
# Sistemas Operativos
# Informe Trabajo Práctico Nº1

Segundo cuatrimestre de 2021

---

**Integrantes:**

| | |
|---|---|
| Morantes, Agustín Omar | 61306 |
| Negro, Juan Manuel | 61225 |
| Sambartolomeo, Mauro Daniel | 61279 |

**Fecha de entrega:** Lunes 13 de septiembre de 2021

## Decisiones tomadas durante el desarrollo

La primera decisión importante tomada en el diseño del programa fue cómo manejar la interacción entre los programas *Application* (padre) y *Worker* (los hijos). Se decidió utilizar un vector de *File Descriptors* desde el proceso padre, el cual guardó los *Pipes* para escribir a los hijos, y otro *Pipe* que heredan todos los hijos del cuál se recibe información. Como todos los hijos escriben al mismo *Pipe*, entonces limitamos la cantidad máxima que pueden escribir para que sea menor a *PIPE_BUF*, de esta forma asegurando atomicidad en la operación de escritura.

Por otro lado, para conectar los datos del programa *Application* con el programa *View* decidimos crear un *TAD* para que utilicen de una memoria compartida. Para implementar esta memoria se utilizó una *POSIX shared memory* en conjunto con un *POSIX semaphore* inicializado en cero para sincronización de lectura y escritura. Este semáforo es guardado dentro de la *shared memory* al crearla para que la vista pudiera acceder a él y utilizarlo con funciones provistas por el *TAD*. De esta manera, la vista hace un "*wait*" antes de leer, mientras que la aplicación hace un "*post*" luego de escribir, cada uno manteniendo su propio puntero de a que parte de la memoria está accediendo. De esta forma, la vista solamente puede leer de la memoria compartida si la aplicación escribió antes y se produce una buena sincronización.

Otra decisión importante fue cómo definimos la cantidad de workers a utilizar para procesar los archivos. Nos terminamos decidiendo por utilizar por defecto como máximo, el mínimo entre la cantidad de núcleos lógicos del procesador y la cantidad de archivos a procesar. Por ejemplo: si se requieren procesar 20 archivos y mi procesador tiene 12 núcleos lógicos, se utilizarían 12 workers, pero si me piden procesar 5 archivos se utilizarían 5 workers. Para esto utilizamos el comando *nproc* invocado desde código con *popen*. El número de workers a utilizar es configurable por parámetro con el flag *-j* ignorando los defaults previamente dichos, por ejemplo "./application.out [FILES] -j 3".

## Instrucciones de compilación y ejecución

**Las siguientes instrucciones también se encuentran en el archivo README.md en el repositorio.**

Para compilar se debe ejecutar el comando `make` dentro de la raíz del repositorio. Esto generará los ejecutables `application.out`, `view.out` y `worker.out` en la raíz. Para ejecutar el programa se debe ejecutar utilizando el siguiente comando:

```
./application.out (-j WORKER_COUNT) [FILES]
```

También opcionalmente puede utilizar el proceso View para ver el output de los workers de la siguiente forma: `./application.out ... | ./view.out`

Opcionalmente si se quiere llamar al proceso View desde otro shell, se le puede pasar como parámetro el nombre de la shared memory que utiliza el proceso application, por ejemplo:

```
./view.out /sharedMemName
```

En caso de querer probar el worker manualmente, se lo puede llamar con el comando `./worker.out` y por entrada estándar se le pueden pasar los paths de los archivos a procesar separados con un newline. Para terminar su ejecución puede enviarle un EOF con `CTRL+D`.

## Limitaciones

- Máximo tamaño del nombre/path de un archivo es de 511 caracteres, y se limita a 1024 bytes la cantidad máxima de caracteres que puede escribir un proceso *Worker* para cada uno de los archivos que analiza.
- La cantidad de archivos no está limitada por factores de nuestra parte, pero dependería de cuestiones externas a nuestro programa (cantidad máxima de argumentos para un programa en el sistema operativo utilizado, tamaño máximo de memoria compartida, tamaño máximo de memoria para dado programa, etc.)

## Problemas encontrados y sus soluciones

Un problema encontrado fue cómo guardar el nombre del archivo en el proceso *Worker*. Al investigar, encontramos que en la librería "*limits.h*" se define una constante llamada *PATH_MAX*, siendo este el tamaño máximo de un *path* en *Linux*. Sin embargo, también encontramos que es igual a 4096 caracteres, y también que *Linux* no la usa realmente y que se pueden crear *paths* de mayores tamaños. Además, la constante *PIPE_BUF* también es de 4096 caracteres, por lo que no podíamos permitir *paths* de su máximo tamaño. Por ende, se decidió limitar el tamaño del *path* a 512 caracteres, pues creemos que es una cantidad suficientemente grande para abarcar las necesidades del usuario. Además, podemos limitar el tamaño máximo de la respuesta del proceso *Worker* por archivo a 1024 caracteres, siendo un cuarto del tamaño del *PIPE_BUF* pues entendemos que nunca se deberá escribir tanto en una respuesta.

Otro problema fue el manejo de errores. Para un correcto manejo de errores, se intentó obtener el valor de retorno de *Minisat*, para comparar lo obtenido y saber qué ocurrió durante la ejecución. Sin embargo, este no se pudo obtener pues en el comando *popen* se "*pipea*" el resultado del *Minisat* al comando *grep*. Por ende, se decidió utilizar el resultado del comando *grep* para el manejo de errores, viendo que si el resultado de la búsqueda era vacío, entonces no se pudo analizar el archivo correctamente con *Minisat*. Sin embargo, se podía pasar un archivo inexistente y de nombre "SATISFIABLE", por ejemplo, logrando que nuestra búsqueda sí encuentre resultado pues *Minisat* repite el *path* de un archivo que no pudo encontrar. Además, al utilizar la herramienta de análisis estático PVS-Studio recibimos un warning sobre la utilización de input del usuario (posiblemente malicioso) en una llamada a la función *popen* en el worker, donde ejecutamos el comando de minisat. Ambos problemas se resolvieron verificando que el *path* pedido sea realmente un archivo, aunque PVS-Studio lo sigue considerando como un error. Creemos que esto es un falso positivo por parte de la herramienta.