ITBA

SISEMAS OPERATIVOS

TRABAJO PRÁCTICO NRO. 1

INTER PROCESS COMMUNICATION

GRUPO Nº3 2C2022

ALEJO FLORES LUCEY | 62622
ANDRÉS CARRO WETZEL | 61655
NEHUÉN GABRIEL LLANOS | 62511

Informe - TP1

Grupo Nº3

Nombre	Legajo
Alejo Flores Lucey	62622
Andrés Carro Wetzel	61655
Nehuén Gabriel Llanos	62511

Introducción

Con el objetivo de interiorizarse con los mecanismos de intercomunicación entre procesos de Linux, se ha desarrollado un sistema para calcular el hash MD5 de archivos. Dicho sistema está compuesto por tres archivos ejecutables:

- `md5`: el programa central. Recibe como argumentos los archivos a analizar y se encarga de delegar su análisis a los procesos hijos. Por último, recibe los resultados y los guarda en un archivo.
- `slave`: el programa que calcula el hash MD5 del archivo que recibe por entrada estándar. Cabe recalcar que dicho ejecutable funciona de manera independiente

al resto del sistema.

- `view`: el programa que, si está siendo ejecutado, imprime a salida estándar los resultados del resto del sistema.

A lo largo de este informe se describirán los procesos de compilación y ejecución así como una explicación detallada de los mecanismos de comunicación entre procesos utilizados y por qué.

Instrucciones de compilación

Prerequisitos

Para poder compilar este proyecto se requieren de los siguientes prerrequisitos:

- Docker
- gcc
- make

Compilación

Por única vez, descargar la imagen de Docker:

```
docker pull agodio/itba-so:1.0
```

Luego, cada vez que se quiera compilar, ejecutar el siguiente comando en el directorio del proyecto:

```
docker run -v ${PWD}:/root --privileged -ti agodio/itba-so:1.0
```

Se iniciará el contenedor de Docker. Ahora, ejecutar los siguientes comandos:

```
cd root
make all
```

Instrucciones de ejecución

Se recomienda ejecutar el sistema dentro de la imagen de Docker, para obtener los resultados óptimos. Por esta razón, las siguientes instrucciones suponen que se está utilizando Docker y que el directorio actual es el del proyecto.

Opción 1 (sin impresión en salida estándar)

Simplemente ejecutar la siguiente línea en la terminal:

```
./md5 <lista_de_paths>
```

`<lista_de_paths>` debe ser una lista separada por espacios de paths relativos apuntando a los archivos que se desea que sean analizados.

Los resultados serán almacenados en el archivo `results.txt`.

Opción 2 (piping)

Ejecutar la siguiente línea en la terminal:

```
./md5 <lista_de_paths> | ./view
```

`<lista_de_paths>` debe ser una lista separada por espacios de paths relativos apuntando a los archivos que se desea que sean analizados.

Los resultados serán almacenados en el archivo `results.txt`. Asimismo, los resultados serán impresos en salida estándar.

Opción 3 (dos terminales)

Se requiere correr dos instancias de la misma imagen de Docker:

1. Abra una instancia de Docker como se explicó anteriormente en una terminal (*Terminal 1*)
2. En una nueva terminal (*Terminal 2*), corra `docker ps`
3. Copie el *Container ID*, pues será utilizado en el próximo paso
4. Corra la siguiente línea de código

   ```
   docker exec -ti <container_id> bash
   ```

5. Cambie de directorio al del proyecto

Ahora, ejecute la siguiente línea en la *Terminal 1:*

```
./md5 <lista_de_paths>
```

`<lista_de_paths>` debe ser una lista separada por espacios de paths relativos apuntando a los archivos que se desea que sean analizados.

Se imprimirá por salida estándar la cantidad de archivos a analizar.

Luego, en la *Terminal 2*, ejecute la siguiente línea:

```
./view <cantidad_de_archivos>
```

Los resultados serán almacenados en el archivo `results.txt`. Asimismo, los resultados serán impresos en la salida estándar de la *Terminal 2.*

Chequeos de *memory leaks* y calidad de código

⚠️ Se requieren dos programas extra para realizar dichos chequeos:

- Valgrind
- PVS-Studio

Chequeo de Valgrind

Correr las siguientes líneas de código en la terminal:

```
make all
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes \
  ./md5 <lista_de_paths> | ./view
```

`<lista_de_paths>` debe ser una lista separada por espacios de paths relativos apuntando a los archivos que se desea que sean analizados.

La resolución de los chequeos será impresa en salida estándar

Chequeo de PVS-Studio

Correr la siguiente línea en la terminal:

```
make check-pvs
```

La resolución de los chequeos será almacenada en el archivo `report.tasks`

Limpieza de recursos

Se pueden eliminar los archivos creados por el sistema:

- Para eliminar los ejecutables y archivos de resultados, `make clean`
- Para eliminar los archivos de los chequeos de PVS-Studio, `make clean-pvs`
- Para eliminar todos los archivos creados por el sistema, `make clean-all`

Desarrollo y mecanismos utilizados

Para lograr la comunicación entre los procesos `md5`, `slave` y `view` se utilizaron semáforos con nombre (*named semaphores*), memoria compartida (*shared memory*) y tuberías (*pipes*).

Véase a continuación el diagrama del sistema completo:

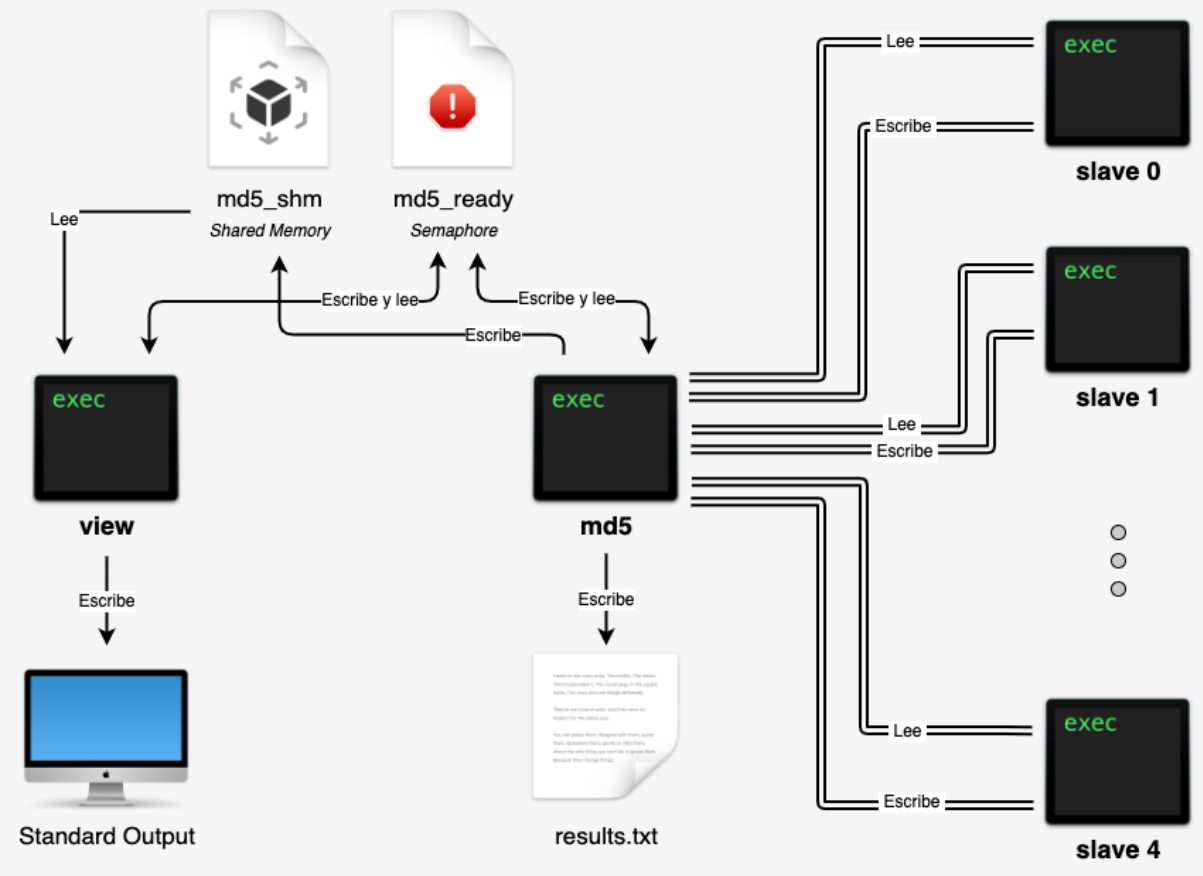

Figura 1. Diagrama del sistema

En primer lugar, el programa principal (`md5`) se comunica con cada esclavo a través de dos *pipes*: uno de ellos se utiliza para la comunicación de `md5` a `slave` y el otro se utiliza para la comunicación de `slave` a `md5`. Se tomó la decisión de hacer transparente la existencia del programa principal para el programa esclavo. Esto quiere decir que `slave` es independiente de `md5` y no sabe de su existencia; de hecho, se puede ejecutar `slave` independientemente y todo funcionará correctamente. Para lograr esto, se debió cerrar la entrada y salida estándar del proceso esclavo y reemplazarlos con las puntas del *pipe* correspondientes. Esto se realizó con la ayuda de la función `dup2()`.

En segundo lugar, se utiliza una memoria compartida y un semáforo con nombre para la comunicación entre el proceso principal y el proceso vista (`view`). El programa `md5` escribe en la *shared memory* a medida que el esclavo devuelve los resultados y el programa `view` consume de ella para imprimir en salida estándar. El semáforo con nombre se utiliza para que el proceso vista se interrumpa en el caso que `md5` no haya escrito los resultados; cada vez que `md5` escribe una línea en la *shared memory*, se realiza un `sem_post()`, que aumenta en una unidad el valor del semáforo. Ahora bien, el proceso vista, cada vez que lee una línea de la *shared*

memory, decrementa en una unidad el valor del semáforo haciendo uso de `sem_wait()`. De esta manera, si el valor del semáforo se encuentra en 0, lo que indica que no hay nada nuevo para leer en la *shared memory*, el proceso vista se quedará esperando hasta que `md5` realize un `sem_post()`.

Cabe recalcar que tanto el proceso `md5` como el proceso `view` abren y cierran la memoria compartida y el semáforo, haciendo uso de las librerías correspondientes. Sin embargo, el proceso principal (`md5`) es el designado para crear y destruir dichos recursos. Esta decisión fue tomada debido a la posibilidad de que no se ejecute el proceso vista.

Por último, el proceso principal escribe los resultados en un archivo de texto.

Limitaciones y problemas encontrados

Una limitación importante del sistema tiene que ver con la creación y destrucción de la región de memoria de la *shared memory.* El proceso principal, `md5`, es el encargado de la creación y destrucción de dicha región. Sin embargo, si se interrumpe la ejecución de dicho proceso la shared memory no se destruirá correctamente y debido a eso las próximas veces que se ejecute el programa éste dará error. Para solucionar esto se cuenta con un *target* del `Makefile`: corriendo `make clean-shm` se elimina la región de memoria residual.

La mayor limitación del proyecto es el máximo arbitrario de procesos esclavos que se pueden crear. Se utiliza una constante con un valor arbitrario (5) para limitar dicha cantidad máxima. De la misma manera, inicialmente se le asignan una cantidad arbitraria (2) de archivos a cada proceso esclavo para que sean analizados.

Por otro lado, no se sufrieron grandes problemas a la hora de desarrollar el sistema. Se debió leer la documentación de muchas funciones para entender su funcionamiento y cómo utilizarlas correctamente; para esto, la herramienta `man` fue de mucha ayuda.

No está de más mencionar un problema que no se pudo sortear de la manera que estabamos esperando: la función `shm_open()` recibe como argumento una máscara con ciertos *flags* que indican el modo de apertura del archivo, entre otras especificaciones. Nos pareció de buen estilo abrir la *shared memory* con el *flag* `O_RDONLY` desde el proceso vista, pues él no escribirá en la memoria compartida. Sin embargo, nos encontrabamos con un error de ejecución al hacer esto. Se terminó descartando la idea y utilizando el *flag* `O_RDWR` para solucionar este error.