# Sistemas operativos

## Trabajo práctico 1: Inter Process Communication

Grupo 6: Pedro Momesso (58027), Franco Baliarda(58306), Agustin Bossi (57068)

# Índice

## Tabla de contenido

# Introducción

El siguiente trabajo tuvo como objetivo hacer uso de los distintos mecanismos que existen para la comunicación entre procesos utilizando el estándar POSIX. La tarea a realizar fue el cálculo de hashes de archivos con el comando md5sum de manera distribuida. Para ello se implementaron tres procesos diferentes, cada uno con una funcionalidad definida y diferente entre sí, de manera tal que para lograr el objetivo final fuera necesario comunicar estos procesos mediante algún tipo de IPC. Estos son “application process”, “slave process”, y “view process”, donde el primero se encarga de distribuir los archivos a hashear entre los distintos “slave process” y luego enviar los resultados a “view process”, el segundo se encarga del cálculo y posterior retorno de este a “application process”, y finalmente el último se encarga de mostrar los resultados por salida estándar.

En el siguiente informe se explicarán los mecanismos utilizados entre cada uno de estos procesos, los problemas y limitaciones encontradas y las instrucciones de ejecución.

# IPCs implementados

## Comunicación entre Application y Slave

Aprovechando que los procesos que calculan hashes se crean a partir de la función fork en la aplicación principal, se decidió utilizar dos pipes unidireccionales con cada esclavo como forma de comunicación. Para evitar problemas de sincronización se estableció un pipe en el que la aplicación escribe la dirección del archivo para ser leída por el esclavo, y otro en el que el esclavo escribe el resultado de la operación de hashing para que la aplicación principal lo lea. Se utilizó como punta de escritura el file descriptor correspondiente a salida estándar, y como punta de lectura el correspondiente a entrada estándar. Esto se definió así para poder ejecutar individualmente los procesos y así detectar más fácilmente los errores que podían llegar a aparecer en el código. Por otro lado, se logró de este modo mantener independencia entre los módulos de ambos códigos: el código del proceso Slave tiene como propósito el cálculo de hashes de archivos únicamente. Por ende, no debe tener presente detalles de cómo se haga uso de él mismo.

El proceso Slave puede ser utilizado como un programa que pide por entrada estándar nombres de archivos para luego imprimir sus hashes por salida estándar. Del mismo modo, puede ser utilizado por un proceso padre (como lo es el proceso Aplicación), para recibir mediante redireccionamiento a su entrada estándar nombres de archivos para luego enviar sus hashes a donde se redireccione su salida.

## Cálculo de hashes en Slave process

Para calcular los hashes de los archivos recibidos de la aplicación padre se decidió utilizar el comando de bash md5sum. Para esto cada proceso slave hace un fork y luego se llama al comando con la función exec pasando como parámetro el archivo.

Para recuperar la salida de md5sum se volvió a aprovechar que el proceso arranca a partir de un fork utilizando un pipe unidireccional donde el proceso md5sum llamado con exec escribe y el esclavo lee, manejando las puntas de la misma manera que se hizo con los pipes entre la aplicación principal y los esclavos.

## Comunicación entre Application y View

Para pasar los hashes recuperados por la aplicación principal al proceso vista se estableció un buffer de tamaño fijo en memoria compartida al cual se obtenía acceso al principio de Application y de View, y se liberaba al final de los

mismos. La aplicación principal escribe los hashes en el buffer y el proceso vista los lee para posteriormente mostrarlos por salida estándar, sincronizandose con un semáforo que protege el buffer dejando que solo un procesos acceda a este en tiempo real y a su vez bloquea al proceso view si no hay nada para leer.

# Problemas y limitaciones encontradas

## Límite de archivos

Debido a limitaciones del sistema, la memoria compartida entre la aplicación principal y el proceso vista tiene un tamaño máximo. Por lo tanto, también hay un límite máximo de archivos que se pueden procesar y escribir en la memoria compartida. Se realizaron estimaciones y se determinó que la cantidad máxima de archivos que se pueden almacenar en la memoria compartida son entre 1000 y 1500 sin que haya ningún tipo de error.

Una posible solución a este problema sería implementar el buffer de la memoria compartida de manera circular, de tal forma que cuando se llegue al final de la memoria se empiece a escribir de vuelta desde el principio de esta. Esta implementación no fue realizada, por un lado ya que no es pertinente al funcionamiento y aprendizaje de la comunicación entre procesos y, por otro, el segundo diseño no fue pensado sino hasta el final de la construcción del sistema. Por ende, dadas las restricciones de tiempo, no se volvió a implementar el sistema de memoria compartida.

## Creación de procesos slaves

Dado el diseño que se pensó consistente de dos pipes por proceso esclavo y dado que se trataron de respetar cuestiones de correctitud, los procesos esclavos fueron desprovistos de todo descriptor de archivo que no usasen. Luego fueron desprovistos de todos aquellos descriptores de archivos pertenecientes a otros esclavos, lo que se tradujo en un tiempo de ejecución mayor. Ésto no habría ocurrido de haber utilizado otros esquemas de creación de procesos slaves, en donde se cuenta con un solo pipe para comunicación aplicación-slaves.

## Comunicación con procesos slaves

Para la lectura de los hashes por parte del padre se debieron esperar varios descriptores de archivos. Ésto se tradujo en tener que aprender sobre la system call select y su funcionamiento, lo cual presentó un desafío. Se logró su correcto uso sólo tras reiteradas pruebas y errores. Por otro lado existe la limitación de impresión de los hashes en verdadero orden de llegada. Este requisito tuvo que ser relajado: si a través de la system call select se obtienen varios pipes disponibles de lectura, no hay manera de recuperar el orden de escritura por parte de los hijos.

# Instructivo de ejecución

1. Extraer archivos a una carpeta con el nombre que desee.
2. Ejecutar en la terminal el script “compileScript.sh”:

```
./compileScript.sh
```

3. Se crearán los binarios:
   a. “mainApplication”: aplicación padre.
   b. “slaveProcess”: procesos esclavos.
   c. “viewProcess”: proceso vista.
4. Ejecutar en la terminal:

```
./mainApplication "dir"/* | ./viewProcess
```

   Siendo “dir” el directorio con los archivos a procesar.

5. Se creará el archivo “hashResults.txt” con los hash de los archivos procesados, para leerlo ejecutar en la terminal:

```
cat hashResults.txt
```

# Testing y comprobaciones

Se utilizó Valgrind en los tres procesos para garantizar que no hay leaks de memoria. Se compiló con -Wall y se corrigieron todas las alertas innecesarias que fueron surgiendo y se comprobó el estilo del código con cpp check.

Dado que no se emplean estructuras de datos en el tp, no hay muchos elementos que pueden testearse. Tampoco hubo necesidad de hacer tests sobre la cobertura de md5sum ya que no se hizo una función que los calculase, sino que se recurrió a bash. Debido a todo esto se decidió hacer un test más integral que comparase los resultados devueltos por los procesos esclavos con md5sum de bash haciendo un assert.

# Conclusión

El trabajo permitió un mejor entendimiento de la concurrencia entre procesos, su manejo y su utilidad. Se pudo ver la importancia de distribuir una tarea en diferentes procesos y cómo afecta al tiempo de ejecución de la misma.

Si bien el trabajo funciona, hay lugar para mejoras, especialmente en la memoria compartida, ya que al tener un tamaño fijo hay un límite en cuanto a la cantidad de archivos que pueden procesarse. Idealmente esta memoria compartida tendría comportamiento de un buffer circular, de manera que la aplicación principal pueda escribir sin quedarse sin espacio en el buffer.

# Bibliografía

Man