PRÁCTICO 2

Ejercicio 1

Objetivo: Implementar dos llamadas al sistema (system calls) que permitan a un proceso ser y no ser visualizado a través del comando ps o top

Los llamados al sistema deben tener los siguientes cabezales:

`long ocultar()` Luego de ser ejecutada el proceso no debe ser visualizado.

`long visualizar()` Luego de ser ejecutada el proceso debe ser visualizado.

Ejercicio 2 (Obligatorio)

Objetivo: Estudiar el uso de diferentes formas de pedir memoria

El kernel de Linux obtiene los PCBs de los procesos (*task_struct*) utilizando el *slab allocator*. Se desea modificar este comportamiento para que utilice:

- *kmalloc* y *kfree* para crear y borrar los PCBs.
- *alloc_pages* y *free_pages* para crear y borrar los PCBs

Se pide:

1. Analizar la performance de las dos variantes y de la implementación original en cuanto a la velocidad de creación de procesos (por ejemplo con un programa que haga una gran cantidad de forks).
2. Explicar las diferencias observadas.

Notas:

- Los caches del *slab allocator* se crean con la primitiva kmem_cache_alloc_node.
- Puede usar la función time para medir el tiempo de ejecución de un proceso.

Ejercicio 3 (Obligatorio)

Objetivo: Analizar las regiones de memoria de los procesos

Como se vio en el teórico Linux representa el espacio de memoria de los usuarios como una serie de intervalos de memoria que un determinado proceso tiene permiso para usar. Estos están representados en el kernel como un grupo de regiones de memoria (`vm_area_struct`) para las que se tiene, entre otros datos, la dirección de inicio y fin, los permisos y los datos del archivo mapeado si corresponde.

Todos estos datos pueden verse desde el archivo `/proc/pid/maps` (donde pid es el identificador del proceso cuyos datos se quieren observar).
En cada línea del archivo maps se muestran en orden de derecha a izquierda:

- El rango de direcciones virtuales de la región (inicio-fin).
- Los permisos de la región (rwx).
- Si es compartida (s) o privada (p).
- Si mapea un archivo el offset dentro del archivo de la sección que se está mapeando.
- Los siguientes campos identifican el archivo que se está mapeando en forma única. Lo primero que se indica es el dispositivo (major:minor del dispositivo correspondiente), su inodo y el nombre del archivo. En caso de ser una región que no mapea un archivo todos estos campos están en 0.

A partir de estos datos se puede deducir a que corresponde cada una de las regiones de un programa que pueden ser entre otras:

- El código del programa (lectura y ejecución, mapea el binario del programa).
- Datos con valor inicial del programa (lectura y escritura, mapea el binario del programa).
- Datos sin valor inicial del programa (lectura y escritura, no mapea un archivo)
- El código de la bibliotecas dinámicas (lectura y ejecución, mapea el binario de la biblioteca).
- Datos con valor inicial de las bibliotecas dinámicas (lectura y escritura, mapea el binario de la biblioteca).
- Datos sin valor inicial de las bibliotecas dinámicas (lectura y escritura, no mapea un archivo).
- Heap del proceso.
- Stack del proceso.

Se pide:

1. Escribir un programa que tenga variables globales con y sin valor inicial y variables locales, tanto escalares como vectores y que utilice la variable errno. El programa debe escribir su mapa de memoria y la dirección de las variables definidas, de errno y de la función main. Se pide analizar los resultados del programa con respecto a la localización de los datos en las regiones.
2. Escribir un programa que realice una llamada malloc reservando 200 KiB de memoria y luego la libere usando free. Mostrar el mapa de memoria antes de pedir la memoria, luego de pedirla y luego de devolverla. Analizar los resultados.
3. Escribir un programa que calcule el coseno de un valor pasado como parámetro y muestre su mapa de memoria. La función cos se encuentra definida en la biblioteca libm por lo que se desea tener dos versiones de este programa: una compilada con bibliotecas estáticas y otra con bibliotecas dinámicas. Se desea analizar los resultados de ambos mapas de memoria.

4. Escribir un programa que mapee un archivo a memoria. Mostrar el mapa de la memoria antes del mmap, mientras el archivo está mapeado y después del munmap. Pruebe con distintos parámetros de mmap y analice los datos obtenidos.

5. Escribir un programa que cree un hijo mediante fork. Mostrar el mapa de memoria del padre antes del fork y el de los dos procesos luego del fork. Analizar los resultados.

Notas:

- Para mostrar el mapa de memoria de un proceso se puede usar:

  `system(‘‘cat /proc/pid/maps’’).`

- Para imprimir las direcciones de memoria se puede usar:

  `printf (‘‘main %p errno %p’’, main, &errno).`

- Para compilar el programa de la parte 3 en forma estática se debe usar:

  ```
  gcc programa.c /usr/lib/libm.a -o programa
  ```

  y para compilarlo en forma dinámica:

  ```
  gcc programa.c -lm -o programa
  ```

- Para mostrar las bibliotecas dinámicas contra las cuales está compilado un programa se puede usar el comando *ldd*.

Ejercicio 4 (Obligatorio)

Objetivo: Implementar un componente para el intercambio de mensajes entre procesos en Linux.

El componente tendrá capacidad para guardar un solo mensaje a la vez y, además, el mensaje tendrá un tiempo de vida limitado.
El componente brindará los siguientes llamados a sistema:

`long tsosend(const char * msg)` Para ser ejecutado por el proceso que quiera enviar un mensaje.

`long tsoreceive(char * msg)` Para ser ejecutado por un proceso que quiera recibir un mensaje.

Consideraciones:

- Un mensaje enviado es solamente recibido por un proceso que ejecute una recepción (el primero que lo haga).

- Si el tiempo de vida del mensaje se vence (ningún proceso pidió la recepción en el transcurso del tiempo de vida), el mensaje es desechado por el componente.

- Si un proceso pide la recepción de un mensaje y no hay mensaje enviado, se debe retornar el mensaje nulo (\0).

- El envío de un mensaje anula un mensaje anterior que todavía este en el componente. El tiempo de vida se resetea.
- El tiempo de vida de un mensaje dentro del componente es de 15000 (interrupciones de reloj).
- Los mensajes tienen un largo máximo de 64 bytes.
- Es necesario utilizar primitivas de sincronización (ej. semáforos) para el manejo de las estructuras dentro del componente.

Se pide:

1. Implementar el componente que permita brindar la funcionalidad propuesta. Se debe utilizar una workqueue para realizar el envejecimiento y eliminación del mensaje si finalizó su tiempo de vida.

Notas:

- Se brindan el archivo *tso-4.7.0.tar.bz2* que contiene la estructura del componente para el núcleo correspondiente. El archivo deberá ser abierto en el directorio raíz de los archivos fuentes del núcleo. Luego de abierto se debe activar la opción TSO->TSO_MSG en los menús de configuración del kernel (make menuconfig). La opción aparece en el menú principal debajo de *Cryptographic API*.
- El archivo tso.c contiene un esqueleto tentativo del componente.