Trabajo Práctico 1: IPC

72.11 - Sistemas Operativos

Segundo Cuatrimestre 2024

Integrantes del **Grupo 21**:

62872	Matías LEPORINI
63382	Camila LEE
63074	Ana NEGRE

Instrucciones de Compilación

Para crear el contenedor dentro del directorio que contiene el repositorio:

```
docker pull agodio/itba-so-multi-platform:3.0

cd $MyDir

docker run -v "{$PWD}:/root" --security-opt seccomp:unconfined -ti
agodio/itba-so-multi-platform:3.0
```

Una vez iniciado el contenedor, correr los comandos:

```
cd root

cd TP1-SistemasOperativos-72.11

make all
```

De quererse eliminar los archivos de output, puede correrse el comando:

```
make clean
```

Instrucciones de Ejecución

Hay tres maneras de correr el programa:

Caso 1: Cálculo de los hash sin la posibilidad de ver los resultados durante el procesamiento.

```
./bin/md5 <archivos>
```

Caso 2: Cálculo de los hash con la posibilidad de ver los resultados durante el procesamiento.

```
./bin/md5 <archivos> | ./view
```

Caso 3: En dos terminales separadas, lo cual equivale al caso anterior.

```
./bin/md5 <archivos>

./bin/view <ar>
```

Decisiones de Desarrollo

En primer lugar, durante el desarrollo se decidió hacer que el programa slave corriera de manera independiente a los otros dos programas. Esto se consideró correcto bajo la premisa de la modularización y la independencia de programas (dentro de lo posible).

Por el otro lado, se debieron tomar varias decisiones respecto a la implementación de la comunicación entre el programa app y view. La comunicación entre estos procesos se dio en dos aspectos. En primer lugar, mediante la *shared memory*, y en segundo lugar, para garantizar la exclusión mutua durante el acceso a la shared memory, mediante dos *semáforos*.

Para compartir la memoria entre ambos procesos, tanto para el caso de querer correr app en una terminal y view en otra como para el caso de correr ambas utilizando el pipe, se decidió dejar el nombre de la shared memory impreso en stdin, para luego ser recuperado por el usuario (o por view). Los nombres de los semáforos, sin embargo, se decidieron mantener como constantes en la biblioteca compartida.

Diagrama de Procesos

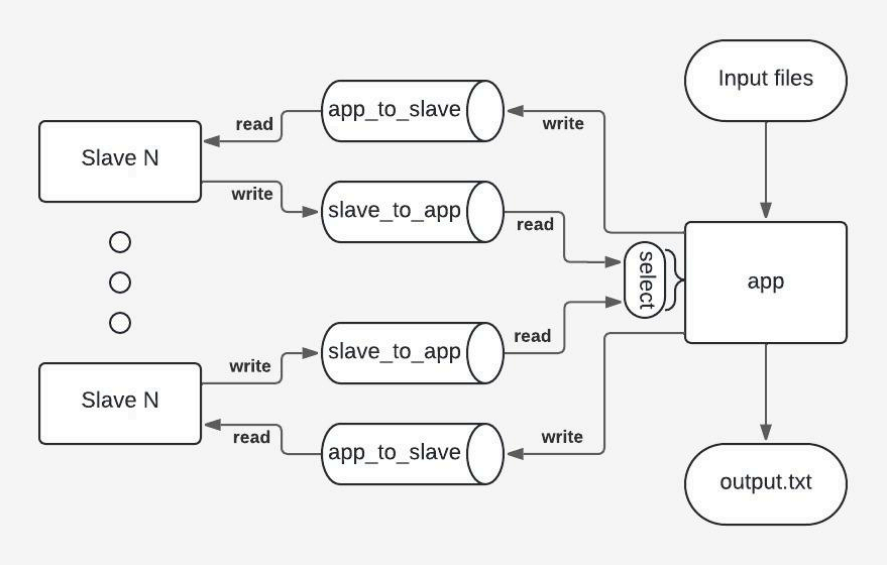

Figura 1. Conexión entre slaves y app

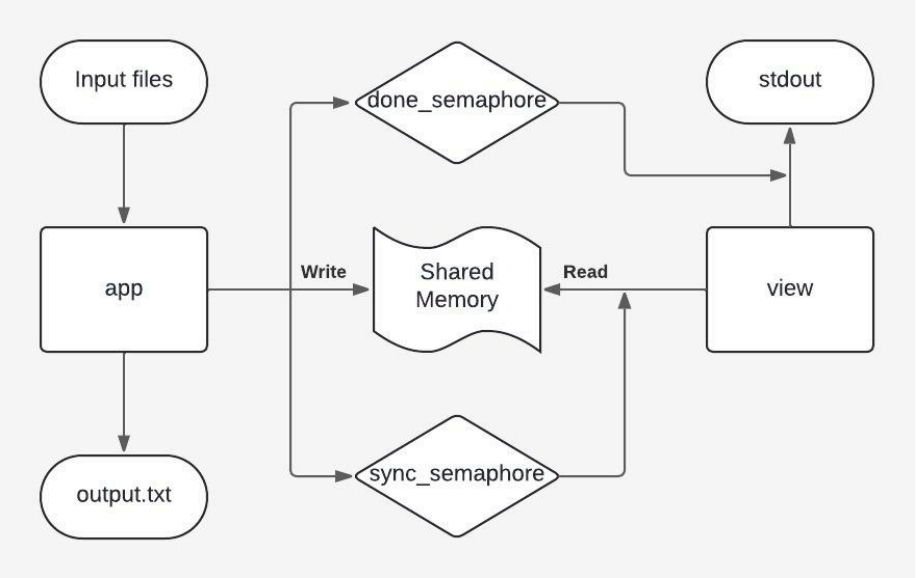

Figura 2. Conexión entre app y view

Limitaciones

A lo largo del desarrollo del trabajo se encontraron algunas limitaciones.

Por ejemplo, en el caso de abortar la ejecución del programa antes de tiempo, la shared memory no se libera. Adicionalmente, de decidirse abortar la ejecución de app y no de view, esta última queda en un bucle aguardando la llegada de información de app que nunca llega. Para solucionarlo se debería o bien elaborar un handler de señales para, en caso de un SIGKILL o SIGTERM en app, ejecutar un kill() de view; o bien transmitir el abort de app a través de la memoria compartida (o posiblemente un pipe del tipo FIFO). Estas soluciones, sin embargo, se consideraron que escapan lo solicitado por el enunciado, por lo que se decidió no implementarlas.

Problemas encontrados durante el desarrollo

Durante el desarrollo se encontraron numerosos problemas, lo cual no es de extrañar. El problema principal, sin embargo, fue la conexión entre app y slave.

Para la utilización del buffer compartido se decidió que, de no poder leer de la memoria compartida, view se bloqueara hasta que existiera información para leer de app (subiendo el semáforo), y que, luego de consumirla, view volviera a bajar el semáforo. En un inicio se había decidido utilizar un único semáforo para efectuar la sincronización entre ambas aplicaciones. El principal problema con esto fue que, al terminar de escribirse la información desde app, nunca se volvía a levantar el semáforo, y view, al "no saber" que debía efectuar un exit(), quedaba bloqueado indefinidamente.

En un principio, este problema se intentó solucionar proporcionándole una estructura a la shared memory, creando una variable "shmDone" que permitiera a view saber cuándo debía salir del loop. Problema: al no ser atómica la asignación de la variable, se producía una race condition en la conexión, donde "once in a blue moon", view continuaba bloqueándose.

Finalmente se decidió utilizar la implementación de dos semáforos, uno con el objetivo previamente explicado, y otro con el propósito de indicar que el buffer está lleno para permitir a view leer del mismo. Luego, para poder salir del loop, se verificó que ya no hubiera más caracteres a leer preguntando si en el índice de lectura (de view) se encontraba un null.

Otro problema importante encontrado durante el desarrollo fue el manejo inadecuado de los *file descriptors* de los pipes. Inicialmente, no se entendía por qué los *file descriptors* de los archivos permanecían abiertos -incluso cerrando los descriptores de archivo después de hacer dup2- al interrumpir el programa con CTRL+Z. Al usar ps aux para identificar los procesos esclavos y luego lsof -p <pid del esclavo>, se observaba que los esclavos con PID más alto tenían dos *file descriptors* adicionales abiertos por cada esclavo anterior. Finalmente, se comprendió que el problema radicaba en que, aunque a nivel de la aplicación principal los esclavos pueden estar trabajando simultáneamente, desde el punto de vista de un esclavo específico, solo sus propios descriptores de archivo deben estar abiertos ya que opera de manera aislada con acceso único a sus propios canales de comunicación. Teniendo en claro esto, se implementaron varios cambios para resolver el problema:

1. Modificar la estructura de los esclavos: en lugar de almacenar solo el descriptor de lectura y escritura (que es lo que se había hecho al principio), se cambió para que guardara los pipes app_to_slave[2] y slave_to_app[2]. Esto permitió un control más sobre todos los *file descriptors* de los esclavos.
2. Se mejoró la gestión de los *file descriptors* agregando en la función de creación de esclavos (create_slave_process) código adicional para cerrar los descriptores de archivo de los otros esclavos. Es decir, dentro del proceso hijo, se cerraron todos los descriptores de archivo que no pertenecían al esclavo actual.

Luego de implementar estos cambios, al ejecutar lsof solo se ven dos descriptores de archivo abiertos, además de los cuatro que siempre deben estar abiertos (0r - STDIN, 1w - STDOUT, 2u - STDERR, 3w - output.txt).

Fragmentos de Código Reutilizado

Para pensar el esqueleto de sincronización entre app y view se utilizó el pseudo-código visto en clase:

```
master(){
while(smth){
      esperar_resultado(...);
      publicar_resultado_en_shm(...);
      avisar_resultado_disponible(...);
}
///
}
vista(){
while(smth){
      esperar_resultado(...);
      leer_resultado_de_shm(...);
      imprimir_resultado_en_stdout(...);
}
}
```

Para la incorporación de los dos semáforos se utilizó como base el ADT de memoria compartida visto en la clase práctica. A su vez, para la sincronización de los semáforos, también se tomó el esqueleto mencionado en la clase práctica.