tales como `shm_open` y `mmap`, que son el estándar **POSIX**.

El verdadero problema, vino al implementar los semáforos. Cuando incluimos los semáforos en el código descubrimos un gran problema al ejecutar la vista, este programa nunca termina. Como el proceso vista solo recibe la información necesaria para conectarse a la "*shared memory*" por entrada estándar no tenemos manera de decirle a el proceso vista cuando ya no hay más archivos para leer. Al ejecutar el programa, podíamos apreciar como había un buen funcionamiento de la shared memory entre el proceso aplicación y el proceso vista, pero cuando terminaba de imprimir los archivos, el proceso aplicación terminaba su ejecución y el proceso vista se queda esperando en el semáforo por el siguiente elemento.

Nuestra primera solución a esto fue investigar la API de semáforos **POSIX**. Investigando, encontramos `sem_timedwait` la cual hace la espera como el semáforo que habíamos incluido anteriormente y si en un rango de tiempo (programado) el semáforo no cambia de valor, inmediatamente se termina este proceso, lo cual permite que el proceso visto pueda terminar su ejecución. Esta solución no es la correcta ya que no funciona en todos los casos. Si da la casualidad de que todos los procesos esclavos estén trabajando en un archivo muy grande, estos pueden llegar a tardar mucho tiempo lo cual llevaría a el proceso vista a terminar antes de lo debería

La solución final que implementamos fue enviar desde el proceso **app** un **EOF** (string vacío) de manera tal que cuando el proceso **vista** reciba dicho string vea que su longitud es nula y sale del ciclo de ejecución.

6.2 Punteros en memoria compartida

Al declarar el TAD de la "*shared memory*" teníamos un error que no nos permitía destruir o desincronizar el bloque de memoria creado anteriormente. Tras mucho "*debugging*" descubrimos que le problema yacía dentro de la declaración de nuestra variable "*path*" dentro del TAD, la cual anteriormente era declarada como puntero. Como era declarado como puntero, al crear la memoria compartida se guardaba el valor "*path*" recibido por parámetro de esta manera. Esto es un error ya que se guardaba este parámetro de manera que apuntara a una dirección de memoria de un proceso especifico. Al querer borrar o cerrar la conexión con la memoria compartido, lanzaba un error, ya que el puntero no correspondía. La solución fue declarar esta variable como un array estático y de esta manera solucionar los errores y poder desacoplar la conexión a la memoria compartido y luego borrar el bloque.

6.3 Uso de select

Más en el comienzo del desarrollo, tuvimos un problema con el uso de `select`, el programa quedaba colgado en el `select`. Resulta que el slave no estaba escribiendo sus resultados correctamente en el `pipe`, `printf` no lograba escribir en el fd 1 (que estaba mapeado al pipe). Finalmente logramos solucionarlo mediante `setvbuffer`.

7 Limitaciones

En cuanto a limitaciones del proyecto, simplemente tuvimos muy en cuenta lo pedido por el trabajo practico. Al leer en varias ocasiones la consigna del primer trabajo practico de la materia, notamos que todo estaba detalladamente explicado como lo deberíamos resolver, dejando pocas decisiones para tomar (mayormente en cuanto a la estructura del código). Al tener tantos puntos en consideraciones, se limita las posibilidades que teníamos para resolver varios problemas, ya que ciertas soluciones a problemas encontrados

en el proyecto podían realizar un conflicto con otros puntos pedidos por el trabajo practico. De esta manera, dejaba pocas posibilidades para cumplir con todas las necesidades o aspectos que pedía el trabajo practico.

8 Código reutilizado

No utilizamos ninguna fuente externa aparte del "*man page*". Basamos partes de nuestro código en los ejemplos que proporciona este "*man page*", más notablemente, para la realización de la librería de "*shared memory*" tomamos mucha inspiración del código de ejemplo que proporciona acerca de `shm_open`.

9 Conclusión

Dado como finalizado el presente trabajo, podemos concluir y afirmar que el sistema planetado nos permitió la familiarización y utilización de diferentes métodos de **IPC** presentes en los sistemas operativos modernos, al igual que encontrarnos con errores y ver las consecuencias al no plantear un sistema de sincronización acorde a las necesidades.

Además, con el presente trabajo aprendimos a la utilización de fuentes importantes de información como los *manpages*.

Por último, nos topamos con errores y eso nos forzó a aprender a utilizar a herramientas de reconocimiento de errores como `strace`, `valgrind` y `pvs-studio`, que son de gran ayuda, no solo para crear un sistema funcional, sino también eficiente y escalable.