Trabajo Práctico 3

Programación de Sistemas Operativos

1er Cuatrimestre 2011

1. Introducción

1.1. Objetivo

En este tercer trabajo práctico utilizaremos las estructuras internas que el procesador ofrece al sistema opertivo para implementar algunas funcionalidades de manera eficiente que de otra forma sería imposible implementar.

1.2. Entrega

Fecha límite de entrega: **Jueves 30 de Junio - 16hs**

La entrega del trabajo práctico debe realizarse en forma digital e impresa. Se debe enviar un correo electrónico a `pso-doc@dc.uba.ar` cuyo asunto (*subject*) sea "Entrega TP3 - Apellido1, Apellido2, Apellido3", conteniendo:

- Nombre, apellido y LU de los 3 (tres) integrantes del grupo
- Código fuente, en un archivo comprimido.
- Informe en formato digital.

Paralelamente, se debe entregar **impreso** el informe del trabajo práctico en una carpeta. No se debe imprimir el código fuente. Tampoco es necesario entregar el código en un medio de almacenamiento físico junto con el informe. El informe del trabajo debe contener la documentación de lo realizado, explicando *cómo* fue realizado, en especial documentando las desiciones que debieron tomar.

1.3. Alcance

El alcance de esta tercer entrega del trabajo cubrirá los siguientes temas o unidades del kernel:

- Inter-Process Comunication (IPC) via *pipes*.
- IPC via memoria compartida.
- Manejo de memoria *on-demand*
  - asignación de memoria *lazy*
  - copy-on-write

1.4. Organización general

Para esta etapa, se agregan al trabajo realizado anteriormente nuevos módulos que exportan la funcionalidad pedida y se modifican otros realizados anteriormente.

2. Inter-process Communication

Comunicación entre procesos o *Inter-process Communication* permite a un proceso, que en principio debe tener la ilusión de estar ejecutando solo en la PC, comunicarse con otros procesos que están también ejecutando. Hay varios mecanismos para realizar esto, uno de ellos es la utilización de *pipes*.

2.1. `pipe` - Pipes

Un pipe es un canal de comunicación unidireccional con dos extremos, donde todo lo que se escribe en un extremo se lee por el otro. El pipe además cuenta con un buffer interno o capacidad del pipe, de modo que es posible escribir en el pipe una cierta cantidad de bytes sin que sean leídos, aún, en el otro extremo. Estos bytes escritos esperan en este buffer a ser leídos en el otro extremo.

Ejercicio 1: Implementar la función `pipe_init` que inicializa el módulo.

Para exportar al usuario la posibilidad de utilizar pipes se aprovechará un mecanismo existente: los *char devices*. Sin embargo, cada **extremo** del pipe tendrá su *char device* asociado, de esta forma se pueden cerrar independientemente el extremo de lectura y el de escritura.

Ejercicio 2: Implementar la función `pipe_read` que permita leer del pipe. Esta función deberá bloquearse si no hay nada que leer del pipe, aunque puede volver sin haber leído todo lo solicitado. Devuelve la cantidad de bytes leídos o 0 en caso de que no haya nada para leer y nunca más vaya a haber algo para leer, devido a que ya se cerró el extremo de escritura.

Ejercicio 3: Implementar la función `pipe_write` que permita escribir en el pipe. Esta función deberá bloquearse en el caso de que el buffer interno del pipe esté lleno, hasta que haya lugar para continuar escribiendo. Además, deberá devolver la cantidad de bytes escritos o 0 en caso de que no sea posible escribir en el pipe dado que el otro extremo ya fue cerrado.

Ejercicio 4: Implementar la función `pipe_open` que crea un nuevo pipe rellenando el array pasado por parámetro con los dos punteros a los char devices. La primer posición corresponde al extremo de lectura y la segunda posición al extremo de escritura. En caso de error deberá devolver un número negativo.

Ejercicio 5: Implementar la syscall `pipe` que cree un nuevo pipe y devuelva 0, o devuelva un número negativo en caso de error. Recuerde que el usuario debe recibir dos *file descriptors* y no los punteros a las estructuras del kernel.

Ejercicio 6: Implementar la función `pipe_flush` que cierre el extremo correspondiente del pipe. Tenga en cuenta que esta función debe desbloquear a los procesos que están esperando en el otro extremo del pipe, si los hubiera.

2.2. `loader` - Extensión a la carga de procesos.

Uno de los problemas de los pipes y otros mecanismos similares es que no es posible realizar un pipe entre dos procesos que ya están ejecutando. Para que efectivamente tenga utilidad este mecanismo, el *char device* debe ser conocido por ambos procesos, para lo cual introducimos un nuevo método para crear procesos.

Ejercicio 7: Implementar la syscall `fork` que duplique el proceso actual de modo que el nuevo proceso creado tenga una copia exacta de la memoria RAM del proceso anterior y una copia de los file descriptors con los mismos números. La única diferencia será entonces que un proceso recibirá el *process id* del proceso nuevo mientras que el nuevo recibirá un 0 como valor de retorno. En caso de error, se deberá devolver un número negativo.

Tenga en cuenta que esto no cambia el comportamiento del otro mecanismo para crear procesos, con la syscall `run`, donde no se comparte nada.

3. Manejador de memoria

La unidad de paginación del procesador permite ofrecer de manera eficiente cierta funcionalidad al usuario. Por ejemplo, si el usuario pide 1MB de memoria pero luego utiliza efectivamente sólo 100KB, los restantes 900KB de memoria física podrían haberse usado para otro proceso.

3.1. `mm` - Extensiones al manejador de memoria

Para los siguientes ejercicios necesitará registrar una rutina de atención de interrupción del Page Fault.

Ejercicio 8: Modifique la función `mm_init` para registrar la rutina de atención de interrupción del Page Fault. En caso de que el programa haga un acceso inválido deberá cerrarlo como si lo hiciera llamando a `exit`.

Memoria on-demand

Ejercicio 9: Modifique la syscall `palloc` para que no reserve la memoria física de la página pedida hasta tanto el usuario no haya efectivamente utilizado la página. Tenga en cuenta que deberá diferenciar este caso de aquél en que realmente se realizó un acceso inválido. En el caso que no haya más memoria disponible en el sistema se deberá terminar el proceso como si lo hiciera llamando a `exit`.

Memoria compartida

Otro mecanismo que puede ser utilizado como comunicación entre procesos, aunque no como sincronización de manera eficiente es el de memoria compartida. Nuevamente, para que dos procesos puedan compartir memoria deberán conocerse, es decir, haber hecho fork del mismo proceso. De este modo, lo único que debemos hacer es *marcar* páginas de memoria como compartidas.

Ejercicio 10: Implementar la función `mm_share_page` que marque una página de memoria *virtual* como compartida. En caso de error, ya sea porque la página de memoria virtual no estaba mapeada o no era del usuario se debe devolver un número negativo.

Ejercicio 11: Exportar la syscall `share_page` que le permita al usuario marcar una página de memoria como compartida, con la misma semántica que `mm_share_page`.

Ejercicio 12: Modificar la syscall `fork` para que al copiar la memoria para crear el nuevo proceso efectivamente permita que la memoria sea compartida por los dos procesos. Tenga en cuenta que si nuevamente ocurre un `fork` en alguno de estos procesos, se deberá compatir la misma página entre entre todos los procesos involucrados.

Copy-on-write

Uno de los efectos negativos del uso de `fork` es la necesidad de copiar toda la memoria del proceso, cuando tal vez el nuevo proceso sólo utiliza una fracción, o incluso ambos utilizan partes de la memoria, como el código a ejecutar, sólo para leer los datos de allí. La manera de aprovechar eficientemente la memoria en este es caso es con la utilización de lo que se conoce como *copy on write*.

Este mecanismo evita la copia física de la memoria que realiza el fork hasta tanto sea necesario. Para ello, los dos procesos involucrados en el fork comparten la memoria física, como ocurría en la memoria compartida, pero sin la posibilidad de escribir.

Al ocurrir la primer escritura se realiza efectivamente la copia y se permite la escritura por parte del proceso en cuestión. Tenga en cuenta que de haber sucesivos forks deberá compartir la memoria de sólo lectura entre más de dos procesos y que si todos los procesos escriben en una misma página el último en hacerlo no necesitará efectivamente realizar una copia.

Ejercicio 13: Modificar la syscall `fork` y la rutina de atención de interrupción del Page Fault para que realice copy-on-write de la memoria al realizar un fork.

Ejercicio 14: Modificar la funciones `loader_exit` para que libere la memoria utilizada por los procesos involucrados con las nuevas modificaciones de manejo de memoria.

4. Tareas

Dado que las lecturas y escrituras sobre los distintos dispositivos de hardware son bloqueantes, un proceso que lea datos de un dispositivo, los procese y los escriba en otro dispositivo deberá utilizar cada uno de los recursos (dispositivo de entrada, procesador y dispositivo de salida) en turnos. Para resolver este problema y mejorar el *troughput* del sistema, se utilizarán tres procesos, comunicados entre sí.

De este modo, el primer proceso leerá del dispositivo de entrada, bloqueándose a la espera del siguiente dato. Por su parte, el segundo proceso leerá los datos enviados por el primer proceso y los procesará, mientras que el tercer proceso estará mayormente bloqueado esperando que los datos se escriban en disco

Ejercicio 15: Implementar el programa PSO `krypto` que al iniciar cree dos procesos más con `fork()`, de modo que los tres procesos ejecuten las funciones `reader`, `encrypt` y `writer`, respectivamente, las cuales al terminar de ejecutar salen del programa. La función `reader` leerá el archivo "`/disk/kernel.bin`" de a bloques de 4KB y los escribirá en un pipe al siguiente proceso, `encrypt`. Por su parte, este proceso encriptará el buffer de 4KB haciendo un `xor` contra un buffer fijo y conocido y luego escribirá el resultado en otro pipe, al proceso `writer`. Por último, este proceso leerá el contenido del pipe y lo escribirá en el archivo "`/serial0`".

Una forma de utilizar semáforos en el usuario cuando no se cuenta con tal sistema subyacente es la utilización de pipes. De este modo, puede utilizarse la llamada a `read` de un caracter como un `wait(1)` y la llamada a `write` de un caracter como un `signal(1)`. Esta implementación permite además un comportamiento extra al cerrar los extremos del pipe.

Ejercicio 16: Implementar el programa PSO `memkrypto` similar al `krypto` pero de modo que la comunicación de los datos en sí sea a través de buffers de memoria compartida, mientras que la sincronización sea a través de pipes. Para ello, utilizar 4 buffers de 4KB cada uno de memoria compartida entre los tres procesos y tres pipes: de `reader` a `encrypt`, de `encrypt` a `writer` y de `writer` a `reader`.

Los 4 buffers se utilizan circularmente, es decir, luego de escribir en el cuarto buffer se comienza a utilizar nuevamente el primero, si está disponible. Cuando el mismo está disponible para que lo utilice el siguiente proceso, ya sea porque se dejó de utilizar como ocurre entre `writer` y `reader`, o porque el dato ya está escrito para ser leído por el siguiente proceso, entonces se envía un `signal` avisando esto.