Instituto Tecnológico de Buenos Aires

TP1: Inter Process Communication

72.11 - Sistemas Operativos

GRUPO N°4

Integrantes:

Aramburu, Paz - 62556
Neimark, Luciano - 62086
Tordomar, Nicolás - 62250

Docentes:

Aquili, Alejo Ezequiel
Godio, Ariel
Mogni, Guido Matías

Fecha de entrega: 17 de Abril 2023

Índice

Introducción

A través de un sistema que calcula el hash MD5 de varios archivos, en este trabajo práctico pusimos en práctica contenidos vistos en la materia como Pipes, procesos, técnicas de IPC, entre otras.

Este trabajo nos hizo comprender aún más sobre la ventaja de qué haya procesos que realizan una tarea trabajando en paralelo, en vez de realizar código secuencial que era lo que se acostumbraba en materias anteriores.

A continuación vamos a explicar el proceso por el que pasamos para llegar a un producto terminado, haciendo hincapié en los problemas que hubo durante el desarrollo, decisiones que tomamos al momento de desarrollar el sistema y las limitaciones que el mismo tiene.

En primer lugar, para comprender un poco mejor sobre cómo funciona el sistema, se presenta una figura donde se ve el flujo de los procesos.

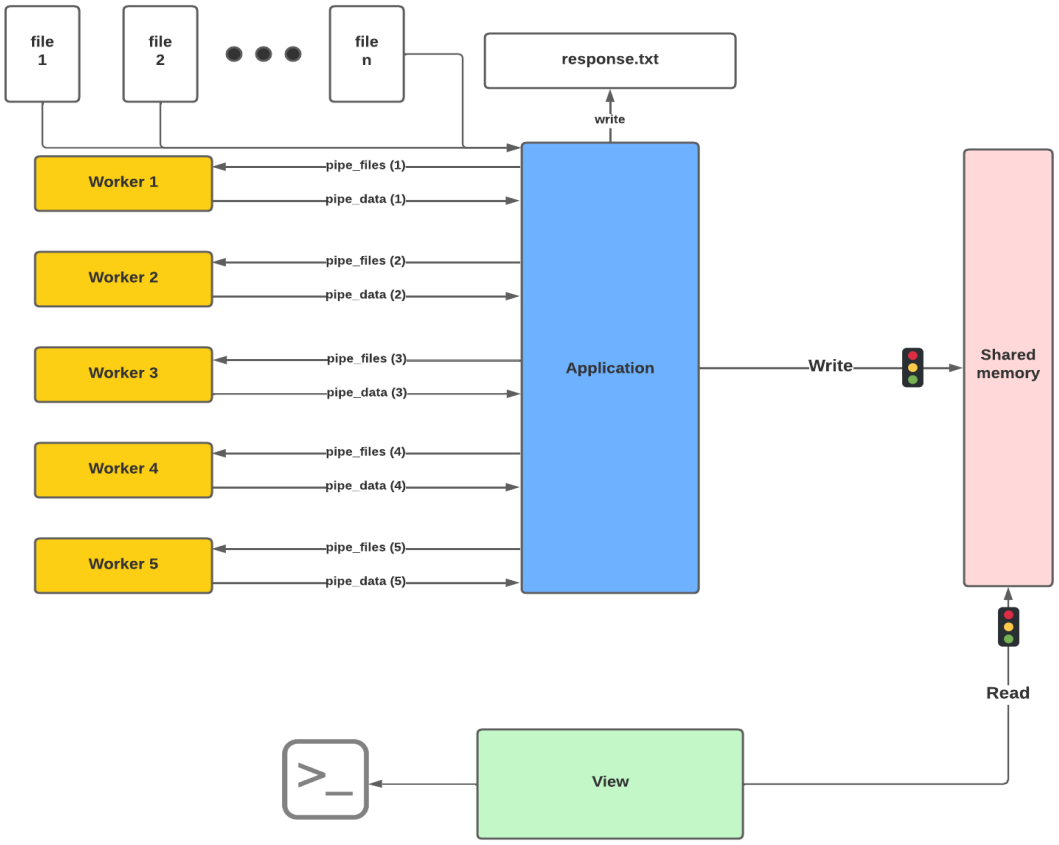

Figura 1: Flujo de los procesos en el caso de que la cantidad de workers es 5

Instrucciones de compilación y ejecución

Aclaración: se asume que el trabajo práctico se va a compilar y ejecutar dentro del container de docker provisto por la cátedra. Si no se cuenta con dicha imagen, se puede descargar desde acá.

1. Para compilar el trabajo práctico se debe correr el Makefile con el comando “make all”
2. Ejecución
    a. Ejecución sin el proceso vista: ./md5 <files>

    b. Ejecución con el proceso vista conectado por pipe: ./md5 <files> | ./vista

    c. Ejecución con el proceso vista corriendo desde otra terminal con parámetros:
    Desde terminal A: ./md5 <files>.
    Desde terminal B: ./vista <arg1> <arg2> <arg3>

    d. Ejecución con el proceso vista corriendo desde otra terminal con valores pasados por entrada estándar:
    Desde terminal A: ./md5 <files>
    Desde terminal B: ./vista
    <arg1>
    <arg2>
    <arg3>

<arg1> <arg2> <arg3> se refiere a los argumentos que enviará el proceso aplicación por la salida estándar.

Decisiones tomadas durante el desarrollo

A lo largo del desarrollo nos encontramos con un montón de opciones distintas para resolver diferentes cuestiones del proyecto. En esta sección del informe vamos a contar qué decisiones se tomaron para poder resolver los problemas. Para comenzar vamos a hablar sobre los mecanismos de IPC.

Al tener procesos trabajadores que se tenían que comunicar con el proceso aplicación y viceversa, podríamos haber utilizado una amplia variedad de mecanismos de IPC, tanto de memoria compartida como de pasaje de mensajes. En esta oportunidad se optó por utilizar un par de pipes anónimos ya que así se pedía desde la consigna.

Otro mecanismo de IPC utilizado en el trabajo práctico fue el de memoria compartida, este se utilizó para que el proceso aplicación pueda comunicarse con el proceso vista si es que esta fue instanciada. El proceso aplicación escribe los resultados en el buffer compartido y el proceso vista accede a dicho buffer para leer la información. Dichos accesos se controlaban con dos semáforos.

El buen funcionamiento de los semáforos garantiza que nuestros procesos estén libres de race conditions. Para lograr esto, se optó por un diseño que utiliza dos semáforos. El primero es un semáforo que el proceso aplicación va a incrementar (sem_post) cada vez que publica información en el buffer de memoria compartida. Luego, el proceso vista decrementa (sem_wait) el valor cada vez que lee un elemento. Esto logra evitar una race condition ya que en el caso de que el proceso view quiera acceder al buffer siempre está obligado a accederlo en una posición de memoria anterior a donde está escribiendo el proceso aplicación y en el caso de que el buffer está vacío (semáforo = 0) el proceso view se va a quedar bloqueado hasta que se ingrese algo al buffer.

Por otro lado el segundo semáforo se utiliza para poder indicar cuando termina el proceso vista ya que esa información es necesaria para poder liberar recursos compartidos entre ambos procesos. Sobre este tema vamos a seguir profundizando en la sección de *“Problemas en el desarrollo”*

Otra decisión que se tomó es como guardar desde el proceso aplicación los valores de los files descriptors asociados a los trabajadores para poder comunicarse con ellos. Para esto se evaluó el uso de un struct pero se optó por utilizar una matriz de 2 x #workers donde guardabamos los dos files descriptors.

En cuanto a cómo recibir la información generada por los trabajadores, creamos un struct que tiene la siguiente estructura

```
typedef struct Response {
  char md5[33];
  int pid;
  char name[100];
} Response;
```

Los 33 caracteres del md5 se definen ya que el md5 siempre cumple con la misma estructura de 32 caracteres + el carácter para el 0, luego el PID se guarda en un int y por último el name del file se guarda en un string de hasta 100 caracteres.

Para decidir la cantidad de trabajadores que se van a ejecutar en nuestro proyecto se establecieron dos casos. El primer caso es cuando la cantidad de archivos que se envía es menor a 5, en este caso se crea un trabajador por archivo para que se ocupe de resolver la tarea. Por otro lado, cuando estamos trabajando con más de 5 archivos, lo que se decidió es que se creen 5 trabajadores para que se ocupen del cálculo del md5.

Una vez creados los trabajadores tuvimos que definir cuál iba a ser la carga inicial que se le asignaba a cada trabajador. Para esto armamos una función matemática que devuelve la cantidad tomando en cuenta que queremos arrancar con el 20% de los archivos divididos en la cantidad de trabajadores que se hayan creado

```
int first_amount = (int) (0.2*real_file_count/num_workers);
```

Problemas encontrados durante el desarrollo

A lo largo del desarrollo nos encontramos con algunos problemas que nos hicieron tener que dar varias vueltas para poder resolverlos. En esta sección detallamos los mismos

Uno de los problemas que más tiempo requirió fue el de leer desde el worker la cantidad de caracteres correcta que nos mandaba aplicación. En una primera instancia intentamos hacerlo con la función read pero nos dimos cuenta que no teníamos como saber cuantos caracteres leer sin tener que hacer dos reads para enviar la cantidad de caracteres primero y luego realizar la lectura. Esta opción quedó descartada ya que hacer dos reads implica hacer el doble de syscalls por lectura y era una alternativa un poco cara.

Luego, se nos ocurrió usar la función getline ya que el proceso trabajador iba a leer desde su entrada estándar (que es el pipe que lo conecta con el MD5) y aprovechar que la misma lee hasta \n.

Otro problema que encontramos fue al momento de utilizar PVS y Valgrind ya que desconociamos los flags que había que utilizar. Este problema se terminó resolviendo ya que al revisar las clases del campus e investigar un poco, encontramos los flags adecuados para correr ambos analizadores de código. Los problemas que reportaba PVS fueron fáciles de entender y resolver ya que proveen una página web que tiene detallados los errores y sus soluciones. Por otro lado, en el caso de Valgrind nos encontramos con errores más complicados de entender pero que terminamos resolviendo luego de algunos intentos y búsqueda.

Otra traba que encontramos fue al momento de usar la función select. Este problema fue causado debido a que era la primera vez que la utilizabamos y para poder resolverlo tuvimos que revisar el manual y dirigirnos a la sección de ejemplos donde pudimos ver cómo se implementa en un caso real y de ahí basarnos para nuestro desarrollo.

Un problema que quizás fue un poco menor pero que también nos consumio tiempo fue al momento de recibir los valores necesarios para que el proceso vista se pueda conectar con el proceso MD5. En el caso donde se recibe por entrada estándar, tuvimos el problema de nuevo de no saber cuánto leer . Esto lo resolvimos utilizando la función de getline() una vez

que lo pudimos resolver en los trabajadores. Esto nos trajo algunas complicaciones porque nos estábamos olvidando de hacer que el string fuera null terminated.

Los últimos problemas que tuvimos fueron con los semáforos. En un principio contábamos con un solo semáforo y hacíamos un sem_destroy para el semáforo y un munmap y un unlink para la memoria compartida desde view.c. Después de pensar un tiempo nos dimos cuenta que como a veces se corría application.c sin view.c, el semáforo y la memoria compartida no se estaban quitando correctamente. Pero en el momento que pasamos esto a application.c, dejó de funcionar el programa. Tanto la memoria como el semáforo nos tiraban error al intentar abrirlas desde view. Imprimiendo el número del semáforo con sem_getvalue y el número del file descriptor de la memoria compartida nos dimos cuenta que estaba tirando números que nosotros no le asignamos. Esto lo solucionamos corriendo el comando rm /dev/shm/*, ya que lo que ocurría era que como en algún momento no se habían cerrado bien el semáforo y la memoria compartida, al intentar acceder desde view tiraba un error porque ya estaban abiertas.

Sin embargo, nos empezó a pasar que en application corria tan rapido que cerraba y hacia el unlink de la memoria y el semáforo antes de que view pueda llegar a abrirlo. Para solucionarlo, creamos otro semáforo y lo llamamos “signal”, ya que daba la señal a application de qué view habia terminado de correr, de manera que pusimos un sem_post al final de el proceso view y un sem_wait de signal justo antes de que application haga los unlink. Acá todo debería haber funcionado bien, pero nos ocurría que al correrlo no termina de ejecutarse nunca. Luego de investigar, nos dimos cuenta que cuando corríamos application y view con un pipe, se hacía de forma secuencial, de manera que se corría todo application primero y luego view, por lo que el semáforo signal trababa toda la ejecución. Esto lo solucionamos con un fflush justo después de mandar los nombres de los semáforos y la memoria por salida estándar de aplicación. Lo que hace fflush es indicarle al proceso (application) que permita que el otro proceso en el pipe (view) pueda ejecutarse.

Limitaciones

En esta sección del informe detallamos las limitaciones que se dieron por las decisiones de diseño tomadas por nosotros a la hora de resolver el TP.

En primer lugar, optamos por una memoria compartida que tiene tamaño fijo por lo que la cantidad de files que puede procesar se establece en 500 files por ejecución.

Luego, al momento de liberar recursos al sistema, nos encontramos con otra limitación que es cuando el usuario corta la ejecución con un ctrl + c. Esta limitación ocurre ya que la liberación de recursos se realiza sobre el final de la ejecución, por lo que si se llegase a interrumpir la misma, nunca llegaríamos a las líneas de código que realizan los frees.

Otra limitación que encontramos es que si ocurre un error en alguno de los esclavos ocurriría un corte en la ejecución de todo el programa. Esto se debe a que el manejo de errores de los esclavos termina en un exit(1) y por otro lado, el proceso aplicación, con el diseño que implementamos, no tendría forma de enterarse que algún trabajador dejó de ejecutar.

Conclusión

En conclusión, en este trabajo práctico se implementó, a partir de lo visto en la teoría, un sistema que calcula el hash MD5 de varios archivos, utilizando técnicas como Pipes, procesos y mecanismos de IPC. Se utilizaron pipes anónimos para la comunicación entre procesos y memoria compartida para la comunicación entre el proceso aplicación y el proceso vista. Además, se implementó un diseño que utiliza semáforos para evitar condiciones de carrera y se creó una función matemática para definir la carga inicial que se le asigna a cada trabajador. En general, este trabajo nos permitió comprender mejor las ventajas de tener procesos trabajando en paralelo en lugar de realizar código secuencial y aprender cómo se puede lograr eso de una forma eficiente.