Trabajo Práctico N°1 Sistemas Operativos: *Inter Process Communication*

Bruno Enzo Baumgart, Santiago José Hirsch, Mauro Joaquín Vella

Instituto Tecnológico de Buenos Aires

72.11 - Sistemas Operativos

Ariel Godio, Alejo Aquili,

14-04-2023

Resumen

En este trabajo se trabajó la comunicación entre procesos (IPC) presentes en un sistema POSIX. Para utilizar los IPC se realizó un sistema de cómputo de hashing "md5" de forma descentralizada entre varios procesos sobre una cantidad N variable de archivos.

Para esto se cuenta con los procesos de aplicación, vista y esclavos (ver **Fig. 1**).

El proceso aplicación (llamado "application") recibe por línea de comando los archivos a procesar y se encarga de iniciar los procesos esclavos repartiendo la carga y de recibir y almacenar los resultados.

El proceso vista recibe por entrada estándar o como parámetro la información necesaria; y se encarga de conectarse al *buffer* compartido, y de imprimir en pantalla los resultados obtenidos en el *buffer* de llegada a medida que se llena el mismo.

El proceso esclavo (llamado "slave") recibe los archivos a procesar; y se encarga de iniciar el programa correspondiente para procesarlos y de enviar la información relevante al proceso aplicación mediante un mecanismo sofisticado de *Inter Process Communication*.

Palabras clave: IPC, Inter Process Communication, comunicación entre procesos, md5, md5sum, POSIX, process, shared memory, memoria compartida, semaphores, semáforos.

Momentos críticos durante el desarrollo

Durante el desarrollo del trabajo se llegaron a varios momentos decisivos, donde se decidió tomar cierto camino. A continuación se detallarán esos momentos cronológicamente y se explicarán los motivos que llevaron a las decisiones.

Estructura del proyecto

En la etapa inicial del proyecto, se ideó una estructura del proyecto en base a lo que expresaba y requería la consigna. En "application.c" se hizo el esquema básico de las principales funcionalidades: el pasaje de argumentos, el manejo de archivos y el cálculo de la cantidad de esclavos para luego construir sobre esa base sólida.

Luego se llevó a cabo el mismo proceso para edificar la estructura de "slave.c". Para esto hizo falta crear una estructura para el resultado ("md5_info"), y encargarse del manejo de los file descriptors con las funciones "FD_ZERO" y "FD_SET" de la librería "sys/select.h".

A partir de este punto se hizo notar la necesidad de modularizar el proyecto. Se creó el directorio "utils/" para contener las librerías necesarias[1] para las funciones que se utilizarían durante el proyecto, cada una con el manejo de sus errores.

Comunicación entre procesos

La comunicación para el paso de archivos entre el proceso de aplicación y el proceso de *slaves* se realizó mediante *pipes*. Estos debían tener un manejo cuidadoso para no dejar extremos abiertos inutilizados.

Estructura de *pipes*

Application debía escribir a slave y slave debe escribir a application por separado, y como se vió en clase, application debe estar conectado con cada slave de forma bidireccional

[1] "fd_utils.c", "file__utils.c", "process_utils.c", "sem_utils.c" y "shm_utils.c"

(debe haber un *pipe* de ida y otro de vuelta). Por lo tanto se decidió tener dos *pipes* por cada proceso *slave* (ver **Fig. 2**).

Control de *pipes*

En primera instancia se cierran los extremos de escritura de "app_to_slave" y de lectura de "slave_to_app", ya que *slave* utiliza únicamente el extremo de escritura de "slave_to_app" y *application* utiliza únicamente el extremo de lectura de "app_to_slave".

Comunicación con *subslaves*

Del mismo modo se manejaron los pipes conectando a cada slave con sus *subslaves*. Para lo que también hizo falta crear una estructura "subslave_info" para mantener un manejo ordenado.

Shared memory* y *semaphores

Para la realización de las funciones de manejo de memoria compartida y semáforos ("shm_utils.c" y "sem_utils.c"), hizo falta leer la documentación para comprender precisamente para qué y cómo se utilizaban las mismas.

Proceso vista

En primer lugar, es necesaria la aclaración acerca del nombre elegido para el proceso. Este se debe a la historia etimológica de la palabra: del latín *visĭtus*. Con respecto a lo que se pedía en la consigna, se pensó en una primera instancia que sería simple ya que se habían realizado todas las funciones de manejo de *shared memory* y *sempahores* con anterioridad. Y pareció serlo, el algoritmo principal no fue más que el llamado a las funciones de creación, apertura, *unlinking* y destrucción ya previamente hechas.

Complicaciones con el programa de prueba

Una vez realizado el algoritmo principal se tomó la decisión de armar un programa de testeo. Desde “application.c” se envíaba a la memoria compartida la estructura de un “md5_info” conteniendo todos los datos de los archivos necesarios para imprimir en “vista.c”, el hash calculado, el nombre del archivo, y el pid del proceso esclavo que lo resolvió.

Aquí fue cuando hubo una serie de problemas que no permitían el correcto funcionamiento del programa de prueba. No se abría la memoria compartida, al llamar a la función de “open_shm” el programa entraba en el manejo de error de la función. Luego se pudo notar que lo mismo ocurría al intentar abrir los semáforos con la función “open_sem”.

Gracias a muchos testeos y pruebas de análisis estático y dinámico del código con los programas “valgrind” y “PVS studio”, se pudo llegar a la conclusión de que el proceso aplicación estaba corriendo normal y completamente, y recién luego corría el proceso vista.

Esto era de gravedad ya que el proceso aplicación estaba cerrando la memoria compartida y los semáforos antes de que vista puedo abrirlos. Se tuvo que tomar una decisión importante, retrasar el armado del proceso vista ya que los errores eran probablemente relacionados al diseño del testeo ideado.

Proceso *slave*

Luego de las complicaciones con el proceso vista, se decidió programar las funcionalidades de *slave* en su respectiva rama de “git” para así cuando ya estuviera funcionando poder volver al proceso vista y arreglar el problema de sincronización con ayuda de *slave* ya funcionando. Fue entonces que se realizó el algoritmo principal del proceso *slave*, y luego siguiendo la metodología llevada a cabo en “vista”, se realizó un testeo.

Programa de prueba

Para testear el proceso de “slave.c”, se le pasaban archivos de prueba a “application.c” tal como expresa la consigna, e imprimía los resultados calculados en el archivo de salida “respuesta.txt”. A diferencia de la experiencia anterior en el proceso de vista, este testeo corrió adecuadamente sin ninguna complicación.

Cálculo de la cantidad de esclavos

Luego de una serie de testeos cuando el proceso se encontró funcionando, se decidió que si bien la cantidad de esclavos debía ser proporcional a la cantidad de archivos envíados, debía también tener un límite superior con el fin de no sobrecargar al procesador. Esto se decidió así ya que era normal que a un proceso *slave* le toque calcular el hash de un archivo con considerable más tamaño que los demás y para esto era conveniente tener la menor cantidad de procesos corriendo en simultáneo, pero a su vez era conveniente dividir en muchos procesos cuando la necesidad de poder de cómputo por archivo era semejante entre los archivos. La fórmula utilizada fue la siguiente: $\{cant\_files \,/\, (INITIAL\_FILES\_PER\_SLAVE \ * 3) \ + \ 1\}$ [2]

“process_utils.c”

Mientras se avanzaba en el desarrollo del proceso *slave* y a medida que fueron de necesidad nuevas funciones que se encarguen de procesos (crear nuevos esclavos, y terminar procesos), se decidió juntarlas en una nueva librería llamada “process_utils.c”.

Fusión de “slave.c” y “vista.c”

Cuando se tuvo el algoritmo del proceso *slave* funcionando, y el algoritmo de vista armado pero sin funcionar, se realizó un *merge* de ambas ramas de desarrollo de “git” a una nueva llamada “dev” con el objetivo de, con un correcto funcionamiento, implementar las tres partes del proyecto (*application*, *slave* y vista), y resolver el conflicto con el proceso vista.

[2] “cant_files” refiere a la cantidad de archivos que se le envíen al proceso *application*, y “INITIAL_FILES_PER_SLAVE” es la macro que refiere a la cantidad de archivos distribuídos a cada *slave* en la distribución inicial.

Problemas

El problema de no poder abrir la memoria compartida seguía existiendo, y además ahora se debía resolver cómo enviarle a un esclavo más de un archivo a la vez (en la distribución inicial) tal como exige la consigna.

Soluciónes

En una primera instancia se decidió enviar de a un archivo a la vez en la distribución inicial, entonces de esta forma se logró una versión del trabajo práctico funcional si y solo sí se le envíaba de a un archivo a cada esclavo.

Luego se intentó implementar un sistema de encolado para el pasaje de archivos entre estos dos procesos. Cada esclavo tendría su propia estructura de archivos encolados de tres posiciones. La idea detrás de esta implementación era a medida que se consumía un archivo para ser resuelto, se desplazaba la cola y añadir un archivo nuevo en la última posición.

Esta aproximación trajo consigo una gran cantidad de problemas de manejo de memoria, que fueron encontrados corriendo el proceso con la herramienta de análisis “valgrind”.

La aproximación de la cola fue descartada por una mucho más simple: se cambió la macro que indicaba la cantidad inicial de archivos enviados a cada esclavo de uno (1) por tres (3) (siempre cuando la cantidad envíada sea de al menos tres). El manejo de la cola lo realiza automáticamente el “pipe” gracias a su implementación.

En cuanto al problema de la memoria compartida en el proceso vista, se decidió agregar un segundo semáforo comunicando el proceso *application* y el proceso vista, encargado de establecer cuándo era posible eliminar los recursos.

Luego de este último arreglo se contaba con una versión funcional entre *application*, *slave* y vista donde se podía enviar un número constante de archivos inicialmente a cada *slave*

para que los resuelva, y el *proceso* vista imprimía en tiempo real los resultados obtenidos. De todas formas existía todavía un error: no se guardaba bien el nombre de los archivos para imprimir en el resultado.

Nuevo problema y su solución

Dentro de la cola automática del *pipe*, si se le pasaban *n* archivos inicialmente (tres en nuestra implementación), los primeros *n-1* nombres de archivo se perdían, y al momento de imprimir repetía el nombre del último impreso.

Para solucionar este problema se decidió enviar desde *slave* hacia *application* no sólamente el hash de md5 calculado, sino el resultado completo del "md5sum" (en formato *string* "md5hash nombre_de_archivo"). De esta forma en *application*, se separaría el string en su parte de hash y su parte de nombre. Para esto se utilizó la función "strtok" y luego de pruebas y errores con el manejo de *strings*, se llegó a la primera versión finalizada del proyecto.

Discusión y conclusiones

En la instancia final, se analizó el código nuevamente y se llegó a la conclusión de que el segundo semáforo añadido en una instancia previa, no era completamente necesario. Por lo tanto se lo eliminó y se volvió a la versión original que era de una mayor simpleza.

También se notó de la necesidad de adicionar una pequeña medida de seguridad en cuanto al manejo de *shared memory* y *semaphores* principalmente en el caso de frenar el programa durante su ejecución con "ctrl + C". En tal caso los nombres de la memoria compartida y del semáforo ya se encontrarían utilizados y podría generar un error al volver a correrse. Es por esto que se hizo de manera preventiva *un link* previo a la creación de los mismos.

En la consigna se habla de un "sleep" antes de la ejecución del algoritmo de *application* para dar tiempo a correr el proceso vista por terminal en caso de ser deseado. Se decidió que este

intervalo sea de cinco segundos que parece un tiempo razonable para correr el script en la terminal.

A lo largo del trabajo práctico se fueron llenando las librerías en “manager.h” y “errors.h” con tal de mantener la modularización y limpieza de recursos del proyecto.

Se realizó en cierto momento del desarrollo del trabajo un archivo “Makefile” con sus dependencias para poder hacer uso de los *shortcuts* que ofrece (“make”, “make clean”), y además poder utilizar “PVS studio” ya que lo requiere.

Para hacer el testeo final del proyecto primero se lo probó con pocos archivos livianos, luego con más de treinta y más de ochenta considerablemente más pesados, y últimamente como prueba de fuego se intentó correr con una carga de ciento cuarenta archivos de los cuales algunos pesaban más de un gigabyte.

A su vez se decidió dejar en los archivos de extensión “.c” el comentario necesario para poder correr fácilmente la herramienta “PVS studio” de análisis estático de código.

Una limitación del programa es que si fuese el nombre de algún archivo mayor a 255 caracteres, sólamente se imprimirán los primeros 255 caracteres del mismo.

En las líneas veintiséis y veintisiete (26, 27) del archivo “vista.c” la herramienta “PVS studio” indica la siguiente advertencia menor de seguridad: “V755 A copy from unsafe data source to a buffer of fixed size. Buffer overflow is possible.”. Esto se debe a una precaución frente a posibles ataques de inyección de código. Siguiendo indicaciones de la cátedra se decidió ignorar únicamente esta precaución ya que escapa el propósito del trabajo y se considera que leer de entrada estándar en dicho proceso vista.c es suficientemente seguro, ya que se realiza un control para que la entrada no sea de una longitud indeseada.

Referencias

No se reutilizaron fragmentos de código de fuentes externas a lo largo del proyecto. Únicamente las funciones de librerías del lenguaje de programación C.

Anexo

Fig. 1

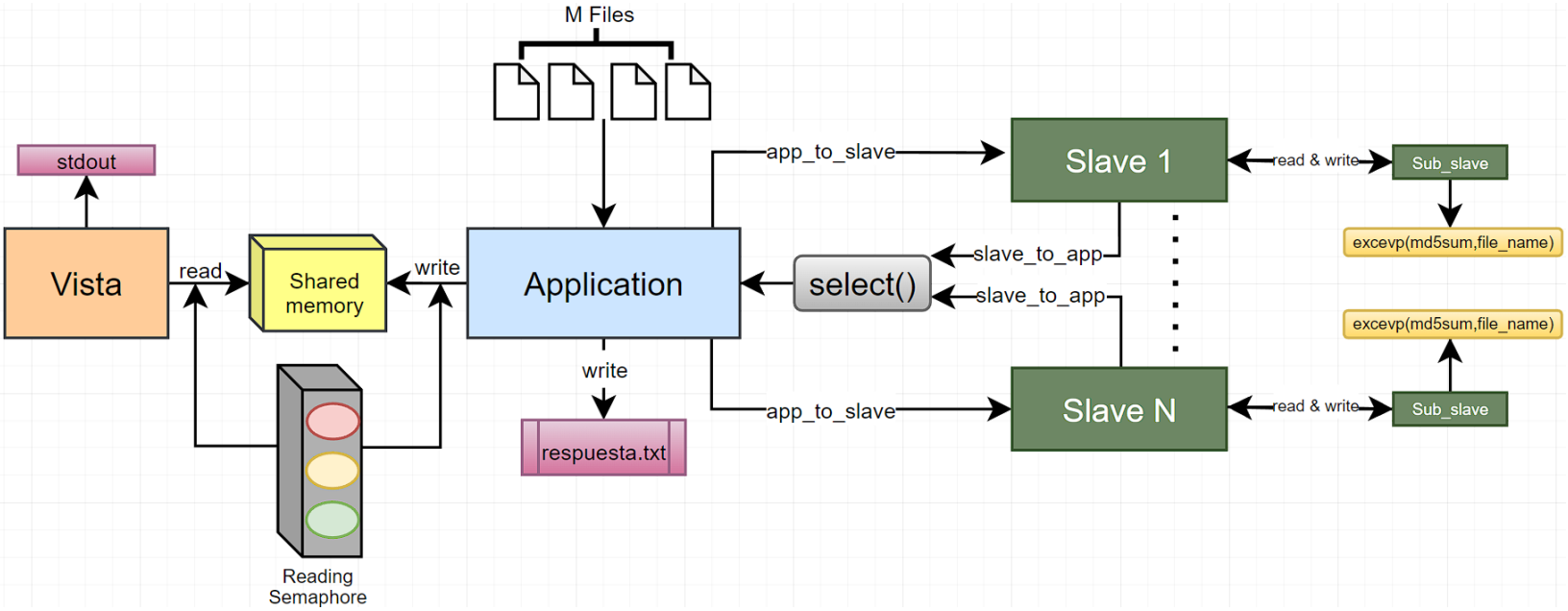

Fig. 2

```
typedef struct slave_info {
    int app_to_slave[2];   // File descriptors connecting app to slave
    int slave_to_app[2];   // File descriptors connecting slave to app
    (...)
} slave_info;
```