**Sistemas Operativos**

(1C2018)

Trabajo Práctico 1

**Inter Process Communication**

**Integrantes:**

Tomás Ferrer - 57207

Felipe Gorostiaga - 57200

Lucas Sanz Gorostiaga - 56312

Santiago Swinnen - 57258

Introducción

El siguiente informe describe el desarrollo del Trabajo Práctico 1 de la cátedra de Sistemas Operativos del ITBA, en el cual se trabajaron los conceptos relacionados con la comunicación entre procesos y el uso de memoria compartida. Para ello se desarrolló en lenguaje C un sistema que consta de múltiples procesos cuyo objetivo final es imprimir en pantalla los hashes MD5 de los archivos incluidos en un directorio pasado como parámetro.

La estructura del trabajo está formada por un proceso aplicación, varios procesos esclavos (cantidad determinada en función del número de archivos a procesar) y un proceso vista, que comparte memoria con el proceso aplicación.

El trabajo se desarrolló y se probó en el sistema operativo Linux. Cabe aclarar que no fue posible correrlo con normalidad en Pampero debido a una falla en la creación de los named pipes, recurso utilizado para la comunicación entre el proceso aplicación y los procesos esclavos.

A continuación se detallarán diversas cuestiones que influyeron en el resultado final del trabajo y son relevantes para el análisis y la evaluación del mismo.

En todo momento se buscó mantener el buen estilo de código utilizando nombres claros para las funciones y las variables, modularizando para evitar funciones largas y difíciles de leer, y separándolas en distintos archivos .c según el objetivo de cada una y de esta forma evitar archivos de código excesivamente extensos.

Principales decisiones

Para comunicar al proceso aplicación con los procesos esclavos se decidió utilizar dos named pipes por cada esclavo, uno para que el proceso aplicación envíe al esclavo los nombres de los archivos a procesar y otro para que el esclavo pueda devolver el código calculado, así como indicar que está libre.

Asimismo cada proceso esclavo crea un unnamed pipe para capturar la salida del programa “md5sum”, el cual se encarga de obtener el MD5 del archivo recibido. Para una operación tan básica como redirigir la salida estándar de un proceso a la entrada estándar de otro (y siendo que cada esclavo hace sólo un fork y exec), se buscó hacerlo de la manera más simple para comunicar dos procesos, que es con el unnamed pipe.

Para descubrir qué procesos esclavos están libres y enviarles más trabajo, se utilizó la system call select, que detecta en qué pipes hay contenido para leer, utilizando como set de file descriptors aquellos entrantes al proceso aplicación.

En cuanto al proceso Vista, se utilizaron semáforos y memoria compartida, tal como lo indica la consigna.

Cabe hacer una aclaración sobre una decisión relacionada con el desarrollo que se tomó sobre la función manage\_children en el archivo “application.c”. Se consideró realizar una modularización para que la misma fuera menos extensa. Sin embargo, esto significaba crear una nueva función con trece parámetros y manipulación de punteros a variables creadas en la función original. La pérdida de simpleza y legibilidad que la modularización implicaba llevó a conservar la función tal como estaba ya que la extensión de la misma no resultó excesivamente grande.

Respecto al número de esclavos utilizados se decidió fijar la cantidad en 1 ya que para la gran mayoría de las carpetas funciona muy rápidamente y si se los hubiera determinado en función de la cantidad de archivos, para casos extremos con carpetas con muchos archivos esto habría significado una cantidad muy grande de esclavos, lo cual no se consideró conveniente. Por otro lado, como posiblemente en el contexto de la utilización de este código se podría querer cambiar la cantidad de esclavos para calcular los hashes de alguna carpeta con muchos archivos, basta con cambiar en el código la constante SLAVE\_NUM.

Compilación y ejecución

Para compilar fue utilizado el compilador de GNU (gcc), con los parámetros -Wall y -pedantic, para los archivos que utilizan semáforos (semaphores.h) se linkeditó con -lpthread y -lrt. Nótese que se proporciona un Makefile. El programa se ejecutó sobre Ubuntu 16.04 x64, en varias computadoras.

Para ejecutar el programa, hay que abrir una terminal y ejecutar application, pasándole como parámetro una lista de nombres de archivos (incluyendo el path), ya sea enumerándolos o de la siguiente manera: “mi\_directorio/\*”. Luego, si se desea que aparezca un proceso vista, es necesario ejecutar desde otra terminal view, pasándole como parámetro el PID del proceso application, el cual se imprime en salida estándar ni bien comienza la ejecución de application.

Desarrollo

La primeras tareas llevadas a cabo por el proceso aplicación al iniciar su ejecución son el cálculo de esclavos necesarios, la creación de los pipes y posteriormente de los esclavos. Estos últimos abren los pipes de entrada y salida como primera acción, para evitar que el proceso aplicación no quede bloqueado al abrirlos desde su extremo, situación que se da inmediatamente después del fork de cada esclavo.

Se decidió en un principio mandar un archivo para procesar a cada esclavo. Una vez finalizada esta primera etapa, el proceso aplicación entra en un loop del cual no sale hasta que no haya recibido tantos hashes MD5 (o la correspondiente respuesta en caso de que se haya enviado un directorio) como archivos haya en el directorio pasado como parámetro. Dentro de este loop se hace la llamada a la previamente mencionada función select y posteriormente se leen los pipes con contenido para leer.

Para lograr una comunicación entre el proceso aplicación y los esclavos lo más eficientemente posible desde el punto de vista espacial se resolvió enviar por los pipes mensajes de la siguiente manera:

1. Se envían por el pipe exactamente 3 caracteres con la longitud del mensaje que se quiere enviar a continuación.
2. El emisor escribe en el pipe el mensaje.
3. El receptor en primera instancia lee 3 caracteres.
4. Se convierten los 3 caracteres a entero y se guardan en una variable.
5. Se lee nuevamente, utilizando como cantidad de bytes el valor de la variable recibida en una primera instancia.

Esta forma de comunicación se respeta en ambos sentidos. De esta manera solamente se utiliza el espacio justo y necesario y se logra que el receptor lea la cantidad justa de caracteres.

Una situación particular se da cuando se encuentra un directorio. Por cuestiones de prolijidad y para no alterar la normalidad del flujo de intercambio de información entre aplicación y esclavo (cada vez que aplicación recibe algo inmediatamente envía algo), el proceso aplicación envía tres ceros indicando que se ha encontrado un directorio y esclavo responde de la misma manera. Se decidió no incrementar directamente el índice del arreglo de archivos ya que en algunos casos extremos esto podría ser causa de problemas y circunstancias no deseadas.

Para la intercomunicación de los procesos vista y aplicación se utilizó, como indicó la cátedra, shared memory y semaphores. Dentro del proceso aplicación se inicializa un pedazo de shared memory, utilizando su pid como key para la función shmget. Cuando se corre el proceso vista, se le pasa como argumento al programa el pid del proceso aplicación lo cual permite conectarse al mismo sector de memoria compartido. Una vez conectado a la zona de memoria compartida, el proceso vista procede seteando el primer byte de esta memoria en 1, indicando al proceso aplicación que la vista está presente. El segundo byte se utilizó para determinar si era el turno de la vista para trabajar (imprimir el contenido de la memoria), es decir el valor de dicho byte era 1 o 0 si era el turno de la aplicación (imprimir en la memoria el valor del md5sum).

También se utilizó un POSIX semaphore que se encargó de pautar los turnos correctamente entre ambos procesos, realizando un sem\_post desde la aplicación y la vista cuando estos procesos terminaban de realizar su trabajo, y hacer un sem\_wait cuando el otro proceso estaba trabajando (viendo el segundo byte de la memoria). Al comienzo de la aplicación, se esperan 15 segundos para que se conecte el proceso vista y en caso negativo procede a volcar la información en un archivo sin preocuparse por la memoria compartida. Si el proceso vista se corre luego de pasados los 15 segundos, mientras la aplicación procesa los hashes, esta comenzará a imprimir en pantalla los resultados de los archivos comenzando desde que se conectó la vista.

Problemas y soluciones

A lo largo del desarrollo del trabajo surgieron diversos inconvenientes y circunstancias a resolver.

El tema más recurrente, que en la gran mayoría de los casos se dio por errores de planificación conceptual, fue el bloqueo no deseado de procesos por llamadas a la función read cuando los pipes estaban vacíos (para la solución final la única situación en la que esto fue buscado fue para cuando un esclavo estaba libre, y no tenía otra función más para cumplir que esperar a que le lleguen nuevos archivos).

Este problema surgió primero al abrir los named pipes para lectura y escritura desde el lado de la aplicación sin antes haber hecho fork y execl del esclavo. Esto generaba que aplicación se bloqueara hasta que el esclavo abriera los correspondientes pipes, cosa que nunca sucedería ya que la creación de los esclavos correspondía a una instancia posterior del programa bloqueado. Esto se resolvió abriendo los pipes desde aplicación inmediatamente después del fork y en las primeras dos líneas luego de las declaraciones en el caso del esclavo

Una segunda instancia en la que surgiría esta cuestión (instancia a la que no se llegó ya que la situación se dio desde lo conceptual previo a la implementación) era al momento de leer a los esclavos desde el lado de la aplicación. Fue claro que si se leía a cada esclavo en cada ciclo, se llegaría a la situación en que un esclavo no tenía contenido para enviar, la llamada al read vacío bloquearía al proceso aplicación, y éste perdería tiempo útil de ejecución en el que podría haber estado leyendo a hijos que sí tuvieran contenido disponible. Para solucionar esta cuestión se recurrió a la utilización de la función select, que mediante la macro FD\_ISSET permite conocer si tiene un file descriptor tiene contenido para leer sin bloquearse.

También se atravesaron diversos problemas prácticos, por fallas en la programación que fueron resueltos con el correr de los chequeos, como por ejemplo segmentation faults al leer strings y envíos de cero caracteres a los esclavos que causaban que estos dejaran de funcionar correctamente.

Para asegurar que no hubiera memory leaks, la herramienta valgrind fue utilizada (con el parámetro -v), y el resultado final es muy positivo, ya que indica que hay tantos alloc como free, por lo que no queda nada sin liberar en el heap.

Asimismo cppcheck fue utilizado (con los parámetros --verbose y --enable=all) para correcciones de estilo, lo cual evidenció un potencial bug en un llamado a scanf, en el cual se podía llegar a leer demasiados bytes, lo cual desestabilizaría el programa.

Conclusión

A partir de las decisiones tomadas y puestas en práctica se logró construir una solución a la consigna respetando todos sus aspectos y garantizando el correcto funcionamiento. Dentro de las limitaciones que todo desarrollo presenta, se buscó maximizar la eficiencia en cada decisión así como la claridad y el estilo del código.

El grupo considera haber incorporado satisfactoriamente conocimientos sobre comunicación entre procesos a pesar de los diversos obstáculos que se presentaron tanto desde lo conceptual como a lo largo de la implementación.

Citas

El código es mayoritariamente de autoría propia (algunas ideas fueron extraidas de la página [www.stackoverflow.com](http://www.stackoverflow.com) pero implementadas por los integrantes del grupo) . El único fragmento de código extraído de Stack Overflow fue la función isFile en application.c, detallada a continuación:

**int**

is\_file (**const char** \*file) {

**struct** stat buf;

stat(file, &buf);

**return** !S\_ISDIR(buf.st\_mode);

}

Se utilizaron las funciones y system calls provistas por el sistema operativo. Para obtener el Hash MD5 de los archivos se siguió la recomendación de la cátedra y se utilizó el programa “md5sum“.

A modo de referencia se utilizó el comando man, los contenidos teóricos provistos por la cátedra así como la bibliografía recomendada, y diversos sitios web además del ya mencionado Stack Overflow.