TP1 - Interprocess Communication: Informe

Integrantes:

- Alejo Caeiro (60692)
- Kevin Catino (61643)
- Román Gómez Kiss (61003)
- Tomás Alvarez Escalante (60127)

Grupo: 2

Fecha de entrega: 31/3/2022

Decisiones tomadas durante el desarrollo

Manejo de git

Decidimos que un integrante del grupo solamente haga los commits al repositorio, y trabajar a través de la extensión de Visual Studio Code llamada *LiveShare* para poder trabajar de forma colaborativa y tener una buena división de tareas.

IPC y sincronización

Se utilizaron *unnamed pipes* para comunicar al proceso principal con los esclavos, y se utilizo *shared memory (System-V* dado que fue visto en la teórica*)* junto a *named semaphores* para comunicar el proceso principal con el proceso vista.

El proceso principal se encarga de crear el shared memory y la vista accede a ella para leer los resultados del procesamiento e imprimirlos. Para sincronizar el acceso a ellos, el semáforo se inicializa en cero desde el proceso principal, se incrementa con `sem_post()` cada vez que se escribe un resultado en la shared memory, y es decrementado luego por la vista cuando este los va leyendo.

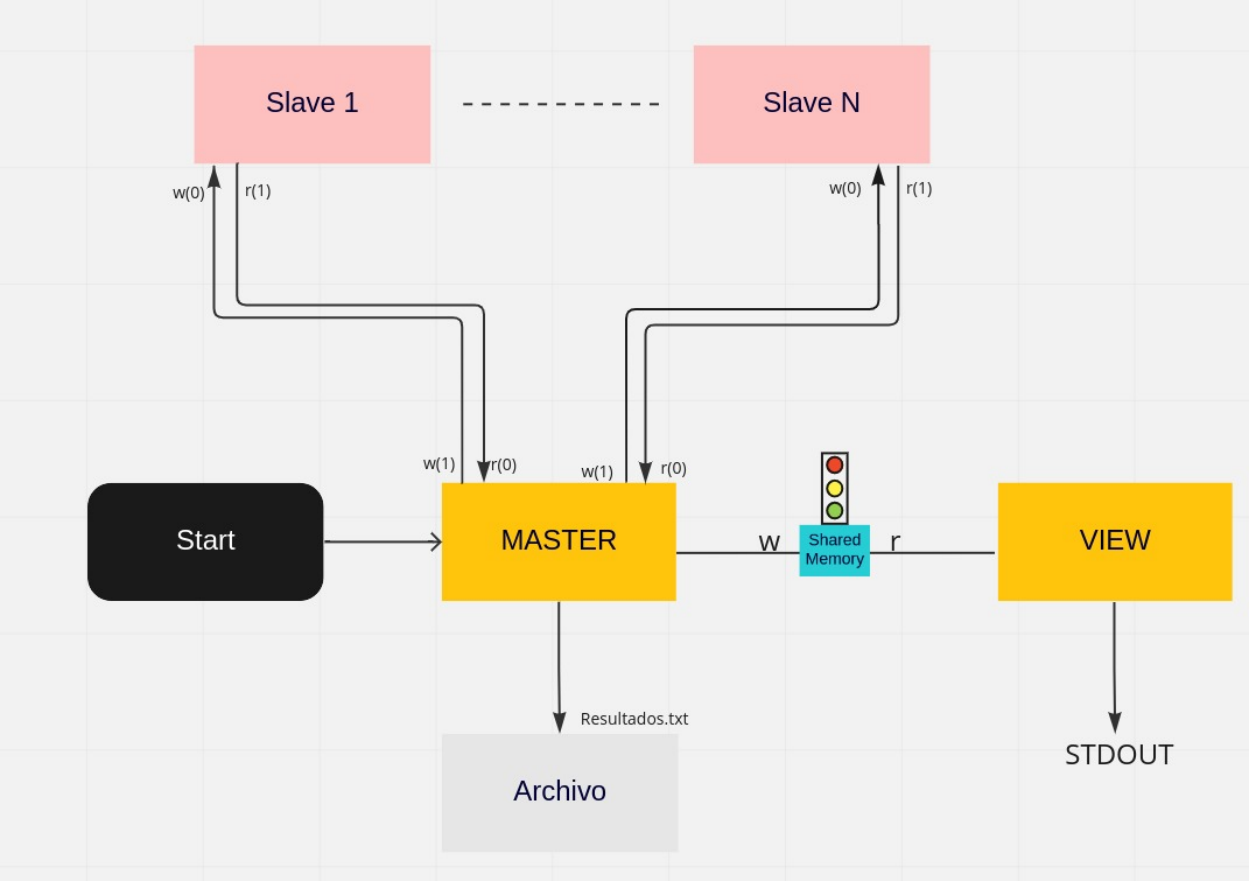
Cantidad de pipes

Como los pipes son unidireccionales, concluimos primero que deben usarse como mínimo 2, pero dado que tenemos varios procesos esclavos que pueden llegar a leer simultáneamente del mismo pipe (y la lectura es sólo atómic hasta 1 byte), es necesario tener un pipe por esclavo a través del cual leerán las tareas a procesar. El manual de linux garantiza la atomicidad del write hasta un cierto numero de bytes (PIPE_BUF=4096 bytes), por lo que podríamos a haber usado un único pipe desde el lado del proceso principal para leer los resultados del procesamiento. Sin embargo, decidimos optar por utilizar 2N pipes (donde N es el número de procesos esclavos) ya que nos pareció más ordenado y claro para implementar. Esto generó la necesidad de utilizar la función `select()` para poder desde el proceso principal elegir entre todos los pipes uno con una tarea procesada pendiente (de no haber hecho esto, podríamos haber caído en un *busy waiting* donde el padre espera a un pipe en específico cuando otros procesos ya terminaron y mandaron su tarea procesada).

Otras decisiones de implementación

- Decidimos que habrían 4 procesos esclavos, cada uno de ellos teniendo asignado 2 tareas inicialmente (todo se puede modificar desde los `#define` en el encabezado del código).
- Tuvimos que utilizar `getline` dado que si usamos `read` desde los slaves, era necesario hacer un close del pipe para que este finalice con un EOF. El problema es que debemos enviar más de una tarea a cada esclavo cuando ellos terminen su tarea asignada, y hacer un close entonces no es adecuado.
- Para la shared memory, decidimos no hacer uso de `ftok` junto al file system, y utilizar directamente la key devuelta por `shmget` para acceder a la memoria compartida.
- Por último, decidimos modularizar el proyecto a través del uso de TADs, uno para el manejo de los procesos esclavos, la asignación de las tareas y el proesamiento de su resultado y otro para el manejo de la memoria compartida. Esto nos permitió dividir mejor el código y mejorar su comprensión durante el desarrollo del trabajo.

Diagrama



Limitaciones

- Pueden procesarse hasta 99 archivos con 256 caracteres cada salida de procesamiento debido a las características de la estructura definida en la memoria compartida.
- La key generada al crear la memoria compartida no debe ser mayor a 20 caracteres.

Compilación y ejecución

Se debe compilar haciendo `make all` desde la carpeta “TP1-SO”, pudiéndose además hacer un `make clean` para borrar los archivos generados y permitir un recompilado del proyecto.

Para ejecutar el programa, estando en la “TP1-SO” hay dos alternativas:

Alternativa 1 de ejecución

```
./app.out <archivos a procesar> | ./view.out
```

Alternativa 2 de ejecución

```
./app.out <archivos a procesar>
```

y durante la ejecución del programa, ejecutar en otra terminal lo siguiente:

```
./view.out <ID>
```

donde <ID> es el número que se imprime en la primera terminal al ejecutar `./app.out`

Inconvenientes durante el desarrollo

- Estuvimos varias horas con el proyecto sin funcionar debido al error en el uso de `&&` en lugar de `&` para la manipulación de flags.
- Inicialmente, de forma incorrecta, destruíamos la shared memory desde el proceso principal antes de que la vista pudiera acceder a ella y leerla.
- No estábamos seguro de cómo podríamos indicar al proceso vista hasta cuándo debe continuar leyendo la memoria compartida una vez que tenía el acceso a ella. Finalmente decidimos pasar como primer item por memoria compartida la cantidad total de archivos que fueron procesados, de forma similar a como en el pipe se indica primero la cantidad de caracteres a leer y luego el mensaje.
- Tuvimos dificultades para la creación de la memoria compartida que nos tiraba errores, lo cual terminó solucionándose al investigar los flags disponibles en `shmget()` y la función de cada uno.
- No teníamos muy claro cómo organizar el espacio de memoria compartida para acceder a ella de forma ordenada y con una lectura y escritura correcta. Se nos ocurrió incluir dentro del TAD de memoria compartida la definición de una estructura en forma de un arreglo de líneas, para que la escritura y la lectura sea

simple: cuando el proceso principal escribe, lo hace en una línea del arreglo y aumenta el contador de líneas, y lo mismo ocurre en la lectura desde el proceso vista. Así cada tarea se encuentra en una línea diferente del bloque.

- No teníamos muy claro cómo sincronizar el acceso a memoria compartida, y habíamos pensado inicialmente en hacer un `wait()` y `post()` desde ambos procesos, utilizándo semáforos binarios para que ambos procesos no puedan acceder a la memoria compartida a la vez. Luego, tomando los principios del problema del “proveedor y consumidor”, optamos por únicamente hacer un `wait()` desde el proceso vista y un `post()` desde el proceso principal, dado que lo único importante a la hora de hacer la lectura era que la línea siendo leída ya haya sido escrita por el proceso padre en algún momento previo, pudiéndose acceder al mismo segmento de memoria siempre y cuando el puntero de lectura sea menor al de escritura.
- Erróneamente estuvimos haciendo `wait()` de los esclavos antes de haber cerrado un lado del pipe desde el proceso principal, esto generaba que los procesos esclavos nunca terminaran la parte de lectura en su ejecución y por lo tanto nunca finalizaban. Cambiar el orden de estas funciones solucionó el problema.

Justificaciones de análisis estático y dinámico

PVS Studio

Al correrlo se obtiene la siguiente advertencia:

> *`/home/khcatino/SO/TP1-SO/src/sh_mem_ADT.c 52 warn V522 There might be dereferencing of a potential null pointer 'sh_mem_handler'. Check lines: 52, 47.`*

El programa considera la posibilidad de que el puntero obtenido al hacer `calloc()` sea `NULL`, y maneja el caso correctamente, por lo que la advertencia es un falso positivo:

```
sh_mem_ADT sh_mem_handler = calloc(1, sizeof(sh_mem_CDT));
```

```
    if (sh_mem_handler == NULL)
        error_exit("Error allocating memory",  MALLOC_ERROR);

    sh_mem_handler->flag = flag;
```

Valgrind y Cppcheck

No se obtienen mensajes de advertencia.