Instituto Tecnológico de Buenos Aires

72.11-Sistemas Operativos (SO)

Trabajo Práctico N°1

Inter-Process Communication (IPC)

Profesores

Merovich, Horacio (Responsable)

Aquili, Alejo Ezequiel (ATP)

Godio, Ariel (ATP)

Mogni, Guido Matías (ATP)

Alumnos

Juan Ignacio García Matwieiszyn (61441)

Leandro Ezequiel Rodriguez (61069)

Jerónimo Brave (61053)

Índice

Introducción

Este informe refleja nuestro aprendizaje sobre los distintos tipos de IPCs (Inter-Process Communication) presentes en un sistema Posix. Pudimos implementar con éxito un sistema que distribuye tareas de SAT solving entre varios procesos tal como lo pide el enunciado, utilizando los conceptos de semáforos, share memory, fork(), exec() y pipes vistos en clase.

A su vez, la implementación permite en tiempo de compilación optar por cuál de las APIs de Share Memory implementadas utilizar. Si bien no era parte del enunciado, al haber comenzado implementando una API para System V y darnos cuenta del error, decidimos no eliminar este desarrollo y, dado que la estructura de nuestro proyecto no presenta acoplamiento, permitirle al usuario decir cual usar.

Cabe mencionar que utilizamos tanto git como las extensiones de Visual Studio Code y CLion de edición de código compartido[1]. Así pudimos trabajar en simultáneo en los mismos archivos sin encontrarnos con conflictos de git a la hora de hacer push o merge, y al mismo tiempo dividirnos en distintos milestones en diferentes ramas de git.

Instrucciones

Compilación:

Para compilar el proyecto es necesario tener ***minisat*** y ***tr*** instalado en el entorno que se esté trabajando, ya sea en docker o de forma local, por lo que primero recomendamos correr los siguientes comandos en la terminal:

```
$ apt install minisat
$ apt install tr
```

De solicitar permisos de sudo, debe cambiar los comandos anteriores por los siguiente:

```
$ sudo apt install minisat
$ sudo apt install tr
```

Tras haber instalado ***minisat*** y ***tr***, estamos listos para decidir qué API de Share Memory utilizar. Para ello hacemos uso de lo implementado en el archivo Makefile:

- Si se deseara utilizar la implementación de Posix, se debería correr:

```
$ make rebuild_posix
```

[1] Esto se ve representado en la discrepancia de líneas de código modificadas en git. La cual no representa directamente la actividad de cada integrante.

- Si se deseara utilizar la implementación de System V, se debería correr:

```
$ make rebuild_sysv
```

Una vez se haya corrido el comando deseado, se habrán generado en el file system ambos ejecutables, del proceso vista y el proceso aplicación, bajo los nombres ***p_master*** y ***p_view***.

Ejecución:

Tras haber creado ambos ejecutables, se debe decidir de qué forma ejecutar el sistema. Siguiendo el enunciado, hay dos formas posibles de ejecutar el mismo. La primera es ejecutando el padre pasándole como argumento el/los archivo/s a ejecutar y utilizar un pipe para que redireccione la salida a la vista de la siguiente forma:

```
$ ./p_master <files> | ./p_view
```

Por otra parte, si se desea, también se podría ejecutar por separado. En una terminal se debería correr el siguiente comando obteniendo por salida estándar dos códigos que representan el nombre identificador de la shared memory y del semáforo :

```
$ ./p_master <files>
0x12345678
0x54321012
```

El proceso padre en este punto está esperando que el proceso vista se conecte desde otra terminal. Si este proceso no se conecta tras un corto lapso de tiempo, el resultado del procesamiento de los archivos pasados por parámetro se plasmará únicamente en el archivo ***resultado.txt***. Tras aguardar unos breves segundos si la vista no se conectó el programa padre terminará con éxito.

Si el proceso vista, en cambio, logra conectarse desde otra terminal a la shared memory utilizando los códigos provistos por el padre antes de que este termine mediante el comando:

```
$ ./p_view 0x12345678 0x54321012
```

No solo el padre escribirá en el archivo mencionado anteriormente, si no que el proceso vista se asegurará de imprimir por salida estándar el resultado de cada archivo procesado, permitiéndole al usuario ver en la terminal lo escrito por el proceso aplicación en el archivo. La salida del proceso vista, si los archivos a evaluar son .cnf bien formados, debería lucir de la siguiente manera:

```
45941    FILENAME: files/s3.cnf    Number of variables: 2
Number of clauses: 0     CPU time: 0 s  UNSATISFIABLE
```

```
45942    FILENAME: files/s4.cnf    Number of variables: 40
Number of clauses: 79    CPU time: 0 s  UNSATISFIABLE
```

Decisiones tomadas

Diseño

Al momento de implementar la shared memory, se optó por diseñar un TAD que maneje una instancia de recursos compartidos, así se escribió el archivo header del cual luego se implementaron las funciones pertinentes.

La memoria compartida se crea o se abre, se puede leer y escribir (con la ayuda de un semáforo desarrollado de manera interna que evita condiciones de carrera, busy waiting y deadlocks), como también cerrar esta memoria una vez se termina de usar. La memoria compartida cuenta con dos punteros, uno de lectura y otro de escritura que manejan las funciones de read y write, con estos dos punteros se asegura que nunca se tratara de escribir a una zona de la cual se está leyendo como también solo leer lo que ya está escrito.

El semáforo mencionado es inicializado en 0. Se incrementa ante cada write (donde internamente se realiza un ***sem_post()***) y decrementa ante cada read (donde internamente se realiza un ***sem_wait()***), esperando en caso de estar en 0 (diagramado abajo).

Al diseñar el proceso vista, se tuvo en cuenta si este recibía parámetros o no para, de esta forma, determinar si se debían leer los argumentos - es decir, los nombres de la shared memory y del semáforo - por entrada estándar o por parámetro. Estos eran fundamentales para permitirle al proceso vista conectarse a los recursos compartidos. Una vez conectada, se tuvo que leer de la memoria la cantidad de archivos escrita en el primer lugar de la misma por el proceso aplicación para luego, dentro de un ciclo, hacer un read de cada una de las salidas de minisat ejecutadas por los esclavos hasta que no hubiera más resultados por leer.

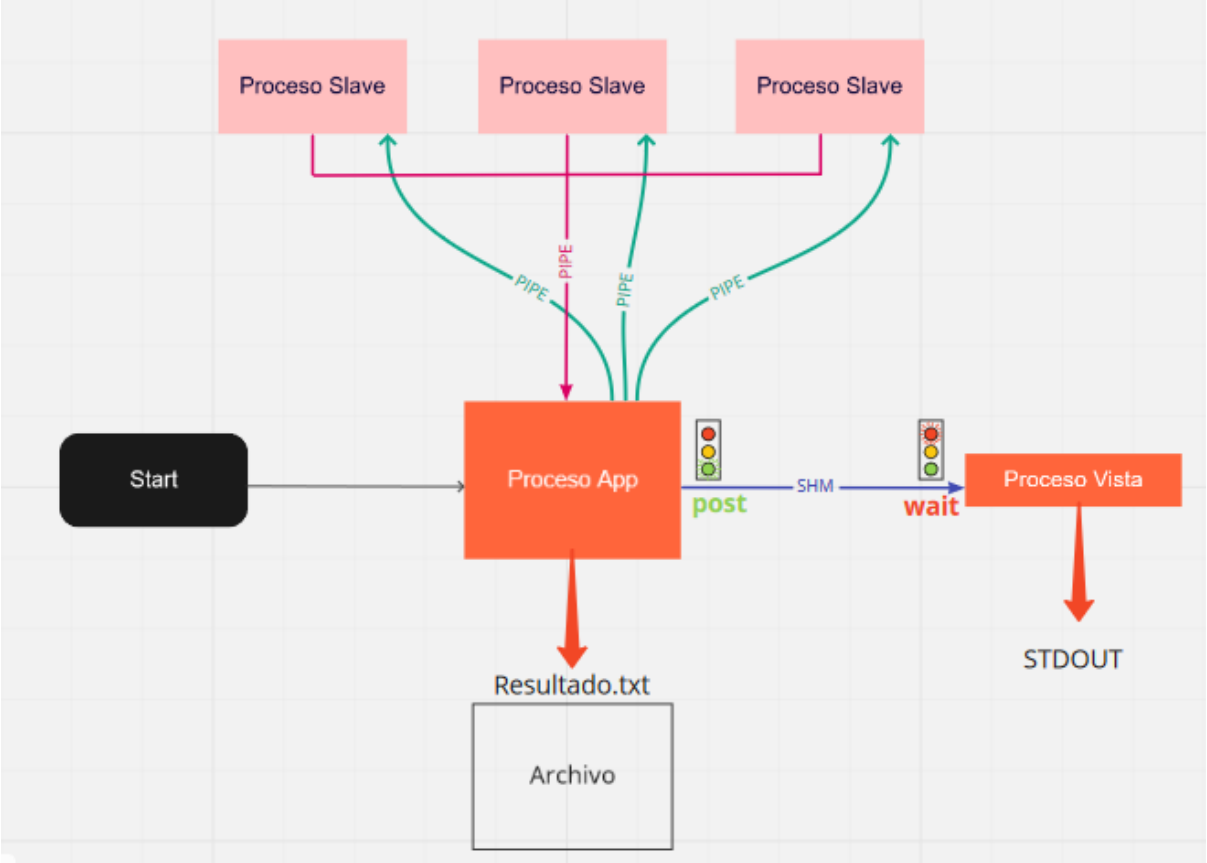
Por otra parte, en el proceso aplicación nos encargamos de inicializar los recursos compartidos (semaforo y shared memory) bajo un nombre arbitrario, abrir los pipes tanto de lectura para cada esclavo como el pipe de escritura compartido por todos, crear los procesos esclavos mediante un fork() y un exec() para luego cerrar los extremos correspondientes del pipe y enviarle a cada esclavo su respectivo archivo a procesar. También se implementó un TAD para almacenar la información de cada esclavo (PID y descriptor de archivo). A medida que los esclavos van procesando sus archivos, estos vuelven a recibir de a un .cnf a la vez.

Diagrama

Como vemos en el diagrama de nuestra implementación, optamos por utilizar un único pipe de escritura de los esclavos al proceso aplicación y un pipe de lectura por cada esclavo. Esto es porque los esclavos pueden escribir atómicamente al pipe si dicha escritura tiene menos de una cantidad determinada de bytes.

Se hizo uso de un único semáforo para administrar el control sobre la shared memory y que no se permita acceder al mismo lugar que se está escribiendo. El proceso aplicación hará un post luego de haber escrito en la memoria, permitiendo en cada llamado a read de parte del proceso vista a salir del wait en el que se encuentre para poder leer una de las respuestas. Al estilo puntero escritura / lectura que implementa linux para manejar sus buffers.

Como podemos ver en el diagrama, es el proceso aplicación el que escribe el resultado en el archivo correspondiente y el proceso vista el que escribe en salida estándar.



Limitaciones

Una de las limitaciones que presenta nuestra implementación es un valor fijo para el tamaño del buffer de la shared memory. El mismo no se basa en la cantidad de archivos recibidos por argumento sino que es un valor representativo de gran tamaño.

Este buffer se puede modificar fácilmente por uno circular de necesitarlo en un futuro. Así el programador deberá seguir las indicaciones del README.md para evitar pérdida de información por salida estándar. De todas maneras en el archivo results.txt mantiene toda la información siempre en cuando haya memoria disponible para el tamaño del archivo.

Problemas encontrados

Algunos de los problemas encontrados a la hora de implementar esta solución fueron los siguientes:

1. Nos dimos cuenta que no estábamos cerrando correctamente varios descriptores de archivos. Para solucionarlo debimos crear una función que cierre todos los pipes abiertos hasta el momento para evitar problemas con la lectura.
2. No es una dificultad puntual pero una de las cosas que más se hizo fue recurrir al ***man*** y leer el mismo para ver como implementar todos los diferentes métodos de IPC. Muchas veces se terminó leyendo el mismo manual varias veces por falta de atención o malas interpretaciones, al igual que por falta de costumbre de utilizar el manual como recurso.
3. Uno de los mayores problemas que nos llevó bastante tiempo resolver fue un error que teníamos con la lectura de la shared memory desde el proceso vista. Esto se debía a que los flags que le estábamos enviando a la función para acceder a la memoria no eran correctos; debíamos enviarle permisos O_RDWR, ya que se le debía permitir a este proceso no solo leer si no también modificar los valores del puntero de lectura que accedía al buffer de la memoria. Nos costó encontrar ese error, pero con la ayuda del debugger de CLion pudimos notar que la causa del segmentation fault generado era por tratar de modificar el r_pointer cosa que requería de permisos de escritura.

Bibliografía

- **https://github.com/bradfa/tlpi-dist/**