



**Trabajo Práctico N°1
Segundo Cuatrimestre 2024**

Profesores:

- Godio, Ariel
- Aquili, Alejo Ezequiel
- Mogni, Guido Matías
- Gleiser Flores, Fernando

Integrantes:

- Bonesi, Franco - 64239
- Romanato, Matías - 62072
- Novillo, Valentina - 63156

Fecha de entrega: 16 de septiembre de 2024

Índice:

1. Introducción
2. Diagrama de procesos
3. Decisiones tomadas en el desarrollo
4. Problemas en desarrollo
5. Instrucciones para compilación y ejecución
6. Limitaciones
7. Códigos externos

1. Introducción

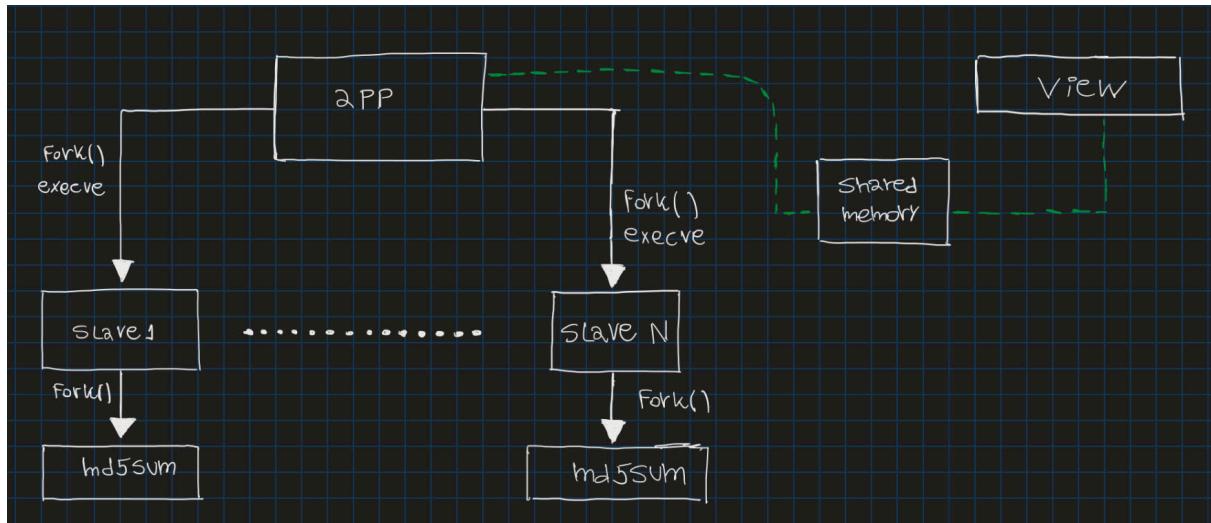
El presente trabajo práctico se centra en la exploración y aplicación de los mecanismos de comunicación entre procesos (IPC) en sistemas POSIX, mediante el desarrollo de un sistema distribuido que calcula el hash MD5 de múltiples archivos. El objetivo principal es distribuir la carga de procesamiento entre varios procesos esclavos y utilizar diferentes IPCs para la coordinación y transmisión de datos entre ellos, así como con los procesos de aplicación y vista.

El sistema se compone de tres procesos principales: la aplicación, que coordina el procesamiento y distribuye las tareas entre los esclavos; los esclavos, que procesan los archivos y envían los resultados; y el proceso vista, que se encarga de visualizar los resultados en tiempo real. Para lograr esto, se emplearon pipes anónimos y memoria compartida con semáforos como mecanismos principales de IPC, garantizando un sistema eficiente y libre de condiciones de carrera o bloqueos mutuos.

Este informe detalla las decisiones tomadas durante el desarrollo, los desafíos enfrentados y cómo fueron resueltos, así como las instrucciones necesarias para compilar y ejecutar el sistema.

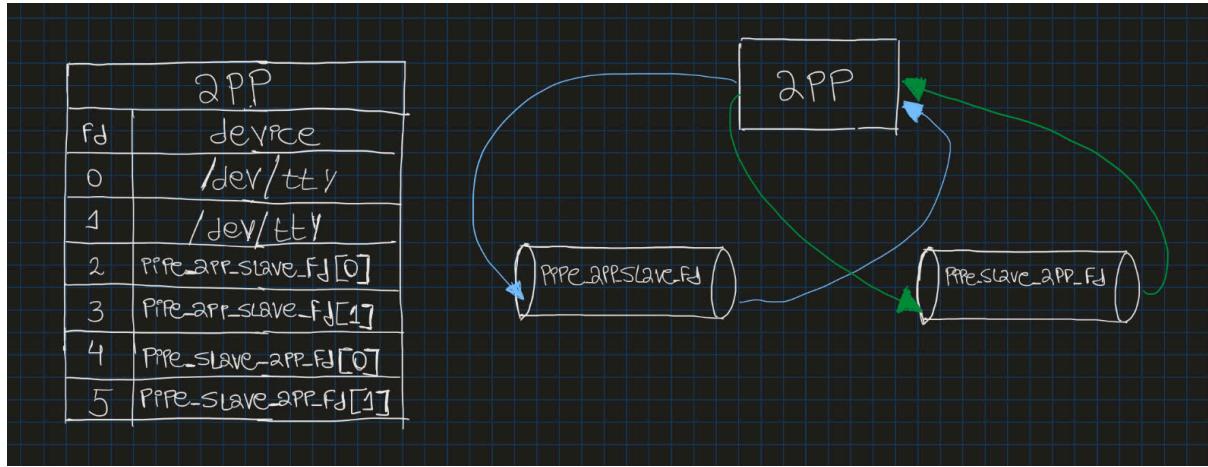
2. Diagrama de Procesos

El proceso aplicación, será mencionado como **app**, el proceso vista, como **view** y los procesos esclavos, como **slave**.

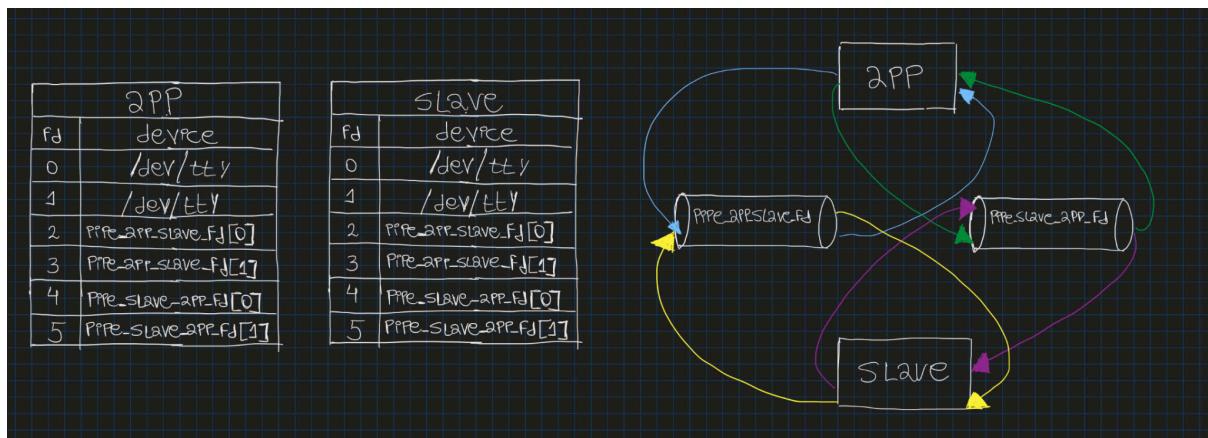


A continuación, se detalla en forma gráfica cómo se realizó el manejo de pipes desde app. Los posicionamientos de los file descriptors en las tablas se colocan para aportar claridad sobre el manejo de procesos y NO hacen referencia a la ubicación real de los mismos.

Al principio, se generan dos pipes en la app, pipe_app_slave_fd y pipe_slave_app_fd:

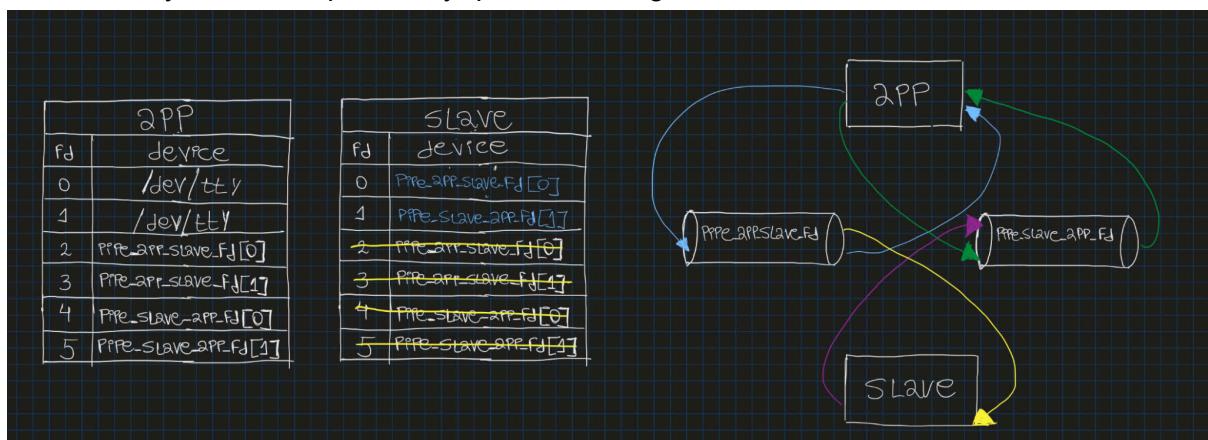


pipe_app_slave_fd representa el pipe por dónde app enviará los paths de los archivos a slave, y pipe_slave_app_fd representa el pipe por donde la app recibirá los md5 computados por slave. Luego, en la función new_baby_slaves(...) se ejecuta un fork() y queda de la siguiente manera:

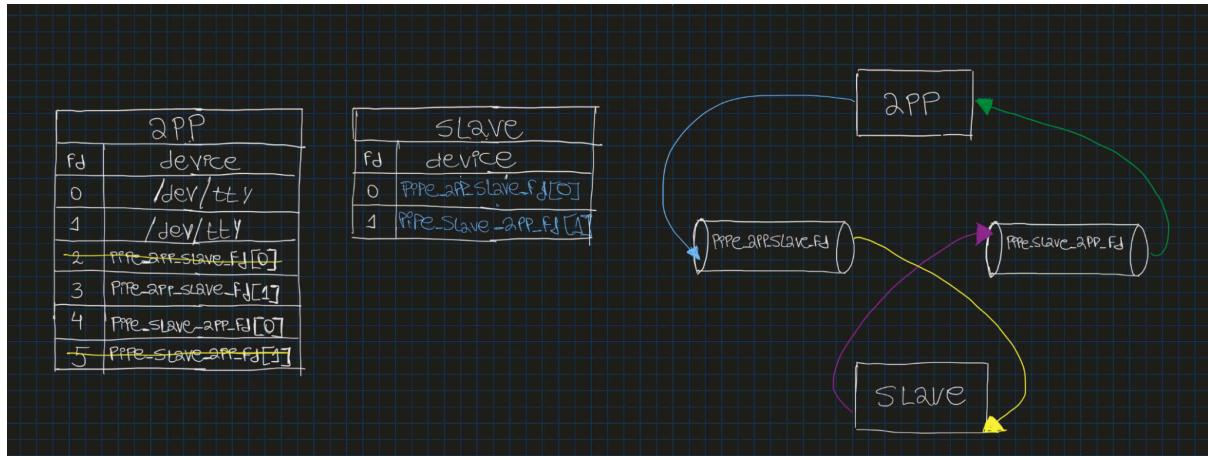


Es entonces que, cuando se está en el proceso hijo, se duplica mediante dup2(...) pipe_app_slave_fd[0] con la entrada de la tabla representada por STDIN_FILENO y pipe_slave_app_fd[1] con la entrada representada por STDOUT_FILENO.

Luego, a través de close(...), se cierran todos los fd dado que los que nos interesaban ya fueron duplicados y queda de la siguiente manera:



Entonces, de la misma manera, se cierran los fd del padre que no vamos a utilizar (pipe_app_slave_fd[0] y pipe_salve_app_fd[1]) y quedan listos los pipes para la ejecución adecuada de los hijos.



Además, al iniciar, la app crea una shared memory, de la cual comparte su nombre por salida estándar, para que el proceso view pueda acceder y leer los md5 computados.

3. Decisiones tomadas en el desarrollo

Durante el desarrollo del TP tomamos distintas decisiones en cuanto a la estructura y funcionamiento del sistema. Las principales fueron:

a) Utilizar un struct para centralizar el manejo de todo lo relacionado a la memoria compartida entre el “app_md5” y el “view”.

Decidimos implementar esta estructura ya que nos pareció la mejor manera de administrar la memoria compartida. Esta estructura contiene un buffer (“buf”), un contador (“total_files”) y un semáforo (“available_files”). El buffer lo definimos para que “app_md5” pueda ir llenándolo con los resultados que los procesos “slave” vayan devolviendo y así “view” pueda recurrir a este buffer para leer los datos e imprimirlos por pantalla. Por otro lado, el contador lo definimos para que el proceso “view” sepa la cantidad de resultados que va a tener que imprimir en total y que no termine antes de imprimirlos a todos. Por último, utilizamos el semáforo para sincronizar la comunicación entre “app_md5” y “view”, ya que necesitábamos que a medida que se vayan procesando los distintos archivos desde “app_md5”, el proceso “view” sepa que hay disponible un dato para imprimir. Además, utilizamos el semáforo para que recién cuando “app_md5” termine de escribir en el buffer, el proceso “view” lea del mismo y no haya un problema de sincronización.

b) Utilización de sem_init(3) en vez de sem_open(3).

Utilizamos esta función, que figura en sem_overview(7), ya que nos resultó más directa para usar y porque pudimos aprovechar la estructura de la memoria compartida y hacer que el semáforo esté almacenado en la misma. Esto nos permitió optimizar el manejo del semáforo ya que así su inicialización y destrucción solo era necesaria de hacer desde “app_md5” y desde “view” podíamos utilizarlo

directamente, una vez mapeada la memoria compartida. Es decir, no hizo falta inicializar y destruir el semáforo desde “view” también.

c) Procesos “slave” leen archivos siempre por entrada estándar.

Decidimos implementar que los procesos “slave” lean siempre de entrada estándar ya que notamos que se podía unificar el comportamiento y no hacía falta diferenciarlo con el caso de que reciban los archivos a procesar por parámetro. Esto lo logramos utilizando pipes que comunican al “app_md5” con los “slave”, mediante el cual únicamente el “app_md5” escribe y los “slave” leen información. Con estos pipes creados, definimos que cada fd que representan sus extremos de lectura sean los devices para cada entrada estándar de los “slave”. Es decir, cada “slave” tiene en su fd 0 (stdin) el extremo de lectura del pipe que lo conecta con “app_md5” como device. Además, fue necesario utilizar la función getline() desde “salve”, a la cual le especificamos por parámetro que lea únicamente por entrada estándar.

d) No mandar una señal para matar a los “slave”

En relación a lo explicado en el punto c), al implementar la comunicación entre el “app_md5” y los “slave” de esta manera, no resultó necesario mandar una señal explícitamente a los “slave” para “matarlos” cuando no haya más archivos por procesar, sino que simplemente cuando estos reciban EOF terminarán su ejecución.

4. Problemas en el desarrollo

Hemos enfrentado distintas complicaciones a la hora de llevar a cabo el proyecto, las cuales desarrollamos en los siguientes ítems:

a) Incorrecto funcionamiento del sem_open(3)

En un comienzo quisimos utilizar la función sem_open para la inicialización de nuestro semáforo. Pero, nos encontramos con que el mapeo para que el proceso “app_md5” y el proceso “view” puedan interactuar con el semáforo tenía problemas. Es decir, no logramos que el mapeo este hecho correctamente. Es por esto que finalmente optamos por utilizar sem_init(3).

b) Decisión en cómo y qué tipo de semáforos utilizar

Un dilema con el cual nos enfrentamos es la decisión de cómo llevar a cabo la sincronización entre el proceso “app_md5” y “view”. Estuvimos bastante tiempo discutiendo y definiendo cómo y cuántos semáforos íbamos a utilizar. Finalmente, decidimos que podíamos utilizar 1 solo semáforo (“available_files”), acompañado por otros componentes para lograr una correcta sincronización.

c) Complicación del proceso “view” para leer desde la memoria compartida

Otro inconveniente fue que en un momento nos figuraba un error que indicaba que “view” no podía leer desde la memoria compartida. Estuvimos muchísimo tiempo tratando de descubrir cuál era el problema hasta que nos dimos cuenta de que en “app_md5”, cuando éste imprime el nombre de la memoria compartida para que el proceso “view” lo lea, estaba haciendo concatenado con un “\n”. Este “\n” lo habíamos puesto para que al imprimir en consola se lea mejor, pero no nos estábamos dando cuenta que correspondía hacerlo después de darle tiempo al “view” para que aparezca y lo lea, sin el “\n”. Es decir, “view” estaba recibiendo que

el nombre de la memoria compartida era “/shm\n” cuando en realidad estaba definida como “/shm”.

5. Instrucciones para compilación y ejecución

Para la compilación del proyecto se creó un archivo Makefile y Makefile.inc el cual proporciona los comandos, **make clean** para borrar los código objetos, **make all** para compilar los 3 programas de forma individual para su uso normal y **make checkval** el cual también compila los 3 programas pero sin el flag -fsanitize=address para usar Valgrind durante la ejecución sin problemas.

Para la ejecución de slave (./slave) no se requieren parámetros y va a esperar por entrada estándar uno o múltiples paths a los archivos a procesar separados por espacios, y continuará su ejecución hasta recibir un EOF. Para el programa app_md5, este espera recibir por lo menos un path a un archivo por parámetro (./app_md5 file1 file2 ...). El programa view necesita recibir por parámetro o entrada estándar el nombre de una Shared Memory válida y se puede ejecutar sólo (./view shm_name) usando el nombre impreso en salida estándar por el programa app_md5 antes de que pasen los 2 segundos del sleep, y pipeado con este mismo programa (./app_md5 file1 file2 ... | ./view).

6. Limitaciones

Las limitaciones encontradas son que los programas asumen que los paths a los archivos son válidos, view asume el formato en el que está la información cargada en el buffer de la shared memory y el tamaño de este es estático, por lo que hay un límite para la cantidad de información que se le puede pasar a view.

7. Códigos externos

No se reutiliza código externo de otras fuentes, sin contar el uso de los manuales de las funciones utilizadas a lo largo del proyecto, y los ejemplos leídos de los mismos.