

Sistemas Operativos

Inter Process Communication

Fermín Gómez

Miguel Di Luca

Pedro Vedoya

Maite Herrán

Decisiones de Diseño


El presente trabajo práctico fue diseñado en el lenguaje C siguiendo la norma de un sistema POSIX para calcular los hashes MD5 de múltiples archivos de forma distribuida. El sistema consta de tres tipos de procesos: el proceso main o aplicación, los n procesos esclavos o hijos del proceso aplicación (siendo n igual a una variable definida en la aplicación llamada numOfSlaves), y el proceso vista.

Para el proceso aplicación, que es el encargado de crear y organizar los esclavos, se establecen los pipes entre él y sus esclavos, reemplazando la entrada y salida estándar de sus esclavos con las entradas a los pipes, que se guardan en un array para poder acceder fácilmente a ellos, y luego por dichos pipes les manda el buffer con los nombres de los archivos. Este buffer es armado concatenando los archivos que fueron mandados por entrada estándar, anteponiendo la cantidad de paths que va a ser mandada, en la que cada path está null terminated. Luego esto es enviado a los esclavos, donde son procesados, y devuelve el md5, que es leído del pipe de cada esclavo, y mandado al view para que sea imprimido. La división inicial procura dividir primero el 40% de la cantidad total de archivos entre los 3 esclavos, y luego se va llenando con otros archivos a medida que los esclavos se van liberando.

Para definir cuándo el proceso aplicación debe enviar más archivos a un esclavo, se decidió utilizar la función select, que espera a que el file descriptor de salida (en este caso el pipe al esclavo liberado) permita que se pueda escribir en él, y cuando esto sucede le envía más archivos para procesar.

Para el proceso esclavo, se recibe un array por entrada estándar con los nombres de los archivos concatenados por el pipe, encabezados por la cantidad de archivos que posee, que luego va iterando sobre cada uno de dichos archivos, agregándole una lista de orden para ser procesadas, y llama a la función popen que consigue su md5. Luego los escribe en su salida estándar para enviarlos al proceso aplicación. En cada pasada de armado de hash, llama a la función select que le permite esperar hasta que reciba el siguiente archivo a procesar, y lo agrega a la lista.

En el caso del proceso vista, se pensaron distintas soluciones. Una de ellas, por ejemplo, fue que cada cierta cantidad de memoria compartida ya leída por la vista, el proceso aplicación podría volver a escribirla desde el principio y así solucionar posibles problemas de escasez de espacio. Descartamos esta solución por el simple hecho de que se irían perdiendo los hashes calculados a medida que avanzara la ejecución.



Para eximir al proceso vista de una espera activa se utilizó un solo semáforo, la aplicación realiza un post cuando envía una tanda (cada tanda finaliza con un '\0'), por lo tanto la vista (que realizará el wait previamente o posteriormente al post de la aplicación) cambiará su estado de espera a uno de preparado, y eventualmente, imprimirá hasta encontrarse con un '\0', luego de eso ejecutará nuevamente un wait, mientras tanto la aplicación no se detiene y continúa calculando otros hashes y escribiendo en la memoria compartida luego del '\0' de la última tanda. La aplicación realiza un post luego de cada tanda, esto permite que la vista pueda saber cuántas tandas pendientes se encuentran listas para ser mostradas.

Se decidió, por último, que el proceso vista vaya recorriendo la memoria compartida hasta encontrar un '\0' o un EOF puesto por la aplicación al final de los hashes ya calculados. Si la vista recibe un EOF significa que la aplicación ya no enviará más hashes, por lo tanto la vista finaliza. Como se explicó anteriormente, si recibe un '\0', significa que la tanda que se estaba imprimiendo finalizó, por lo tanto, la vista deberá esperar a que, mediante el semáforo, la aplicación le indique que hay una nueva tanda disponible. La próxima tanda de lectura se hará desde la posición a la que llegó en la tanda anterior. Así hasta que finalice la ejecución del programa o se termine la memoria compartida.

Desde el proceso vista también se hace un kill del proceso aplicación, utilizando su pid que fue pasado por este mismo. El kill comprueba si le puede enviar una señal al proceso aplicación, de no ser así, devuelve -1 significando que el proceso terminó por lo que la vista comprueba mediante el valor del semáforo si le quedan tandas para imprimir, de ser así imprimirá en pantalla hasta la última tanda que recibió y luego finaliza su ejecución.

Instrucciones de compilación y ejecución

Para el correcto compilado y ejecución del programa, se recomiendan seguir los siguientes pasos, ingresando los comandos dados por terminal, con todos los archivos en el mismo directorio:

1) \$gcc slave.c -o Esclavo -Wall -pedantic -std=c99

2) \$gcc main.c shmSem.c -o Main -Wall -pedantic -std=c99 -pthread -lrt

3) \$gcc view.v -o Vista -Wall -pedantic -std=c99 -pthread -lrt

4) \$gcc testing.c -o Testing (Opcional para testeo del resultado de los hashes, más instrucciones abajo)

5)\$. /Main .* | ./View

Para la ejecución del programa, basta correr “./Main <...>”, donde <...> es o una carpeta del modo “./carpeta/*”, o archivos separados por un espacio “./archivo1 ./archivo2 ...”.

6)./Testing (Opcional)

Testing

El programa tiene una función que le permite verificar que el hash de los archivos haya sido correctamente calculado, es decir va archivo por archivo calculando nuevamente su md5 y comparándolo con el que fue calculado por los esclavos. Esto debe ejecutarse primero compilando el código, segundo corriendo el código principal y finalmente corriendo el testing.

Para esto, se debe correr el comando “gcc testing.c -o Testing”, y luego ubique Testing en la misma carpeta donde se ubica el .txt creado por el programa Main, con todos los archivos a procesar y sus respectivos hashes, o corra nuevamente el programa Main si quiere tener en cuenta el hash de Testing. Finalmente corralo usando “./Testing”, y le dirá si hay problemas, y de haberlos, en que archivo se encuentra.


Limitaciones

El programa encuentra limitaciones a la hora de tener que calcular el hash md5 de una cantidad demasiado grande de paths, no solo por tener que invertir mucho tiempo en todo el proceso de cálculo, sino que también corre el riesgo de que se llene el buffer, y haya errores de sobreescritura.

El programa tampoco se encuentra preparado para lidiar con el caso en el que haya menos files que esclavos, dado que no posee una cantidad dinámica de esclavos. En este caso procesa los paths pedidos, pero no lo hace de manera eficiente,, siempre se terminan creando esclavos de más(estos ocurren en pocos casos, dado que el programa tiene 3 esclavos).

Problemas encontrados

Un problema recurrente fue el de poder lograr una comunicación entre el proceso aplicación y sus esclavos, de manera que la aplicación sepa cuándo matar a los esclavos y cortar el



programa, y que no haga esto prematuramente. Esto se soluciono poniendo un contador que informa al ciclo en cada iteración cuantos archivos fueron procesados hasta ese momento, cuando se procesan todos, deja de entrar en el while y mata a los hijos y luego al programa padre(finaliza).

Otro problema fue el de los semáforos. Nos fue un problema poder decidir si usar uno o dos semáforos, y cómo implementar la memoria compartida con respecto a esa decisión. Se consideró la opción de utilizar dos semáforos, de esta manera, solo escribe la aplicación o lee la vista al mismo tiempo, esta opción funcionaría correctamente, pero no seria la optima. El problema es que mientras la vista lee información anterior, la aplicación podría continuar calculando y escribiendo hashes en zonas a la que la vista no va a leer aún, pero como la aplicación se encuentra bloqueada deberá esperar a que termine la vista. Por lo tanto, como se explicó, se utilizó un solo semáforo, delimitando la zona hasta donde puede leer la vista insertando un '\0' al final de cada tanda.

Fuentes

Código utilizado para calcular el md5:

<https://stackoverflow.com/questions/3395690/md5sum-of-file-in-linux-c>

Fundamentos De Sistemas Operativos - 7° Edición - Autor: Silberschatz, Galvin y Gagne.