Trabajo Práctico 1

Sistemas Operativos

Integrantes:
Alfredo Santiago (66601)
Ramiro Luis Garcia (61133)
Simón Marabi (61626)

Año: 2024-1C

Grupo: 12

Profesores: *Ariel Godio, Alejo Ezequiel Aquili, Fernando Gleiser Flores, Guido Matias Mogni*

Decisiones

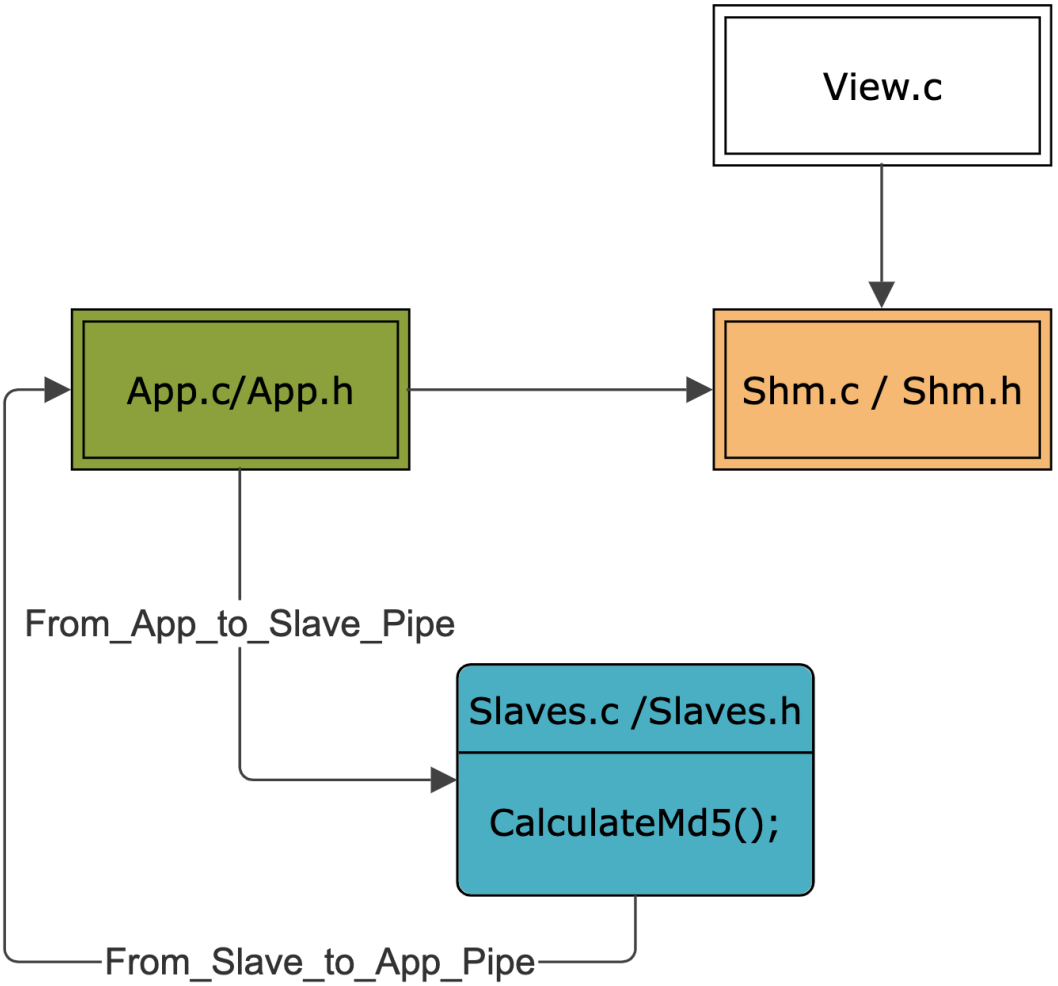
El IPC utilizado para la comunicación entre el proceso app y los slaves fue mediante **pipes bidireccionales anónimos**. Esto se realizó con el objetivo de evitar un 'busy waiting', dado que el proceso padre (app) enviaba información a los hijos (slaves) para procesar el MD5 de los archivos, y no queríamos que el padre se quedara esperando mientras recibía o enviaba información. Para lograr esto, utilizamos un select que monitorea los **file descriptors** de las entradas de información proporcionadas por los hijos al proceso padre.

Para compartir la información recibida por el proceso app al proceso view hicimos uso de **shm posix**, de manera que la creación y conexión de la **shared memory** fuera más sencilla, debido a su fácil implementación y portabilidad.

Para escribir información en la shm hicimos uso de la función **write**, donde se plasman los resultados recibidos por los slaves y un salto de línea, esto con el fin de que cada vez que se llamara nuevamente a la función para escribir más información, el puntero de escritura continuará desde el último lugar alcanzado. De manera similar hicimos la lectura de la información de la shm mediante la función **read**, implementada de manera que leyera byte por byte hasta alcanzar los saltos de línea proporcionados por write.

Finalmente, para la sincronización entre el proceso app (que escribe la información recibida por los slaves a la shm) y el proceso view (que lee la información escrita por app en la shm), utilizamos un **semáforo posix** inicializado en 0, de manera que fuera un contador de la información escrita por el proceso app, de este modo, el proceso view solo puede leer hasta que exista información escrita, de esta manera evitamos race conditions al entrar en una zona de información compartida.

I. Diagrama



II. ¿Cómo compilarlo y conectarlo?

Utilizar el makefile ubicado en la misma ruta que los archivos .c. Para compilar, simplemente ejecutar make en la terminal.

Conexión mediante argumento:

Inicializar el proceso app proporcionando como argumento los paths a los archivos de los cuales buscamos su MD5. Se pueden proporcionar los paths de manera individual, separados por espacios, o mediante el uso de /* para expandir la carpeta de los archivos.

Ejemplo: ./app files/*

Una vez inicializado el proceso app, ya sea en una terminal o en background, inicializar el proceso view en otra terminal o en foreground agregando la información impresa por el proceso app como argumento.

Ejemplo: ./view <parámetro proporcionado por app>

Conexión mediante pipe:

Inicializar el proceso view mediante un pipe con el comando proporcionado a continuación.

Ejemplo: ./app files/* | ./view

Nota: Tener en cuenta que “files” es la carpeta donde están los archivos a ser procesados.

III. Limitaciones

Una de las limitaciones que nos encontramos fue que PVS lo utilizamos al final, de manera que no pudimos hacer chequeo de la calidad del código a la par que elaboramos el trabajo.

IV. Problemas

Al momento de que pasabamos los paths de los archivos desde el proceso app a los slaves del programa, cada uno era enviado con el respectivo path y un carácter nulo al final de cada uno, al llegar al slave hicimos una separación por tokens usando el limitador de carácter nulo, sin embargo, a la hora de entrar a la función que ejecuta el comando md5sum, el path terminaba agregando o quitando letras, de manera que no se pasaba el archivo a procesar. Probamos con cambiar el limitador a espacios en blanco (cambiándolo también desde el proceso app), pero seguía sin funcionar de manera correcta. La solución fue ocupar el carácter de salto de línea.

En el proceso de vista (view), para recibir el parámetro que permite la conexión a la memoria compartida (shm), el método por argumento no tenía problemas, sin embargo, cuando se pasaba la información mediante pipe, creamos una variable de buffer de entrada y, mediante la función fgets, metíamos la información recibida por entrada estándar en la variable. El problema ocurría cuando el tamaño del buffer era mayor que la información recibida, de manera que el proceso view quedaba esperando a que el tamaño del buffer se llenara o el proceso app acabara, teniendo como consecuencia que para ese punto la memoria compartida y el semáforo ya han sido destruidos. La solución fue cambiar la función fgets por read. De esta forma, la lectura se detenía una vez que se encontraba el carácter nulo, evitando así la espera indefinida.

V. Citas de Fragmentos reutilizados de otras fuentes

Msync:

https://man7.org/linux/man-pages/man2/msync.2.html

Strtoul:

https://www.tutorialspoint.com/c_standard_library/c_function_strtoul.htm

fd_set y select:

https://www.topcoder.com/thrive/articles/Linux%20Programming%20-%20Getting%20Started%20with%20the%20Select%20Model