Trabajo Práctico 1: Inter Process Communication

Sistemas Operativos

Juan Bensadon
Rodrigo M Navarro Lajous
Esteban Kramer
Agustín Calatayud

Índice

Decisiones tomadas

La principal decisión que tomamos fue elegir el sistema que emplearíamos para intercomunicar al proceso Aplicación con los procesos Esclavos. Se decidió utilizar un sistema de pipes organizado utilizando estructuras que funciona de la siguiente manera: el proceso aplicación contiene un arreglo de estructuras Esclavo, y en cada una se guardan los file descriptors de escritura hacia el Esclavo y lectura desde el Esclavo. El esclavo en sí copia los lados correspondientes a *su* lectura/escritura en sus file descriptors STDIN y STDOUT antes de ejecutar su proceso principal. De esta manera, cada proceso esclavo no tiene más que acceder a dichos FD para comunicarse con la Aplicación, mientras que está simplemente selecciona un Esclavo de su arreglo de estructuras y toma sus file descriptors de ahí.

También tuvimos que elegir cómo implementar la cola de tareas (es decir, de archivos para procesar) del proceso Esclavo. Tras deliberar un poco, llegamos a la conclusión de que el funcionamiento de las pipes nos serviría en sí como una cola, ya que los mensajes recibidos se irían acumulando en orden hasta ser leídos. Hubo, sin embargo, que considerar el tamaño máximo de este "buffer", por lo cual acudimos a la información de man7.com, en la cual se afirma lo siguiente:
"In Linux versions before 2.6.11, the capacity of a pipe was the same as the system page size (e.g., 4096 bytes on i386). Since Linux 2.6.11, the pipe capacity is 16 pages (i.e., 65,536 bytes in a system with a page size of 4096 bytes). Since Linux 2.6.35, the default pipe capacity is 16 pages, but the capacity can be queried..."
De esta manera, aun en el *improbable y peor* de los casos (aquel en que una máquina corra una versión anterior a la 2.6.11), el pipe tendría 4Kb de espacio para guardar nombres de archivos, lo cual es más que suficiente para la mayoría de los propósitos. En el caso de propósitos más extremos, sería entonces ridículo estar utilizando una versión tan antigua de Linux en primer lugar.

Para determinar cuándo la Aplicación le debe enviar más trabajo a un esclavo, se pensó inicialmente en guardar el estado de los esclavos en el arreglo de structs, pudiendo el estado ser IDLE (libre) o BUSY (ocupado). Al enviarle trabajo al esclavo, su respectivo estado en el arreglo se cambiaría a BUSY y, al finalizar su último trabajo, el esclavo enviaría por pipe una señal especial indicando que se encontraba libre. Para evitar las complicaciones de utilizar diversos mensajes distintos en la comunicación por pipe y para reducir su uso en general, se optó por guardar en el arreglo de structs la cantidad de trabajos asignados a cada esclavo, y reducir este valor cada vez que se recibe un trabajo de dicho esclavo. Así, la aplicación lleva cuenta sola de qué esclavos están ocupados y libres, simplificando la sincronización.

Con respecto a qué archivos se pueden pasar al programa decidimos que no se consideren los directorios pero si todos los archivos enviados.

Se decidió utilizar la implementación de shared memory y semáforos de System V IPC.

Con respecto a cómo recorre la view los archivos se pensaron un par de alternativas. La primera fue dejar el número de archivos finalizados en el primer lugar de la memoria. Se pensaron además otras alternativas como por ejemplo cada una cantidad específica de memoria recorrida empezar desde el principio nuevamente para evitar la cantidad limitada de memoria pero eso significaba que los primeros archivos hasheados se perderían en el caso de que la vista lo tomará luego de hacerse la nueva recorrida. También se analizó usar un semáforo binario pero sin tener un número de archivos o alguna forma de definir hasta dónde podía recorrer la view la descartamos a la opción. Finalmente se tomó la decisión de usar un semáforo no binario que básicamente bloquee la vista hasta tener archivos sumando al semáforo con cada archivo analizado y por lo tanto bloquea la vista cuando se acaban los archivos para mostrar.

Compilación y ejecución

El programa debe ser compilado y ejecutado en Linux.

Para la compilación, se debe ejecutar el comando make con el que se compila todo automáticamente.

Para ejecutar la aplicación con los esclavos, se debe ejecutar el archivo application.out, pasando como parámetros los archivos a los que se les quiere calcular el hash. Finalmente, para ejecutar la vista se debe ejecutar el archivo view.out, pasando como parámetro el pid de la aplicación.

Limitaciones

- Por el funcionamiento del comando md5sum, se debe escapar los espacios en un archivo que tiene espacio en su nombre. Por como se pasan los archivos con espacio en argv los espacios no se escapan, por lo cual se asume que los archivos tendrán nombres sin espacios.
- Como se ha mencionado anteriormente, la cantidad de archivos que se puede enviar a un esclavo a la vez es limitada, más allá de que el número máximo sea grande. En casos muy particulares, esto podría suponer un problema.
- El tamaño de la Shared Memory, utilizada para la comunicación entre el proceso Aplicación y el proceso Vista, es limitado, por lo cual se debe elegir a la hora de la compilación y no puede ser modificado después.
- El algoritmo de distribución de archivos es fijo, es decir siempre toma un mismo número de archivos para distribuir por cada esclavo cuando estos se desocupan. Una mejora posible sería adaptar la cantidad de archivos a distribuir según la cantidad de archivos restantes y esclavos libres por ejemplo.
- Al usar un semáforo no binario para la vista que decrementa cada vez que imprime un archivo no se podría utilizar dos procesos vistas al mismo tiempo ya que el proceso no sería útil, se decrementaría el semáforo desde la otra vista y no se verían todos los archivos.

Testeos

Para detectar posibles leaks de memoria, utilizamos valgrind. Al correrlo, no encontró errores de este tipo. A continuación, se muestra el resultado de su ejecución:

```
==1379==
==1379== HEAP SUMMARY:
==1379==     in use at exit: 0 bytes in 0 blocks
==1379==   total heap usage: 7 allocs, 7 frees, 6,920 bytes
allocated
==1379==
==1379== All heap blocks were freed -- no leaks are possible
==1379==
==1379== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
from 0)
==1379== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
from 0)
```

Para los tests se probó si los semáforos bloquean correctamente, dado que la view es un ejemplo concreto de esto vimos que de hecho se bloquean cuando deberían.

Además se testeo el parseo de archivos y además se testeo la creación de esclavos, la distribución de archivos y la comunicación por los pipes.

Fuentes

Shared Memory: http://users.cs.cf.ac.uk/Dave.Marshall/C/node27.html
Semaphores: http://man7.org/linux/man-pages/man2/semget.2.html
Pipe: http://man7.org/linux/man-pages/man7/pipe.7.html