Trabajo Práctico 3 ∼ PSO

Liptak, Leandro - leandroliptak@gmail.com
Reboratti, Patricio - darthpolly@gmail.com
Silvani, Damián - dsilvani@gmail.com

July 14, 2011

Documentación

Inter-process Communication

`pipe` - Pipes

Un pipe está compuesto por dos char devices denominados *endpoints* o extremos. Cada extremo tiene un puntero a un buffer común de tamaño fijo (4Kb), un puntero a su extremo opuesto, y una cola. Uno de los extremos sólo puede leer del buffer, mientras que el otro sólo puede escribir. Los char devices se almacenan en un arreglo de tamaño fijo.

El buffer está implementado como un arreglo de 4Kb circular. Para mayor eficiencia en la utilización del espacio, el buffer se utiliza *circularmente*: Cada char device extremo tiene un puntero de posición y un entero que indica la cantidad de bytes utilizados o disponibles para leer. Los punteros son individuales a cada extremo, pero el entero de "bytes disponibles" es compartido. Cuando el extremo de escritura almacena n bytes, se incrementa la variable de "bytes disponibles" y se avanza el puntero. Cuando un puntero se excede del límite del buffer, comienza de nuevo al principio del arreglo (*wrap around*). Por otro lado, el extremo de lectura sólo puede leer hasta que su puntero de posición no sobrepase el de escritura (puesto que ya no hay más bytes para leer, el resto es basura).

Cuando se intenta leer del buffer y éste está vacío, el proceso se bloquea hasta que el buffer contenga nueva información, y esto únicamente sucede cuando otro proceso escribe en el pipe. Al terminar de escribir, el proceso despierta a todos los que están esperando en la cola del otro extremo, el de lectura. El proceso es análogo cuando un proceso intenta escribir en un buffer lleno. De esta manera, el mecanismo de pipes se utiliza de manera transparente para los procesos, éstos no necesitan conocer el tamaño del buffer interno. Más importante aún, permite

una manera de sincronizar la comunicación entre dos o más procesos de manera óptima, puesto que los procesos se van a dormir cuando tienen que esperar para procesar la información proveniente de otros procesos, y se despiertan exactamente cuando la información está disponible.

La función de escritura devuelve 0 inmediatamente cuando el otro extremo par se cierra. Este comportamiento es conocido como *broken pipe*. La razón es simple: no tiene sentido escribir en un buffer que ya no se puede leer. Por otro lado, en la función de lectura se suma la condición de que el buffer esté vacío. Es decir, si un proceso escribe en el pipe y cierra el extremo de escritura, el proceso que lee del pipe puede leer lo que queda del buffer antes de que la comuncación se corte.

`loader` - Extensión a la carga de procesos

Implementar la llamada al sistema `fork()` resultó más sencillo que la implementación de `loader_load` o `run` en parte por la capacidad de reutilizar funcionalidades diseñadas para estas últimas. Se solicita un nuevo PID, un nuevo directorio de páginas (con el área del kernel mapeada) y se procede a crear el PCB para el proceso hijo. Luego (según la primer implementación de la syscall) se copian todas aquellas páginas presentes en el esquema de paginación del proceso padre y se mapean dichos frames con el contenido copiado y la misma dirección virtual en el esquema de paginación del hijo. La excepción a dicha copia son las páginas de kernel del área de 0 a 4MB y la página de la pila de anillo cero pues el contenido de ésta no era el adecuado para un proceso recién lanzado.

Posteriormente, se arma una pila de anillo 0 tal como lo haría `loader_load` y se mapea en la dirección esperada. Finalmente, se copian los descriptores de archivos incrementando el número de referencias en el descriptor de dispositivo asociado y se encola el proceso en el scheduler.

Una vez implementadas las funcionalidades vinculadas al manejador de memoria, fue necesario pulir el mecanismo de copia de páginas del proceso padre al hijo pues en las entradas de las tablas de páginas había mucha más información que la simple presencia de la página. Por este motivo la copia se extendió también a las entradas de las tablas de páginas transportando así la información presente en ellas al nuevo proceso. Vale aclarar que en algunos casos no se trata de una copia directa de la entrada sino que la información presente en ella/s puede verse alterada.

Manejador de memoria

Memoria on-demand

Brindar dicha funcionalidad fue relativamente sencillo: a medida que se solicitaban páginas con `palloc()` el rango de direcciones virtuales crecía pero la solicitud del frame correspondiente se postergaba. Es así que la entrada en la tabla de páginas correspondiente a dicha página continuaba marcada como no presente pero uno de los bits restantes de dicha entrada se seteaba indicando que la página había sido previamente solicitada. Esto permitía posteriormente, frente al fallo producido a causa del acceso a dicha página poder discernir entre un intento de acceso a una dirección inválida o a una página previamente demandada. Esta lógica se incluyó en el handler del *Page Fault* y si el caso fuera el último mencionado se procede a solicitar un frame y mapearlo en dicha dirección virtual, retornando luego a la instrucción que causó la excepción.

Memoria compartida

Para implementar la funcionalidad que permite compartir una página entre dos o más procesos, primero elegimos uno de los tres bits disponibles para el usuario en la page-table entry, que nos va a servir para marcar la página como compartida. Una vez hecho eso, cuando el proceso mediante syscall solicita compartir una página, se chequea el estado actual de la mísma. En caso que ésta esté presente, se enciende el bit de compartida mencionado antes. Si no está presente se chequea el valor de *requested* (Ver Memoria on-demand). Si la página está *requested* entonces antes de marcarla como compartida, se pide a memoria el page-frame y se mapea a la dirección virtual solicitada (es decir, se completa el proceso que se inició cuando el usuario solicitó esa página y el mecanismo de Memoria on-demand la marcó como solicitada)

Para que la marca de compartida tenga efecto, la rutina que copia las páginas del proceso padre al hijo al llamar al syscall `fork()` reconoce el bit de página compartida y para cada página presente con dicho bit encendido, en vez de marcarla para copiar en la próxima escritura (Ver copy-on-write), sólamente copia la entrada de la page-table del proceso padre al directorio del hijo. De esta manera, en el directorio del proceso hijo habrá una entrada que apunta a la misma dirección de memoria física que el padre, que tiene el bit de presente y el bit de compartida encendidos. Asimismo, si el proceso hijo llama a `fork()`, la rutina encontrará el mismo bit de compartida y la página será accesible también por el nuevo hijo.

A la hora de liberar memoria, cuando un proceso que tiene mapeada una página compartida ejecuta el syscall `exit()` no queremos que la página se libere a menos que éste sea el último proceso que la está usando. Para eso la rutina que libera memoria chequea para todos los directorios, la page-table entry correspondiente

a esa dirección virtual. Si encuentra otro proceso con esa página compartida, entonces no la libera.

Copy-on-write

Para implementar el mecanismo de *copy-on-write* se modificó, como se ha mencionado, aquella sección de la llamada al sistema `fork()` en que se copiaban las páginas de un proceso a otro. En lugar de esto, se procedió a crear el mapeo en el esquema de paginación del hijo, para la misma dirección virtual que el padre e indicando la página como presente, pero con acceso de sólo lectura. Tambíen la entrada del padre es marcada como de sólo lectura y adicionalmente ambas entradas se marcan indicando que han de ser copiadas ante una escritura.

Posteriormente, frente a una eventual escritura se corrobora si la página debe ser copiada o no. En caso de haber sido marcada para ser copiada, pero ser el único proceso con dicho mapeo (pues los otros ya han accedido y han hecho la copia), dicha página se otorga nuevamente con permisos de lectura/escritura sin ser copiada. En otro caso, la copia se efectúa, indicando luego que dicha página ya no necesita copia en el esquema de paginación de dicho proceso.

Tareas

Como lo describe el enunciado, las tareas `krypt` y `memkrypt` leen un archivo (por defecto, el binario del kernel) del disco rígido, lo encriptan con cifrado XOR usando una llave definida en el código, y lo escriben en el puerto serie. Dado que hay partes del proceso que son intensivas en CPU y otras en E/S, la idea es paralelizar estas etapas separándolas en distintos procesos, permitiendo que se comuniquen entre sí con algún mecanismo de IPC. En este trabajo práctico se implementaron dos mecanismos: pipes y memoria compartida.

`krypt`

Esta tarea utiliza pipes para comunicar cada parte del proceso. Se crean un pipe antes de cada fork, de modo que uno de ellos lo comparten el proceso que lee del disco (`read_proc`) y el que encripta (`encrypt_proc`), y otro lo comparten `read_proc` y el proceso que escribe al puerto serie (`write_proc`).

Se pidió la condición de que los procesos transfieran los datos de a bloques de 4Kb, y es por esto que cada proceso tiene un buffer de un bloque que es utilizado para leer y escribir en el pipe correspondiente.

El ciclo de `krypt` es el siguiente: `read_proc` lee de a 4Kb del disco en el buffer y escribe la cantidad de bytes leídos en el pipe. Inmediatamente `encrypt_proc` se despierta porque hay datos nuevos para leer del pipe. Luego de encriptar,

escribe el resultado en el segundo pipe. `write_proc` se despierta, lee el contenido del pipe en un buffer y lo transfiere al puerto serie.

Este ciclo continua hasta que la lectura al disco que realiza `read_proc` devuelve 0 por que ya no hay más datos que leer del archivo (End-Of-File). El proceso cierra el extremo de escritura, y esto inmediatamente causa un "broken pipe" desde el lado de `encrypt_proc`, desencadenando en la terminación de este proceso, y a su vez del tercero, `write_proc`, finalizando correctamente la tarea como era esperado.

El hecho que los pipes permitan ser leídos a pesar de que el extremo de escritura ya fue cerrado es clave para que la tarea termine como coresponde. Si esto no fuese posible, `encrypt_proc` probablemente no llegaría a leer del buffer antes de que `read_proc` termine, desencadenando en una finalización prematura.

`memkrypt`

A diferencia de `krypt`, `memkrypt` utiliza un buffer de 4 páginas compartidas de 4Kb de manera circular. `read_proc` lee del disco y escribe en la página 1, luego hace lo mismo con la página 2, 3 y 4, y vuelve a comenzar por la página 1. La primera dificultad que encontramos es que necesitábamos una forma de sincronizar los procesos, para que uno no sobreescribiera una página que todavía no había sido escrita, o que otro proceso no leyera de una página con información inútil.

Tomando la sugerencia de la consigna del trabajo práctico, escribimos un conjunto de funciones que implementan *semáforos* a nivel de usuario utilizando pipes. De esta manera, pudimos simular un `signal(n)` o un `wait(n)` escribiendo o leyendo `n` bytes en un pipe.

`read_proc` comienza haciendo `signal(4)` para "marcar" las 4 páginas del buffer como disponibles, dado que la tarea acaba de comenzar y el buffer está vacío. El ciclo comienza con un `wait(1)`. Así, cuando el proceso termine de escribir en todas las 4 páginas, no llegue a pisar la página 1 si no fue leída por `encrypt_proc` aún. Luego de trabajar sobre una página, `read_proc` hace `signal(1)` sobre el semáforo de `encrypt_proc` para "avisarle" que tiene una (y sólo una) página para encriptar. La situación es similar en `encrypt_proc` y `write_proc`.

El segundo problema que nos surgió fue el de comunicar la cantidad de bytes leídos por `read_proc` a los otros dos procesos. Para resolverlo, pedimos una página extra y la marcamos como compartida. En esta página almacenamos el entero que corresponde a la cantidad de bytes leídos del disco por `read_proc`. Luego `encrypt_proc` y `write_proc` leen esa variable para saber cuantos bytes deben encriptar y escribir al puerto serial, y cuando deben terminar. Para evitar condiciones de carrera al leer o escribir sobre esa variable, utilizamos un *mutex*.