Sistemas Operativos

TP1: Inter Process Communication (IPC)

Lucas Lonegro – 63.738
Tobías Noceda – 63.422
Tomás Raiti – 62.887

Decisiones tomadas durante el desarrollo

Para el uso de la shared memory entre el proceso aplicación y el proceso vista, se tomó la decisión de que el proceso aplicación imprimiera en pantalla el nombre de la misma. Para el nombre se tomó la decisión de que el mismo dependiera del PID del proceso aplicación haciendo así que sea más seguro crear una shared memory y que no existiera una bajo ese mismo nombre. A su vez, se decidió que sea esa misma memoria la que tuviera almacenados los semáforos que permiten la sincronización de los procesos. Para esto se usó un struct que permite a ambos procesos (vista y aplicación) crear y acceder a la shared memory de la forma correcta.

Como se hace referencia anteriormente, la sincronización de los procesos vista y aplicación se da mediante semáforos. Los mismos son necesarios porque ambos procesos acceden a un mismo bloque de memoria. Si bien se suelen utilizar principalmente para evitar problemas cuando dos o más procesos deben ambos leer o escribir en la misma memoria, en este caso, aunque haya sólo un proceso leyendo y uno escribiendo, es necesario la utilización de estos para asegurarse de que el proceso que está leyendo no lea más bytes de los que escribió el proceso aplicación. Para esto, se utiliza un semáforo que incrementa en cada escritura de aplicación y decrementa en cada lectura de vista. Por otro lado, se decidió utilizar otro semáforo para asegurarse de que el proceso aplicación no le cierre la shared memory a vista antes de que termine de imprimir. Para ello, en el proceso aplicación se inicializa un segundo semáforo en 1. Si vista no se "acopla", el mismo permanece en 1 durante toda la ejecución lo que permite al proceso aplicación saber que al terminar de escribir todo puede terminar (estará haciendo un wait de este semáforo). Por el contrario, si el proceso vista si se acopla, al iniciar hace un wait de este semáforo, decrementándolo así a 0. El mismo no vuelve a incrementar hasta que vista no termine su lectura por lo que el proceso aplicación estará esperando hasta que vista le "encienda" este semáforo.

En cuanto al proceso aplicación y sus procesos hijos (esclavos) la comunicación fue mucho más sencilla y se realizó a través de pipes anónimos que el padre creó antes de crear a los hijos. Donde el padre le manda por un pipe el nombre de un archivo, y recibe por el otro la salida deseada y es recién en ese momento que le envía un nuevo nombre.

En cuanto a los procesos vista y esclavo se tomaron 2 decisiones con la intención de independizarlos. En cuanto a esclavo, se tomó la decisión de que cada proceso hijo ejecute el código de esclavo, lea la salida del pipe por el que se comunica el padre como su stdin y escriba en la entrada del que leerá aplicación como si escribiera en stdout. Esto permite que el código del programa esclavo en sí mismo, pueda ejecutarse de forma independiente ya que al ejecutarlo y escribir en la terminal el nombre de un archivo, permitirá obtener el resultado deseado. En cuanto a vista, como simplemente lee una shared memory y los semáforos dentro de la misma, cualquier proceso con un comportamiento similar, una shared memory estructurada de la misma forma y que imprima en pantalla el nombre de la shared memory que creó puede acudir al proceso vista para que este le imprima texto en la terminal.

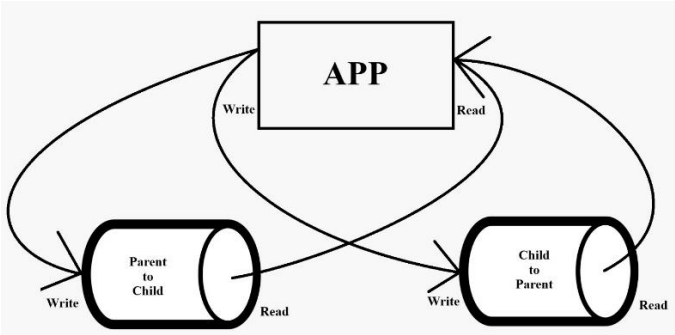
Observaciones

Con la intención de evitar la repetición de código se escribió en aplicación la macro CATCH_IF. La misma es un switch en cascada que según en qué momento ocurrió un error en el código, se entra a la misma con parámetros distintos para liberar todo aquello que sea necesario. Esta parte del código genera problema con PVS ya que dice que el switch podría ser incorrecto, pero consideramos que el “if” previo se encarga de solucionar el problema por más que PVS no lo interprete así.

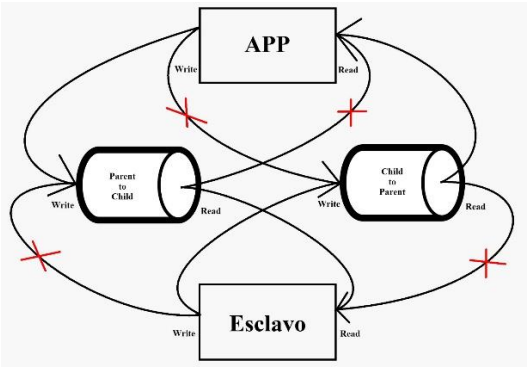
Por otra parte, se decidió que el tiempo de espera para que se conecte el programa vista ascienda a 10 segundos ya que debido a la unicidad y la longitud del nombre de la shared memory creada, se hace muy complicado lograr la conexión antes de que transcurran los 2 segundos.

Diagramas de conexión entre procesos:

App y esclavos:

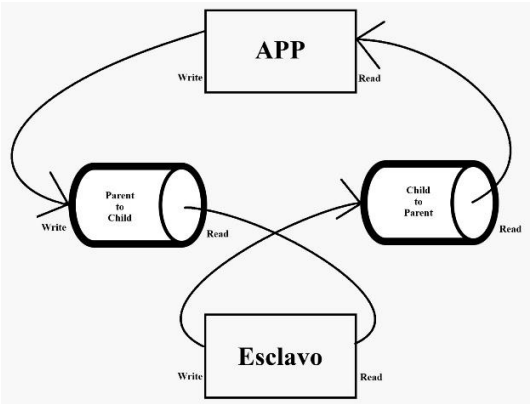


Antes de la creación de los hijos, app crea 2 pipes por hijo.



Al crearse el proceso hijo ambos tienen las mismas conexiones.

Como hay conexiones de sobra, se cierran, los pipes tachados (los dos de arriba los cierra app y los dos de abajo los cierra el esclavo



Conexión final

App y vista:

semData							
semExit							
S	A	L	I	D	A	1	

→ Indica la cant. de data a leer por vista

→ En 1 con vista desconectada 0 con vista conectada

La grilla representa el buffer.

→ Puntero verde en aplicación
→ Puntero rojo en vista

Instrucciones de compilación y ejecución

Para abrir la imagen de Docker

./agodio

Para compilar los archivos:

make

Guarda los binarios en el directorio bin.

Para correr el programa:

Opción 1: (sin vista que podría ir en otra terminal) ./bin/md5 (files)

Opción 2: (con vista) ./bin/md5 (files) | ./bin/vista

Una vez corrido el programa:

make clean

Se encarga de eliminar los binarios, la carpeta bin y el archivo output creado.

Limitaciones

Una limitación que se encontró es que, si el proceso se cancela o elimina antes de tiempo, la shared memory creada no se elimina. Si bien al asociar el nombre de la shared memory al PID del proceso aplicación se evita que esto genere problemas al intentar ejecutar nuevamente la aplicación, estas shared memories quedan obsoletas lo cual no es ideal.

Problemas encontrados y sus soluciones

El mayor problema que nos encontramos fue la sincronización entre vista y aplicación, principalmente para que la aplicación no cierre la shared memory antes de tiempo a vista. Pero en cuanto a eso le encontramos esta solución con la inclusión de un segundo semáforo creado exclusivamente para asegurarse de que eso no ocurra (como se explicó anteriormente).