Trabajo Práctico 1
72.11 Sistemas Operativos

Grupo 1

Integrantes

Matías Juan Rossi Seifert - 63202
Luciano Stupnik - 64233
Federico Viera - 62022

Índice

1. Introducción

El objetivo de este trabajo práctico es aprovechar las ventajas de la programación paralela poniendo en práctica los conocimientos adquiridos en clase sobre algunos de los distintos mecanismos de intercomunicación entre procesos (IPCs) de un sistema POSIX. Eso fue logrado mediante la implementación de un sistema de cómputo de MD5 para varios archivos, dividiendo la carga entre los esclavos y encargando la presentación de los resultados a un proceso vista.

2. Decisiones Tomadas Durante el Desarrollo

Para la comunicación entre los esclavos (*slave.c*) y el master (*md5.c*) nos fue indicado que utilicemos un par de pipes por cada esclavo. Uno para el envío de paths de archivos del master al esclavo (*pipe_path_to_slave*) y otro para recibir los resultados del procesamiento del archivo que le envía el esclavo al master (*pipe_hash_to_master*).

Luego decidimos que el master escriba en un buffer en memoria compartida, el segundo mecanismo de IPC utilizado en el trabajo práctico. De está manera el proceso vista podrá ir mostrando aquello que el master va escribiendo. El nombre de la memoria compartida es escrito en *stdout* para que el proceso vista pueda usar dicho nombre para abrirla, ya sea pipeando en la línea de comandos o pasándole como parámetro a la vista.

Para el manejo de la memoria compartida decidimos crear una librería de autoría propia, *shm_lib.c.* Esta contiene de forma condensada las funciones necesarias para poder crear y abrir (*get_shm*), cerrar (*close_shm*), escribir *(shm_write)* y leer *(shm_read)* del buffer en memoria compartida.

Decidimos que cada escritura del proceso md5 (realizada con *shm_write*) en la memoria compartida estará delimitada con un ‘\0’ al final, para que a la hora de realizar una lectura desde la vista (con *shm_read*) se lea hasta dicho punto. Estás funciones *shm_read* y *shm_write* van actualizando los offsets *write_offset* y *read_offset* del ADT para que la siguiente vez escriban y lean desde donde dejaron la última vez.

Para la sincronización de la memoria compartida entre el proceso md5 y el proceso vista, optamos por usar un semáforo *data_available* que indica la cantidad de lecturas disponibles. De está manera al terminar de escribir (*shm_write*) un dato se hace un *sem_post(&data_available)* para indicar que hay un dato más para que sea leído y a la hora de leer (*shm_read*) se hace un *sem_wait(&data_available)* para garantizar que el proceso vista no lea cuando no hay datos disponibles y se quede bloqueado evitando el busy waiting. Sumado esto a nuestra implementación del *write_shm* y *read_shm* permitimos que el proceso vista pueda leer datos previamente escritos a la misma vez que el proceso md5 escribe nuevos. Es decir, el proceso vista y el proceso md5 pueden acceder al buffer a la misma vez, pero siempre en partes distintas del mismo, puesto que el proceso vista solo puede leer escrituras anteriores.

En el esclavo tuvimos que poder crear un proceso que corra el programa hashmd5sum con el path recibido y utilizar la salida estándar del mismo. Para esto encontramos la función *popen()* la cual recibiendo en nuestro caso el parámetro "**r**" y el comando a ejecutar, se encarga de crear un pipe, crear un hijo, redireccionar la salida estándar del hijo al pipe, y ejecutar el comando indicado. El retorno de está función es un puntero de archivo que nos permite acceder al extremo de lectura del pipe. De está manera, en el esclavo recibimos el resultado de *popen()*, le agregamos nuestro PID y lo escribimos en salida estándar (que en realidad es el extremo de escritura de un *pipe_hash_to_master*) para que sea leído por el master.

3. Diagrama de Procesos

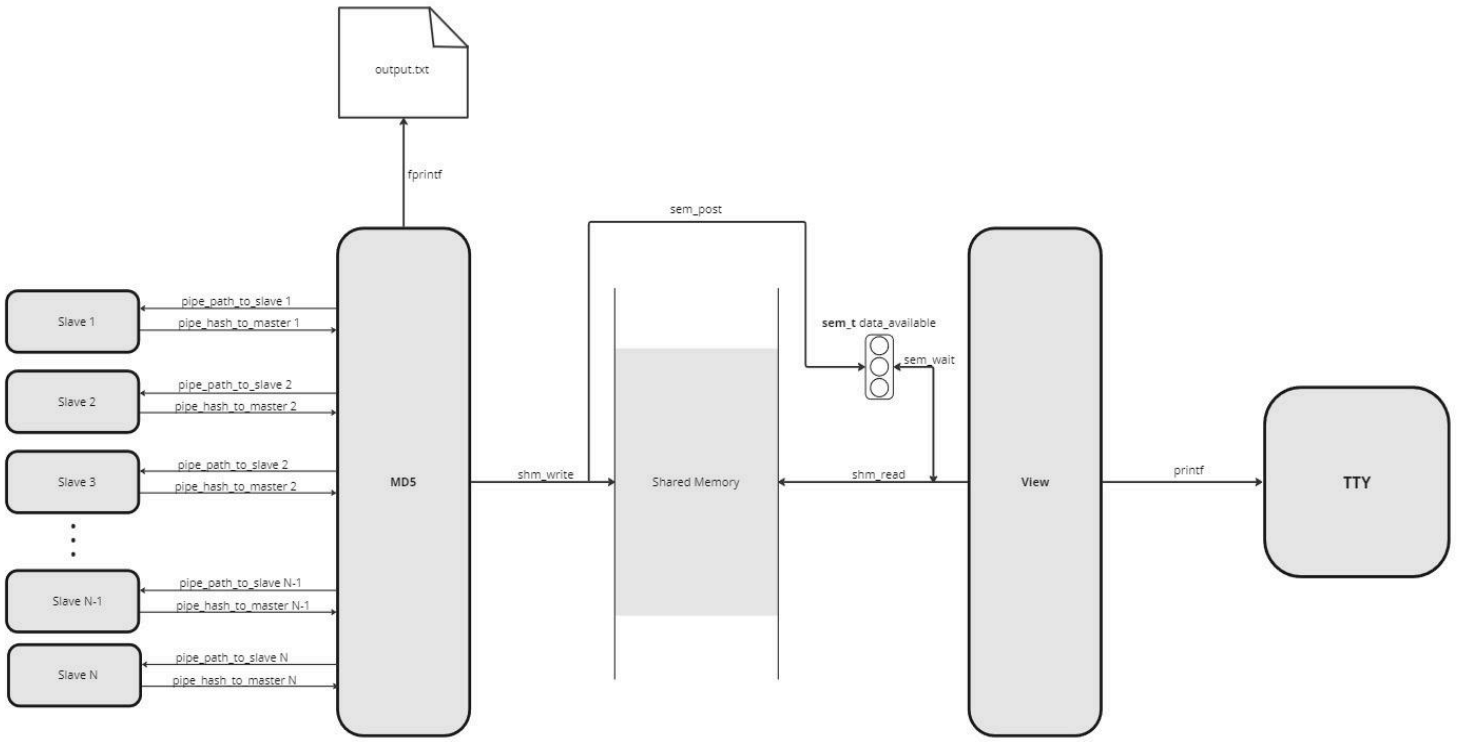

Figura 1. Diagrama del Trabajo Práctico, caso N slaves.

4. Instrucciones de Compilación

El trabajo práctico **debe ser compilado y ejecutado** usando la imagen provista por la cátedra:

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

Para **compilar** el trabajo práctico se debe usar el archivo "**Makefile**" correspondiente, corriendo con el comando:

- *make all*

Para ejecutar el trabajo práctico se puede elegir entre:

- Correrlo en una terminal
  - **Sin vista:** ./md5 files/*
  - **Con vista:** ./md5 files/* | ./view

- Correrlo en dos terminales
    - **Terminal 1:** ./md5 files/*
    - **Terminal 2:** ./view <*info*>
        - Dónde <*info*> en nuestro view es un solo argumento, el nombre de la Shared Memory, la cual será impresa por el ./md5.

5. Limitaciones del Trabajo Práctico

Usamos la cantidad máxima de caracteres definidas en *limits.h PATH_MAX* el cuál es de 4096 caracteres en el container.

Otra limitación, que es autoimpuesta por el grupo, es el tamaño fijo de la memoria compartida, la cual es de *413100* bytes. Consideramos que este tamaño es lo suficientemente grande como para contener toda la salida, puesto que asumiendo un tamaño máximo de 4096 caracteres por PATH cada respuesta tendría un máximo de 4131 caracteres (PATH + hash + PID + espacios). Tomando como partida este número que ya de por sí es muy grande nos permitiría procesar un total de 100 archivos. De superar está cantidad de bytes el md5 hace un retorno y escribe un mensaje de error apropiado. El view termina.

6. Problemas Durante el Desarrollo y su Solución

Principalmente, tuvimos muchos problemas con el manejo de errores. Muchas funciones podrían retornar un error y se estaba volviendo engorroso nuestro código. Para esto decidimos crear una función que cerrara los recursos y haga el exit. Sin embargo, en otros lugares los descriptores a cerrar eran otros, pero nos enteramos que al hacer exit el sistema operativo cierra los descriptores y pudimos emprolijar bastante nuestro código.

Luego al principio no cerrábamos los file descriptors de los esclavos anteriores cuando hacíamos el *fork()* en *md5.c*, pero usando la herramienta *lsof* nos dimos cuenta rápidamente de lo que estaba sucediendo.

También tuvimos problemas al principio porque a pesar de redirigir la salida estándar y la entrada estándar para los procesos correspondientes, no se estaba recibiendo por el otro proceso la información escrita con por ejemplo, printf. Leyendo en el foro nos dimos cuenta que el problema era que no estábamos usando el *setvbuf()* que desactiva el buffer de salida para que los datos se escriban en tiempo real. En ese momento lo resolvimos sin mayores problemas.

En la librería de *shm_lib.c*, no estábamos registrando problemas de sincronización y no encontramos deadlocks, race conditions ni busy waiting. Sin embargo, nos dimos cuenta al final que estábamos usando un semáforo mutex extra, que no era necesario. Puesto que como mencionamos en nuestras decisiones tomadas en el desarrollo, nuestro semáforo *data_available* junto con nuestra implementación de la librería de shared memory evitaban que el md5 y la vista accedan simultáneamente a los mismos bytes del buffer.

Con respecto al view.c, teniamos el problema de que los Paths se mandaban '**n**'-terminated y no podíamos llegar de manera correcta al break del while en el view.c, ya que nosotros usábamos un '**n**' como condición de corte y al nunca leerla, no se cortaba. La solución del equipo fue cambiar la condición de corte, "**END_OF_READ**" a '**0**'. Este error de "**END_OF_READ**" también nos dio una dificultad mayor para conectar el master y el view, ya que él view no sabía por dónde empezar a leer los archivos enviados por el master y finalmente, por nunca salir del while loop dentro del view, la Shared Memory nunca era correctamente cerrada.

Después nos costó mucho resolver algunos errores pequeños. Por ejemplo estuvimos mucho tiempo porque la vista no empezaba a leer hasta que terminara el md5. Nos dimos cuenta que era porque la vista se nos quedaba bloqueada en el fgets para conseguir el nombre de la shared memory, ya que md5 no escribía el '\n'.

Por último, el PVS Studio nos mostraba el siguiente error "Unchecked tainted data is used in expression". Leyendo, observamos que era una vulnerabilidad de nuestro código, ya que corríamos el comando "md5sum <path>" usando *popen()*, donde el path era recibido por entrada estándar (el pipe). De esta manera, como no se valida que la variable path sea efectivamente un path, nos podrían inyectar código para que se corra otro comando. Para arreglar este problema decidimos validar el path usando la función realpath() . El enunciado decía que podemos asumir que todos los path corresponden a archivos existentes y regulares, pero creímos que era una buena práctica validar el path de todas maneras.

7. Citas de fragmentos de código reutilizados de otras fuentes.

Tomamos ciertos esqueletos de funciones para la *shm_lib,* de la clase práctica del 09/09, presentada por el profesor Alejo Ezequiel Aquili.