Procesos en Memoria y Disco en SODIUM

Juárez Nelson, Grimau Darío, Saura Alan, Gronski Fernando, Barbieri David
Universidad Nacional de La Matanza

Resumen: La investigación trata sobre el manejo de procesos en disco y en memoria para el sistema operativo SODIUM. El objetivo de la investigación es implementar y dejar funcional una llamada genérica para crear procesos con pasaje de parámetros; y además, implementar la funcionalidad de permitir copiar procesos desde memoria a disco, dejando una interfaz simple para facilitar la compatibilidad de una conexión futura con el Sistema de Ficheros (File System).

Palabras claves: Proceso, Partición, Exec , File System, Archivo , Fork, Disco

1. Introducción

Al comenzar a investigar acerca de cómo se manejan los procesos en SODIUM nos encontramos con funciones que se implementan bien y otras que no, en algunos casos para determinadas necesidades será conveniente crear algunas nuevas o reutilizar otras. Principalmente, nuestro punto de interés es tratar de implementar la familia de funciones "EXEC", las cuales ejecutan un proceso luego de que el mismo sea creado, como objetivo se impone copiar los procesos existentes en memoria principal hacia disco en el momento de booteo de la máquina. Esto es de importancia dado que sólo se cargarían en memoria los procesos que se estarían por ejecutar. Al lograr esto, estaremos dejando puertas abiertas para que en un futuro se pueda implementar un módulo de Swapping en SODIUM, y a su vez, dejar una conexión fácilmente extendible para cuando se incorpore el módulo de File System. Actualmente en SODIUM hay algunas funciones para crear procesos, pero no soportan pasaje de parámetros ni tampoco está estandarizado, por lo tanto se las debe reconstruir o buscar la manera de modificarlas para que cumplan con nuestros objetivos. En dichas funciones se encontraron fallas tanto en la creación de determinadas estructuras de datos como en la llamada a la función fork de archivos particulares. Para poder llevar a cabo el objetivo planteado se tendrá que estudiar el flujo de la creación de procesos, las estructuras que manejan y todo lo necesario para resolver el problema.

Frente a esta situación, nos planteamos los siguientes objetivos.

- Implementar una función que cargue todos los programas usuario alojados en la carpeta /usr a disco.
- Crear una tabla de partición en la cual se alojen las direcciones tanto de inicio como de fin de dichos programas en un dispositivo específico (esto sólo a nivel de cumplir el objetivo, luego se lo reemplazará con el File System).
- Crear una tabla en la cual se almacenen los archivos binarios generados por los programas usuario.
- Dejar una interfaz simple para la conexión de los módulos Swapping y File System.
- Generar una función que realice la búsqueda de un archivo binario en el dispositivo utilizado.
- Implementar una función que cargue en memoria un archivo binario.
- Generar una función que permita a un proceso reemplazarse a si mismo por un archivo binario desde disco.
- Implementar una función que cree un proceso a partir de un archivo binario.
- Implementar todos los comandos de la familia "exec" para ejecutar procesos.
- Generar una función que valide si un archivo binario fue modificado o no.

2. Elementos de trabajo y metodología

A continuación detallaremos los problemas, inconvenientes y modificaciones que se realizaron para cumplir con los objetivos impuestos al comienzo del presente documento:

El primer inconveniente con el que nos encontramos es la falta de documentación en ciertas partes del código, la incomprensión de ciertas funciones ocasionadas por la falta de comentarios concisos

que reflejen de una manera simple la funcionalidad de lo que se hace, la dificultad de encontrar donde se produce la creación de los procesos, el traspaso de los procesos usuarios a memoria (utilizada como disco) , el tamaño que ocupa cada proceso, así como la dirección de inicio donde se aloja dicho proceso y la lógica de ejecución de un proceso. Algunas de estas dudas fueron despejadas por profesionales en la materia y muchas otras investigando el código del SODIUM, en todo momento se recurrió a información de la red para terminar de comprender la situación.

En lo que conlleva a la investigación del código se descubrió que la carpeta /usr de la raíz del SODIUM está dedicada a automatizar la compilación de los procesos del usuario. Aquí, se colocan los archivos fuentes con extensión ".c", y al momento de cargar el Sistema Operativo, los mismos se compilan mediante el script "listar_binarios_usuario" localizado en la carpeta herramientas, mediante este script se genera un archivo binario por cada archivo fuente. Más adelante, entre otras cosas, se llama a la función crearProceso(). Esta función no es la forma estándar de instanciar un proceso, ya que debería realizarse mediante un fork y luego algún comando de la familia "exec", como ser "execl", "execlp", "execle", "execv" o "execvp". Además, la misma pasa el enviroment por defecto, y sólo un parámetro: el nombre del binario sin el .BIN. De todas maneras, dentro de crearProceso() hay una llamada a una función denominada "iFnLeerCabeceraEjecutable", esta función lo que hace es obtener toda la información de la cabecera del archivo ejecutable. Se envía por parámetro a la función una estructura del tipo stInfoExe, y la misma se llena con los datos relevantes del proceso, tales como la posición inicial de memoria y el tamaño. A partir de estos datos pudimos comenzar a crear los primeros comandos, utilizando las nuevas funciones de lectura/escritura en disco implementadas en el SODIUM.

En primer lugar hicimos el comando "vFnFormatParte", que crea una partición en un determinado dispositivo, dedicada al almacenamiento de procesos usuario. Además, la función inicializa la tabla necesaria para la administración del espacio de la partición, además de inicializar la tabla de administración de procesos en esa misma partición. La llamada a la función desde la línea de comandos es la siguiente:

*vFnFormatParte(char *stAux, char *stCantsec)*

Para verificar la correcta generación de dicha tabla de partición, nos vimos en la necesidad de crear una función que nos permita visualizar la correcta creación de la tabla de partición, con lo cual hicimos un comando para obtener información sobre la misma. Dicha función se llama "vFnVerTablaParticion". La tabla de particiones de los dispositivos está estructurada de la siguiente manera:

dispositivo | sector comienzo partición | cantidad sectores partición

En segundo paso, implementamos un comando llamado "vFnCopiarEjecutableDisco", que lo que hace es copiar un proceso alojado en memoria a disco y además lo agrega a la tabla de de binarios. La idea es tratar de hacer una función genérica que no sólo copie a disco sino hacia cualquier dispositivo IDE conectado en el bus de datos. El objetivo de dicha función es poder cargar todos los procesos usuarios en disco en vez de cargarlos en memoria, como se hace actualmente, y solo levantar a memoria cuando se los utilice. La llamada a la función desde la línea de comandos es la siguiente:

*vFnCopiarEjecutableDisco (char * cDispositivo, char * cNombre_archivo_bin)*

Para administrar y ver el espacio ocupado por los procesos en la partición, se creó el comando "vFnVerTablaBinarios", que lo que hace es mostrar cómo están alojados los procesos en cada partición., en dicha tabla se puede ver el sector de inicio y la cantidad de sectores que ocupa dicha binario. La tabla de procesos está estructurada como se muestra a continuación:

nombre_archivo |dispositivo | sector inico | cantidad sectores

Se creó una función para poder determinar la ubicación de un archivo en la tabla de procesos donde fue almacenado al comienzo, esto es necesario porque luego se necesitará ejectutar dicho archivo binario y mediante el acceso a la tabla obtenemos la dirección de inicio y fin del proceso en el dispositivo.

*iFnBuscarArchivoHdd (char * cNombreArchivo, struct _PosicionEnDisco sUbicacionEnDisco)*

Se creó una función que reemplaza un proceso por otro y ejecuta desde el comienzo, Esto es usado para reemplazar el proceso cuando se ejecuta algún comando de la familia "exec". Como parámetros recibe el PID del proceso a reemplazar, la ubicación en disco del proceso que quiero que reemplace al anterior, el nombre del archivo binario y el offset.

*iFnReemplazarProcesoHdd(unsigned long ulPid, struct _PosicionEnDisco sUbicacionEnDisco, char * stArchivo, unsigned int uiOffset)*

Creamos una función que se encargue de leer la información de la cabecera de un archivo ejecutable, mediante esta lectura podemos obtener el tamaño que ocupa dicho proceso y también la dirección de inicio del mismo, esto es útil para cargarlo en la tabla de procesos y posteriormente utilizarlo para traer procesos desde disco, la sintaxis de la función es la siguiente:

iFnLeerCabeceraEjecutableHdd (struct _PosicionEnDisco sPDisco, stuInfoEjecutable pstuInfo)*

Por otra parte, para terminar de complementar el objetivo propuesto implementamos la función "lFnSysExecveHdd", la cual tiene como fin ejecutar un determinado archivo binario para un proceso que será enviado por parámetro, como resultado obtenemos la ejecución de un programa usuario. La función realiza la funcionalidad de la familia exec, la cual ejecuta un proceso. Por parámetros se le pasa el nombre del archivo binario que se quiere ejecutar y los parámetros, a lo sumo pueden ser dos. La sintaxis es la siguiente:

*lFnSysExecveHdd (const char *cARCHIVO, char * cArgv[], char * cEnvp[])*

Por ejemplo: si creamos el programa llamado SUMA que lo que hace es sumar 2 números pasados por parámetro, vamos a crear el proceso de la siguiente manera: "lFnSysExecveHdd ("SUMA.BIN", 1, 2)". El proceso recibe como parámetros los números 1 y 2, los suma, y los muestra por pantalla.

Al igual que antes, la ausencia de un File System acarrea el problema de implementar ciertos miembros de la famila de funciones Exec , como por ejemplo , execl , execle , y execv, ya que las mismas tienen como parámetro un path, lo cual es imposible de implementar sin un Sistema de Ficheros, por lo cual procederemos a implementar las variaciones de exec que no pidan un path en sus parámetros. Igualmente se deja una interfaz para implementar en un futuro el file system y así continuar con la implementación para toda la familia exec. Para todas las funciones y variables utilizadas se creó un archivo "exec.c" el cual contiene todo lo necesario para poder utilizar la familia exec.

Uno de los problemas encontrados al probar el programa fue que cuando se modificaba algún archivo binario y luego se lo quería ejecutar el mismo perdía estabilidad, para solucionar esto se utilizó un método que determine, antes de ejecutar el proceso, si su archivo binario fue sufrió modificaciones, en caso que fuese cierto evita la ejecución informando el problema. El método utilizado es CRC el cual

asigna un número al archivo binario, determinado por una serie de cálculos, al momento de quererlo ejecutar se calcula dicho número y si cambia significa que el binario fue modificado, muestra un mensaje y corta la ejecución.

3. Conclusión

En lo que respecta a lo propuesto como objetivo al comienzo del documento, se logró llegar a dicho fin. Hoy en día en SODIUM contamos con un comando que realice la funcionalidad de la familia EXEC, para llegar a dicho objetivo se tuvo que generar una serie de funciones que nos ayudaron a cometer el objetivo. Lo primero que se realizó fue una función que guarde en disco todos los procesos usuario que anteriormente se almacenaban en memoria (parte dedicada al almacenamiento), la función que realiza dicho cometido es *vFnCopiarEjecutableDisco.* Otra necesidad que tuvimos fue la de crear una partición en el dispositivo de almacenamiento en el cual se guarde una tabla con datos necesarios de los procesos existentes, el propósito de dicha tabla es dar la funcionalidad de un File System dado que hoy en día no contamos con dicho módulo en SODIUM, la función que comete dicho fin es *vFnFormatParte.* Una vez que se implemente el File System dicha función será obsoleta y estas tablas y espacio reservado no deberán de utilizarse.

Dado que todavía no existe un File System y que algunas de las funciones EXEC (como ser execl, execle y execv) reciben como parámetro el nombre y la ruta del archivo a ser ejecutado y el resto de las funciones de la familia (execlp y execvp) reciben el nombre del archivo, se utiliza una función genérica para todas dejando lugar a la implementación del file system en un futuro. Las interfaces y la nomenclatura utilizada para la familia de funciones EXEC es la siguiente:

e - Se pasa a la función exec un array de punteros a las variables de entorno.
l - Se pasa a la función exec los argumentos introducidos por línea de comando, cada uno individualmente, hasta ingresar un 0.
p - La función exec utiliza la variable de entorno PATH para encontrar la ruta absoluta del archivo pasado por parámetro.
v - Se pasa a la función exec los argumentos mediante un array de punteros.

Función : **lFnSysExecl**
Descripción : Ejecuta un proceso a partir del path del archivo binario pasado por parámetro. La letra 'l' viene de "Command-line arguments", por lo tanto esta implementación de exec recibe por parámetro todos los argumentos separados por coma, hasta que el último argumento sea un '0'.
Declaración: int lFnSysExecl(char const *cPath, char const *cArg0 , …)

Función : **lFnSysExecle**
Descripción : Ejecuta un proceso a partir del path del archivo binario pasado por parámetro. Además recibe por parámetro todos los argumentos separados por coma, hasta que el último argumento sea un '0', y también recibe por parámetro un puntero a vector que contiene variables del entorno.
Declaración: int lFnSysExecle(char const *cPath, char const *cArg0, … , char const *cEnvp[])

Función : **lFnSysExeclp**
Descripción : Ejecuta un proceso a partir del nombre del archivo binario pasado por parámetro ((se buscará el archivo en el directorio indicado por la variable de entorno PATH).Recibe por parámetro todos los argumentos separados por coma, hasta que el último argumento sea un '0'.
Declaración: int lFnSysExeclp(char const *cFile,char const *cArg0, …)

Función : **lFnSysExecv**
Descripción :Ejecuta un proceso a partir del path del archivo binario pasado por parámetro. Recibe los argumentos mediante un vector.
Declaración: int lFnSysExecv(char const *cPath, char const *cArgv[])

Función : **lFnSysExecvp**
Descripción : Ejecuta un proceso a partir del nombre del archivo binario pasado por parámetro (se buscará el archivo en el directorio indicado por la variable de entorno PATH). Recibe los argumentos mediante un vector.
Declaración: int lFnSysExecvp(char const *cFile, char const *cArgv[])

Para finalizar se hicieron modificaciones que corresponden a validaciones y ayuda sobre la utilización de las funciones desarrolladas. Un ejemplo de estas validaciones es verificar que cuando se realiza la modificación de un archivo binario, muestre un mensaje de error y aborte la ejecución. Para dicho fin se generó una validación CRC, el cual asigna un número al archivo binario y antes de ejecutar verifica que dicho número sea el mismo, si cambió significa que el binario fue modificado y, por lo tanto, no se debería permitir la ejecución del mismo. Para aportar una ayuda al usuario sobre la correcta utilización de los comandos implementados a nivel de parámetros enviados y sintaxis se genera una función de ayuda en la cual se muestra un ejemplo con todos los parámetros que se deberían poner, el valor devuelto por dicha función y una breve descripción de su fin.

4. Bibliografía

[CHA01] Charles M. Kozierok, “The PC Guide: SFF-8020 / ATA Packet Interface (ATAPI)”, 2001-04-17.

[TEC96] Technical Committee T13 Process, “PR Memory and Process (MR)”, Global Engineering Documents, 1996.