Guía del Diseño y la Implementación de GTL 0.1

Maximiliano Cristiá Gisela Giusti Felipe Manzano
{mcristia, ggiusti, fmanzano}@fceia.unr.edu.ar

Grupo de Investigación y Desarrollo en Ingeniería de Software
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario, Argentina

17 de febrero de 2004

Resumen

Este documento contiene la siguiente información:

— Relación entre cada llamada al sistema (de las que se han considerado importantes para esta versión) y las operaciones del modelo formal

— Descripción de los TADs a utilizar así como también de las funciones auxiliares

— Tópicos que deben investigarse para obtener una implementación correcta respecto del modelo formal

La organización del documento no es la mejor para ser usado como referencia pues se lo ha organizado de manera tal que cada sección contenga la asignación de trabajo para cada equipo de desarrollo.

Índice

1. Generalidades

1.1. Metodología de trabajo

Todos los equipos de desarrollo deben leer este documento y el que describe el modelo formal [1] por completo, antes de comenzar a programar.

Cada equipo de desarrollo debe trabajar en el orden estipulado en cada sección pues hay otros equipos que dependen de la terminación de tareas efectuadas por otros equipos.

Los equipos que deban implementar piezas de software que están especificadas formalmente en [1] deben tener en cuenta que el modelo es una abstracción del sistema de archivos de Linux (y en general de su núcleo) y que no lo representa fielmente. En algunos casos, el modelo establece predicados que ya están implementados en Linux y en otros la implementación difiere notablemente de lo expresado en Z (aunque en esencia es lo mismo). En consecuencia, una parte importante de su trabajo es saber qué deben implementar y qué deben revisar para verificar que cumple con la especificación. Dado que este punto es muy delicado deben recurrir al equipo de coordinación en caso de duda.

1.1.1. Asignación de tareas a los equipos

A continuación se indica qué tarea le corresponde a cada equipo de trabajo. Cuando se indica una sección de este documento significa que el equipo debe realizar todos los estudios y modificaciones detalladas en esa sección.

— Coordinación general, integración y aprobación: Maximiliano Cristiá, Gisela Giusti y Felipe Manzano

— Sección 2: Andrés Krapf y Pablo Menichini

— Sección 3: Federico Wiecko

— Sección 4: Lorena Glassel y Brenda Lieber

— Sección 5: Javier Murillo y Emilio Siningen

— Sección 6: equipo de Lucas Charles (FaMAF)

1.1.2. Documentación que debe ser presentada

Todos los equipos de trabajo deben entregar la siguiente documentación (más allá de la que explícitamente se pida en cada asignación de tareas).

— Código comentado con `doxygen` tal cual se hizo con Lisex 0.0.

  Es decir, al integrar el sistema debe ser posible generar un Manual de Referencia para el Programador semejante al de Lisex 0.0 (mismas secciones, mismos parámetros de documentación, mismo estilo, etc.). Recomendamos ver cómo se documentó el código de Lisex 0.0 para trabajar de la misma forma. Junto con la distribución de Linux que reciban se entrega el archivo `agroups.dox` el cual contiene la definición de los grupos de documentación a utilizar (i.e. "Nuevas llamadas al sistema", "TAD sc", "Constantes o parámetros de compilación definidos en Lisex 0.0", etc.). Si es necesario generar nuevos grupos de documentación esto debe comunicarse al equipo de coordinación.

— Aquellos equipos que deban documentar la interfaz de TADs deben hacerlo tal cual se lo hizo en Lisex 0.0. Es decir deben dividir la documentación del TAD en:

  - Estructuras de datos privadas (ocultas)
  - Constantes (públicas)
  - Funciones de la interfaz
  - Tipos definidos
  - Funciones privadas

  Para lograrlo sugerimos que vean la documentación del TAD `sc` de Lisex 0.0.

- Para documentar una función (llamada al sistema, función auxiliar del núcleo, función de librería, etc.) se deben incluir los siguientes datos (usando los *tags* apropiados de `doxygen`)

  - Parámetros de entrada y/o salida
  - Valores de retorno
  - Precondiciones en lenguaje natural (si es posible tomarlas de [1]).
  - Postcondiciones en lenguaje natural (si es posible tomarlas de [1]).
  - Descripción corta
  - Descripción larga

  En cualquier caso si el texto de documentación hace referencia a un nombre de un término (variable, constante, función, tipo, etc.) que también fue documentado se debe utilizar la nomenclatura de `doxygen` para que este pueda establecer el *link* adecuado.

- Se debe documentar el cuerpo de cada función que tenga una especificación formal, poniendo un comentario que indique el comienzo y el fin de la implementación de una precondición o postcondición (especificada formalmente).

  Por ejemplo, si la especificación de la función `f` indica que se debe verificar que el parámetro de entrada `num` pertenezca a los enteros no negativos, el código debe verse de la siguiente forma:

```
int f(..., int num, ...)
{
.....
//! Verificacion de la precondicion ...
if (num < 0) return ERROR;
//! Fin verificacion de la precondicion ...
.....
}
```

  Por lo tanto, el programador tiene la obligación de estructurar el código de manera tal que la documentación del mismo sea simple y cumpla con el estándar. Sin embargo, no se espera que se modifique completamente la implementación existente con el sólo fin de cumplir con este requisito: el programador debe encontrar el punto medio entre aumentar la calidad de su trabajo (entendida como el apego a los estándares de trabajo) y terminar la tarea en el plazo previsto.

1.1.3. CVS

CVS significa *Control Version System* o *Concurrent Version System*. Es un programa que permite coordinar el trabajo de varios programadores que trabajan sobre el mismo programa. El CVS se compone de una base de datos (en Linux es uno o varios directorios con archivos) y un programa, ambos se suelen llamar también cvs.

El programa `cvs` tiene varios comandos, los más importantes son:

`checkout` (o `get`) el cual permite sacar un (o varios) programa (un `.c` o `.h`) de la base para que el programador lo modifique

`commit` el cual permite actualizar un programa que está en la base con otro que el programador tiene en su directorio de trabajo

`add` que permite agregar un nuevo archivo (`.c` o `.h`) a la base

`diff` que permite hacer un patch como si se hiciera un `commit`. Es decir, es un `commit` sin modificar la base. Este comando es muy útil para nosotros.

Como dijimos la base es un árbol de directorios. Cada programador tiene su propio directorio de trabajo. El directorio de trabajo tiene la misma estructura que la base; lo único que cambia es la raíz. Por ejemplo, la base puede estar en `/gtl/cvs` y el directorio de trabajo del usuario `ggiusti` estará en `/home/ggiusti/gtl` pero a partir de allí ambos árboles de directorios son semejantes. A `ggiusti` le pueden faltar archivos o directorios porque sólo trabaja sobre una parte de la base, o le pueden sobrar porque aun no ha hecho un `add` de un programa nuevo.

(Cuando decimos programa nos referimos a cualquier archivo de texto, pueden ser `.tex` o cualquier otra cosa que se escriba en modo texto.)

Nosotros hemos creado una base de CVS en el sitio de SourceForge. Como acceder esa base requiere de una conexión a Internet decidimos hacer un checkout de todos los programas, grabar esa imagen en un CD y dársela a ustedes (trabajen o no con todos los módulos que están allí). Para poder usar el CVS de SourceForge tuvimos que dar de alta un proyecto en el sitio, pueden verlo en `http://sourceforge.net/projects/gtl/`. Allí también podríamos poner documentación, tener una lista, un foro y mil cosas más; veremos con el tiempo qué vamos usando y qué no.

El programa cvs está diseñado según el estilo cliente/servidor por lo que desde sus casas podrían hacer commits, checkouts, adds o diffs contra el sitio de SorceForge. Pero, obviamente necesitan un cliente CVS. Linux lo tiene y es compatible con el de SourceForge. Por lo tanto esta es una razón más para trabajar desde Linux (lo que incluye conectar la PC a Internet desde Linux).

El programa `cvs` es suficientemente inteligente como para hacer un `commit` (o `diff`) sólo de lo que ha cambiado en el directorio de trabajo del programador que ejecuta el comando. La base del CVS no almacena todo el archivo modificado sino que guarda un único archivo en el cual va anotando las sucesivas modificaciones (para eso usa el comando `diff` de UNIX). Por lo tanto, cada vez que hacen un `commit` (o `diff`) es muy rápido si han trabajado sobre pocos módulos. Esta es una de las razones por las que les dimos el resultado de un `checkout` completo (la otra es porque no estamos seguros sobre qué módulos tienen que trabajar según el enunciado que les tocó).

Entonces para minimizar el uso de Internet y para minimizar los conflictos debidos a cambios hechos por distintos grupos, vamos a trabajar de la siguiente forma:

1. Cada grupo va modificando los módulos que necesita según el enunciado que le tocó.

2. Cuando tiene una serie de cambios que considera que forman un incremento mínimo (recordar diseño) hace un `diff` (no un `commit`). Para esto debe conectarse y ejecutar el comando. El resultado de ese `diff` es un *patch*. Ese patch nos lo deben mandar a nosotros. Más adelante les daremos el comando diff preciso que tienen que usar para minimizar el tiempo de conexión.

3. Cuando a nosotros nos llega un patch lanzamos un `commit`. Ese `commit` puede generar conflictos con cambios previos o cambios hechos por otros grupos. Si nosotros podemos resolver esos conflictos lo hacemos, si no podemos tendremos que reunirnos con el grupo.

Como siempre para más detalles leer la página de manual o el info de cvs.

1.1.4. Contenido del CD

El CD contiene lo siguiente:

`gtl.tar.bz2` que es la imagen de un `checkout` del CVS

`lisex0.0.tar.bz2` que son todos los fuentes usados en Lisex

`modelo.pdf` que es la descripción formal del modelo para GTL (está en inglés pero es una versión preliminar, suficiente como para programar pero debe tener multitud de errores de inglés así que les ruego que los señalen en la copia impresa que tienen)

`diseno.pdf` que contiene los enunciados para cada grupo más unas cuantas pautas de desarrollo y comentarios sobre el diseño y la implementación de GTL

`doxygen-1.3.3.linux.bin.tar` que es un binario de `doxygen` (la herramienta de documentación que vamos a usar) listo para instalar en Linux

`doxygen_manual-1.3.3.pdf` que es el manual de `doxygen`

1.1.5. Reuniones y contactos

La idea es perder la menor cantidad de tiempo posible en reuniones, pero a la vez puede ser que 5 minutos de charla produzcan un avance muy grande si el grupo está trabado con algo. Por lo tanto, preferimos mantener un esquema de reuniones bajo demanda con algunas reuniones fijadas por nosotros. Otra cosa que hace perder mucho tiempo es juntar a todo el equipo de desarrollo por lo que pensamos que, al menos inicialmente, es preferible que las reuniones sean entre cada grupo y nosotros (posiblemente dentro de 6 semanas nos juntemos todos para discutir el avance en general del equipo).

Usemos el mail todo lo que podamos pero tengamos en cuenta que no todos tenemos mail en casa y que no todos estamos viendo mails todo el día y que no todo se puede poner en un mail (a veces escribir algo que se dice en 5 minutos lleva una hora).

Los contactos de todos son los siguientes:

Nombre	Teléfono	Dirección	Email
Ulises Cerviño	4253435	Urquiza 1429 11° B	`ulises.cervino@totalise.co.uk`
Maximiliano Cristiá	4385559	Tucumán 4142	`mcristia@fceia.unr.edu.ar`
Gisela Giusti	155449244	3 de febrero 546 PB B	`ggiusti@fceia.unr.edu.ar`
Lorena Glassel	4263455	J.M. de Rosas 1238 8° B	`lorenaglassel@hotmail.com`
Ignacio Grima	4812209		`nacho@fceia.unr.edu.ar`
Andrés Krapf	4486783	Mendoza 6° B	`k0293@lucre.fceia.unr.edu.ar`
Brenda Lieber	4514389	Brassey 8135	`brendalieber@yahoo.com.ar`
Felipe Manzano	4482830	Mendoza 173 2° A	`fmanzano@fceia.unr.edu.ar`
Pablo Menichini	4384407	San Lorenzo 2582	`m3425@lucre.fceia.unr.edu.ar`
Javier Murillo	4515307	Pje. Cadrón 7610	`jimtito@arnet.com.ar`
Emilio Siningen	4512483	Micheletti 8065	`gringonob@arnet.com.ar`
Federico Wiecko	4485549	Mitre 1568 5° A	`efedoblev@myrealbox.com`

1.2. TADs

Los TADs para clases de acceso y listas de control de acceso son los mismos que los utilizados en Lisex 0.0, excepto que se modifica la implementación de algunas estructuras de datos ocultas. La implementación de las ACL será exactamente la misma que la de Lisex 0.0. La implementación de las clases de acceso se cambia a un arreglo de bits.

En el i-nodo del VFS se agregan dos campos de tipo `sc` (en lugar de uno como se hizo en Lisex 0.0). En en el i-nodo de GTLFS (el sistema de archivos físico de GTL generado a partir de EXT2), se mantienen los campos para almacenar la clase de acceso usados en Lisex 0.0

La razón de este cambio está en el corazón del modelo formal [1]. Allí se considera que las terminales lógicas deben clasificar la entrada provista por el usuario en cierta clase de acceso (que cambia según lo indica el usuario), y que deben poseer una clase de acceso máxima para la salida enviada por los procesos. Por otro lado, el mismo documento estipula que el resto de los objetos manejados por el sistema deben tener una única clase de acceso (tal cual lo era en Lisex 0.0).

Entonces, con el fin de unificar ciertas operaciones y de hacer más elegante (y posiblemente más modificable) la implementación, decidimos trabajar de la siguiente forma:

— En el i-nodo de GTLFS se guarda una única clase de acceso, la cual se accede de la misma forma que en Lisex 0.0.

— La clase de acceso mencionada en el punto anterior es la que se puede modificar con la llamada `chobjsc` y la que retorna la llamada `oscstat`. Sin embargo, la modificación debe hacer utilizando una interfaz definida en la sección 5.

— El i-nodo del VFS llevará dos clases de acceso que son accesibles únicamente por medio de las funciones `msc` y `csc` (por *maximum security class* y *current security class*[1]).

  La relación de estas funciones con el modelo [1] es la siguiente:

  — `csc` y `msc` para el i-nodo de un archivo que no sea una terminal lógica corresponden a la aplicación de *osc* sobre ese i-nodo. Es decir, ambas retornan el mismo valor.

    Por lo tanto, para trabajar con archivos ordinarios puede usarse cualquiera de la dos según convenga. No es simple en este estado del desarrollo saber si en futuras versiones retornarán valores diferentes.

  — `csc` para el i-nodo de una terminal lógica corresponde a la aplicación de *ltsc* a esa terminal

  — `msc` para el i-nodo de una terminal lógica corresponde a la aplicación de *mptsc* a la terminal física asociada con esa terminal lógica.

  Para más detalles, leer con atención la sección 5 y el documento [1].

— Cuando se invoca la llamada al sistema `open` sobre un i-nodo, el VFS se las arregla para que ambas funciones retornen el mismo valor que a su vez es igual al alamacenado en GTLFS.

— Luego, el VFS invoca a la operación de i-nodo `open` de GTLFS, la cual puede cambiar el valor que retorna `csc`

— Por medio de otras funciones descriptas en la sección 5 es posible alterar los valores retornados por las funciones mencionadas en estos párrafos

Todos aquellos grupos de trabajo que deban trabajar dentro del núcleo y que requieran "la" clase de acceso de un objeto deben, entonces, leer atentamente la sección 5 para encontrar una descripción de la interfaz disponible.

[1] La signatura de todas las funciones mencionadas en estos párrafos se brinda en la sección 5.

1.3. Programas confiables

Un programa confiable es una extensión del (micro)núcleo de seguridad que por alguna razón no se lo incluye dentro de aquel. En el peor de los casos, un programa confiable está eximido de todos los controles de seguridad que el núcleo impone a todos los otros programas. En esta versión de GTL los programas confiables pueden hacer muy pocas cosas más que los programas ordinarios (ver [1], operaciones *Exec*, *Chobjsc*).

Por otro lado, en algunos casos es necesario otorgarle confianza a un programa pues debe preservar la integridad de ciertos datos o efectuar ciertas tareas administrativas de una única forma (recordar que los sistemas MLS está orientados fundamentalmente a la protección de la confidencialidad y que por lo tanto las medidas para preservar la integridad rara vez son consideradas). En esta versión de GTL los programas confiables tienen esta misión. Además, los programas confiables deben ser protegidos contra cualquier modificación no autorizada, razón por la cual se los incluye en la TCB. Más aun, un programa es confiable sí y sólo sí está dentro de la TCB. Para determinar si un programa pertenece o no a la TCB se debe usar la función `istrusted` definida en la sección 6.

Dado que la hipótesis de conflicto bajo la cual se desarrolla GTL es la existencia de caballos de Troya, la modificación no autorizada de los programas de la TCB incluye ataques por medio de caballos de Troya. `root` no está exento de ejecutar un caballo de Troya por lo que procesos lanzados por este usuario no tienen autorización para cambiar esos programas. Sin embargo la TCB implementada en esta versión de GTL no es suficientemente fuerte como para impedir estos ataques (futuras versiones contendrán mejores implementaciones de la TCB).

La heurística para programar un programa confiable está constituida por las siguientes reglas.

— La simplicidad y sencillez deben primar sobre cualquier otra cualidad a excepción de la corrección funcional. De aquí se desprende que cuanto más pequeño y cuanta menos funcionalidad tenga el programa, mejor.

— Un programa confiable no puede usar librerías o programas no confiables. Por lo cual, se debe minimizar el uso de `libc`, y en caso de ser necesario, las porciones que se utilicen deben ser asiladas en una nueva librería para su posterior verificación.

  Una alternativa para no usar librerías es utilizar llamadas al sistema.

— El código debe estructurarse de manera tal de facilitar la verificación (formal) del programa. Por lo tanto, se desalienta el uso de abreviaturas provistas por el lenguaje, sentencias de salto incondicional, punteros, etc.

— El programa debe escribirse en un lenguaje que al ser compilado produzca código directamente ejecutable por el procesador. Es decir, por ejemplo, no puede ser un *shell script* ni un programa Perl.

— Todo programa confiable debe comunicar sus estados más significativos (entre los que se encuentra invariablemente el inicio y fin de ejecución) utilizando la denominada *ventana de confianza del núcleo* (ver sección 3 para más detalles).

2. Llamadas al sistema

Esta sección describe los cambios más profundos al sistema, los que en general están especificados en [1].

Está dividido en dos partes siendo la primera muy simple pues se trata de modificar levemente la función `vfs_permission`. En la segunda parte se listan todas las llamadas al sistema relacionadas con el VFS que deben ser modificadas. Cada una de ellas se asocia con una o varias operaciones de [1] y

se explica el motivo de la asociación. Todas las llamadas de VFS que no estén listadas a continuación no deben modificarse.

En la sección 5 se enuncian los cambios a las estructuras de datos del VFS y de GTLFS así como también las interfaces que tendrán esas estructuras. Por tanto el equipo a cargo de la implementación de lo que se describe en esta sección sólo deberá usar (y no programar) esas interfaces.

2.1. Grupo `allgrp` definido implícitamente

En Lisex 0.0 se definió el grupo `allgrp` (GID 666) como el grupo que contiene a todos los usuarios. Sin embargo, la implementación elegida está lejos de ser la mejor (e incluso puede estar errada).

Para solucionar este problema se debe modificar la función auxiliar del núcleo `supplemental_group_member` de forma tal que cuando se verifican los permisos de los grupos secundarios del usuario se considere siempre como grupo secundario del usuario al grupo con GID 666. De esta forma, sea cual fuere el usuario se lo considerará como miembro de `allgrp`.

2.2. Llamadas de Lisex 0.0

Las siguientes llamadas deben copiarse directamente de la implementación de Lisex 0.0:

— `acladd`

— `acldel`

— `chmod`, `fchmod`, `lchmod`

— `chown`, `fchown`, `lchown`

Las llamadas al sistema `owner_close` y `sscstat` presentes en Lisex 0.0 deben ser removidas en esta versión de GTL.

2.3. sys_access

Signatura `long sys_access(const char * filename, int mode)`

Archivo `open.c`

Corresponde a *Stat*

Comentario Efectúa una comprobación de permisos, lo que equivale a solicitar información de control o de los atributos de seguridad por lo que se le deben aplicar las mismas restricciones que a `stat` y `oscstat`.

2.4. sys_aclfstat

Signatura

```
long sys_aclfstat
   (unsigned int fd, struct acl_entry * acl_statbuf, int len)
```

Archivo `stat.c`

Corresponde a *Stat*

Comentario Llamada agregada debido al nuevo modelo de seguridad.

2.5. sys_aclstat

Signatura

```
long sys_aclstat
   (char * filename, struct acl_entry * acl_statbuf, int len)
```

Archivo `stat.c`

Corresponde a *Stat*

Comentario Llamada agregada debido al nuevo modelo de seguridad.

2.6. sys_close

Signatura `long sys_close(unsigned int fd)`

Archivo `open.c`

Corresponde a *Close*

2.7. sys_creat

Signatura `long sys_creat(const char * pathname, int mode)`

Archivo `open.c`

Corresponde a *Create*

2.8. sys_execve

Signatura

```
int sys_execve (const char * filename, const char * argv[], const char
   * envp[])
```

Archivo `exec.c`

Corresponde a *Exec* (y *Mmap* parcialmente)

Comentario Notar que la especificación de *Exec* indica un comportamiento diferente según la llamada la ejecute un programa confiable o uno no confiable. Por tanto se debe determinar:

a. Qué programa está ejecutando el proceso, lo que se hace... (consultar a Felipe)
b. Si el proceso ejecuta un programa confiable, lo que se hace con la función `istrusted` definida en la sección 6

Además, la llamada utiliza las facilidades de mapeo en memoria de archivos con lo cual, al implementarla, se debe tener en cuenta la especificación de *Mmap*. En particular observar que la llamada `mmap` realiza el trabajo de mapeo en una función auxiliar del núcleo llamada `do_mmap_pgoff` En esta función auxiliar se incluyen los controles especificados en *Mmap*. Al mismo tiempo, `execve` usa .... para realizar el mapeo, por lo que no hace falta modificarla según la especificación de *Mmap*.

2.9. sys_fsstatfs

Signatura `long sys_fsstatfs(unsigned int fd, struct statfs * buf)`

Archivo `stat.c`

Corresponde a *Stat*

2.10. sys_fstat

Signatura

```
long sys_fstat (unsigned int fd, struct __old_kernel_stat * statbuf)
```

Archivo `stat.c`

Corresponde a *Stat*

Comentario Si el archivo ya fue abierto en modo `read` no hace falta verificar la precondición de *Stat*.

2.11. sys_fstat64

Signatura

```
long sys_fstat64 (unsigned long fd, struct stat64 * statbuf, long
   flags)
```

Archivo `stat.c`

Corresponde a *Stat*

Comentario No hay página de manual disponible. Debería ser muy semejante a 2.10.

2.12. sys_getdents

Signatura

```
long sys_getdents (unsigned int fd, void * dirent, unsigned int count)
```

Archivo `readdir.c`

Corresponde a *Read*

Comentario GTLFM supone que el directorio del cual se van a leer las entradas ya ha sido abierto por `open` en modo `read`.

2.13. sys_getdents64

Signatura

```
long sys_getdents64 (unsigned int fd, void * dirent, unsigned int
   count)
```

Archivo `readdir.c`

Corresponde a *Read*

Comentario No hay página de manual disponible, debería ser semejante a 2.12. GTLFM supone que el directorio del cual se van a leer las entradas ya ha sido abierto por `open` en modo `read`.

2.14. sys_link

Signatura `long sys_link(const char * oldname, const char * newname)`

Archivo `namei.c`

Corresponde a *Link*

2.15. sys_lstat

Signatura

```
long sys_lstat
   (char * filename, struct __old_kernel_stat * statbuf)
```

Archivo `stat.c`

Corresponde a *Stat*

Comentario En GTLFM sólo se especifica la información de control de acceso que este tipo de llamadas debe devolver, la implementación debe retornar los datos especificados más todos aquellos datos que retorne la implementación actual de la llamada, a menos que haya que modificar la signatura de la llamada.

2.16. sys_lstat64

Signatura

```
long sys_lstat64
   (char * filename, struct stat64 * statbuf, long flags)
```

Archivo `stat.c`

Corresponde a *Stat*

Comentario No hay página de manual disponible. Debería ser muy semejante a 2.15.

2.17. sys_mkdir

Signatura `long sys_mkdir(const char * pathname, int mode)`

Archivo `namei.c`

Corresponde a *Create*

2.18. sys_mmap

Signatura

```
void  *  mmap(void  *start,  size_t length, int prot , int
       flags, int fd, off_t offset);
```

Archivo

Corresponde a *Mmap*

Comentario Mapea archivos abiertos en memoria de manera tal que su contenido se puede acceder sin usar `read` o `write`.

La mayor parte de las verificaciones de seguridad las hace una función auxiliar del núcleo llamada `do_mmap_pgoff`. Por lo tanto, los controles de seguridad se incluyen en esta función. Además `do_mmap_pgoff` es utilizada por otras llamadas al sistema como `uselib` y `execve`.

2.19. sys_newfstat

Signatura `long sys_newfstat(unsigned int fd, structu stat * statbuf)`

Archivo `stat.c`

Corresponde a *Stat*

Comentario No hay página de manual disponible. Debería ser muy semejante a 2.10.

2.20. sys_newlstat

Signatura `long sys_newlstat(char * filename, struct stat * statbuf)`

Archivo `stat.c`

Corresponde a *Stat*

Comentario No hay página de manual disponible. Debería ser muy semejante a 2.15.

2.21. sys_newstat

Signatura `long sys_newstat(char * filename, struct stat * statbuf)`

Archivo `stat.c`

Corresponde a *Stat*

Comentario No hay página de manual disponible. Debería ser muy semejante a 2.32.

2.22. sys_open

Signatura `long sys_open(const char * filename, int flags, int mode)`

Archivo `open.c`

Corresponde a *Open*

2.23. sys_oscstat

Signatura

```
long sys_oscstat (char * filename, int * level, int * categories, int
   len)
```

Archivo `fs_sc.c`

Corresponde a *Oscstat*

Comentario Llamada agregada debido al nuevo modelo de seguridad.

2.24. sys_pipe

Signatura

```
int pipe(int descf[2])
```

Archivo

Corresponde a *Open*

Comentario Abre el archivo que se usará como tubo por lo tanto se la considera como un refinamiento de *Open*.

Es muy importante notar que al crearse un i-nodo de tipo *pipe* se proveen las operaciones de i-nodo correspondientes, las que son invocadas por VFS para efectivizar entre otras cosas la lectura y la escritura en el tubo. Por lo tanto, deben revisarse todas esas operaciones de manera tal que sea implementadas como indica el modelo. En particular, notar que el tubo se vacía una vez que el proceso consumidor lee lo suficiente por lo que es posible alterar la clase de acceso del tubo si así fuese necesario.

2.25. sys_pread

Signatura `ssize_t sys_pread(unsigned int fd, char * buf, ...)`

Archivo `read_write.c`

Corresponde a *Read*

2.26. sys_pwrite

Signatura `ssize_t sys_pwrite(unsigned int fd, char * buf, ...)`

Archivo `read_write.c`

Corresponde a *Write*

2.27. sys_read

Signatura `ssize_t sys_read(unsigned int fd, char * buf, size_t count)`

Archivo `read_write.c`

Corresponde a *Read*

2.28. sys_readdir

Signatura `int readdir(unsigned int fd, struct dirent *dirp, unsigned int count)`

Archivo `readdir.c`

Corresponde a *Read*

2.29. sys_readlink

Signatura

```
long sys_readlink(const char * path, cgar * buf, int bufsiz)
```

Archivo `stat.c`

Corresponde a *Read*

2.30. sys_readv

Signatura

```
ssize_t sys_readv
   (unsigned long fd, const struct iovec * vector,
    unsigned long count)
```

Archivo `read_write.c`

Corresponde a *Read*

2.31. sys_rename

Signatura `long sys_rename(const char * oldname, const char * newname)`

Archivo `namei.c`

Corresponde a *Create*

2.32. sys_stat

Signatura

```
long sys_stat
   (char * filename, struct __old_kernel_stat * statbuf)
```

Archivo `stat.c`

Corresponde a *Stat*

Comentario En GTLFM sólo se especifica la información de control de acceso que este tipo de llamadas debe devolver, la implementación debe retornar los datos especificados más todos aquellos datos que retorne la implementación actual de la llamada, a menos que haya que modificar la signatura de la llamada.

2.33. sys_stat64

Signatura

```
long sys_stat64
   (char * filename, struct stat64 * statbuf, long flags)
```

Archivo `stat.c`

Corresponde a *Stat*

Comentario No hay página de manual disponible. Debería ser muy semejante a 2.32.

2.34. sys_statfs

Signatura `long sys_statfs(const char * path, struct statfs * buf)`

Archivo `open.c`

Corresponde a *Stat*

2.35. sys_symlink

Signatura

```
long sys_symlink(const char * oldname, const char * newname)
```

Archivo `namei.c`

Corresponde a *LinkS*

2.36. sys_uselib

Signatura `long sys_uselib(const char * library)`

Archivo `exec.c`

Corresponde a *Open* en modo *read* seguido de *Read* (y *Mmap* parcialmente)

Comentario Selecciona la librería que usará el proceso que invoca la llamada. Por tal motivo se considera que el archivo que contiene la librería deberá ser abierto en modo de lectura, lo que implica que debe verificarse la especificación de *Open* en modo *read*.

Además, la llamada utiliza las facilidades de mapeo en memoria de archivos con lo cual, al implementarla, se debe tener en cuenta la especificación de *Mmap*. En particular observar que la llamada `mmap` realiza el trabajo de mapeo en una función auxiliar del núcleo llamada `do_mmap_pgoff` En esta función auxiliar se incluyen los controles especificados en *Mmap*. Al mismo tiempo, `uselib` usa .... para realizar el mapeo, por lo que no hace falta modificarla según la especificación de *Mmap*.

2.37. sys_write

Signatura

```
ssize_t sys_write(unsigned int fd, char * buf, size_t count)
```

Archivo `read_write.c`

Corresponde a *Write*

Comentario Tener mucho cuidado al implementar esta operación pues en [1] se definen dos operaciones: *Write* que se aplica a todos los objectos que no sean terminales lógicas y *WriteLT* que se aplica unicamente a las terminales lógicas. Consultar con el equipo de coordinación.

2.38. sys_writev

Signatura

```
ssize_t sys_writev
   (unsigned long fd, const struct iovec * vector,
    unsigned long count)
```

Archivo `read_write.c`

Corresponde a *Write*

Comentario Tener mucho cuidado al implementar esta operación pues en [1] se definen dos operaciones: *Write* que se aplica a todos los objectos que no sean terminales lógicas y *WriteLT* que se aplica unicamente a las terminales lógicas. Consultar con el equipo de coordinación.

3. Ventana del núcleo, dispositivos, terminales y programa `chinsc`

3.1. Ventana del núcleo (camino confiable)

Esta versión de GTL implementará un suerte de camino confiable (ver [2, páginas 170-172] para una descripción conceptual). El camino confiable se utilizará para que los programas confiables y el núcleo se comuniquen con el usuario de forma tal que este no pueda ser engañado. En otras palabras el camino confiable le permitirá al usuario corroborar que ciertos mensajes y ciertas acciones del sistema provienen realmente de software que no es un caballo de Troya.

Usualmente un camino confiable se puede establcer de dos maneras.

— Por medio de una combinación de teclas que es capturada por el núcleo lo que hace que este suspenda procesos no confiables y se comunique directamente con el usuario.

— Por medio de un sector de la pantalla que sólo puede ser utilizado por software de confianza; precisamente es el núcleo del sistema el que se encarga de evitar que cualquier proceso utilice ese sector de la pantalla.

Por lo tanto, este equipo debe modificar el núcleo del sistema de manera tal que exista un sector de la pantalla, denominado *ventana de confianza*, en el cual sólo puedan escribir los programas confiables y el núcleo. Más concretamente, la última línea de la terminal (siempre pensando en modo texto) estará reservada a tal fin. Es atribución del equipo buscar la mejor implementación posible pero deben tener en cuenta lo siguiente:

— El software existente (no confiable) no debe notar el cambio; es decir, cualquier programa Linux debe ejecutar sin problemas pudiendo escribir en la pantalla todo lo que desee.

— En lo posible debería evitarse la introducción de una nueva llamada al sistema para que los procesos confiables utilicen esa zona de la pantalla. Dado que el núcleo tiene la capacidad de definir el tamaño de la terminal, debería ser posible que si un proceso confiable desea utilizar la ventana de confianza, solicite al núcleo escribir en la línea siguiente a la máxima admitida por la dimensión vigente. Este verificará que la petición proviene de un programa confiable y permitirá el uso de la ventana. Por el contrario si un programa no confiable intenta lo mismo, puede deberse a dos motivos:

  a. El programa contiene un error
  b. El programa es un caballo de Troya

  En cualquiera de los dos casos el núcleo puede negar la petición sin comprometer la compatibilidad.

Recordar que un programa es de confianza sí y sólo sí pertenece a la TCB; y que la pertenencia de un programa a la TCB se determina mediante la función `istrusted` definida en la sección 6.

La interfaz de que dispondrán los programas de confianza para utilizar la ventana de confianza debe quedar definida cuanto antes, y se la debe comunicar al equipo de coordinación. Su implementación puede hacerse algo más adelante.

3.2. Dispositivos

Sacar del núcleo toda referencia a dispositivos que no estén entre los siguientes:

— disco rígido

— disquetera

— cdrom (lectora)

— terminal (teclado y pantalla)

— impresora

— puerto serie

— puerto paralelo

— mouse

— null

— zero

Modificar la operación de i-nodo `open` para la terminal de forma tal que ponga la clase de acceso retornada por `csc` a L.

3.3. Relación entre terminales física, lógica y virtual

Revisar con sumo cuidado la relación entre terminales física, lógica y virtual. En particular, como se indica en [1], debe ser imposible que un proceso (incluso uno de `root`) cambie la relación entre una terminal física y su correspondiente terminal lógica. Ademá el modelo asume que lo que un usuario ingresa por una terminal física llega a una única terminal lógica, y que lo que un proceso escribe en una terminal lógica es enviado a la "pantalla" de una única terminal física. Por lo tanto, esta hipótesis debe ser verificada en profundidad.

3.4. Revisar y modificar IOCTL y las llamadas POSIX equivalentes

Como parte del estudio mencionado en la sección anterior, se debe analizar profundamente la llamada al sistema `ioctl` y las llamadas POSIX equivalentes (página de manual de `termios`): `tcgetattr`, `tcsetattr`, `tcsendbreak`, `tcdrain`, `tcflush`, `tcflow`, `cfmakeraw`, `cfgetospeed`, `cfgetispeed`, `cfsetispeed`, `cfsetospeed`, `tcgetpgrp` y `tcsetpgrp`. En particular se deben realizar las actividades mencionadas a continuación.

Sacar todos los comandos de todos los dispositivos que no estén en la lista anterior; es decir, eliminar todo lo necesario para que sea imposible utilizar los dispositivos que no estén en la lista anterior a través de `ioctl` o de las llamadas POSIX equivalentes.

De los que quedan, revisar y documentar cada uno de los comandos y llamadas desde el punto de vista del flujo de información entre la memoria de cada proceso y los objetos que el comando o la llamada acceden. En general debe tenerse en cuenta el modelo presentado en [1]. Allí, se establece claramente cuándo hay flujo de información. El flujo de información deben ser explícito, en el sentido de que si un proceso puede obtener información clasificada a través de las respuestas (binarias) de ciertas llamadas, tales canales no deben ser tenidos en cuenta aunque sí deben ser documentados.

Además, si es necesario, deben modificarse los comandos y las llamadas para que verifiquen el modelo. Lo que se busca es que cualquiera de estos comandos o llamadas no altere la semántica de las llamadas representadas en el modelo. Es decir, por ejemplo, si un comando permite escribir en la terminal, este debe reprogramarse de forma tal que cumpla con la especificación de $Write$; si un comando permite enviar el input ingresado por un teclado a la pantalla de otra terminal, debe cancelarse. Otro ejemplo son las terminales virtuales: todas ellas deben tener la misma clase de acceso `msc` aunque pueden tener diferentes clases `csc`; de todas formas, todas deben tomar input de un único y mismo teclado y envían output a una única y misma pantalla.

3.5. Comando `chinsc`

Este equipo debe programar el programa de confianza `chinsc` que debe responder a la especificación de la operación *Chinsc* detallada en [1]. Para poder programar este comando es necesario que el equipo de trabajo a cargo de las tareas descriptas en la sección 6 defina la interfaz de una librería que debe ser utilizada por `chinsc`.

El parámetro *sc*? mencionado en *Chinsc* estará en uno de los siguientes formatos:

— $nivel : cat_1 : ... : cat_n$ donde *nivel* está en formato LHRL y todas las cat_i están en formato CHRL (ver sección 6 para una descripción de LHRL, CHRL y SCHRL)

— *scLabel* que es una etiqueta en formato SCHRL

Aunque no está dicho en *Chinsc*, el programa `chinsc` debe mostrar, en la pantalla de la terminal desde la cual fue invocado, la nueva clase de acceso (es decir, *sc*?) una vez que ha sido establecida.

Es absolutamente necesario que el usuario esté completamente seguro que la clase de acceso que ve escrita en su pantalla fue escrita por `chinsc` y no por otro programa (no confiable). Por este motivo, el comando debe utilizar la ventana de confianza para tal fin.

4. Familia de llamadas SUID y comandos de usuario

4.1. Familia de llamadas SUID

Debe modificarse la familia de llamadas SUID de forma tal que cumplan con la especificación de la operación *Setuid* según se describe en [1]. Es decir, que *root* no puede hacer SUID a *secadm* a menos que esté ejecutando un programa confiable (es decir un programa incluido en la TCB, ver sección 6). Para saber qué programa está ejecutando un proceso se debe ... (consultar con Felipe).

Como `init`, `agetty` y `login` son parte de la TCB y los ejecuta *root*, entonces pueden hacer SUID a *secadm*.

4.2. Comandos de usuario

Programar y modificar comandos a nivel de usuario para trabajar con clases de acceso y listas de control de acceso. Para poder programar algunos de estos comandos es necesario que el equipo de trabajo a cargo de las tareas descriptas en la sección 6 defina la interfaz de una librería que debe ser utilizada por estos programas. Los programas que se deben modificar son los siguientes:

`chmod` Debe permitir modificar la ACL de cualquier archivo según lo especificado en [1]. La interfaz con el usuario debe ser idéntica a la definida para `chacl` (ver más abajo).

`chown` Debe permitir modificar los dueños de la ACL de cualquier archivo según lo especificado en [1]. La interfaz con el usuario debe ser idéntica a la definida en `chacl` (ver más abajo)

`ls` Debe tener una opción que liste la ACL de los archivos que se le pasan como parámetro, y otra que liste la clase de acceso de los archivos que se le pasan como parámetro. En este último caso debe haber tres alternativas:

- Mostrar los números internos usados por el núcleo
- Mostrar los niveles y las categorías según el contenido de LHRL y CHRML (ver sección 6)
- Mostrar el nombre de la clase de acceso según SCHRL (ver sección 6).

Los programas que se deben agregar son los siguientes:

`chacl` Debe permitir modificar la ACL de cualquier archivo según lo especificado en [1]. La interfaz con el usuario debe ser tal que permita el mismo cambio en diferentes archivos y distintos cambios en el mismo archivo y ambos a la vez. En particular debe ser posible indicarle una serie de cambios desde un archivo. El contenido de este archivo se define utilizando un lenguaje simple que permita indicar todos los tipos de cambios.

`oscstat` Debe listar la clase de acceso de los archivos que se le pasan como parámetro. Debe haber tres alternativas

- Mostrar los números internos usados por el núcleo
- Mostrar los niveles y las categorías según el contenido de LHRL y CHRML (ver sección 6)
- Mostrar el nombre de la clase de acceso según SCHRL (ver sección 6).

`sscstat` Debe listar la clase de acceso de los usuarios que se le pasan como parámetro. Debe contemplar las mismas tres alternativas que `oscstat`.

5. GTLFS, VFS y programa de instalación

En esta sección se describen los cambios que deben efectuarse sobre las estructuras de datos del VFS y de GTLFS. Estos cambios, en algunos casos implican eliminar, modificar y/o agregar código para que el resto de los módulos del núcleo puedan interactuar de manera segura con las capas inferiores del sistema de archivos. Además se enuncian los requerimientos para el programa de instalación de GTL.

5.1. GIDIS Trusted Linux File System (GTLFS)

El sistema de archivos de GTL (GTLFS) será una nueva versión EXT2 compatible con este último; en otras palabras EXT2 soportará el almacenamiento de los nuevos atributos de seguridad (clases de acceso y ACL). Esto se logra programando en EXT2 una serie de modificaciones que le indican al programa qué debe hacer ante la presencia de estos nuevos atributos. La posibilidad de hacer esto se debe a que EXT2 reserva en el super-bloque dos campos denominados *revision levels*.

Estos campos se utilizan para indicar la versión de EXT2 y actuar en consecuencia; es decir, el código hace diferentes cosas para diferentes versiones. Entre las funciones que se deben modificar para que tengan en cuenta la nueva versión de EXT2 (es decir, GTLFS) están: `ext2_update_inode`, `ext2_read_inode`, `ext2_read_super`, `etx2_remount`, etc. En la sección 5.2 se dan más detalles.

Desactivar todos los sistemas de archivos físicos que no estará permitido montar. Sólo se podrán montar: EXT2, FAT, ISO9660 (ver sección 5.3 para algunos detalles más).

Se debe modificar el programa `mkfs` de manera tal que pueda ser utilizado para trabajar con GTLFS; es decir, este programa debe ser capaz de formatear particiones para GTLFS. Al formatear una partición debe poner la clase de acceso de `/` y `lost`found+ a L. En particular debe ser posible formatear un sistema de archivos GTLFS indicando la cantidad de categorías que soporta el conjunto de categorías de las clases de acceso.

5.2. Clases de acceso a nivel de GTLFS y VFS

Cambiar la implementación de la estructura de datos que almacena el conjunto de categorías de las clases de acceso a nivel de GTLFS y VFS. Usar un mapa de bits como se sugirió durante el curso. Pedir al resto de los equipos la resolución que cada uno haya hecho y seleccionar la mejor.

En el i-nodo del VFS se agregan dos campos de tipo `sc` (en lugar de uno como se hizo en Lisex 0.0). En en el i-nodo de GTLFS (el sistema de archivos físico de GTL generado a partir de EXT2), se mantiene un único campo de tipo `sc`. Estos campos deben ser tratados como TADs de tipo `sc` (más detalles a continuación).

La razón de este cambio está en el corazón del modelo formal [1]. Allí se considera que las terminales lógicas deben clasificar la entrada provista por el usuario en cierta clase de acceso (que cambia según los indica el usuario), y que deben poseer una clase de acceso máxima para la salida enviada por los procesos. Por otro lado, el mismo documento estipula que el resto de los objetos manejados por el sistema deben tener una única clase de acceso (tal cual lo era en Lisex 0.0).

Entonces, con el fin de unificar ciertas operaciones y de hacer más elegante (y posiblemente más modificable) la implementación, decidimos trabajar de la siguiente forma:

— En el i-nodo de GTLFS se guarda una única clase de acceso, la cual se almacena en dos campos (uno para el nivel y otro para el conjunto de categorías) como se hizo en Lisex.

— El i-nodo del VFS llevará dos clases de acceso que se obtienen únicamente por medio de las funciones:

```
sc * msc(struct inode *inode)
```

```
sc * csc(struct inode *inode)
```

La relación de estas funciones con el modelo [1] es la siguiente:

— `csc` y `msc` para el i-nodo de un archivo que no sea una terminal lógica corresponden a la aplicación de *osc* sobre ese i-nodo. Es decir, ambas retornan el mismo valor.

Por lo tanto, para trabajar con archivos ordinarios puede usarse cualquiera de la dos según convenga. No es simple en este estado del desarrollo saber si en futuras versiones retornarán valores diferentes.

  - `csc` para el i-nodo de una terminal lógica corresponde a la aplicación de *ltsc* a esa terminal
  - `msc` para el i-nodo de una terminal lógica corresponde a la aplicación de *mptsc* a la terminal física asociada con esa terminal lógica.

- Consultar la clase de acceso de un objeto cualquiera (invocando *Oscstat*) debe ser equivalente a usar `msc`. Es decir, "la" clase de acceso de una terminal es la que retorna `msc`.

  (Para los archivos ordinarios podría usarse cualquiera pero para simplificar conviene usar `msc`.)

- Cuando se invoca la llamada al sistema `open`, esta debe estar programada de forma tal que ambas funciones retornen el mismo valor que a su vez es igual al almacenado en GTLFS.

  Luego, el VFS invoca a la operación de i-nodo `open` de GTLFS, la cual debe cambiar el valor que retorna `csc` sólo en el caso de que se trate de una terminal lógica. Más precisamente, como se indica en [1], el valor que retorne `csc` debe ser L.

- Para modificar el valor de las clases de acceso retornadas por `sc` y `msc` se deben utilizar las siguientes funciones:

  ```
  set_msc(struct inode *inode, sc *sc)
  ```

  ```
  set_csc(struct inode *inode, sc *sc)
  ```

  En el uso de estas funciones deben aplicarse las siguientes reglas:

  - Estas funciones pueden ser invocadas sólo si `inode` existe y `sc` es una clase de acceso válida.
  - Invocar *Chinsc* (oviamente sobre una terminal) implica invocar `set_csc`
  - Invocar *Chobjsc* sobre cualquier objeto implica invocar `set_msc`
  - Si se debe cambiar la clase de acceso de un objeto que no es una terminal lógica (por ejemplo cuando se modifica la clase de acceso de un objecto vacío) debe invocarse `set_msc` .

5.3. Interacción con otros sistemas de archivos físicos

Se debe reprogramar la llamada al sistema `mount`. Ahora sólo se podrán montar FAT, ETX2, e ISO9660 pero siempre de sólo lectura y sin dispositivos (opciones: `MS_RDONLY` y `MS_NODEV`). Se supone que estos sistemas de archivos sólo se montan desde disqueteras y CD-ROMs.

La clase de acceso de todos los archivos contenidos en el dispositivo montado debe ser igual a la retornada por `csc` para la terminal desde donde se monta. Entonces deben modificarse las operaciones de i-nodo de los sistemas de archivos físicos mencionados para que las clases de acceso verifiquen la propiedad enunciada.

5.4. Programa de instalación

El proceso de instalación descripto a continuación no es en absoluto confiable por lo cual la instalación que se haga de GTL puede no arrancar desde un estado seguro[2].

Debe generarse un CDROM buteable que contenga todos los archivos para una distribución GTL. Al butear éste debe hacer lo siguiente:

1. Pedir al usuario en qué partición debe instalarse GTL
2. Usar la versión GTL de `mkfs` para construir un GTLFS en esa partición
3. Montar el GTLFS creado en el paso anterior.

[2]En realidad, esta versión de GTL no es suficientemente segura como para plantear un proceso de instalación seguro.

4. Copiar los archivos de la distribución en el GTLFS montado

Luego sólo falta hacer algo para que se pueda arrancar desde esa partición:

a. Arreglárselas para usar un *boot loader* tradicional

b. Crear un diskette GTL-buteable que conozca la partición antes elegida

6. TCB, `agetty`, `login` y cambio de clases de acceso

6.1. *Human Readable Labels* (HRL)

Dentro de la TCB (ver sección siguiente) se deben agregar los siguientes archivos:

LHRL (por *Level Human Redeable Labels*) que asocie los números utilizados por el núcleo para los niveles de seguridad con etiquetas legibles por el usuario. Los números usados por el núcleo son los que documenta la llamada al sistema `oscstat`.

CHRL (por *Categories Human Redeable Labels*) que asocie los números utilizados por el núcleo para las categorías con etiquetas legibles por el usuario. Los números usados por el núcleo son los que documenta la llamada al sistema `oscstat`.

SCHRL (por *Security Classes Human Redeable Labels*) que asocie clases de acceso con etiquetas legibles por el usuario. En este caso las clases de acceso se describen usando LHRL y CHRL, y a cada SC se le asocia una nueva etiqueta (distinta de las usadas en los otros archivos) que identifica a la SC. De esta forma los usuarios podrán referirse rápidamente a las SC más usadas.

Además (esto es lo más importante) se debe programar un TAD (en forma de librería C de funciones) para cada uno de estos archivos que permita interactuar con ellos. Es decir debe haber funciones para consultar una categoría, para modificar una asociación entre número y etiqueta, etc. Estos TADs son software confiable, por lo que para programarlos se debe leer con atención la sección 1.3.

Como paso previo fundamental a la implementación de los TADs se deben definir y documentar sus interfaces. Es muy importante que esta tarea se haga rápida y correctamente pues hay otros equipos que dependen de estas interfaces.

6.2. Trusted Computer Base (TCB)

En esta versión de GTL, la TCB será simplemente un directorio (por ejemplo, `/etc/tcb`), lo que evidentemente no es una TCB. La TCB contiene un conjunto de archivos de datos y programas cuya integridad debe ser protegida; además, los programas son considerados confiables, es decir, se asume que no son caballos de Troya (y por tanto tienen ciertos privilegios que otros procesos de `root` no tienen).

El equipo debe definir ubicación de la TCB, es decir el directorio donde se guardarán los archivos y programas de la TCB; y programar la siguiente función (auxiliar del núcleo) que recibe un XXX e indica si pertenece o no a la TCB:

```
int istrusted(struct inode *inode)
```

que retorna 1 cuando el i-nodo apuntado por `inode` esta en la TCB y 0 en caso contrario. Si `inode` apunta a un i-nodo que es una liga a otro i-nodo, la función debe seguirla (recordar que ya hay funciones auxiliares del núcleo que hace ese trabajo).

La TCB está compuesta por los siguientes programas:

— La imagen del núcleo del sistema

— `init`

— `agetty`

— `login`

— `passwd`

— `chsubsc` (descripto más abajo)

— `chobjsc` (descripto más abajo)

— `chinsc` (descripto en la sección 3)

6.3. Persistencia clase de acceso usuarios

La persistencia de clases de acceso de usuarios se logrará incluyendo las clases de acceso de los usuarios en un archivo dentro de la TCB. Este archivo debe tener la siguiente estructura:

— Una entrada de la forma *login* : *clase de acceso* por línea

— Cada *clase de acceso* se escribirá en uno de los dos formatos

  — $nivel : cat_1 : ... : cat_n$ donde *nivel* está en formato LHRL y todas las cat_i están en formato CHRL

  — *scLabel* que es una etiqueta en formato SCHRL

Además (esto es lo más importante) se debe programar una librería C de funciones que permitan interactuar con este archivo. Esta librería es un software confiable, por lo que para programarla se debe leer con atención la sección 1.3.

6.4. `agetty` y `login`

El programa `agetty` debe ser modificado de forma tal que luego de recibir el login, constante que es un usuario del sistema (sin verificar su contraseña pues lo hace `login`) y en caso de que lo sea debe llamar a `login` poniendo el UID del nuevo proceso igual al del usuario. Por el momento la funcionalidad restante de `agetty` debe permanecer sin cambios. Además, en tanto programa confiable, debe utilizar la ventana de confianza para comunicar al usuario sus estados más significativos (ver sección 3 para más detalles).

El programa `login`, al igual que `agetty`, será parte de la TCB (es decir será un programa confiable). La modificación que debe introducirse se resume en los siguientes puntos:

— Será ejecutado por `agetty` pero su UID será el del usuario que recibió `agetty` y no 0 como en Linux.

— La contraseña de cada usuario la buscará en un archivo personal del usuario (y no en `/etc/passwd` ni en `/etc/shadow`). Estos archivos deberían estar dentro de la TCB y cada uno de ellos tendrá la clase de acceso del usuario al que corresponde. `login` debe verificar esta condición, caso contrario no debe permitir el ingreso del usuario. Si el archivo de un usuario no existe o está vacío el usuario no podrá ingresar al sistema[3].

[3]Tener en cuenta que aun con estas medidas la seguridad del sistema no es muy grande pues un caballo de Troya podría cambiar la contraseña de cualquier usuario (lo mismo ocurriría de mantener el esquema tradicional). De todas formas para remediar esto se debería tener una TCB verdadera lo que está fuera del alcance de esta versión de GTL.

— Luego de autenticar al usuario deberá lanzar el shell utilizando la llamada `execve` la cual (siendo `login` un programa confiable) pondrá la clase de acceso del shell a L. Claramente, este programa no debe (ni puede) cambiar el UID del shell por lo dicho en el punto 1.

— Debe utilizar la ventana de confianza para comunicar al usuario sus estados más significativos (ver sección 3 para más detalles). De esta forma se evita que un usuario de bajo nivel deje abierta su terminal, pero de forma tal que parece estar lista para recibir el login de un usuario, con el objetivo de capturar la contraseña de otros usuarios. Como el programa que deja corriendo el usuario de bajo nivel no puede utilizar la ventana de confianza, otros usuarios podrán notar que no se están logueando ante el programa oficial.

6.5. Cambio de clases de acceso de usuarios y objetos

A nivel de usuario hay que programar dos programas (confiables) que permitan cambiar la clase de acceso de usuarios y objetos. Los nombres de los programas deben ser `chsubsc` para los usuarios y `chobjsc` para los objetos. Estos programas serán el shell de los administradores MAC.

Los programas deben ser sumamente sencillos pues se espera que sean usados en muy raras ocasiones (tener en cuenta que a `chsubsc` sólo lo pueden ejecutar los administradores MAC, y que a `chobjsc` lo pueden ejecutar los usuarios ordinarios pero sólo sobre objetos vacío y los administradores MAC sobre cualquier objeto). `chsubsc` debe utilizar la librería descripta en la sección 6.3, y `chobjsc` debe usar la llamada al sistema del mismo nombre descripta más abajo.

En particular el formato de la entrada que reciban debe ser mínimo y cualquier error en la entrada debe abortar el programa. Los programas deben recibir de a un par (uid, sc) o (obj, sc) (según sea `chsubsc` o `chobjsc`, respectivamente), es decir no hace falta que trabajen sobre expresiones regulares, listas tomadas de archivos, etc. Cuanta menos funcionalidad tengan, mejor. La interfaz con el usuario también debe ser mínima: un menú de una o dos opciones es suficiente. Tomar como muestra los programas implementados en Lisex 0.0.

Estos programas deben utilizar la ventana de confianza para comunicar al usuario sus estados más significativos.

A nivel del núcleo se debe programar la llamada al sistema `chobjsc` la cual está especificada en [1] con la operación $Chobjsc$.

Referencias

[1] Cristiá, M. *System and security model of GIDIS Trusted Linux 0.1 - Z version.* GIDIS, www.fceia.unr.edu.ar/gidis, July 2003.

[2] Gasser, M. *Building a Secure Computer System.* Van Nostrand Reinhold, 1988.