72.11 - Sistemas Operativos

Instituto Tecnológico de Buenos Aires

Trabajo Práctico Nro. 1

Inter Process Communication

Profesores

Godio, Ariel

Aquili, Alejo Ezequiel

Mogni, Guido Matías

Merovich, Horacio Víctor

Alumnos

Bartellini Huapalla, Mateo (61438)

Estevez, Franco Nicolas (61452)

VIttor, Lucas Agustin (61435)

Instrucciones de compilación y ejecución

En el **README** del proyecto se detalla mejor como realizar la compilación y ejecución. En este informe haremos un breve resumen con lo necesario para crear la imagen, el contenedor y los archivos binarios para ejecutarlo.

Antes que nada, es importante renombrar el archivo **sample_dot_env** del proyecto por **.env** para que el archivo encargado de manejar docker funcione correctamente.

Docker

Dentro de la carpeta **docker/** existe un archivo llamado **docker.sh** que es el encargado de crear la imagen y el contenedor.

A partir de ahora se asume que los comandos van a ser ejecutados desde la carpeta **dsat/** (carpeta root del proyecto).

Primero, dar permisos de ejecución al archivo **docker.sh**

```
$ chmod +x ./docker/docker.sh
```

Para crear la imagen y correr el contenedor[1]

```
$ cd docker/ && ./docker.sh && cd ..
$ ./docker/docker.sh run
root@e698a84bf24c:/dsat#
```

Makefile

Luego de correr el contenedor, para crear los ejecutables:

```
$ make
```

Para correr los ejecutables:

Opcion 1

```
$ out/app $(DIRECTORIES) &
<info>
$ out/view <info>
```

Opcion 2

```
$ out/app $(DIRECTORIES) | out/view
```

[1] Por defecto, la imagen será **dsat:1.0**, el nombre del contenedor **dsat** y el nombre del volumen **dsat.** Esto se puede modificar editando el archivo **.env** del proyecto.

Decisiones tomadas durante el desarrollo

Slaves

Debido a los requerimientos del proceso aplicación, decidimos tomar las siguientes decisiones para la creación de los procesos esclavos:

- Mediante la constante **NUMBER_SLAVE_REQUEST** se define la máxima cantidad de procesos esclavos.
- En el caso de haber menos archivos que procesos esclavos solicitados, se crearán los necesarios para tener uno por archivo.
- Mediante la constante **INITIAL_FILES_PER_SLAVE** se define la cantidad de archivos iniciales que le otorgarán inicialmente a los procesos esclavos.
- Si la cantidad de archivos totales no alcanza como para repartir la cantidad deseada de archivos iniciales, se le repartirá uno a cada proceso esclavo.
- Como indica la consigna, en caso de haber más tareas para realizar, al liberarse un proceso esclavo (i.e. ya no tiene ningún archivo que procesar) se le otorgará un nuevo archivo.
- Cada esclavo termina cuando lee un **EOF** por su entrada estándar.

Pipes

Para la comunicación entre el proceso aplicación y el proceso esclavo decidimos utilizar dos *pipes* para cada par. Como opciones, podríamos haber elegido un solo *pipe* para la conexión esclavo → aplicación, pero debido a lo visto en clase, optamos por la mencionada en un principio, ya que nos facilitaba el reconocimiento de qué proceso hijo respondía y así asignarle una nueva tarea.

Tomando en cuenta lo mencionado anteriormente, poseemos dos *pipes* para cada par aplicación esclavo. El *pipe* de salida (aplicación → esclavo) servirá para otorgarle al proceso esclavo el *path* hacia los archivos que deberá procesar. Si hablamos del pipe de salida (esclavo → aplicación), el esclavo le entregará mediante éste al proceso aplicación el resultado de haber procesado el archivo. Con la función *select()* podemos supervisar los distintos *file descriptors* de entrada desde los procesos esclavos hacia el proceso aplicación para saber cuándo están listos para otorgarnos un resultado, y posteriormente, de ser necesario, otorgarle otro *path* para que procese el siguiente archivo.

Se decidió como protocolo a usar en estos pipes, el uso de ***"\n"*** como delimitador tanto al momento de enviar nombres de archivos al esclavo como al momento de enviar los resultados a la aplicación. También se había considerado la opción de que los esclavos escribieran siempre mensajes del mismo largo, para así leer de a bloques desde la aplicación, pero al momento de implementarla se optó por la primera opción por estilo.

Shared Memory

Por otro lado, para la comunicación entre el proceso aplicación y el proceso vista, comparten información mediante el uso de una memoria compartida (*shared memory*). El

proceso aplicación escribirá en este segmento de memoria los resultados obtenidos de los procesos esclavos, y el proceso vista procederá a leerlos.

Para evitar que el proceso vista busque leer información inexistente, hicimos el uso de un semáforo como mecanismo de sincronización. Dado que en este caso, sólo eran necesarios un escritor y un lector, esta opción brinda toda la funcionalidad necesaria.

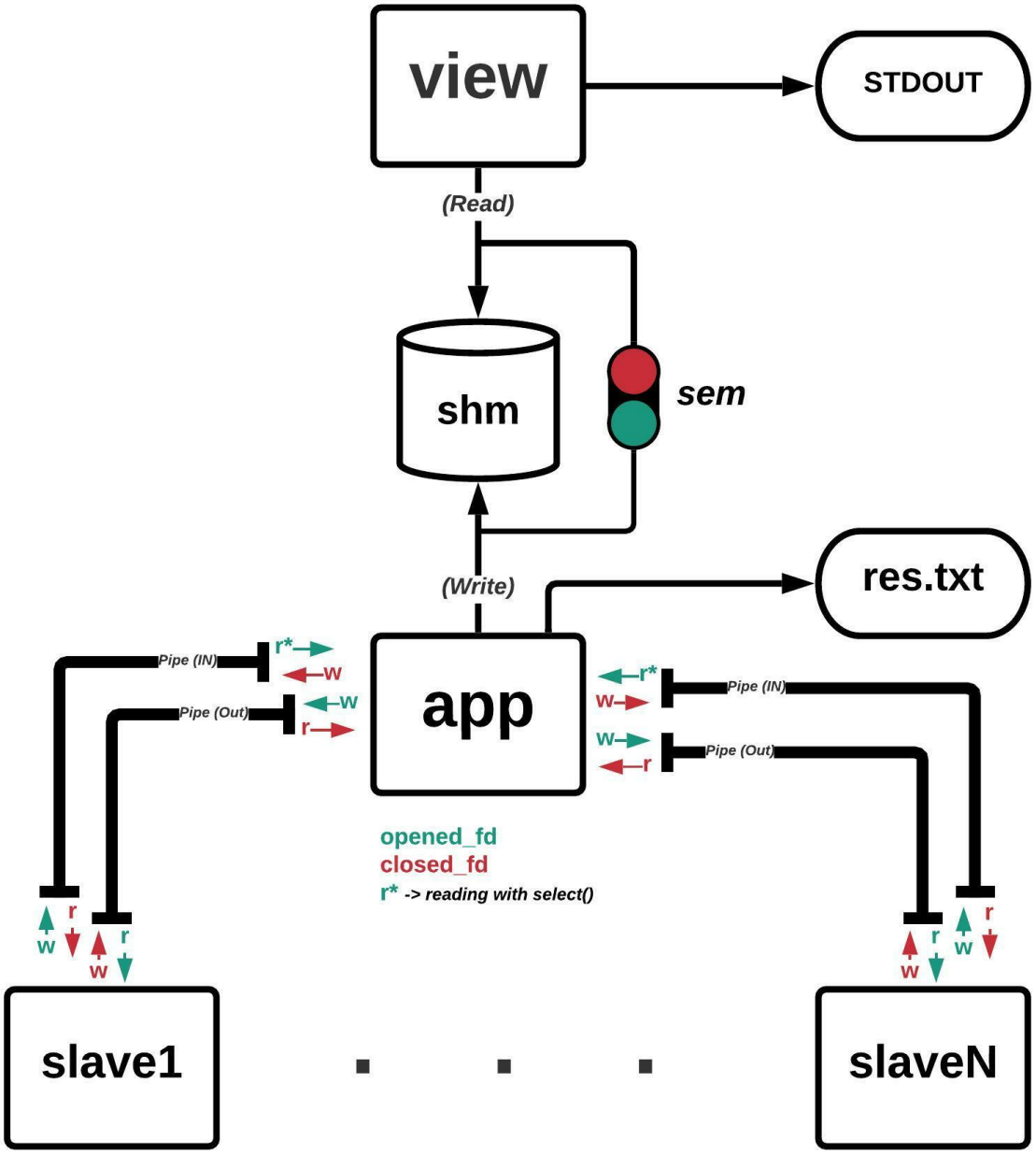
En su implementación, decidimos hacer uso de un *buffer* lineal, tomando como tamaño máximo la cantidad de caracteres de un bloque definido previamente (mediante la constante **MAX_SLAVE_OUTPUT**, que definiría el máximo tamaño de una respuesta de un proceso esclavo luego de procesar un archivo) multiplicado por la cantidad de archivos. Cada resultado se escribirá en una posición distinta de memoria que luego no será modificada, para que de esta manera el proceso vista los pueda leer sin inconvenientes.

Mecanismos de sincronización (semaphore)

Como mencionamos anteriormente, utilizamos un semáforo con nombre para la sincronización de los procesos aplicación y vista. El objetivo sería lograr que el proceso vista sólo lea de la memoria compartida en el caso de existir información relevante.

El semáforo iniciaría en 0, incrementando con cada ***post*** que realiza el proceso aplicación, y decrementando con cada ***wait*** que realiza el proceso vista. Recordemos que el ***wait*** bloquearía al proceso vista hasta que el valor del semáforo sea mayor que 0. Por ende, podemos pensar como que el valor actual del semáforo le indica al proceso vista cuántos resultados debe leer. Con esto, evitamos condiciones de *busy waiting* debido a que sólo se leerá cuando exista un resultado.

Diagrama de comunicación entre procesos



Limitaciones

Para la memoria compartida, como elegimos un *output* máximo de salida de cada proceso esclavo, e hicimos uso de un *buffer* de tamaño lineal, se debe tener en cuenta que existe un máximo de archivos que se pueden procesar, debido a que cada uno de estos aumentaría el tamaño necesario a reservar en la memoria. Además, si se quisiera modificar el programa esclavo a ejecutar, se tiene que tener en cuenta la limitación de tamaño de la salida, y garantizar el uso del ***"\n"*** como limitador. Por último, la cantidad de procesos esclavos que se pueden crear también está limitada por sus *files descriptors*, pues la función *select()* posee un valor máximo permisible para su *file descriptor*. Como futura mejora, se podría investigar el uso de la función *poll()* que trae mejoras en estos aspectos.

Links de utilidad

https://www.softprayog.in/programming/interprocess-communication-using-posix-shared-memory-in-linux

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

https://www.man7.org/tlpi/code/online/index.html