Trabajo Práctico 1 - Grupo 7

(72.11) Sistemas Operativos

Integrantes

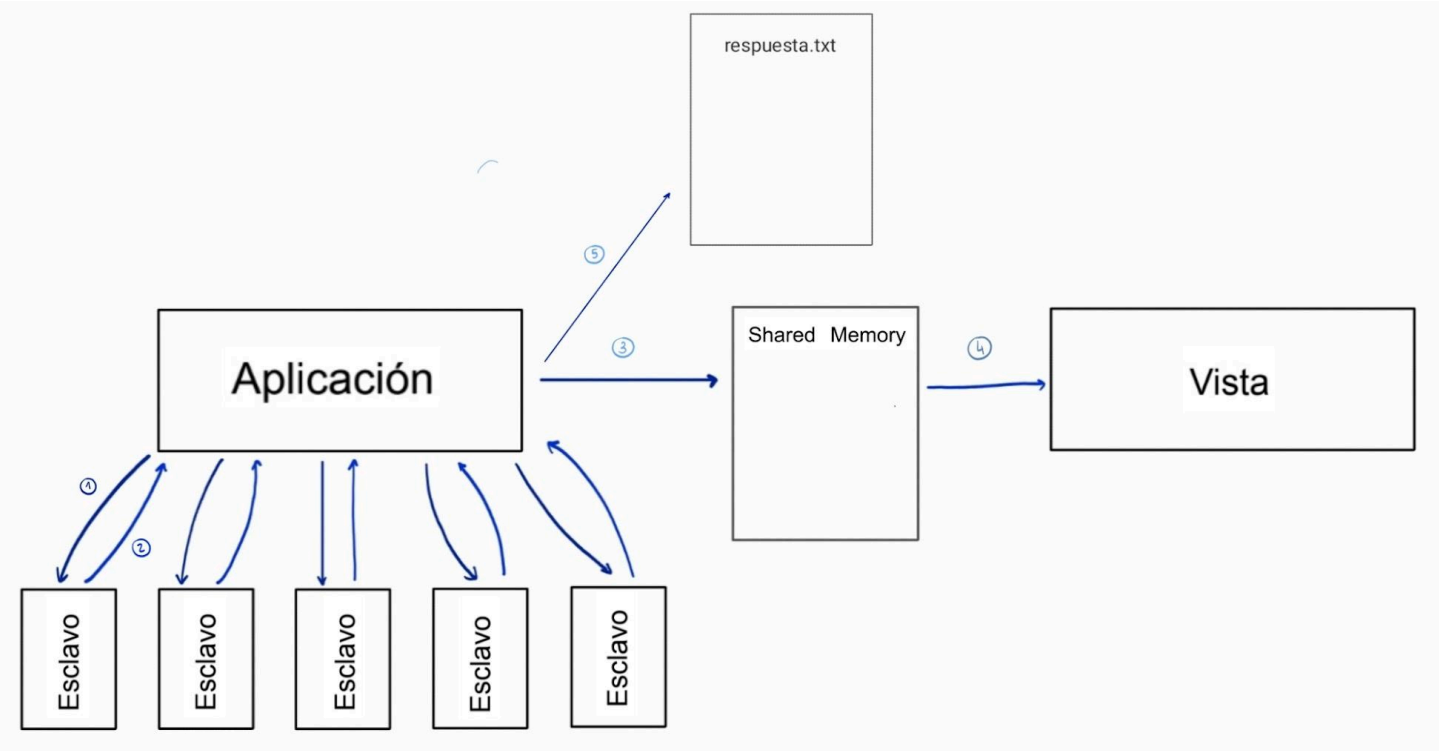
- Martina Schvartz Mallone - 62560
- Josefina Míguez - 63132
- Miguel David Taylor

Decisiones tomadas durante el desarrollo:

En primer lugar creamos el espacio de memoria compartida en *application.c* que luego sería abierta, si es que se conecta, por *view.c*. La memoria compartida permite la comunicación entre ambos procesos: *application* a medida que consigue lo deseado de lo pasado por línea de comando, lo va a escribiendo en esta porción de memoria, mientras que la vista la va leyendo. El método que utilizamos para que *application* le avise a *view* que ya no hay más material para leer, fue enviar un string *TERMINATION* que cuando la vista lo reconociera, termine de recorrer la memoria compartida. Para amortiguar los problemas de sincronización, utilizamos un semáforo llamado *can_read* que incrementa su valor a medida que datos son escritos en la memoria compartida por el proceso padre. Por otro lado, la vista cada vez que consulta un dato de la memoria, decrementa el valor del semáforo: si el mismo indica que hay 0 valores que se pueden leer, se bloquea entonces deberá esperar a que el semáforo aumente su valor (es decir, *application* indicó que hay más información en memoria).

Podría pasar que la vista no se conectara. Para ese escenario creamos otro semáforo con el nombre *check_view*. Este semáforo es creado por la aplicación e inicializado en 0. La vista, si se conecta, es quien se encarga de incrementar ese valor. *Application* hace un *timedwait* en el cual se bloquea hasta que se conecte la vista o por un tiempo determinado en caso contrario. Con esta función nos aseguramos que la aplicación no se bloquee para siempre en caso que la vista nunca se conecte.

Diagrama



1. La aplicación le pasa uno de los argumentos al esclavo
2. El esclavo le devuelve a la aplicación el código md5 y el id del proceso
3. La aplicación ubica lo calculado en la memoria compartida
4. La vista lee de la memoria compartida e imprime
5. La aplicación también escribe lo calculado en un archivo aparte llamado resultado.txt

Instrucciones de compilación y ejecución

Para compilar:
`make`

Para ejecutar:

Con pipes:

```
./app <files> | ./view
```

En terminales separadas[1]:

Terminal 1: `./app <files>`

[1] Observación: En la terminal 1 se imprimirá `/shm_md5`, ignorar.

Terminal 2: `./view /shm_md5`

Limitaciones

Como en nuestra implementación decidimos establecer un número mínimo fijo para la cantidad de archivos que sea el doble de la cantidad de hijos que se van a crear, necesitamos el compromiso del usuario que ejecute nuestro programa con un mínimo de 10 archivos para calcular su código md5.

Problemas encontrados y cómo se solucionaron

Estuvimos bastante tiempo intentando resolver un segmentation fault que resultaba la minoría de veces que ejecutábamos el cual no encontrábamos su origen. Luego de varias técnicas de debugeo, encontramos que el problema residía en que estábamos abriendo el semáforo can_read dando por hecho que primero se crearía en *application*. Notamos que habían situaciones aleatorias, asumimos gestionadas por el Scheduler, en el cual el proceso aplicación abría el semáforo luego de que la vista lo creara con parámetros de *flag* y *mode* con basura. Con eso corregido, el fallo dejó de aparecer.

Observación

Al correr el comando nos dieron 3 warnings que habiendo leído la documentación de pvs studio, decidimos ignorarlos ya que son una falsa advertencia: el uso de un puntero tira el warning aún si la posibilidad de que apunte a null ya fue descartada.