Cátedra de Sistemas Operativos UNLaM
Trabajo Práctico Nro. II
Implementación estándar POSIX

Gutierrez Jose L.
Maugeri Daniel J.
Medina Emanuel D.
Regueira Vazquez Walter D.
Sureda Martín

Resumen. El presente trabajo contiene un marco teórico complementario al trabajo práctico solicitado por la cátedra. El tema principal de este documento es dar a conocer la implementación de un comando del SO SODIUM respetando el estándar POSIX.

Contenido

1 Introducción

Cuando un usuario del sistema desea o necesita ejecutar algún comando, como por ejemplo una lectura a una unidad de disco o un programa del usuario, un proceso llamado intérprete de comando o "shell" lee los comandos a partir de una Terminal. La interfaz entre el Sistema operativo y los programas del usuario se define como el conjunto de instrucciones ampliadas que proporciona el sistema operativo. A estas instrucciones se las denomina llamadas al sistema (system calls). Los programas del usuario se comunican con el sistema operativo y le solicitan servicio mediante las llamadas al sistema. A cada una de estas llamadas le corresponde un procedimiento de la biblioteca al que pueden llamar los programas del usuario. La solicitud de un servicio al sistema operativo es tratada como una interrupción a nivel de software (trap) y, para atenderla, el sistema pasa de modo usuario a modo kernel.

Para que la aplicación usuario pueda emitir una system call, es necesario realizar la llamada por medio de una API. Las APIs (Application Program Interfase, Interfaz de Programación de Aplicaciones) son un conjunto de funciones residentes en bibliotecas que permiten que una aplicación de usuario pueda realizar pedidos al sistema operativo. Un ejemplo de esto son las APIs de Unix basadas en el estándar POSIX. Este estándar agrupa una serie de estándares y recomendaciones del instituto IEEE, que intentan definir la base para la construcción de sistemas operativos abiertos. El acrónimo de POSIX es Portable Operating System Interface (la X proviene de Unix).

El objetivo de este segundo trabajo práctico es precisamente implementar uno de los comandos del SO SODIUM respetando la sintaxis y semántica que propone el estándar POSIX mencionado.

Para llevar a cabo el objetivo propuesto, se ha determinado investigar y modificar el comando PS, el cual ya fue desarrollado en la primera parte del año e implementado mediante las denominadas CALL GATES.

2 Estándar POSIX

2.1 Actualidad

POSIX está siendo desarrollado en el marco de la *Computer Society* de IEEE, con la referencia IEEE 1003, y también a nivel de estándar internacional con la referencia ISO/IEC 9945. Su objetivo es generalizar las interfaces de los sistemas operativos para que una misma aplicación pueda ejecutarse en distintas plataformas.

Debe quedar claro que el estándar solo define la interfaz, en ningún caso la implementación, y que está pensado para ser utilizado tanto por desarrolladores de aplicaciones como por programadores que implementen sistemas operativos. POSIX es por tanto una colección de documentos que definen claramente una interfaz estándar entre el sistema operativo y sus aplicaciones a nivel de código fuente. En otras palabras, POSIX define los servicios que debe proveer un sistema, especificando de forma exacta los prototipos de las funciones de biblioteca y llamadas al sistema, los tipos de las variables utilizadas, las cabeceras, los códigos de retorno de las funciones, su comportamiento concurrente, etc. Además también especifica otros aspectos, como por ejemplo la estructura general del sistema de archivos, consideraciones sobre el set de caracteres usado, sobre las expresiones regulares, las variables del entorno, la interacción con el terminal o las secuencias de escape, especifica las interfaces de usuario y software al sistema operativo en 15 documentos diferentes. La línea de comandos estándar y las interfaces de *scripting* se basaron en Korn Shell. Otros programas a nivel de usuario (*user-level*), servicios y utilidades incluyen AWK, echo, ed y cientos de otras. Los servicios requeridos a nivel de programa incluyen la definición de estándares básicos de I/O, (archivo, terminal, y servicios de red). También especifican una API para las bibliotecas de threading, que es muy utilizada en una gran variedad de sistemas operativos. En realidad estos estándares surgieron de un proyecto de normalización de las API y describen un conjunto de interfaces de aplicación adaptables a una gran variedad de implementaciones de sistemas operativos.

Algunos de los estándares POSIX ya han sido aprobados, mientras que otros están siendo desarrollados todavía. Los podemos agrupar en tres categorías:

1. *Estándares base*: Definen la sintaxis y semántica de interfaces de servicios relacionados con diversos aspectos del sistema operativo. El estándar no especifica cómo se implementan estos servicios, sino únicamente su semántica. La mayoría de los estándares base están especificados para el lenguaje de programación C. La siguiente tabla lista algunos de los estándares básicos POSIX más importantes para las aplicaciones de tiempo real.

Tabla 1: Estándares POSIX para Tiempo Real

1003.1 Servicios básicos del Sistema Operativo
1003.1a Extensiones a los servicios básicos
1003.1b Extensiones de tiempo real
1003.1c Extensiones de threads
1003.1d Extensiones adicionales de tiempo real
1003.1e Seguridad
1003.1f Sistema de ficheros en red (NFS)
1003.1g Comunicaciones por red
1003.1h Tolerancia a fallos
1003.1j Extensiones de tiempo real avanzadas
1003.1m Puntos de chequeo y reintento
1003.2 Shell y utilidades
1003.2b Utilidades adicionales
1003.2d Ejecución por lotes (batch)
1003.3 Métodos para probar la conformidad con POSIX
1003.21 Comunicaciones para sistemas distribuidos de tiempo real

2. *Interfaces en diferentes lenguajes de programación ("bindings")*: estos estándares proporcionan interfaces a los mismos servicios definidos en los estándares base, pero usando otros lenguajes de programación. Los lenguajes que se han usado hasta el momento son el Ada y el Fortran. En la siguiente tabla se muestran algunos de ellos:

Tabla 2: Estándares POSIX para Lenguajes

1003.5 *Binding* de Ada para 1003.1
1003.5b *Binding* de Ada para 1003.1b y 1003.1c
1003.5c *Binding* de Ada para 1003.1g
1003.5f *Binding* de Ada para 1003.21
1003.9 *Binding* de Fortran 77 para 1003.1

3. *Entornos de sistemas Abiertos*: Estos estándares incluyen una guía al entorno POSIX y perfiles de aplicación. Los perfiles son un subconjunto de los servicios POSIX que se requieren para un determinado ámbito de aplicación. Representan un mecanismo para definir de forma estándar un conjunto bien definido de implementaciones de sistemas operativos, adecuadas para áreas de aplicación específica.

2.2 Características más relevantes

- Algunos tipos de datos utilizados por las funciones no se definen como parte del estándar, pero se define como parte de la implementación. Estos tipos se encuentran definidos en el archivo de cabecera <sys/types.h>. Estos tipos acaban con el sufijo "_t". Por ejemplo: uid_t, es un tipo que se emplea para almacenar un identificador de usuario (UID).

- Los nombres de las llamadas al sistema en POSIX son en general cortos y con todas sus letras en minúsculas. Ejemplo: fork, close, read.

- Las funciones, normalmente devuelven cero si se ejecutaron con éxito, o -1 en caso de error. Cuando una función devuelve -1, el código de error se almacena en una variable global denominada *errno*. Este código de error es un valor entero. La variable *errno* está definida en el archivo de cabecera <errno.h>.

- La mayoría de los recursos gestionados por el sistema operativo se referencian mediante descriptores. Un descriptor es un número entero mayor o igual que cero.

Desde que la IEEE empezó a cobrar altos precios por la documentación de POSIX y se ha negado a publicar los estándares, ha aumentado el uso del modelo Single Unix Specification. Este modelo es abierto, acepta entradas de todo el mundo y está libremente disponible en Internet. Fue creado por *The Open Group*.

2.3 Partes

Haciendo una comparación con la tabla 1 (Estándares para Tiempo Real), observamos que para Single Unix Specification se mantiene la misma estructura, como se puede ver el POSIX 1b (extensiones para aplicaciones de tiempo real).

POSIX.1, Core Services (Servicio Central), implementa las llamadas del ANSI C estándar. Incluye:

- Creación y control de procesos.
- Señales.
- Excepciones de punto flotante.
- Excepciones por violación de segmento.

- Excepciones por instrucción ilegal.
- Errores del bus.
- Temporizadores.
- Operaciones de archivos y directorios (sobre cualquier file system montado).
- Tuberías (*Pipes*).
- Biblioteca C (Standard C).
- Instrucciones de entrada/salida y de control de dispositivo (ioctl).

POSIX.1b, extensiones para tiempo real:

- Planificación (*scheduling*) con prioridad.
- Señales de tiempo real.
- Temporizadores.
- Semáforos.
- Intercambio de mensajes (*message passing*).
- Memoria compartida.
- Entrada/salida sincrónica y asincrónica.
- Bloqueos de memoria.

POSIX.1c, extensiones para hilos (threads):

- Creación, control y limpieza de hilos.
- Planificación (*scheduling*).
- Sincronización.
- Manejo de señales.

POSIX.2, Shell y Utilidades (IEEE Std 1003.2-1992)

- Intérprete de Comandos
- Programas de Utilidad

Los siguientes Sistemas Operativos son 100% compatibles con uno o varios estándares POSIX:

- GNU/Linux
- A/UX
- AIX
- BSD/OS
- HP-UX
- INTEGRITY
- IRIX
- LynxOS
- Mac OS X v10.5 en Procesadores Intel.
- MINIX

- MPE/iX
- QNX (IEEE Std. 1003.13-2003 PSE52;
- RTEMS (POSIX 1003.1-2003 Profile 52)
- Solaris
- OpenSolaris
- UnixWare
- velOSity
- VxWorks (IEEE Std. 1003.13-2003 PSE52)
- Windows NT (y posteriores)

2.2 API / SYSCALL

Para que nuestra aplicación de espacio de usuario pueda emitir una syscall, es necesario poder realizar la llamada mediante un API (Interfaz de Programación de Aplicaciones) que alguien nos proporcione. Esta API es proporcionada por la librería estándar de C de nuestro sistema, la libc o glibc en el caso de Linux. Una de las APIs más comunes del mundo Unix es la basada en el estándar POSIX, como ya mencionamos en el apartado anterior; además Linux es considerado compatible POSIX, e intenta ajustarse al estándar SUSv3 (Single UNIX Specification Version 3 o UNIX 03) cuando es aplicable.

Hay que decir que el estándar POSIX se refiere a las APIs, no a las syscalls, es decir, define un comportamiento pero no cómo lo debe hacer. Un sistema puede ser certificado como compatible POSIX pues ofrece un apropiado conjunto de APIs a los programas, sin importar cómo las funciones se han implementado. Desde el punto de vista del programador de aplicaciones, la distinción entre una API y una syscall es irrelevante, lo único que importa es el nombre de la función, los tipos de los parámetros y el valor devuelto.

Por otro lado, también hay que destacar que el estándar POSIX recomienda que exista una correlación uno a uno entre la función de API y la llamada al sistema. Es decir, el API de una syscall debe ser igual al formato usado en la syscall del kernel. Sin embargo no se garantiza que detrás de un API de una syscall se invoque la pertinente syscall, sino que puede que la funcionalidad de la misma se esté proporcionando desde la propia librería de C, sin invocar al kernel o que el resultado sea la combinación de varias syscalls del kernel. Lo ideal, en aras de la eficiencia y velocidad de ejecución es que la syscall exista realmente en el kernel.

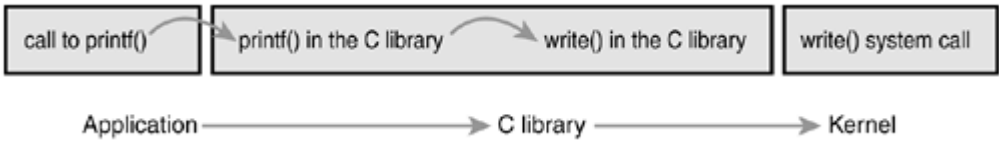

Ilustración 5: API

En UNIX es común encontrar implementada API en la biblioteca C incluyendo la System Call interface. Dicho de otra forma, en Unix las llamadas a funciones de la POSIX API tienen una fuerte relación con las system calls. En cambio la interfase de system calls en Linux es provista en parte por la biblioteca de C. Sin embargo desde el punto de vista del programador la API es lo esencial, mientras que para el Kernel, las system calls son todo.

3 Implementación estándar POSIX en SO SODIUM

Dada la introducción teórica, detallaremos a continuación la sección práctica de este trabajo de investigación.

Como hicimos referencia en la introducción, se designó al comando PS para la implementación del estándar POSIX.

Como primer paso, dejamos de lado el concepto de *call gates* estudiado y desarrollado en el primer trabajo de investigación. Acto seguido determinamos las funciones necesarias a utilizar por el comando elegido, que son *open, read, write y close* respectivamente. Dado que las mismas son utilizadas por el SO Linux nos posibilitó copiar y respetar el estándar de llamadas de las mismas, como ser la cantidad de parámetros, los tipos de datos y qué devuelve cada una luego de ser llamada.

Estas funciones descriptas fueron codificadas en nuestra biblioteca *libsodium*, donde los parámetros son recibidos y preparados para realizar a posteriori la correspondiente llamada al sistema o *syscall*.

Por último y volviendo al comando PS, se desarrolló el mismo llamando a estas funciones básicas mencionadas. A continuación se observa el detalle de la codificación realizada para este trabajo.

3.1 Comando PS

FILE: PS.C
PATH: /usr/bin/ps.c

```
int main(int argc, char* argv[])

   /* Sección de ayuda */
   …
   /* Chequeo de parámetros de entrada */
   …

   /* A continuación efectuamos el llamado de la
   función OPEN. La misma abrirá un archivo de
   configuración con los datos referentes al
   comando PS. Han sido respetados la cantidad y
   los tipos de datos de la función según lo
   investigado en el SO Linux. Dicha función
   devuelve un entero que representa al File
   descriptor creado.

   iFd = open("Nombre de archivo","Flag","Modo");

   /* Luego de abrir el archivo, procedemos a
   efectuar la lectura del mismo utilizando el
   comando READ. Al igual que el comando OPEN, se
   respetaron la cantidad y los tipos de datos.
   Primero llamaremos a este comando para
   determinar la cantidad de procesos a mostrar
   (almacenado en la variable iCantproc), y así
   conocer el espacio a reservar en nuestra
   estructura de datos stuPCB.

   iCantproc = read(iFd,0,0);
   if(iCantproc <0)

      return -1;

   /* Realizamos el malloc correspondiente según el
   espacio requerido para la estructura stuPCB */

   if((stuPCBP=malloc(sizeof(stuPCB)*iCantproc)
   ==NULL))

      return NULL;
```

```
/* Ahora llamaremos nuevamente al comando READ
para leer propiamente los datos a almacenar en
nuestra estructura temporal y mostrar por
pantalla para el usuario. */

if(read(iFd,stuPCBP,0)<0)

   return -1;

/* Una vez leído el archivo, procedemos a llamar
a la función CLOSE para cerrarlo, pasando por
parámetro el File Descriptor correspondiente */

close(iFd);

/* Imprimir por pantalla los datos almacenados
en la estructura temporal */

…

/* Se libera espacio reservado para la
estructura stuPCB, y finalizamos main del
comando PS */

free(stuPCBP);
return 0;
```

3.2 Biblioteca LIBSODIUM (API)

FILE: libsodium.c
PATH: usr/lib/libsodium

Se detalla a continuación la modificación de la biblioteca de funciones de usuario, con el agregado de las funciones *open, read, write* y *close.*

Función OPEN:

```
/*
@brief Abre un archivo
```

```
 @param stNombreArchivo Nombre del archivo que se
quiere abrir
@param iFlag Tipo de apertura del archivo
@param iMode Modo de apertura del archivo
@returns 0 OK.
*/

int open( const char *stNombreArchivo, int iFlag,
int iMode ){

long liRetorno;
while(liRetorno == 1)
{

   /* Llamada al sistema para la apertura del
   archivo por parte del SO */

   SYS_CALL_3(   liRetorno,   errno,   __NR_open,
   stNombreArchivo, iFlag, iMode );

if(liRetorno == 1)
   sched_yield();
}

return liRetorno;
}
```

Función READ:

```
/*
  @brief Lectura de un archivo
  @param iFd File descriptor del archivo
  @param vpBuffer Estructura de datos
   @param uiTamanio Tamaño del buffer
   @returns liRetorno
*/

int read(int iFd, void *vpBuffer, size_t uiTamanio){

long liRetorno;

/* Llamada al sistema para la lectura del archivo por
parte del SO */

SYS_CALL_3(   liRetorno,   errno,   __NR_read,   iFd,
vpBuffer, uiTamanio );
```

```
return liRetorno;
}
```

Función CLOSE:

```
/*
  @brief Cierra un archivo
  @param iFiledescriptor Descriptor del archivo que
      se dquiere cerrar
  @returns 0 OK.
*/

int close( int iFiledescriptor ){

long liRetorno;
while(liRetorno == 1)
{

     /* Llamada al sistema para liberar memoria del
    archivo antes abierto por parte del SO */

    SYS_CALL_3(   liRetorno,   errno,   __NR_close,
    iFiledescriptor,0 ,0 );

    if(liRetorno == 1)

       sched_yield();
}

  return liRetorno;

}
```

Función WRITE:

```
/*
  @brief Escribe un archivo
  @param iFd File descriptor del archivo
  @param covpBuffer Estructura de datos
   @param uiTamanio Tamaño del buffer
  @returns
*/
```

```
int write( int iFd, const void *covpBuffer, size_t
uiTamanio)
{

long liRetorno;


  /* Llamada al sistema para leer un archivo por
  parte del SO */

  SYS_CALL_3( liRetorno, errno, __NR_write, iFd,
  covpBuffer, uiTamanio);

   return liRetorno;
}
```

4 Inconvenientes encontrados

Para la resolución de este trabajo, tanto en la etapa de análisis como en la de desarrollo nos encontramos con distintos problemas que debimos resolver.

El primer obstáculo surgió al momento de comprender el alcance del trabajo. Si bien pudimos entender la funcionalidad de las APIs y del estándar POSIX con relativa facilidad, nos resultó complicado llegar a entender claramente qué funcionalidad programar en SODIUM. Debíamos tener en cuenta que muchas de las llamadas que programaríamos no estarían implementadas siquiera al terminar el trabajo (porque estaban en manos de otros grupos y se incorporarían al final), y por momentos nos fue complicado definir el alcance del código que debíamos desarrollar. Por ejemplo, comenzamos a programar la funcionalidad del syscall *read*, pero interrumpimos luego de que los profesores nos aclararan el tema.

Otro inconveniente que tuvimos fue la comprensión en detalle del código de las llamadas en SODIUM, al tener que seguir el camino de ejecución a través de varios archivos fuente.

Una vez que tuvimos en claro qué camino seguir y cómo debíamos resolverlo, el proceso de desarrollo fue mucho más dinámico y entre todos pudimos lograr el resultado que necesitábamos.

5 Conclusión y pasos a seguir

Nuestra propuesta fue adaptar los programas de usuario creados a partir del trabajo práctico nro. 1, para que dependan únicamente de bibliotecas provistas por SODIUM, cuya interfase sea consecuente y se mantenga dentro del estándar POSIX. Debíamos entonces corregir o crear funciones e interfases, y adaptarlas para los casos en que fuese necesario. Tuvimos que ser cautelosos a la hora de modificar o quitar alguna *Syscall* que ya estuviera desarrollada, además de respetar la numeración tal como indica POSIX.

En el anterior trabajo práctico de investigación nos basamos en pasar los comandos en archivos de la misma familia o funcionalidad. En esta ocasión, tomamos directamente como referencia el comando PS. Comenzamos así a pensar la forma en que debíamos dejar la funcionalidad de dicho comando de acuerdo al estándar POSIX, lo cual esto fue complicado de analizar y resolver.

Si bien teníamos bien definida la funcionalidad del comando, el punto fue que debíamos incorporar las llamadas a otras funciones de la manera que el estándar POSIX indica, tal cual lo hace Linux (por ejemplo). Algunas de estas funciones son las mencionadas en el punto 3 de este documento: *open, read, write*, etc. En muchos de los casos, las funciones no estaban implementadas, pues nosotros debíamos dejar cada función armada correctamente con sus parámetros de entrada y el valor de retorno tal como lo indica el estándar, pero con la salvedad que la resolución la podíamos definir con valores prefijados.

Luego de analizar el código existente y de consultar tanto la bibliografía como también a la cátedra, logramos dejar implementado el comando PS.

Dada esta investigación y su correspondiente parte práctica, dejamos un precedente para que en lo sucesivo pueda continuarse la estandarización de otros comandos, como por ejemplo el comando *read* del file system. Sólo se debe pensar en la forma de codificarlo verificando el estándar, sin preocuparse todavía por los parámetros que recibe o el valor que devuelve.

6 Referencias

1. http://www.uv.es/gomis/Apuntes_SITR/POSIX-RT.pdf
2. http://www.ctr.unican.es/asignaturas/MC_ProCon/Doc/ProCon_II_01-POSIX_intro.pdf
3. http://www.dachary.edu.ar/materias/SOII/docs/Programacion_en_Linux.pdf
4. http://upcommons.upc.edu/revistes/bitstream/2099/7851/6/p164.pdf
5. www.uv.es/gomis/Apuntes_SITR/Trasparencias/POSIX_rt.pdf
6. UNIX Programación Avanzada, Francisco M. Marquez, Ed Alfaomega Ra Ma tercera edición, octubre 2004, Pág. 243,298.
7. Apunte de Programación en Linux, Cátedra de Sistemas de Computación II, Universidad Nacional de la Matanza, páginas 88, 90.
8. Sistemas Operativos Modernos, Andrew S. Tanenbaum, Ed. Prentice Hall primera edición, Copyright 1993, Pág. 1 – 21.
9. http://syscalls.kernelgrok.com/