Universidad Nacional de La Matanza
Florencio Varela 1903 - San Justo - Buenos Aires - Argentina

Ingeniería en Informática

Sistemas Operativos

Trabajo Práctico N° 1

SHELL NIVEL 3

	Nicanor Casas
Equipo	Graciela De Luca
	Waldo Valiente
Docente	Gerardo Puyo
	Martín Cortina

GRUPO: 1

Alumnos		
Apellido	**Nombre**	**DNI**
Mosso	Juan Pablo	33.040.517
Martin	Sergio Miguel	32.691.890
Brieva	Leonardo	32.090.398
Vicente	Fernando	30.047.687

Acreditación:

Instancia	Fecha	Calificación
ENTREGA FINAL	09/09/2009	

-- 2009 --

Universidad Nacional de La Matanza
Florencio Varela 1903 - San Justo - Buenos Aires - Argentina

Ingeniería en Informática

Sistemas Operativos

Trabajo Práctico N° 1

ACTIVACIÓN DE PROCESOS A NIVEL DE USUARIO EN SODIUM

	Nicanor Casas
Equipo	Graciela De Luca
	Waldo Valiente
Docente	Gerardo Puyo
	Martín Cortina

GRUPO: 1

Alumnos		
Apellido	**Nombre**	**DNI**
Mosso	Juan Pablo	33.040.517
Martin	Sergio Miguel	32.691.890
Brieva	Leonardo	32.090.398
Vicente	Fernando	30.047.687

Acreditación:

Instancia	Fecha	Calificación
ENTREGA FINAL	09/09/2009	

-- 2009 --

Observaciones:

ÍNDICE

INTRODUCCIÓN

Objetivos y Alcance del Trabajo Práctico

El objetivo general del trabajo práctico es trasladar la ejecución del shell del nivel 0 al 3. Como objetivos implícitos se encuentran:

- Subdividir la programación del shell con la del kernel. Hoy rutinas del shell acceden a estructuras de datos propias del kernel y hasta las podrían modificar.
- Que cualquier proceso pueda ejecutar en nivel 3 (usuario) y utilizar rutinas del kernel (nivel cero). Para esta entrega, con que sea un shell alcanza.
- Dar seguridad y estabilidad al kernel de SODIUM.
- Aprender sobre el uso de CALL GATE y INTERRUPT GATE. Con sus saltos en distintos niveles.

Asimismo la resolución del mismo está dividido en dos partes:

Primera parte

Objetivo:

El objetivo de la primera parte de entrega del presente trabajo práctico es lograr ejecutar un proceso en nivel 3 y que desde este se vayan comunicando a los distintos niveles, según un parámetro pasado desde el shell (que ejecuta en nivel 0).

Alcance:

- El proceso siempre ejecutara en nivel 3.
- Ejecutar otro proceso o función que se encuentre en un nivel inferior.
- Proporcionar los mecanismos de prueba de cada nivel y la forma de probarlo.

Segunda parte

Objetivo:

Reutilizar el proceso desarrollado en la primer entrega, para migrar el shell a nivel 3.

Alcance:

- Realizar un interprete de comandos que ejecute en nivel 3 y utilice rutinas del kernel en nivel 0.

Sobre el formato de entrega

Esta entrega de documentación sobre el Trabajo Practico realizado está dividido en dos partes de la misma manera que la consigna por razones de simplicidad y cronología.

La primer parte contiene modificaciones e información importante que luego son cambiadas o reemplazadas en la segunda parte pero que son de importante mención para el aprendizaje sobre el funcionamiento del Sistema Operativo.

Se hará un desarrollo independiente para cada parte con su consigna, introducción, detalles y conclusión. Al final de las mismas se adjuntan también secciones de código importantes citados y un historial de modificaciones.

Al final del documento se hará una conclusión final sobre el desarrollo completo del Trabajo Práctico y se citará toda la bibliografía consultada.

PRIMERA PARTE

Base Teórica

Nos basamos en todo momento para investigar los manuales de Intel y los apuntes provistos por la materia, además de investigación por internet (links en bibliografía).

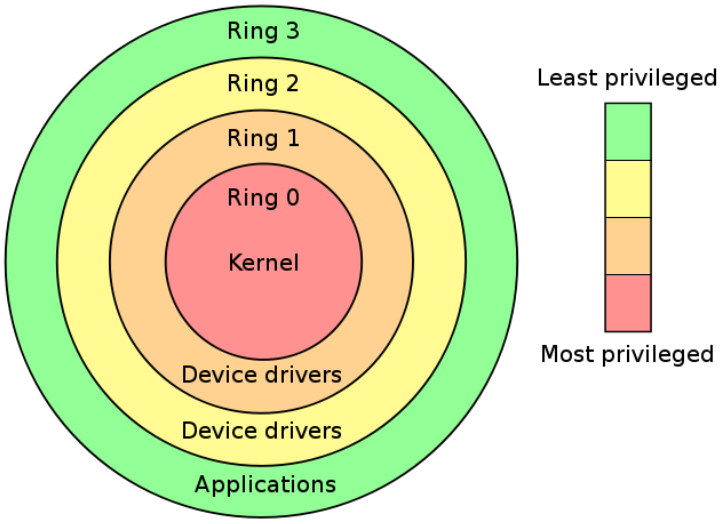
Análisis del Caso

El problema de la seguridad es importante en cualquier Sistema Operativo ya sea comercial o académico. Un Sistema Operativo seguro debe restringir las operaciones que pueden hacer los usuarios para evitar que éste modifique partes del sistema importantes de manera involuntaria provocando errores indeseados o, peor aún, lo haga voluntariamente.

Es necesario primero distinguir qué operaciones están permitidas a un usuario normal y cuales están reservadas al Sistema Operativo para operaciones de administración.

Las operaciones de administración pueden incluyen manejo de memoria, planificación de memoria, administración de usuarios y servicios de hardware. En un funcionamiento normal, estas operaciones son "transparentes" al usuario y no deberían ser accesibles a él (ni por error de programación ni por deliberada intención de parte del mismo).

Por lo tanto hay que definir un límite en cuales operaciones se permiten al kernel y cuales al usuario. Intel propone una división de seguridad esquematizada como Anillos (rings) de seguridad donde el menor nivel es el mas privilegiado y el mas alto es el menos privilegiado.



Para cada nivel se definen operaciones permitidas y operaciones prohibidas. Este chequeo de seguridad es efectuado por Hardware en la ejecución de cada instrucción.
Por lo tanto, se está proveyendo al Sistema Operativo una base hardware para el control de la seguridad.

El Sistema Operativo (kernel corriendo a nivel 0) tiene permisos para acceder a todas las regiones de memoria y ejecutar todas las instrucciones del set x86, y puede definir mediante configurar en la tabla GDT el privilegio de cada Code Segment de cada proceso.

Pero, ¿Qué pasa si un proceso usuario (nivel 3) requiere ejecutar una instrucción que solo puede ser utilizada a nivel 0?

Se hace necesario un mecanismo por el cual, estos procesos puedan acceder de manera controlado a operaciones privilegiadas. Esto se realiza mediante Servicios proveídos por el Sistema Operativo, por los cuales, de manera controlada y a pedido de un proceso usuario se efectúa una operación restringida (Ej: Acceso a disco rígido) utilizando un mecanismo estándar para todos los procesos y que retorne al proceso a nivel 3 luego de haber efectuado esa operación.

Opciones disponibles

- Callgates a servicios de kernel.

  Fue la primera implementación que intentamos para la resolución del problema de la falta de privilegios a nivel usuario. A pesar de ser funcional tiene algunas desventajas serias:
  No es Posix. Al no ser una forma de llamado Standard, por mas que sea funcional, no podrían compilarse programas multiplataforma para correr en sodium.
  No hay infraestructura previa. El sodium está basado en cierta medida en los standares posix, reflejado especialmente en la API libsodium.c que si bien funcionaba a nivel 0 todavía, constituye una base ya formada para la utilización de syscalls.
  Es poco claro y complicado para programar. La forma de acceder a una callgate es haciendo un salto a la callgate (con cualquier offset). Esto puede ser complicado de recordar para cualquier programador.

- Syscalls POSIX

  Cambiamos a esta alternativa en la mitad de desarrollo al darnos cuenta que es una opción mucho mas viable y facil, por oposición a las desventajas propias de los callgates para servicios a procesos usuario:
  Son Posix. Pueden compilarse programas multiplataforma para correr en sodium.
  Hay Infraestructura previa. Pudimos reutilizar casi la totalidad del código ya programado y testeado en libsodium.c
  Es simple. Al manejarse con una interrupción (0x80) y con un estandar con el pasaje de parámetros, es mucho mas facil de programar.

Elección y Justificación de la elección

Nuestra elección respecto a la resolución de esta primera parte del trabajo práctico fue la adaptación de las funciones del sodium de creación y administración de procesos para que puedan correrse en los diferentes niveles descriptos por la arquitectura Intel x86 y utilizando el estandar de Syscalls POSIX para los servicios a los procesos usuario.

Introducción Práctica

Partimos de un sodium que corre únicamente a nivel 0, tanto el kernel, como los procesos nulo, reloj y shell.
Como primera instancia necesitamos adaptarlo para correr cualquier aplicación simple sin necesidad de ejecutar llamadas al sistema (por ejemplo, una cuya única instrucción sea while(1)).

Para esto tuvimos que adaptar todas las funciones correspondientes a la creación de procesos nuevos incluídos en gdt.c para que acepten como parámetro el nivel de privilegio del programa a instanciar.

Una vez logrado esto pasamos a crear la infraestructura para los syscalls.

Resumen de modificaciones

- Makefiles: No se ha modificado directamente ningun makefile, pero si indirectamente. Habiendo creado una herramienta (herramientas/crear_proc_usuario.sh), que modifica varios Makefiles con un nombre de proceso pasado por usuario de manera de que automáticamente el sodium cuente con este la próxima vez que inicie, permitiendo trabajar sobre nuevos procesos a medida que avanzamos en vez de solo trabajar sobre el init.

- Programas: Dicho lo anterior, no hubo modificaciones sobre el INIT, solo sobre los programas que nosotros creamos:
  - usuario.c: Este es nuestro proceso usuario a nivel que vamos a utilizar para las pruebas, contiene una instrucción privilegiada de prueba para saber que una vez pasado el nivel de privilegio, funcione o no dependiendo de este último.

- Funciones: Las funciones afectadas estan casi todas dentro de gdt.c

- iFnCrearProceso: Fue cambiada para ejecutar cualquier proceso que necesitemos, no solo init. Aparte se le pasa por parámetro el nivel de privilegio con el que queremos que ejecute.

- iFnInstanciarTSS: Aquí agregamos descriptores en la gdt que seran usados por funciones correspondientes llamadas por main (en -main se especifica cuales)

- iFnCrearTSS: Esta también fue cambiada con el mismo propósito que la anterior.

- Funciones del shell: Aquí modificamos todo lo concerniente a ejecutar un proceso al estilo DOS. Si no se encuentra ningun nombre de comando asociado a la entrada, se pasa a buscar un nombre de proceso asociado al comando. Si se encuentra se ejecuta, sino se reporta error.

- Main: Se han agregado descriptores para los stacks de los rings 0,1 y 2. Aparte de descriptores de callgate y un selector de código para agregar una syscall (provisorio). Agregados tambien descriptores de código para los codigos de prueba test.

- Variables: Las variables clave que han sido editadas para hacer funcionar un proceso a nivel usuario son:
    - Selectores CS, DS y SS en iFnCrearTSS: Estas variables fueron clave para la ejecución del proceso a nivel 3. No bastaba con asignarle a los descriptores el DPL correspondiente, sino que los selectores en la TSS a esos descriptores debían también llevar la flag del RPL equivalente, una vez seteado esto (en orden con las reglas de seguridad especificadas por Intel) pudimos efectuar el task switch.
    - Selector ss0 en iFnCrearTSS: Aquí especificamos un segmento de datos para el stack a nivel 0, escencial para hacer funcionar el task switch de 0 a 3 y viceversa.

Documentación de Funciones y programas

En esta entrega, puede probarse que el proceso está ejecutando diferentes niveles de código al examinar su salida por pantalla, comparandola con los anexos de los diferentes códigos de usuario.c test2.c test1.c test0.c, sc.c y libsodium.c

usuario.c: Es el programa de prueba que recibe un parámetro int de la linea de comando $usuario [Parámetro] cuya funcionalidad es exactamente la pedida por el enunciado del TP.

test2.c, test1.c, test0.c: Contienen los códigos al nivel que indica su numero para ser ejecutado por el proceso generado de usuario, a travez de callgates en el orden estipulado por el parámetro, escriben un mensaje por pantalla y regresan.

sc.c: Es el código que utilizamos como puente hacia las instrucciones propias para el syscall ubicadas en el kernel. Hay necesidad de hacerlo por separado en su propio segmento para que jamas cambie el selector ni el offset. Es referenciado por la callgate y su primera instrucción es un int $0x80 que puede ejecutar debido a que su CS tiene un DPL = 0. Y Ret Far que devuelve el control al programa que utilizó la callgate.

Libsodium.c: Contiene las funciones del lado usuario para la ejecución de syscalls. Adaptamos la interrupción int $0x80 para que pueda ser ejecutada a nivel usuario.

Descripción de nuevas funcionalidades

- ✓ Ejecución de un binario en shell, si no encuentra un comando, busca el binario.
- ✓ Posibilidad de enviar un parámetro int al main del programa invocado (al DS+BP+Offset
- ✓ Posibilidad de ejecutar un proceso en cualquier nivel de privilegio (0,1,2,3)
- ✓ Posibilidad de un proceso menos privilegiado de ejecutar código privilegiado mediante la implementacion de callgates
- ✓ Syscalls adaptados para ejecución desde cualquier nivel de privilegio
- ✓ Los programas ahora manejan 2 segmentos de memoria, uno para el código y otro para datos y stack, contrariamente a lo que se usaba (1 solo segmento para todo)

Estado del Trabajo Practico al finalizar la Parte I

Posibilidad de instanciar un nuevo proceso simple usuario a nivel 3 (mediante la invocación de la funcion iFnCrearProceso, que no es POSIX para la creación de nuevos procesos, lo que es solucionado en la Parte II)
Éste proceso puede realizar instrucciones que no requieran un llamado al sistema, además de poder imprimir mensajes en pantalla y salir sin generar errores.

Terminado el proceso de prueba de navegación de niveles mediante call gates (a solo modo demostrativo de que funciona correctamente, pero a ninguna funcionalidad útil)

Todavía se utiliza el shell a nivel 0, por lo que el siguiente paso para la Parte II es la generación de todos los syscalls necesarios y la transición a nivel usuario del shell.

Conclusión Parte I

Adaptar un Sistema Operativo que ha llegado a un punto de desarrollo importante que ha contemplado parcialmente la ejecución de un proceso a nivel usuario complicó bastante la transición. Por una parte, nos apoyamos en gran medida en los manuales de intel que explicaban claramente los mecanimos para efectuar la transición, pero por el otro fue un gran trabajo de investigación sobre como funciona el Sodium, puesto que de hecho tenía gran cantidad de codificación apuntada solamente a la ejecución de procesos a nivel 0 solamente, lo que requirió un trabajo de reingeniería y adaptación que fue la mayor cantidad de esfuerzo requerido.

ANEXO I: HISTORIAL DE MODIFICACIONES

Checkpoint 1: Ejecutar Archivo

```
- Herramienta para agregar un proceso nuevo en
/herramientas/crear_proc_usuario.sh
Modo de uso: ./crear_proc_usuario.sh [NOMBRE]

- Shell.c: Agregada una funcion para extraer el comando y
otra para llevarlo a mayusculas y agregarle un .BIN.
  Editado el caso para cuando no encuenta ningun comando
para que pase a la busqueda de ejecutables. Llama a la
funcion de crear Proceso que antes se referia a crear Init.

- gdt.c: Acá se cambiaron todas las funciones de creacion de
INIT a creacion de Proceso  en general y todas reciben una
cadena que varia segun el archivo que queremos ejecutar.
Es importante que empieces desde este punto fernando si vas
a editar este mismo archivo.
```

Checkpoint 2: Proceso Usuario

```
-(kernel/gdt.c) -Agrego nueva funcion
"iFnCrearProcesoUsuario"
                -En las opciones de la funcion
"uiFnAgregarDescriptorGDT" agrego lamascara "D_DPL3"
                -En los selctores pasados como param en la
funcion "iFnCrearTSS", agrego la mascara               "0x3"

-(include/kernel/gdt.c) Agrego definicion de funcion
"iFnCrearProcesoUsuario"
```

Checkpoint 3: Proceso Usuario a nivel 3 Parte 1

```
-(kernel/gdt.c) - iFnCrearTSS:
    * Agregado el flag de DPL a los selectores de codigo,
datos y segmento (uiCS, uiDS, uiSS respectivamente) mediante
| 0x3 (2 primeros bits)
    * Agregado dirección de stack a nivel 0 (Data segment
del kernel ubicado en el tercer selector del GDT = 0x10)
    * Cambiado eFlags a 0x3202L;
    * Definidos espacios de esp (opcional)

Falta:
    - Pasar nivel de ejecución por parámetro.
    - Desarrollar primer callgate para el syscall de exit
(en este momento las aplicaciones usuarios no pueden
terminar normalmente sin un syscall)
```

Checkpoint 4: Proceso Usuario a nivel 3 Parte 2

```
-(kernel/gdt.c)
    - Se pasa como parametros en todos niveles de creacion
de proceso el parámetro iPrivilegio y se parametrizan a
partir de esa variable los flags y seteos para   los
diferentes niveles de permisos.
    - Nueva funcion iFnCrearStack(int iPrivilegio); permite
crear un stack de privilegio pasado por parámetro,
unicamente a las posiciones 3, 4 y 5 respectivamente
      del gdt.
    - Los ss0, ss1, ss2 ahora apuntan a nuevos stacks
creados con la anterior funcion. Incluso el de nivel ss0
tiene nuevo stack para no "molestar" al kernel.
    - La funcion inicializarTSS ahora reserva 3 posiciones
para los stacks mencionados, después de la nula y las 2 del
kernel requeridas por intel.

-(kernel/main.c)
    - La creacion de los 3 nuevos segmentos de stack se
hacen acá después de crear la tarea nula como otro proceso
de inicio.

-(kernel/shell.c)
    - Ahora se puede ejecutar un proceso de 2 maneras:
      * Sin parametros:  $[NombreDeProceso]
        Ejecuta el proceso en nivel 3 por defecto (salvo que
se haya usado la siguiente nomenclatura antes, caso en el
que queda el nuevo nivel)
      * Con parametro:   $[NombreDeProceso]
[NivelDePrivilegio]
        Ejecuta el proceso en el nivel pasado por parámetro
(un numero del 0 al 3). Cualquier otra entrada ejecuta al
proceso en modo 3.
```

Checkpoint 5: Proceso Usuario a nivel 3 Parte 2

-(kernel/gdt.c)
- Agregados 2 descriptores en la gdt, uno para Callgate y otra para SysCall
- Creada la funcion iFnCrearCallGate que traduce una entrada GDT del formato CS-DS-SS a formato Call Gate (una ruptura de huevos). Y la instancia.
En este momento está apuntando hacia la entrada 7 de la GDT, osea al SYSCALL con offset 0.
- Creada la funcion iFNCrearSysCall que carga cualquier binario a memoria. SOLO EL CODIGO, y a nivel 0, de manera que cualquier programa que le pasemos va
a estar como código y NO como proceso en memoria (sin area de datos ni stack), y a nivel kernel, de manera de que sea como una libreria de funciones
que puede ser accedida solo atravez de la callgate.

-(usr/nuevo.c)
- Agregada la linea de comando asm volatile("lcall $(0x30), $(0x30)"); que llama al segmento 0x30 (osea la posicion 6 de la GDT). No se cual es el offset y cual el segmento, en todo caso no importa porque no se toma en cuenta el offset. Llama con RPL = 0 para evitar problemas.

-(usr/sc.c)
- Este es el codigo del syscall. Si, esta en la carpeta usr, no importa por ahora porque se carga al CS en nivel 0, aparte es solo para pruebas, lo importante es usar el syscall.c que ya esta definido, ese es la posta.
* Para probarlo, fijense el codigo de sc.c y el de nuevo.c. Vayan al qemu y ejecuten nuevo. Van a ver que se queda en un loop infinito pero en realidad es que está ejecutando el while de sc.c. Evidencia de que a pesar de que nuevo está en un cs nivel 3, está ejecutando el de sc.c que está nivel 0.

-(kernel/main.c)
- La creacion del callgate y syscall al comienzo.

Falta:
- Desarrollar primer callgate para el syscall de exit (en este momento las aplicaciones usuarios no pueden terminar normalmente sin un syscall)
- Parametrizar niveles de syscall y callgates
- Return para el syscall

Checkpoint 6: Primer Syscall Funcional

-(kernel/gdt.c)
    - Cambiado el offset de la callgate para que apunte a la int 0x80 del sc.c

-(usr/sc.c)
    - Utiles solo las instrucciones int 0x80 y Ret Far (escenciales para que funcione todo). Podría y debería hacerse en un .asm y quizás anexarlo al kernel...

-(usr/usuario.c)
    - Incluye a libsodium.c y tiene una funcion para imprimir y una para exit. Ambas modificadas en el libsodium (ver siguiente)

-(usr/libsodium.c)
    - Los syscalls en asm tienen ahora una llamada a la callgate que lo único que hace es ejecutar el 0x80, en vez de ejecutarlo nativamente (era asi de facil...)
      y por lo visto por ahora funciona perfectamente

Falta:
    - Empezar a hacer los syscalls para shell bien prolijos.

Checkpoint 7: Primer Syscalls a todo nivel y saltos entre niveles

-(kernel/gdt.c)
    - Parametrizados los valores de las callgates para crear una por nivel
    - Reservados descriptores para 3 programas de prueba

-(usr/usuario.c)
    - Ahora reporta su nivel y llama a una rutina de nivel 2

-(usr/test0 usr/test1 usr/test2.c)
    - Creados para aprobar el TP1 demostrando que es posible pasar entre codigos de diferente niveles de privilegio
      mediante callgates correspondientes a cada segmento.

Falta:
    - Cambiar de utilidad al parametro que se le pasa al proceso nuevo al invocarlo desde shell.
       Fué util lo de poder decidir nivel de ejecución, pero para aprobar el tp hay que usarlo
       para las diferentes demostraciones de cambio de nivel.

Checkpoint 8: Cambio argumento de shell y pasaje de parámetros al main

-(kernel/gdt.c)
    - Se agrega un parametro a CrearProceso que de no ser 0, lo envia como argumento (int main(int)) al main del proceso receptor. Típicamente en DS+BP inicial

-(kernel/shell.c)
    - Ahora el parametro sirve como parametro real, como en DOS y no para determinar el nivel de proceso.
      Aunque se dejó la funcionalidad, por ahora en el shell eso está fijo, habría que verlo a futuro cuando se pase el kernel a nivel 3.

-(usr/usuario.c)
    - Recibe un int de parametro en el main para decidir como se va a ejecutar, para el tp1

Falta:
    - Separar los segmentos de datos y codigo en memoria, y matar al que se le ocurrió ponerlos en el mismo lugar.

Checkpoint 9: Fin TP1 Parte 1

-(usr/usuario.c)
    - Pasado por argumento al main, la opcion marcada en el tp para la ejecucion del proceso (entre 1,2, y 3).
      $usuario [Parametro]

-(kernel/gdt.c)
    - Ahora se crean dos segmentos en memoria en vez de uno. Uno para datos y otro para código. Esto evita que
      el stack o los datos mismos reemplacen el código. Aparte sobra memoria para hacerlo
    - Creada la macro para realizar el pasaje del parametro al main en assembler (Base del segmento de datos +
      Base Pointer inicial + Pequeño Offset ($0x04))

ANEXO II: CODIGO FUENTE DE USUARIO

Usuario.c

```
#include <usr/libsodium.h>

int main(int iArgumento){

  iFnImprimir("\nEjecutando en nivel: ", 3);

  // Ejecuta una diferente cadena de saltos dependiendo del Argumento
  if (iArgumento == 1)
  asm volatile("lcall $(0x50), $(0x00)"); // 3-2-1-0

  // Ejecuta una diferente cadena de saltos dependiendo del Argumento
  if (iArgumento == 2)
  asm volatile("lcall $(0x48), $(0x00)"); // 3-1-0

  // Ejecuta una diferente cadena de saltos dependiendo del Argumento
  if (iArgumento == 3)
  asm volatile("lcall $(0x40), $(0x00)"); // 3-0

  iFnImprimir("\nEjecutando en nivel: ", 3);
  exit(1);

}

int iFnImprimir( const char *cncpBuffer, int iNumero ){
   write( 0, cncpBuffer, 0 );
   systest( iNumero );
   return 0;
}
```

sc.c

```
int main(){
    asm volatile("int $0x80");
    asm volatile(".byte 0xCB"); // Ret Far
}
```

test2.c

```
#include <usr/libsodium.h>

int iFnImprimirNumero( const char *, int  );


int main(){
    asm volatile("sub $0x08, %esp");
    /* Por convención de Intel / C la siguiente función reemplaza
     8 bytes del stack que no le pertenecen así que resto 2 bytes
     al stack pointer */
    iFnImprimirNumero("\nEjecutando en nivel: ",2);
    /* Y ahora se los sum */
    asm volatile("add $0x08, %esp");
    asm volatile("lcall $(0x48), $(0x00)");
    asm volatile(".byte 0xCB"); // Ret Far
}


int iFnImprimirNumero( const char *cncpBuffer, int iNumero ){
   write( 1, cncpBuffer, 0 );
   systest( iNumero );
   return 0;
}
```

test1.c

```
#include <usr/libsodium.h>

int iFnImprimirNumero( const char *, int  );


int main(){
    asm volatile("sub $0x08, %esp");
    /* Por convención de Intel / C la siguiente función reemplaza
     8 bytes del stack que no le pertenecen así que resto 2 bytes
     al stack pointer */
    iFnImprimirNumero("\nEjecutando en nivel: ",1);
    /* Y ahora se los sum */
    asm volatile("add $0x08, %esp");
    asm volatile("lcall $(0x40), $(0x00)");
    asm volatile(".byte 0xCB"); // Ret Far
}


int iFnImprimirNumero( const char *cncpBuffer, int iNumero ){
   write( 1, cncpBuffer, 0 );
   systest( iNumero );
   return 0;
}
```

test0.c

```
#include <usr/libsodium.h>

int iFnImprimirNumero( const char *, int  );


int main(){
    asm volatile("sub $0x08, %esp");
    /* Por convención de Intel / C la siguiente función reemplaza
     8 bytes del stack que no le pertenecen así que resto 2 bytes
     al stack pointer */
    iFnImprimirNumero("\nEjecutando en nivel: ",0);
    /* Y ahora se los sum */
    asm volatile("add $0x08, %esp");
    asm volatile(".byte 0xCB"); // Ret Far
}


int iFnImprimirNumero( const char *cncpBuffer, int iNumero ){
   write( 1, cncpBuffer, 0 );
   systest( iNumero );
   return 0;
}
```

SEGUNDA PARTE

Base Teórica

Para la segunda parte del Trabajo Práctico nos basamos en la estructura existente del shell a nivel 0 y en el estándar POSIX para la utilización de System calls.

Análisis del Caso

En este punto del Trabajo Práctico, contamos con las bases funcionales para ejecutar un proceso a nivel básico. Pero aún quedan muchos problemas por solucionar para el pasaje de shell 0 a nivel 3..
Entre estos problemas se encuentran:

- La forma de elaborar incrementalmente y probar el shell a nivel 3 sin contar con un shell a nivel 0.
- Establecer la manera de capturar el teclado desde el nivel 3 y utilizar un buffer propio para el comando ingresado.
- Adaptar los syscalls y crear nuevos según se requiera para la transición.
- Establecer la forma de ejecución de los comandos debido a que ya no cuentan con los permisos necesarios para hacer todo lo que hacían antes.

Cada uno de estos problemas se analiza y resuelve en la siguiente sección.

Opciones disponibles

Enumeraremos por separado los problemas presentados y sus opciones:

- La forma de elaborar incrementalmente y probar el shell a nivel 3 sin contar con un shell a nivel 0.

  En este punto nos encontramos con dos opciones de desarrollo.

  - Desarrollar el nuevo shell como proceso usuario ejecutado desde shell 0
  Las ventajas de este método es su fácil puesta en funcionamiento, con la ejecución de un programa a nivel usuario garantizada al final de la Parte I, cualquier desarrollo de shell partiendo como programa usuario partía de una base sólida para el desarrollo. En cambio, si bien este desarrollo era seguro, no garantizaba la futura implementación sea correcta, porque si bien ejecutaría correctamente con el shell anterior como backup, es imposible saber si librado a una ejecución propia funcionaría. Un fallo en este punto podría significar un trabajo de debugging importante hasta encontrar la causa.
  Otra desventaja es el hecho de que sería difícil diferencia entre ambos shells y sus funcionalidades.

  - Desarrollar el nuevo shell ejecutandose desde el inicio como único shell
  Si bien este desarrollo toma mucho mas esfuerzo de debugging inicial y de implementación, es realmente una mejor alternativa. Todo desarrollo tomado desde el punto en que finalmente debía estar implementado nos garantizó que el shell corriera perfectamente en última instancia, a pesar de que costó mas desarrollarlo inicialmente.

- Establecer la manera de capturar el teclado desde el nivel 3 y utilizar un buffer propio para el comando ingresado.

  Para la captura de teclado tuvimos en cuenta tres alternativas.

  - Espera activa en kernel
  El shell a nivel usuario solicita una syscall para leer un carácter, y se realiza una espera a nivel kernel hasta que se recibe una tecla y se devuelve al proceso. La desventaja de esto es que es completamente bloqueante hasta que se recibe una tecla y totalmente inefectivo.

  - Espera activa en Shell usuario
  La espera activa a nivel shell, propone un ciclo infinito en el usuario. Dentro del ciclo se solicita una syscall “preguntando” si ha habido un ingreso de teclado dentro del buffer de teclado del kernel. Esta alternativa es viable y fácil de implementar.

- Espera pasiva en Shell usuario

La espera pasiva a nivel shell, implementa un vector de solicitudes de teclado manejado por el planificador de tareas. El syscall bloquea al proceso a la espera de actividad del teclado. Esta opción es mucho más eficiente aunque mas costosa de implementar.

- Adaptar los syscalls y crear nuevos según se requiera para la transición.

Se tuvieron que hacer modificaciones en los syscalls fork, waitpid, exit, exec a fin de que funcionen correctamente en nivel tres.
Además se ha creado un syscall específico para el shell con sub-servicios para la ejecución de comandos.

- Establecer la forma de ejecución de los comandos debido a que ya no cuentan con los permisos necesarios para hacer todo lo que hacían antes.

Todos los comandos del shell anterior a nivel cero cuentan con los privilegios suficientes como para efectuar todas las operaciones necesarias para funcionar. Sin embargo en nuestro shell a nivel usuario esto es un poco mas complicado, y se presentaron las siguientes opciones:

- Invocación total del comando en nivel 0

El shell a nivel usuario reconoce los comandos pero solo ejecuta un syscall hacia los formulados en nivel 0 del shell anterior. Esto es factible para algunos comandos pero genera muchos conflictos con algunos otros, especialmente los que requieren ingreso de datos o esperas, ademas de que algunos comandos no requieren un nivel de privilegio de kernel para ejecutarse. Por lo que esta opción además de inviable es poco eficiente.

- Ejecución parcial en nivel 3 y prestación de servicios a nivel 0

Con esta alternativa propusimos ejecutar en shell usuario todo lo que no requiera servicios del kernel (salvo, por supuesto, imprimir en pantalla y recibir teclado) en funciones propias y, a medida que fuera necesario, utilizar un syscall específico de servicios de shell (en nuestro caso llamado shellfc) en el que se proveen las soluciones para la ejecución privilegiada de los comandos en particular (mas detalles sobre esto en la parte práctica)

Elección y Justificación de la elección

Para la transición del shell a nivel 3, decidimos desarrollarlo completamente desde cero (copiando algunas características del original) como único shell, con espera activa a nivel usuario adaptando los syscalls necesarios y con ejecución parcial en nivel 3 y prestación de servicios a nivel 0 para la ejecución de comandos.
La elección de espera activa a nivel usuario para el teclado se debió a la falta de tiempo disponible para el desarrollo del TP, eso junto a muchas otras cosas que deberían ser refinadas realmente hubieran valido la pena la extensión del plazo de desarrollo.

- Segmentos de memoria y sycalls Fork y Exec:
Vale hacer mención aparte a estos syscalls y la forma en que corren los procesos. En este punto fue decisión del grupo el separar la asignación de memoria de los procesos en 2 partes. Un segmento de memoria para código (CS) y otro para Datos y Stack (DS y SS). La decisión de implementarlos tuvo varias razones.

- Forma de compilar en C.
  El pasaje de parámetros se hace en las primeras posiciones del SS por lo que si ocupa el mismo segmento de memoria que el código este parámetro estaría pisando las primeras instrucciones (conclusión teórica y práctica a la vez al comprobar mediante debugger que esto efectivamente pasaba).

- Integridad del código.
  Es muy probable y ha pasado durante todo el desarrollo del trabajo práctico que el stack invada porciones del código impidiendo muchas veces el retorno de una función o directamente invalidandola. La separación hace imposible este problema (sin embargo sigue pasando a nivel kernel, detalle que debe ser completamente evaluado antes de seguir con cualquier desarrollo).

- Sencillez en la implementación.
  La separación a nivel segmento de memoria y no tan solo a nivel selector da al programador la libertad de manipular el stack y los datos sin riesgos de pisar el código. Además permite manipular el tamaño por separado. (Esto fue de suma utilidad al hacer los syscalls fork y exec).

Esto sin embargo ha traído algunos problemas que por falta de tiempo no han podido ser solucionados de raíz aunque si de manera provisoria. El problema es el sodium toma todos los datos respecto a la posición 0 para la toma de variables por ejemplo en vFnImprimir, y la suma se hace respecto al CS. Esto debe ser adaptado para tomar en cuenta que C utiliza DS y SS para variables y CS para constantes. La separación produjo errores respecto desde donde tomar el offset de la variable (solucionado por ahora), sería recomendable retomar la investigación del funcionamiento interno del sodium en este punto (entre muchos otros) antes de continuar agregando módulos.

Introducción Práctica

Partiendo de un ejecutable contieniendo un while(1) como shell inicial a nivel usuario comenzamos a agregarle todas las funcionalidades propias de un shell.

Resumen de modificaciones

La mayor parte de las modificaciones fueron hechas sobre shellusr.c, libsodium.c, syscall.c, shell.c, y system_asm.asm

- shellusr.c: Se comporta de manera similar al anterior shell, tomando del buffer de teclado la entrada y guardandola en un buffer local. Al presionar ENTER se interpreta el comando y sus parámetros.

- libsodium.c: Se agregó una API para los servicios de shell llamada shellfc.

- syscall.c: De la misma manera se agregó una syscall shellfc para los servicios requeridos por el shell a nivel usuario. Además se adaptó para poder recibir los parámetros definidor por POSIX (ebx, ecx, edi, y esi) ya que previamente estaba solamente implementado ebx.

- Shell.c: El código del shell.c viejo no se descartó ya que muchas de sus funcionalidades siguen siendo útiles a traves de syscalls (muchas de ellas adaptadas para funcionar correctamente) aunque realmente debería tener una depuración de las cosas que ya no son funcionales.

  system_asm.asm: Se han hecho modificaciones para el pasaje de mas parámetros POSIX.

Documentación de Funciones y programas

- Shellusr.c contiene un set diferente de funciones respecto al shell.c original ya que muchas son directamente referenciadas completamente por el syscall shellfc a las originales, otras son implementadas parcialmente (en parte a nivel usuario y en parte a nivel kernel) y otras son resueltas completamente a nivel usuario. Las funciones resultantes son:

```
int iFnConvertirCmd(char*);
int iFnObtenerCmd(char*, int*, int*);
int iFnEsCmdShell(const char*);
int iFnStrcmp(const char*, const char*);
int iFnMayusculas(char*);
int iFnImprimir( const char*);
int iFnImprimirNumero( int);
int iFnLeerCaracter( char* );
int iFnImprimirCadena(char*);
int iFnEjecutarBinario(char*, int);
int iFnFuncionShell(int, int, int);
void vFnMenuAyuda();
void vFnMenuSet();
void vFnMenuPlanif();
int iFnFuncionIdle();
int iFnCtoi (char*);
int iFnLongitudCadena (const char*);
void vFnMenuDescUsr(int);
void vFnMenuDumpUsr(int,int);
```

Además de definir un tamaño para el buffer de teclado:

```
#define                TAMANIO_BUFFER_SHELL        255
```

- syscall.c: El syscall SHELL_FC contiene todos los servicios a shell usuario que se necesitaron para el pasaje de los comandos, cada servicio tiene un número como está definido en syscall.h:

__FC_LOG	1	__FC_PAG	16	__FC_LOTE	31		
__FC_COLOR	2	__FC_TSS	17	__FC_PCB	32		
__FC_CLS	3	__FC_STACK	18	__FC_IDLE	33		
__FC_GDT	4	__FC_EXEC	19	__FC_DESC	34		
__FC_VERSHM	5	__FC_PLANIF	20	__FC_EJECUTAR	35		
__FC_VERSEM	6	__FC_SYSK	21	__FC_FD	36		
__FC_GETPID	7	__FC_LEER	22	__FC_SUMAFPU	37		
__FC_GETPPID	8	__FC_DUMP	23	__FC_VER	38		
__FC_REBOOT	9	__FC_KILL	24	__FC_WINM	39		
__FC_PS	10	__FC_CD	25	__FC_INTER	40		
__FC_SEGS	11	__FC_LS	26	__FC_CAMB	41		
__FC_MEM	12	__FC_RM	27	__FC_SETENV	42		
__FC_BITMAP	13	__FC_WRITE	28	__FC_UNSETENV	43		
__FC_IDT	14	__FC_CHECK	29	__FC_SHOWENV	44		
__FC_YIELD	15	__FC_WAITPID	30	__FC_GETENV	45		

En la sección de anexos se incluirá una parte del código respecto a las acciones de cada una de estos servicios al shell usuario.

Descripción de nuevas funcionalidades

- ✓ Shell completamente funcional a nivel usuario implementado desde el inicio del Sodium.
- ✓ Ejecución de archivos binarios respetando el estandar POSIX (fork+exec)
- ✓ Los comandos existentes fueron satisfactoriamente adaptados para funcionar en el nuevo shell, con los siguientes comentarios:

Comando	Comentario
ayuda	Funcional
bitmap	Funcional (Solo en Modo Paginado)
verSem	Funcional
verShm	Funcional
check	Funcional
sysgetpid	Funcional
syswaitpid	Funcional
syskill	Funcional
planif	Funcional
lote	Funcional
log	Funcional (No permite scroll)
pcb	Funcional
leer	Requiere revisión (Con errores de origen)
idle	Funcional
reboot	Funcional
cambiateclado	Funcional
cls	Funcional
desc	Funcional
dump	Funcional
ejecutar	Funcional (No Posix)
stack	Funcional
execve	Funcional
exec	Funcional
gdt	Funcional
idt	Funcional
kill	Funcional
cd	Indeterminado (no falla pero no toma parámetros)
ls	Requiere revisión (Con errores de origen)
rm	Requiere revisión (Con errores de origen)
mem	Funcional (se congela al final)*
write	Funcional
pag	Funcional (Solo en Modo Paginado)

yield	Funcional
interval	Funcional
fd	Funcional
ps	Funcional
segs	Funcional
set	Funcional
sumafpu	Funcional (se congela al final)*
tss	Funcional
ver	Funcional
winm	Funcional

* Las operaciones con numeros de punto flotante funcionan pero al momento de volver a la función llamadora fallan. Probablemente estén reemplazando el stack de retorno, hay que revisar porque está sucediendo.

Estado del Trabajo Practico al finalizar la Parte II

El Sodium ahora funciona mucho más como lo hacen los sistemas operativos basados en UNIX. Con excepción del reloj y el proceso nulo, los demas procesos, incluído el Shell, son archivos ejecutables. El shell corre a nivel 3 y depende de los servicios prestados por el kernel para la ejecución de comandos y la creación de nuevos procesos.

Por mas que quedan muchos detalles por pulir y poco tiempo para hacerlo, el Sodium respeta muchos de los standares POSIX por lo que no tomaría mucho trabajo poder compilar aplicaciones simples de, por ejemplo Linux, para que corran satisfactoriamente en Sodium.

Conclusión Parte II

El desarrollo de esta segunda parte ha sido mucho menos investigativo que la primera. En este caso lo más complicado consistió en solucionar problemas preexistentes. Podría decirse que la mayoría de horas invertidas en este proceso fue de debugging a bajo nivel (utilizando el, por suerte muy útil, debugger de bochs). Surgieron muchas complicaciones imprevisibles sacando a flote malas implementaciones de años previos obligandonos a resolverlas antes de seguir con nuestro desarrollo. Entre ellas la más grave y todavía sin solucionar fue el problema del stack de kernel.
La administración de memoria y la decisión de donde ubicar las pilas para el momento de las interrupciones, es algo que debe ser solucionado antes de seguir implementando funcionalidades debido a que fueron necesarios arreglos provisionales utilizando constantes y posicionando el stack en lugares fijos antes de decidir una estrategia realmente óptima para esta decisión.
Los syscall Fork y Exec son especialmente sensibles a esta situación, por lo que su implementación fue por demás complicada (aunque exitosa). Al ser tan vulnerables los stacks en este punto, la recuperación de información de retorno (eip, cs, eflags, etc). Se vió afectada muchas veces por la naturaleza de la llamada (por ejemplo: procesos grandes o la recursividad tienden a pisar el stack).

ANEXO III: HISTORIAL DE MODIFICACIONES

Checkpoint 1: Inicio de Parte II

```
-(usr/libsodium.c)
    * No se utiliza mas sc.c. Antes funcionaba desde el
libsodium haciendo un callgate a sc.c y este hacia el int
0x80, ahora se cambiò el DPL de la int 80 y se hace
directamente desde el libsodium

-(usr/sc.c) BORRADO

-(usr/shellusr.c) NUEVO
    * Implementando funciones  que invoquen las API de
libsodium.h Para leer y escribir caracteres en pantalla. Por
ahora es solo lo básico.

-(kernel/syscall.c)
    * Creada nueva lFnRead a partir de lFnWrite. No estaba
hecha...
    * Incluída en header las funciones propias de
/shell/teclado.h para sacar de el buffer de teclado de shell
los char recibidos por teclado.

-(kernel/shell.c)
    * Borrado Todo menos la inicializaciòn del teclado.
Podrìa llamarse de ahora en mas TeclInicializer.c

-(kernel/main.c)
    * El shell a nivel usuario y la inicialización de
teclado se hacen al final, porque los mensajes ya no son
procesos por separado
      y no son llamados por el planificador. Salvo el shell
mismo que es un proceso usuario y no funciona hasta que se
activan las
      interrupciones y el plainficador lo llama.
    * Llama a iFnProcShell() para la inicialización del
teclado y demas mensajes.
```

Checkpoint 2: Creando al Shellusr

```
-(usr/shellusr.c)
    * Creado lo bàsico del shell user de 0 pero basandonos
en el shell a nivel 0 que ya estaba

-(kernel/gdt.c)
    * vFnImprimir(" "); Por alguna razon imposible de
determinar, esta linea salva que el shell no se tilde...
```

Checkpoint 3: Instanciar Binario y Fork

```
-(kernel/gdt.c)
    - iFnDuplicarProceso: La funcion estaba bien
bàsicamente, solo le faltaba el poder definir el D_DPL3 al
crear
         los segmentos. Aparte estaban muy mal seteados los
comienzos de los stacks de kernel y usuario y los offsets
para
         llegar a los resultados. Ahora esta corregido.
         Adaptado todo para usar 2 segmentos, uno de codigo
y otro de datos en iFnDuplicarProceso y también en
iFnCrearProceso
    - Agregada la siguiente linea que es fundamental para el
funcionamiento de la doble segmentación.
       // Esta lìnea deberìa ser incluida en iFnCrearPCB
pero dado que esta usado en tantos
       // lugares, cambiarla demanda mucho esfuerzo siendo
que esta en etapa de prueba
       // En cuanto se confirme su efectividad debe ser
integrada a dicha funcion
       pstuPCB[iPosicion].uiDirBaseDatos =
uiBaseSegmentoDatos;

-(kernel/pcb.h)
   - Nueva variable agregada a stuPCB: uiBaseSegDatos. Para
el segmento de datos.
```

Checkpoint 4: Duplicar y reemplazar proceso

-(kernel/gdt.c)
    - iFnReemplazarProceso: Ajustados todos los offsets a la nueva configuraciòn de stacks. Todos los esp0 y stacks de usuario se calculan como 0x3ff0 - 0x200 * Posicion dentro del pcb.
    - iFnDuplicarProceso: Acà tambien se hizo el ajuste.

-(kernel/syscall.c)
    - En la syscall de execve se hicieron los ajustes para poder pasar parametros.

-(usr/shellusr.c)
    - iFnEjecutarBinario es completamente funcional y el shell espera a que termine el proceso con waitpid.

-(kernel/system_asm.asm.c)
    - Increíble, pero edx no estaba pudiendo ser pasado como parametro en los syscalls. EDX era el que fallaba. Ahora hay que ver cuales mas faltan.

Checkpoint 5: Implementando Servicios de Shell

-(kernel/gdt.c)
    - Todavía hay problemas de tamaño del segmento de datos, hay que setearlos manualmente multiplicandolos para que entre todo y no se "pelee" con el stack. Verlo bien.

-(usr/shellusr.c)
    - Implementadas nuevas funciones propias del anterior kernel mediante al syscall Shellfc.

Checkpoint 6: Fin Parte II

-(kernel/shellusr.c)
    - Se han agregado todos los comandos reconocidos anteriormente por el shell a nivel 0 de tres maneras diferentes:
    * Ejecución total en kernel: los comandos son referenciados completamente mediante un servicio de shellfc
    * Ejecución total usuario: Algunos comandos no requieren servicios del kernel mas alla de escribir en pantalla y han sido programados integramente en shellusr
    * Ejecución parcial: Algunos comando tienen combinación de factores de las dos anteriores, debiendo programarlos en shellusr pero utilizando funcionalidades provistas por el syscall shellfc

-(usr/syscall.c)
    - Se completaron todos los servicios a shell usuario en la función lFnSysShellFc
 Algunas de estas funciones referencian a
  pstuPCB[iFnBuscaPosicionProc(ulProcActual)].uiDirBaseDatos + iPar1
 Esto se utiliza en el pasaje de punteros a char debido a la limitación de Sodium/c de siempre referenciar a el code segment+offset. Las cadenas asignadas a variables se guardan en el data segment por lo que la corrección debe ser hecha a "mano" utilizando la dirección de base del segmento de datos sumado a iPar1 que es el offset utilizado por c en el data segment.

Checkpoint 7: Modificaciones sugeridas por la cátedra

-(kernel/gdt.c)

- Se han removido las funciones que ya no se utilizaban: iFnCrearTss2 y iFnCrearTarea

Mientras que se les ha dado un nombre mas adecuado a las siguientes funciones

iFnCrearTareaEspecial -> iFnInstanciarReloj

iFnCrearTSSTareaEspecial -> iFnCrearTSSReloj

Que solo servían para instanciar un solo proceso (el de reloj). Por eso no vale la pena removerlas pero tampoco su nombre reflejaba su verdadera funcionalidad

-(usr/shellusr.c)

- Se han comentado todas las funciones con formato doxygen.

ANEXO IV: SHELL_FC, LISTA DE SERVICIOS A SHELLUSR

Syscall.c (fragmento)

```
LONG LFNSYSSHELLFC(INT IOPCION, INT IPAR1, INT IPAR2)
{
    IF(IOPCION == __FC_LOG)    VFNCAMBIARTERMINAL();
    IF(IOPCION == __FC_COLOR)  VFNSYSSETEARCOLOR(IPAR1, IPAR2);
    IF(IOPCION == __FC_CLS)    VFNMENUCLS();
    IF(IOPCION == __FC_GDT)    VFNMENUGDT(IPAR1);
    IF(IOPCION == __FC_VERSEM) VFNMENUVERSEMAFOROS();
    IF(IOPCION == __FC_VERSHM) VFNMENUVERSHM();
    IF(IOPCION == __FC_GETPID) VFNIMPRIMIR("\NEL SYSCALL GETPID() ME
DEVOLVIO %D", LFNSYSGETPID());
    IF(IOPCION == __FC_GETPPID) VFNIMPRIMIR("\NEL SYSCALL GETPPID() ME
DEVOLVIO %D", LFNSYSGETPPID());
    IF(IOPCION == __FC_REBOOT) LFNSYSREBOOT(0);
    IF(IOPCION == __FC_PS)     VFNMENUPS();
    IF(IOPCION == __FC_MEM)    VFNMENUMEM();
    IF(IOPCION == __FC_SEGS)   VFNMENUSEGS();
    IF(IOPCION == __FC_BITMAP) VFNIMPRIMIRMAPABITS();
    IF(IOPCION == __FC_IDT)    VFNMENUIDT(IPAR1);
    IF(IOPCION == __FC_YIELD)  LFNSYSSCHEDYIELD();
    IF(IOPCION == __FC_PAG)    VFNMENUPAG(IPAR1);
    IF(IOPCION == __FC_TSS)    VFNMENUTSS(IPAR1);
    IF(IOPCION == __FC_STACK)  VFNMENUSTACK(IPAR1,IPAR2);
    IF(IOPCION == __FC_EXEC)   VFNMENUEXEC(IPAR1,IPAR2);
    IF(IOPCION == __FC_PLANIF) IF(IPAR2==5)
VFNMENUPLANIF(PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBASE
DATOS+IPAR1,IPAR2);  ELSE VFNMENUPLANIF(IPAR1,IPAR2);
    IF(IOPCION == __FC_LEER)
VFNMENULEERARCH(PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRB
ASEDATOS+IPAR1);
    IF(IOPCION == __FC_DUMP)   VFNMENUDUMP(IPAR1,IPAR2);
    IF(IOPCION == __FC_CD)
VFNMENUCD(PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBASEDAT
OS+IPAR1);
    IF(IOPCION == __FC_LS)
VFNMENULS(PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBASEDAT
OS+IPAR1);
    IF(IOPCION == __FC_RM)
VFNMENURM(PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBASEDAT
OS+IPAR1);
    IF(IOPCION == __FC_WRITE)
VFNMENUWRITEARCH(PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIR
BASEDATOS+IPAR1,PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBA
SEDATOS+IPAR2);
    IF(IOPCION == __FC_KILL)   VFNMENUOPTKILL(IPAR1);
    IF(IOPCION == __FC_SYSK)   RETURN LFNSYSKILL(IPAR1, IPAR2);
```

```
    IF(IOPCION == __FC_CHECK)   {IF (UIMODOMEMORIA == MODOPAGINADO)
VFNMENUCHECK(IPAR1);
                        ELSE  VFNIMPRIMIR("\NERROR: COMANDO NO APLICABLE.");}
    IF(IOPCION == __FC_WAITPID)  VFNESPERARPID(IPAR1);
    IF(IOPCION == __FC_LOTE)    {INT *IP;
IP=PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBASEDATOS+IPAR2;
VFNMENULOTES(PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBASE
DATOS+IPAR1,
PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBASEDATOS+IP[0],
PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBASEDATOS+IP[1]);
                        }
    IF(IOPCION == __FC_PCB)     VFNMENUPCB(IPAR1);
    IF(IOPCION == __FC_DESC)
VFNMENUDESC(PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBASED
ATOS+IPAR1,IPAR2);
    IF(IOPCION == __FC_EJECUTAR)
VFNMENUEJECUTAR(PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRB
ASEDATOS+IPAR1);
    IF(IOPCION == __FC_FD)      VFNMENUFD();
    IF(IOPCION == __FC_SUMAFPU)
VFNMENUSUMAFPU(PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBA
SEDATOS+IPAR1,
PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBASEDATOS+IPAR2);
    IF(IOPCION == __FC_VER)     VFNMENUVER();
    IF(IOPCION == __FC_WINM)    {VFNCAMBIARVISIBILIDADVENTANAPROCESO();
VFNIMPRIMIRCONTEXTO();}

    IF(IOPCION == __FC_INTER)   {VFNIMPRIMIR("\NPARECE SER EL INTERVALO");
                                 INT LAUX=LFNSYSSCHEDRRGETINTERVAL( 0,0);
                                 VFNIMPRIMIR("\NTIME SLICE: %D",LAUX);}
   IF(IOPCION == __FC_CAMB)     VFNMENUCAMBIATECLADO(IPAR1);
   IF(IOPCION == __FC_SETENV)
VFNSETENV(PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBASEDAT
OS+IPAR1,
PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBASEDATOS+IPAR2);
  IF(IOPCION == __FC_UNSETENV)
VFNUNSETENV(PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBASED
ATOS+IPAR1);
  IF(IOPCION == __FC_SHOWENV)       VFNGETALLENV();
  IF(IOPCION == __FC_GETENV)
{IF(CPFNGETENV(PSTUPCB[IFNBUSCAPOSICIONPROC(ULPROCACTUAL)].UIDIRBASE
DATOS+IPAR1) != NULL) RETURN 1; ELSE                              RETURN
0;}


        RETURN 0;
}
```

PARTE FINAL

Conclusión

Durante el transcurso del trabajo práctico hemos tenido la necesidad de aprender conceptos sobre el funcionamiento interno de los sistemas operativos modernos y la arquitectura Intel:

- El formato y utlización de las tablas GDT e IDT.
- Los pasos del context switching y la tabla TSS
- La estructura y funcionamiento del Planificador de Tareas de sodium
- El manejo de stack de usuario, de kernel y los punteros de transición para llamadas o cambios de contexto (esp0,esp1,esp2)
- Diferentes formas de solicitar servicios al kernel por parte de un proceso usuario
- Formato y funcionamiento de las Callgates
- Concepto de Syscall y su funcionamiento.
- Estandar POSIX para el uso de syscalls
- Diferentes modos de crear un shell y sus comandos
- Estructura interna del sodium y el boot loador
- Herramientas de debugging
- Modo de funcionamiento de las maquinas virtuales

Podríamos afirmar que del tiempo total de desarrollo del trabajo práctico, la parte de aprendizaje ocupó un tercio del mismo.

El segundo tercio fue de adaptación (en mayor medida en la parte I) donde tuvimos que adaptar un sistema operativo planificado en cierta medida para funcionar con varios niveles de privilegio pero completamente programado en nivel 0. En este punto debimos trabajar a la par con los manuales de Intel y los debuggers. La verdadera complicación en este punto es la incapacidad de ir paso a paso en el desarrollo cuando es un cambio tan importante (donde un proceso ejecuta en modo kernel o modo menos privilegiado). Esto hizo que haya que ser muy detallistas y debuggear a nivel bit ante cada paso porque un error de programación en ese punto es muy difícil de tracear.

El último tercio consistió en el desarrollo gradual del shell a nivel usuario, lo que implicó un esfuerzo mucho menos intensivo pero si mas extensivo. Pudimos avanzar incrementalmente dado que una vez logrado correr a nivel usuario pudimos construir sobre una base estable. Las mayores complicaciones surgieron al incrementarse el tamaño del shell y al utilizar las syscalls. Muchas fallas casi imprevisibles se presentaron en este punto. Tras mucho debugging pudimos observar que muchas de estas surgieron por lo ya explicado en conclusiones anteriores sobre el tamaño y posición de los stacks.

Como conclusión final estamos en condiciones de afirmar que a pesar de que entender la estructura del sodium toma un tiempo razonable, el hecho de lidiar con malas implementaciones de funcionar anteriores retrasa el desarrollo a pasos ínfimos. Esta entrega contiene muchísimas correcciones a TPs anteriores, que es algo que no estaba en el alcance del TP sin embargo fueron necesarias (algunas no tan necesarias pero si hechas a conciencia). A su vez, si bien, la funcionalidad de este TP está probada en múltiples plataformas de manera satisfactoria, no podemos asegurar que futuros desarrollos esten exentos de fallar por faltas incluídas en esta versión que no son detectadas al no ponerse a prueba.

Este grupo considera y recomienda que antes de seguir desarrollando módulos importantes se trabaje y ponga énfasis en la solidez y confiabilidad del sodium en este punto para que los siguientes desarrollos puedan ser hechos en una base sin fallas (no se puede puede garantizar esto en este punto). Esto será tanto más difícil cuanto mas pasen las cursadas, debido a que el conocimiento de estos detalles que este grupo posee no estarán a disposición de los grupos del próximo año que tendrán que lidiar con ellos en su proceso de descubrimiento y desarrollo.

Adjuntamos a esta entrega los archivos fuente y los disquetes booteables consistes en una versión de Sodium con capacidad para correr procesos de nivel de 0 a 3 y un shell interactivo que corre a nivel usuario utilizando POSIX como estándar para los servicios de kernel.

Bibliografía

- Intel Architecture Software Developer's Manual Volume 3: System Programming
- Intel Architecture Software Developer's Manual Volume 4: Protection
- Intel Architecture Software Developer's Manual Volume 6: Task Management
- Apuntes de la cátedra.
- http://en.wikipedia.org/wiki/Global_Descriptor_Table
- http://www.jamesmolloy.co.uk/tutorial_html/4.-The%20GDT%20and%20IDT.html
- Documentación automática proveída por el Doxygen
- Stallings W. "Sistemas Operativos" Edición 2da