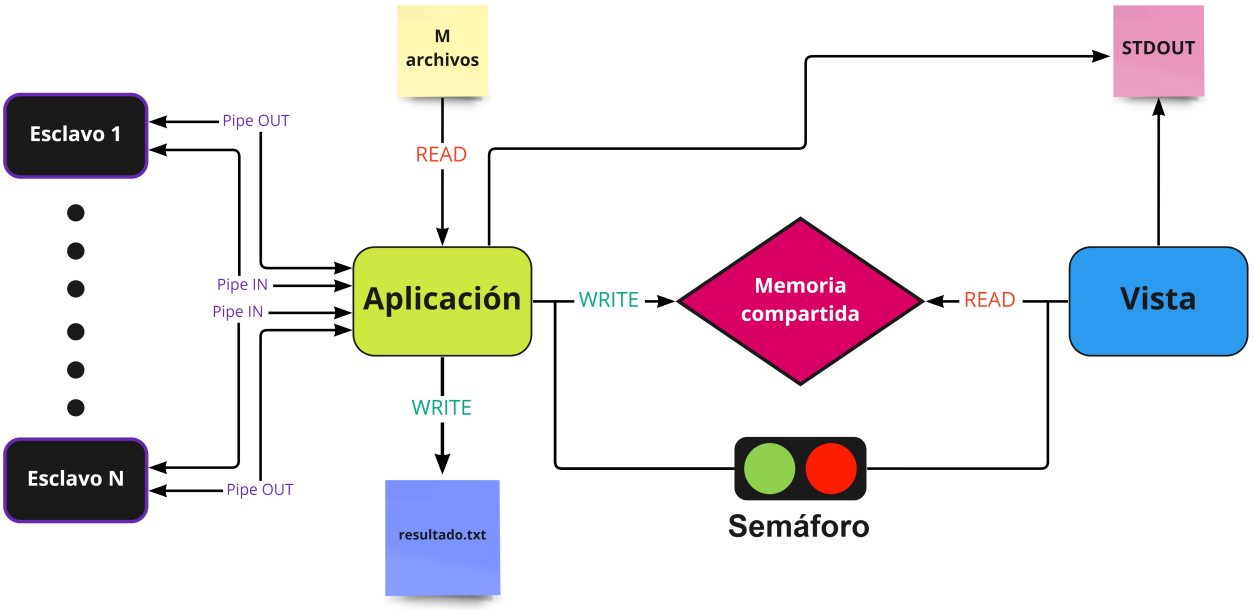
Trabajo Práctico Nro. 1

Ferreiro, Lucas Agustín (61595)
lferreiro@itba.edu.ar

Di Toro, Camila (62576)
cditoro@itba.edu.ar

Chayer, Iván (61360)
ichayer@itba.edu.ar

Inter Process Communication



72.11 Sistemas Operativos
Instituto Tecnológico de Buenos Aires

Indice

Instrucciones de compilación y ejecución

Contenedor Docker

Para construir la imagen de Docker y poder correr el proyecto, se deben ejecutar los siguientes comandos:

```
chmod +x ./docker.sh
./docker.sh build
./docker.sh run
```

Luego de haber hecho esto debería correrse el container:

```
root@3d7b38678779:/code#
```

Compilación y ejecución

Una vez dentro del contenedor de Docker, para compilar y crear los ejecutables se debe correr:

```
make all
```

Este comando generará los ejecutables solve, slave y view en la carpeta **src**. Luego, para correr la aplicación, entrar al directorio mencionado y elegir una de las siguientes opciones:

Opción 1

Utilizar un pipe para iniciar el proceso aplicación y enviar su salida como entrada al proceso vista:

```
./solve files/* | ./view
```

Opción 2

Iniciar primero solamente el proceso aplicación. Luego ejecutar el proceso vista, pasándole manualmente (como parámetro) el valor que imprime en pantalla el proceso aplicación.

Forma 1: en la misma terminal, enviando el proceso aplicación al background.

```
./solve files/* &
>> <files_quantity>
./view <files_quantity>
```

Forma 2: utilizando dos terminales.

Terminal 1

```
./solve files/*
>> <files_quantity>
```

Terminal 2

```
./view <files_quantity>
```

En cualquier caso, al finalizar la ejecución del proceso aplicación se creará un archivo llamado **results.txt**, con la resolución de los archivos procesados. Si el proceso vista lograra conectarse antes de que el proceso aplicación haga un *shmClose()* el mismo resultado se obtendrá por salida estándar.

Testeos

Se realizaron pruebas con PVS-Studio, Cppcheck y Valgrind existentes en la imagen provista por la catedra.Corriendo el siguiente comando podemos obtener comentarios de PVS-Studio y cppcheck:

```
make check
```

Podremos encontrar los respectivos resultados de cada análisis en la carpeta **check** (dentro de **src**):

- Cppcheck: cppout.txt
- PVS-Studio: report.tasks y strace_out

Nota: para testear con Valgrind recomendamos el uso del siguiente comando:

```
valgrind --leak-check=full --show-leak-kinds=all
--track-origins=yes -v ./solve <samples_folder>/* >
./view  2> ./check/pipe.valgrind
```

Clean

Corriendo el siguiente comando se podrá eliminar la carpeta **check** con los archivos generados de los análisis, los ejecutables y el archivo **results.txt**:

```
make clean
```

Decisiones tomadas durante el desarrollo

En la portada podemos apreciar un diagrama de implementación del trabajo. A continuación, comentaremos brevemente como las distintas partes realizan su trabajo para obtener el resultado esperado.

Docker

Nos pareció interesante aprender a generar nuestra propia imagen de Docker para que los integrantes del grupo puedan trabajar bajo las mismas condiciones.

Pipes

Para lograr la comunicación entre el proceso aplicación y los procesos esclavos, se decidió utilizar dos pipes para cada esclavo. En base a esto decidimos implementar un ADT para el manejo de la asignación de tareas a los esclavos y para almacenar la información de cada esclavo (descriptores de archivo, PID, etc.). El ADT está implementado de tal manera que a medida que los esclavos terminan de procesar sus archivos, estos vuelven a recibir de a un **archivo.cnf** a la vez.

Memoria compartida

Para lograr la comunicación entre el proceso aplicación y el proceso vista, el primero de ellos crea una zona de memoria compartida para que pueda compartir información con la vista y un semáforo para sincronizar dichos procesos (a partir de ahora objetos). El proceso aplicación será el encargado de escribir en esta zona de memoria el resultado obtenido por cada proceso esclavo, con el objetivo de que el proceso vista pueda ir leyéndolos e imprimiéndolos por salida estándar. Todo esto, se maneja a través de un ADT que se encarga de almacenar la información de la memoria compartida y el semáforo, distinguir entre proceso aplicación y vista, y saber cuál de los dos procesos es el encargado de hacer un *unlink()* de ambos objetos.

Semáforo

Como mecanismo de sincronización, se decidió utilizar un único semáforo nombrado por la simple razón de que se puede acceder al mismo a través de su nombre. Gracias a él se evitan condiciones de carrera, deadlocks y busy waiting. El semáforo iniciara en 0, se incrementará con cada *sem_post()* que el proceso aplicación realiza cuando escribe en la memoria compartida y se decrementará con cada *sem_wait()* que el proceso vista realiza cuando lee de la memoria compartida. En base a esto podemos deducir que el valor del semáforo indica al proceso vista la cantidad de resultados que puede leer.

Proceso Aplicación

El proceso aplicación se encarga de inicializar los recursos compartidos, los pipes de comunicación con los esclavos y crear los procesos esclavos. También se encarga de liberar estos recursos.

Proceso Vista

El proceso vista sufrió un cambio visible en el repositorio a lo largo del trabajo. En un principio, este fue implementado para que reciba como argumento el nombre de la memoria compartida y el nombre del semáforo para poder hacer uso de ellos. Sin embargo, teníamos el problema de que no sabíamos que utilizar como condición de corte para la vista, lo cual nos hizo replantearnos el desarrollo del mismo. Finalmente, decidimos que la mejor opción era mover las constantes de los nombres de los objetos al ADT del manejo de memoria y que vista reciba su condición de corte (cantidad de archivos que va a procesar la aplicación) por entrada estándar.

Proceso Esclavo

El proceso esclavo se encarga de hacer uso de la aplicación **minisat** cuyo output se imprimirá por salida estándar.

Limitaciones

Una de las principales limitaciones es que fijamos un SLAVE_MAX_OUTPUT cuando procesamos las cadenas de texto que genera cada uno de los esclavos. Definimos esta constante con el valor de 4196. Este valor fue establecido teniendo en consideración que en la librería *limits.h* se define la constante PATH_MAX con el valor 4096. De esta forma, nos aseguramos tener 100 bytes extra para almacenar el resto de la información, de los cuales menos de 70 se usan para almacenar el resto del texto (constante) y nos quedan libres bytes suficientes como para almacenar el pid del esclavo.

Problemas encontrados

A continuación, se listan los problemas encontrados mientras se implementaba la solución de este trabajo practico:

- Al utilizar el comando lsof, pudimos notar que no estábamos cerrando correctamente todos los pipes, cuando creíamos que si lo estábamos haciendo. También nos ocurrió que los pipes de los nuevos esclavos que abríamos podían acceder a los pipes de los esclavos anteriores. De todas formas, analizando detalladamente el código nuevamente, pudimos encontrar los errores.
- También tuvimos la disyuntiva de cómo crear la shared memory: ¿es mejor implementarla con un buffer lineal o un buffer circular? y ¿qué tamaño deberíamos asignarle? Finalmente nos decidimos con utilizar un buffer lineal y otorgarle un máximo de SLAVE_MAX_OUTPUT * filesQty.
- Como se mencionó anteriormente, tuvimos dificultades con los parámetros que debía recibir el proceso vista. Al comienzo del trabajo decidimos enviarle como argumento el nombre de la memoria compartida y el nombre del semáforo, pero nos terminamos decidiendo con solo enviarle la cantidad de archivos que debe leer.
- Nos llevó bastante tiempo determinar la condición de corte del proceso vista.
- Comprender el funcionamiento de la función *select()*.