Sistemas Operativos

Trabajo Práctico 1

Grupo 1

Mariano Agopian, legajo 62317, maagopian@itba.edu.ar

Santiago Tomas Medin, legajo 62076, smedin@itba.edu.ar

Sofia Paula Altman Vogl, legajo: 62030, saltman@itba.edu.ar

Índice

Introducción

El objetivo de este informe era aprender a utilizar los distintos tipos de IPCs presentes en un sistema POSIX, mediante la implementación de un sistema que distribuirá el cómputo del md5 de múltiples archivos entre varios procesos.

En la siguiente imagen (Figura 1) se puede observar un diagrama de cómo están conectados los distintos procesos.

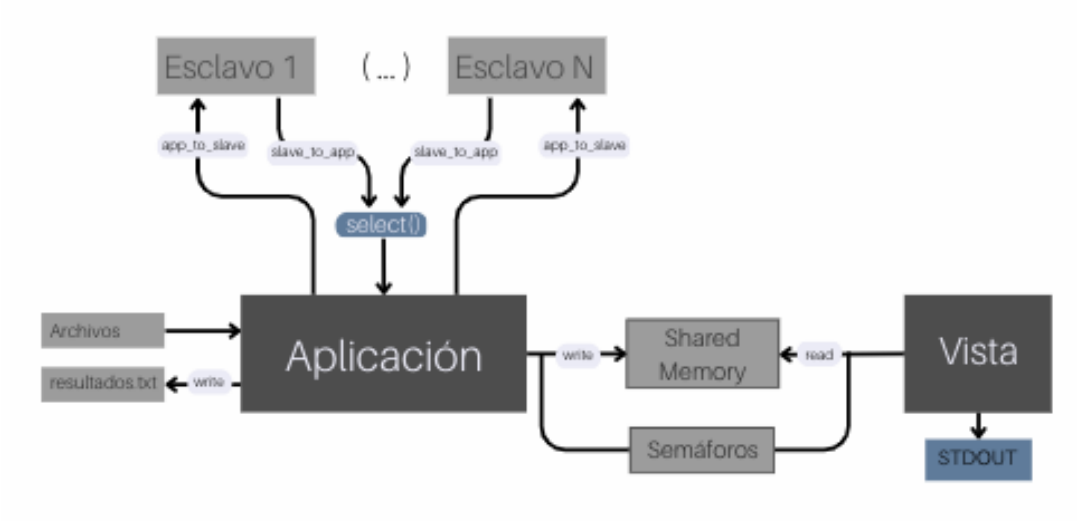

Figura 1: *Diagrama ilustrando cómo se conectan los diferentes procesos*

Como se puede observar, los esclavos (de 1 hasta n siendo este un número natural), se comunican con la aplicación de dos maneras, dependiendo de si el flujo es del esclavo a la aplicación o viceversa. Esto funciona mediante los pipes slave_to_app y app_to_slave respectivamente. Cabe destacar que previo a que la aplicación lea de los pipes slave_to_app, se realiza un select para determinar si este debe ser leído, mediante el chequeo de si se encuentra vacío o no. Por otro lado, al recibir los archivos y ser manejados mediante los esclavos, la aplicación (depende de cómo se pasen los parámetros) o volcara los resultados en un archivo .txt (resultados.txt específicamente), o lo escribirá en la memoria compartida para que lo lea el proceso vista y lo imprima por salida estándar. Los semáforos son utilizados para evitar la condición de carrera en el proceso de escritura y lectura de la memoria compartida.
En el desarrollo de este trabajo se indicarán las instrucciones de compilación y ejecución, las decisiones tomadas durante el desarrollo con sus limitaciones y problemas surgidos y

cómo se solucionaron, finalizando con citas de fragmentos de código reutilizados de otras fuentes.

Instrucciones de compilación y ejecución

Requerimiento: Compilar los programas dentro de Docker en el entorno provisto por la cátedra.

Compilación:

1. Para abrir el entorno de Docker

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

   ```
   $  cd  tp1-so
   ```
   (Para dirigirse hacia la carpeta donde se encuentra el trabajo práctico).

   ```
   $ docker run -v "${PWD}:/root" --security-opt
seccomp:unconfined -ti agodio/itba-so:1.0
   $ cd root
   ```

2. Compilación del programa

   ```
   $ make all
   ```

Ejecución: Hay 3 formas de ejecutar el programa

1. Cálculo de los md5 hash

   ```
   $ ./md5 <files>
   ```

2. Cálculo de los md5 hash y ver resultados mientras son procesados

   ```
   $ ./md5 <files> | ./vista
   ```

3. Similar al caso 2 pero en dos terminales separadas

   a. En la primera terminal:

      ```
      $ ./md5 <files>
      ```

   Este comando imprimirá 3 nombres que serán los argumentos a introducir en la segunda terminal.

   b. Segunda terminal:

      ```
      $ ./vista <argument1> <argument2> <argument3>
      ```

Decisiones tomadas durante el desarrollo

Para empezar, se detallarán las decisiones vinculadas con los mecanismos de IPC que fueron utilizados. Por un lado, se utilizaron pipes anónimos para la comunicación entre el proceso aplicación (denominado *main* en nuestro trabajo) y los procesos esclavos (*slaves*). Por otro lado, con el fin de obtener el resultado del comando *md5sum* se decidió redireccionar la salida estándar de los procesos subesclavos a la entrada estándar del proceso esclavo, evitando así la creación de archivos intermedios (como fue visto en clase).
Finalmente, se utilizó un bloque de memoria compartida (*shared memory*) entre la app y el proceso vista, además de dos semáforos.
Luego, en relación con el proceso vista, había dos formas de invocarlo a este: mediante distintas terminales o mediante la utilización de un pipe. Para esta última se decidió utilizar la función fgets ya que permite la lectura de entrada estándar. Además, se decidió implementar dos semáforos como fue indicado previamente ya que con uno solo surgieron problemas en su implementación
En relación con el proceso de la impresión y visualización de los datos pedidos, se decidió definir estructuras que permitian agrupar esta información para una implementación más ordenada. De la misma manera, se decidió definir estructuras para los esclavos en donde se declaran los pipes, el pid y el nombre del archivo que estuviese manejando en el momento (no se vio necesario definir un array, y en cambio ir pisando este dato a medida que se iban procesando) con el fin de acceder fácilmente a su información tanto para la lectura y escritura de pipes como para su lectura para la impresión.

Problemas durante el desarrollo

Los problemas principales que surgieron fueron los siguientes:

- Al compilar surgieron problemas debido a que no eran encontradas las funciones de semáforos y shared memory repetidas veces. Para solucionarlo se descubrió que hacía falta definir unas variables para que luego las librerías funcionen correctamente.
- Otro problema encontrado fue que los pid obtenidos por salida estándar y volcados en el archivo.txt diferían cuando estos tenían que ser iguales. Esto fue solucionado limpiando la shared memory, ya que se concluyó que se estaban leyendo cosas de programas anteriores.
- Hubo muchas funciones que requirieron de gran parte de nuestro tiempo entender bien cómo funcionaban, entre ellas la función select(). Esto se solucionó consultando en diferentes foros y documentaciones para tener un buen manejo de las funciones.
- Al querer implementar el programa con un solo semáforo tuvimos algunos problemas de sincronización y como fue mencionado previamente, se decidió entonces utilizar dos.
- A partir del pvs-studio en donde se indicaba que el strlen() estaba siendo mal manejado, incluimos los chequeos necesarios para arreglar esto, al manejar el caso en el que diera cero, en donde se tendría que terminar su ejecución.
- Relacionado con la devolución del valgrind, nos vimos obligados a modificar el código ya que se indicaba que la variable buffToSend en el main.c no era inicializada, lo cual fue solucionado con un simple llamado a la función memset().

Limitaciones

Las limitaciones que se tuvieron fueron las siguientes:

- Para empezar, una de las limitaciones encontradas fue que en caso de interrumpir el programa con el comando CTRL+C los recursos no serían liberados. Lo mismo ocurre en el caso de que el programa aborte con error.
- Otra limitación encontrada fue que si no se desea correr el programa de la forma explicada en caso 2, se debe tener dos terminales abiertas en simultáneo. Esto se debe a que utilizan la misma shared memory y además, si no se corre el proceso vista dentro de los 2 segundos del sleep del proceso md5, la memoria compartida no podrá ser abierta en el proceso vista.
- Finalmente, el proceso aplicación siempre abre el semáforo y shared memory por más de que no sea necesario (por ejemplo, los abre aunque no se esté corriendo el proceso vista). Esto no es lo más eficiente y puede traer errores innecesarios.

Citas de fragmentos de códigos reutilizados de otras fuentes

- La función is_regular_file(), encontrada en el archivo main.c se extrajo de una publicación de Stack Overflow:

Jookia, et al. “Checking If a File Is a Directory or Just a File.” Stack Overflow, 13 Marzo 2015, https://stackoverflow.com/questions/4553012/checking-if-a-file-is-a-directory-or-just-a-file.