Inter Process Communication

1er Trabajo Práctico - Grupo 12

72.11 Sistemas Operativos

Segundo cuatrimestre 2023

Integrantes:

Deyheralde, Ben (Legajo: 63559)

Gonzalez Rouco, Lucas (Legajo: 63366)

Mutz, Matias (Legajo: 63590)

Profesores:

Aquili, Alejo Ezequiel

Godio, Ariel

Mogni, Guido Matias

Índice

1. Introducción

El objetivo del 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.

El sistema consiste de tres partes, la aplicación, los esclavos y la vista.

Para conectar la aplicación con los esclavos se utilizaron pipes. Para conectar la aplicación con la vista, se utilizó una memoria compartida. Para evitar problemas de sincronización se utilizaron semáforos

La aplicación recibe por línea de comandos los archivos que debe procesar, se encarga de iniciar los esclavos, mandar los archivos a procesar y escribir en la memoria compartida. Además, guarda en un archivo el resultado del procesamiento, independientemente de si aparece un proceso vista o no.

Se crea un esclavo por cada 20 archivos y cada esclavo tiene una carga inicial de 5 archivos. Los esclavos procesan los archivos y devuelven el resultado a la aplicación a través de pipes. Una vez que terminan con la carga inicial, se les envía de a un archivo hasta que ya no queden archivos por procesar.

La vista lee de la memoria compartida y muestra la información por pantalla.

2. Decisiones tomadas durante el desarrollo

Para el trabajo utilizamos los siguientes IPCs y mecanismos de sincronización:

IPCs:

Para conectar el proceso de aplicación con sus esclavos, se utilizaron pipes anónimos, dos pipes por cada aplicación – esclavo. Uno que se utiliza para la escritura desde la aplicación al esclavo y otro que sirve para la escritura del esclavo a la aplicación. La aplicación, en búsqueda de aprovechar simultaneidad, utiliza select() para estar pendiente a que ya se esté listo para leer el resultado del esclavo y a continuación mandarle otro archivo si es que todavía quedan por procesar.

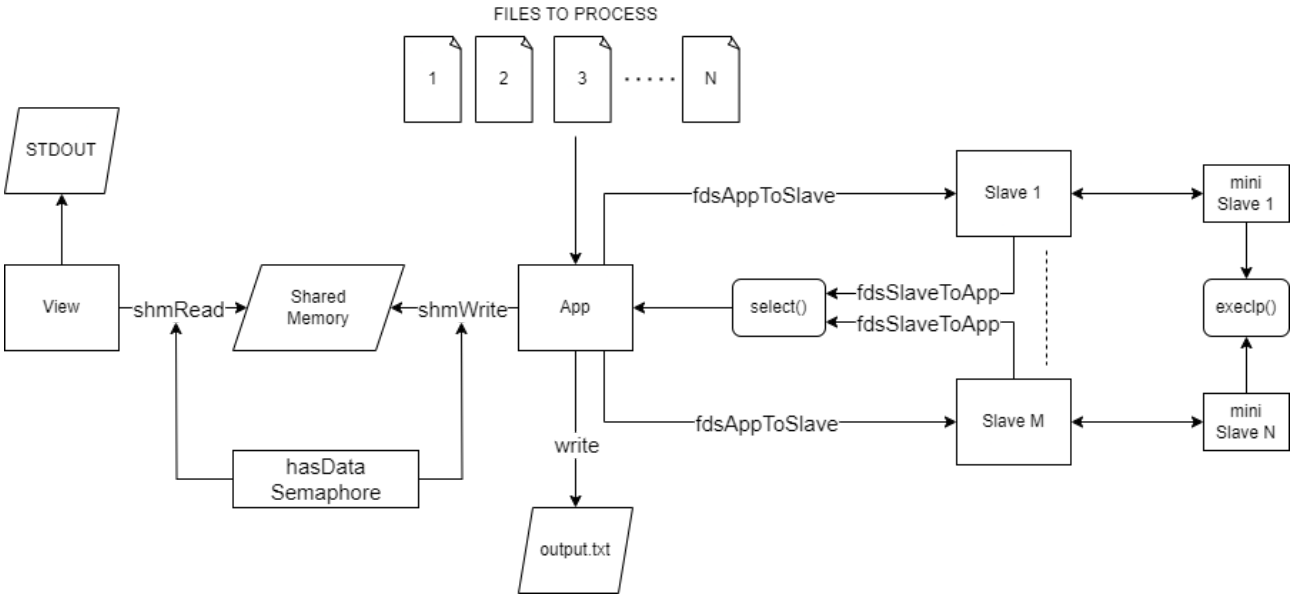
En la comunicación que se hace entre la aplicación y la vista, se utiliza memoria compartida con semáforos. Al escribir, se realiza un writeShm (no la syscall "write") en la shared memory. Este método logra que se escriba la información indicada en el buffer. A la vez, se le realiza un post en el semáforo hasData. Luego para leer desde la view primero se realiza un wait del mismo semáforo para saber si hay datos para leer. Utilizando estas dos directivas evitamos tener problemas de race condition a la hora de comunicarnos entre la aplicación y la vista.

Mecanismos de sincronización:

Para evitar race conditions, como mencionamos antes, utilizamos un semáforo para controlar el acceso a los recursos compartidos.

Para evitar deadlocks, tuvimos en cuenta la ubicación del post y del wait del semáforo, para evitar que en algún context switch se llegue a un deadlock.

3. Diagrama de conexión de procesos



4. Instrucciones de compilación y ejecución

Antes de compilar, se recomienda ejecutar el comando **‘make clean’**. Esto va a remover los archivos ejecutables creados en una compilación anterior, los archivos objeto y los archivos de texto.

Para compilar el sistema, se utiliza el siguiente comando **‘make’.**

El sistema se puede utilizar de dos maneras distintas. La primera es corriendo todo de una misma terminal en donde el nombre de la memoria compartida se envía mediante un pipe al proceso vista. Para ejecutar de esta manera se debe usar la siguiente línea: **‘./app files/* | ./view’** . En donde files/* es el path de los archivos que se busca hacerles el md5.

Para la segunda forma, se debe primero ejecutar el comando **‘./app files/*’** , que iniciará el proceso aplicación, que distribuirá las tareas de procesamiento entre los esclavos y a su vez creará la memoria compartida y allí dejará la información lista para que luego desde la vista se puedan obtener esa información. Al ejecutar el comando obtendremos por pantalla el path de la memoria compartida, esto le pasaremos a la vista como parámetro para poder conectarse a la memoria. El comando para ejecutar la vista es **‘./view <pathToSHM>’**.

Al ejecutar la app se genera un archivo output.txt en donde se puede ver información del md5 del archivo, junto con el nombre y también el process id del proceso esclavo que se encargó de hacer el md5 para ese archivo. Al ejecutar la vista, a medida que lee la información de la memoria compartida, se imprime en la terminal esta misma información.

En caso de querer probar el comando “strace” o ejecutar el programa con Valgrind, hay que sacar el flag “-fsanitize=address” en el archivo Makefile.

5. Limitaciones

El sistema tiene las siguientes limitaciones:

El buffer de llegada es un buffer lineal, por lo que no se puede garantizar que los resultados se muestran en pantalla en el mismo orden en que se procesaron los archivos.

El buffer de la memoria compartida tiene un tamaño fijo y no es circular, por lo que no es virtualmente infinito. Habrá una limitación de cantidad de archivos, sin embargo, esta cantidad es muy alta.

6. Problemas encontrados durante el desarrollo

Uno de los principales problemas que encontramos durante el desarrollo fue evitar las race conditions. Se utilizó un semáforo, mencionado anteriormente, para controlar el acceso a los recursos compartidos. Una vez que la solución fue encontrada con la ayuda de una clase práctica donde se habla de este problema, se llegó a la conclusión de que no era demasiado complejo el uso del semáforo pero se utilizó una gran cantidad de tiempo pensando en cómo implementar esta solución de la manera ideal sin usar recursos que no eran necesarios, como por ejemplo el hecho de utilizar más de un semáforo.

Otro problema que encontramos fue evitar los deadlocks. Para ello, se plantearon distintas situaciones, pensando en dónde podría generar uno. Esto es un proceso tedioso, pero la solución lograda, los evita completamente.

También a medida que se avanzaba, varios problemas surgían pero se iban resolviendo en el momento. Lo correcto fue, desde un primer momento, utilizar las recomendaciones de la cursada para detectar leaks de memoria y posibles fallos. Con esto se logró que en el final de trabajo, habiendo terminado con todas las funcionalidades, que la corrección de errores que nos mostraba PVS-studio sea muy fácil y rápida.

7. Citas y fragmentos de código reutilizados

El ejemplo provisto en la entrada del manual de “shm_open” fue utilizado como base para la creación, conexión y escritura de la memoria compartida. En el mismo se puede encontrar lo siguiente:

- Definición del struct de la memoria compartida.
- Un programa que pasa a mayúscula lo escrito en el buffer.
- Un programa que escriba en el buffer.

8. Conclusión

Como conclusión, este trabajo práctico nos permitió entender mediante la práctica el funcionamiento de los tipos de Inter Process Comunication (IPC) y mecanismos de sincronización que vimos teóricamente en clase ya que tuvimos que utilizar pipes anónimos, shared memory y semáforos.

También nos ayudó mucho a familiarizarnos con el manual de Linux ya que tuvimos que leerlo para aprender sobre el uso de varias syscalls y funciones de C como por ejemplo execlp, shm_open, mmap, ftruncate, sprintf, entre otras.

Además, aprendimos a utilizar herramientas como Valgrind y PVS-studio y strace para poder detectar errores.