1. Procesos y Threads

1.1. Proceso

El modelo de procesos consiste en que, todo el software ejecutable de una computadora, a veces incluso el sistema operativo, está organizado en un número de procesos secuenciales. Un proceso es una instancia de un programa en ejecución, incluyendo los valores del program counter, registros y variables. Conceptualmente, cada proceso tiene su propia CPU virtual, aunque en realidad, la CPU cambia de un proceso a otro. El cambio rápido de un proceso a otro se llama multiprogramación.

1.1.1. System calls para manejar procesos

La system call *fork* es la única manera de crear un proceso nuevo en POSIX. Crea un duplicado del proceso original, incluyendo todos los file descriptors y registros. La llamada a *fork* devuelve un valor, que es 0 en el hijo y el PID del nuevo proceso en el padre. La llamada *exec* reemplaza la imagen por el archivo pasado en el primer parámetro, es decir, se ejecuta el archivo especificado con los parámetros pasados. Por ejemplo:

```
#define TRUE 1

while (TRUE) {

        show_message();
        read_command(command, parameters);

        if ( fork() != 0 ) {
        /* Parent code */
                waitpid(-1, &status, 0);
        } else {
        /* Child code */
                execve(command, parameters, 0);
        }

}
```

Estado del PCB (Process control block): En el padre, al realizarse el *fork* el proceso se bloquea, esperando a que termine el proceso hijo. En el hijo, por el otro lado, el proceso está corriendo. Por otro lado, los process id del padre y del hijo, como se explicó antes, son distintos.

Estado de U AREA (user area): el user area, en estos casos es distinto ya que, el programa que realiza el *fork* es distinto a aquel llamado mediante *execve*. Al ser procesos distintos, el puntero a la tabla de procesos en la U AREA es distinto.

1.2. Thread

En los sitemas operativos tradicionales, cada proceso tiene un espacio de direcciones y un thread bajo su control. Sin embargo, hay muchas situaciones en las que se desea tener múltiples threads bajo el control de un mismo proceso, en el mismo espacio de direcciones, corriendo casi en paralelo como si fueran procesos distintos (con la diferencia de que comparten el espacio de direcciones). Las razones por las que se querría utilizar threads son muchas. La más sencilla puede ser porque hay muchas actividades ejecutandose en el proceso, muchas de las cuales pueden bloquear por acceso a disco u otros motivos. De esta manera, mientras un thread espera una respuesta desde un dispositivo I/O, otro thread del mismo proceso puede ser ejecutado.

1.2.1. Modelo clásico de threads

Una manera de ver a un proceso es la de agrupar recursos relacionados. Un proceso tiene un espacio de direcciones que contienen el texto del programa y datos, junto con otros recursos. Los recursos pueden ser archivos abiertos, procesos hijos, información de cuentas y otras cosas. Otro concepto que los procesos trabajan son el de thread de ejecución. Lo que los threads agregan al modelo de procesos es la posibilidad de tener

múltiples ejecuciones en el ambiente de un mismo proceso. El término multithreading se refiere a la situación en la que se tienen múltiples threads dentro de un mismo proceso. Ejemplo de threads en C:

```
#include <pthread.h>

void* print_function( void *ptr );

main() {
     pthread_t thread1, thread2;
     char* message1 = ``Thread 1'';
     char* message2 = ``Thread 2'';
     int  iret1, iret2;

    /* Se crean threads, ambos con referencia a algun string con mensaje */
    /* Se les pasa una funcion definida dentro del mismo programa, no
     * reemplazan esta imagen por la de otro programa al ejecutar
     * (execve en procesos si lo hace) */
     iret1 = pthread_create( &thread1, NULL, print_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_function, (void*) message2);

    /* Se espera que terminen de ejecutar la funcion */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);
}
```

1.2.2. Implementación de threads en espacio de usuario

Una implementación del modelo de threads es la de poner dicho paquete íntegramente en el espacio de usuario, con el kernel no sabiendo nada acerca de ellos. El kernel solo sabe que esta manejando procesos ordinarios con un solo thread. La primer ventaja de esto es que threads a nivel usuario pueden ser implementados en un sistema operativo que no soporta threads. Con esta aproximación, los threads son implementados a modo de biblioteca.

Con esta implementación, cada proceso necesita su tabla de threads privada para poder monitorear a los threads de ese proceso. Esta tabla es análoga a aquella de los procesos a nivel kernel, solo que se mantiene información acerca de cada thread individual, como puede ser el program counter de cada thread, stack pointer, registros, estados y demás. Al implementar threads a nivel de usuario, se puede hacer switching de un thread a otro sin la necesidad de hacer un trap a nivel kernel. Además, de esta manera, cada proceso puede tener su propio algoritmo de scheduling para los threads.

A pesar de que los threads tienen mejor rendimiento, tienen ciertos problemas, como puede ser la ejecución de system calls que son bloqueantes. Seria inaceptable dejar que el thread realice la system call, ya que bloquearía al proceso, afectando a otros threads. Una alternativa sería cambiar las system calls para que sean no bloqueantes, pero eso implicaría un cambio en el sistema operativo. Otra alternativa más viable sería realizar un wrapper para las llamadas bloqueantes, que solo se realizarían cuando estan son seguras (no van a bloquear). Otra desventaja es que, si un thread comienza, ningún otro thread va a correr en CPU hasta que este ceda voluntariamente la misma.

El mayor argumento contra esta implementación es que generalmente se quiere hacer una aplicación con multithreading en donde hay muchas system calls. Una vez que ha llegado una trap al kernel no sería mucho trabajo para este switchear para que otro thread ejecute, y dejando que el kernel haga esto hace innecesario el uso de wrappers de llamadas que pueden llegar a bloquear.

1.2.3. Implementación de threads en kernel

Bajo esta implementación no hay tabla de threads en cada proceso. En cambio, el kernel tiene una tabla de threads que mantiene información de todos los threads del sistema. Cuando un thread intenta crear o destruir otro thread, se realiza la llamada al kernel, que actualiza la tabla de threads. Cuando un thread bloquea, el kernel puede correr otro thread, tanto del mismo proceso como de otro. A diferencia de thread en espacio de usuario, el kernel le puede sacar tiempo de CPU para otorgarselo a otro thread.

Debido al alto costo que implica crear y destruir threads, se recurre al reciclado de los mismos. Los threads a nivel kernel no requieren system calls no bloqueantes. Además, cuando un thread causa una page fault, el kernel puede chequear si el proceso tiene otro thread y correr ese mientras espera que la página requerida sea traída desde el disco.

Esta implementación traer varios problemas. Por ejemplo, que pasa cuando un proceso con multithreading realiza un *fork*? El nuevo proceso tiene la misma cantidad de threads que el anterior? Depende de lo que se vaya a hacer con este nuevo proceso cada una de las opciones será válida. Si se va a llamar a *exec* para llamar a otra aplicación sería lógico que el nuevo proceso tenga un solo thread. Si se va a continuar la ejecución del programa que originó el proceso sería correcto replicar los threads.

Otro problema de los threads a nivel kernel es con respecto a las señales. Las señales son enviadas a los procesos, no a los threads, al menos en el modelo clásico. Si viene una señal a un proceso, qué thread debería manejarla? Posiblemente cada thread podría indicar que señales maneja, pero nuevamente, tendríamos un problema si varios threads manejan una misma señal.

2. Procesos en Android

2.1. Arquitectura Android

La arquitectura del sistema operativo está dividida en cuatro capas:

- Aplicaciones
- Frameworks
- Bibliotecas y Android Run Time (Dalvik)
- Kernel Linux

2.2. Aplicaciones Android

Una aplicación Android tiene cuatro componentes que se activan con un mensaje llamado "Intent":

- Activities
- Services
- Content providers
- Broadcast receivers

Los componentes están descriptos en un archivo llamado "AndroidManifest.xml". El uso de recursos a los que el sistema les da un resource ID, los mecanismos para compartir datos entre distintas aplicaciones y la previsión para notificaciones asincrónicas se define aquí.

2.3. Activity

Una Activity es una aplicación que se comunica por medio de una pantalla con el usuario. Generalmente esta comunicación se realiza full-screen, pero puede ser una pantalla flotante. Una aplicación esta compuesta por una o más activities, pero solo una puede estar activa a la vez. El resto de las activities se guarda en un stack (que puede ser recorrido con la tecla back).

Cada aplicación corre en su propio proceso con su propia copia de Dalvik. Los procesos son provistos por el kernel y manejados por el Android Run Time. Para mantener la respuesta del sistema, el sistema Android puede matar sin aviso a procesos que considera que no están respondiendo (y las aplicaciones contenidas dentro del mismo).

La prioridad de una Activity se determina mediante su proceso. El programador extiende de la clase Activity y programa los eventos a los que responde. Prioridades:

1. Active Process
2. Visible Process
3. Started Service Process
4. Background Process
5. Empty Process

2.4. Ciclo de vida de una Activity

- Active: está al tope del stack e interactuando con el usuario.
- Paused: visible pero sin foco.
- Stopped: queda en memoria pero ya terminó. Candidata al kill.
- Inactive: fuera de la memoria. Debe lanzarse nuevamente.

3. Manejo de memoria

3.1. Espacio de direcciones

Si los programas del usuario pueden acceder cualquier byte de la memoria, fácilmente estos pueden interferir con el trabajo del sistema operativo. Con este modelo, además, es complicado tener más de un programa siendo ejecutado a la vez. El espacio de direcciones es una abstracción que representa un conjunto de direcciones que un proceso puede utilizar para direccionar memoria.

La solución más simple para implementar dicho espacio es la del uso de registros base y límite. Mediante este método, cada vez que se hace una referencia a memoria, el hardware de la CPU le suma el registro base y chequea que la dirección obtenida sea menor que el valor límite.

3.2. Swapping

La estrategia que se emplea consiste en ubicar cada proceso íntegramente en memoria, dejar que este corra por un determinado tiempo y luego volver a ubicarlo en el disco. Los procesos que estan inactivos se encuentran guardados en disco, de manera de no consumir memoria mientras no estén ejecutándose. Cuando esta estrategia crea múltiples agujeros en la memoria, es posible combinarlos para generar un espacio de memoria mayor. Esta técnica se conoce como compactación de memoria.

3.3. Memoria virtual

Es necesario tener algún mecanismo para correr programas que son tan grandes que no caben en memoria. El método que se diseño para atacar este problema es el de la memoria virtual. La idea básica es que cada programa tiene su propio espacio de direcciones, que está dividido en pedazos llamados *páginas*. Cada página es un rango contínuo de direcciones. Estas páginas son mapeadas en memoria física, pero no todas las páginas deben estar en memoria para correr el programa. Cuando se hace referencia a una parte del espacio de direcciones que no esta en memoria se genera un *page fault*, se alerta al sistema operativo para que vaya a buscar la página faltante y se vuelve a ejecutar la instrucción que generó la falla.

3.3.1. Tabla de páginas

En una implementación simple, el mapeo de una dirección de memoria virtual puede ser resumida de la siguiente manera: la dirección virtual se separa en un número de página (bits de alto orden) y un offset (bits de bajo orden). El número de página virtual se usa como índice en la tabla de páginas para encontrar la entrada de esa página virtual. De la entrada de la tabla se obtiene el *marco* de la página. Luego, con la dirección del marco y el offset de la página se obtiene una dirección de memoria física.

3.3.1.1. Tabla de páginas multinivel El secreto detrás de las tablas multinivel es el de evitar tener todas las tablas de páginas en memoria todo el tiempo. En particular, aquellas que no se necesitan no deben estar alrededor. Lo que se hace es tener en memoria las tablas que están siendo utilizadas y se deja en disco aquellas que no están siendo referenciadas. Para agilizar la búsqueda de las tablas se utiliza lo que se conoce como Translation Lookaside Buffer, que acelera el proceso de traducción de direcciones virtuales a direcciones físicas.

Como una primera aproximación podríamos separar una dirección virtual de 32 bits en un campo de tabla 1 de 10 bits, un campo de tabla 2 de 10 bits y un offset de 12 bits. Como el offset es de 12 bits, las páginas son de 4 KB y tenemos 2^{20} páginas.

Por ejemplo:

VA = 0x00403004 = $\underbrace{0000000001}_{PT1=1}\underbrace{0000000011}_{PT2=3}\underbrace{000000000100}_{Offset=4}$

3.3.1.2. Tabla de página invertida Para direcciones virtuales de 32 bits, las tablas multinivel funcionan razonablemente bien. Sin embargo, para 64 bits, la situación cambia drásticamente. Si el espacio de direcciones es de 2^{64}, con páginas de 4 KB necesitamos una tabla con 2^{52} entradas. Si cada entrada es de 8 bytes, tenemos una tabla con más de 30 millones de gigabytes. La solución a esto es utilizar una tabla de páginas invertida. En este diseño, hay una entrada por marco en memoria real.

El problema de este método, aunque ocupa mucho menos en memoria, es que el mapeo de una dirección virtual a una real es más complicado. No existe una relación directa entre la dirección virtual y la ubicación del

marco de página. De esta manera se debe buscar en la totalidad de la tabla para encontrar el marco adecuado. Esto debe ser realizado por cada referencia a memoria, no sólo cuando ocurre una *page fault*. Para facilitar esta busqueda se suele recurrir a una tabla de hash que maneja colisiones encadenando dichas páginas virtuales con su frame.

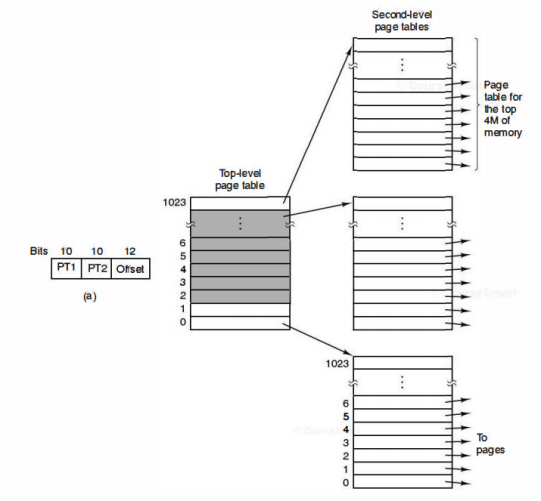

Figura 1: Tabla de página multinivel.

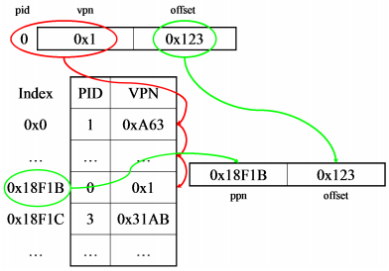

Figura 2: Tabla de página invertida.

4. Linkers y Loaders

4.1. Traducción

La traducción es el proceso mediante el cual el código fuente de un programa es convertido a un programa objeto. Si el lenguaje es un assembler, la traducción es un ensamblado hecho por un programa ensamblador. Este programa convierte código memotécnico a códigos de operación y resuelve identificadores a posiciones de memoria. Si se trata de un lenguaje de alto nivel, la traducción es una compilación.

4.2. Link-editor

La tarea que realiza el link-editor es la de mezclar las direcciones de cada módulo (programa objecto) en un único espacio de direcciones. La salida del link-editor, además de ser un ejecutable puede ser otro programa objeto o una biblioteca. La entrada también puede ser un ejecutable o una biblioteca estática.

4.3. Loader

La tarea que realiza el loader es la de pedir la memoria en la que se ubicará el programa, mapear las direcciones del archivo en direcciones de memoria y resolver los nombres de las bibliotecas dinámicas. Ejemplo dinámico:

```
main                                          /* Ejecucion principal */
....
lib_handle = dlopen(lib_name,RTLD_LAZY);      /* Se abre la biblioteca dinamica */
....
lib_func = dlsym(lib_handle, ''pedida'');     /* Se obtiene la funcion pedida */
.....
(*lib_func)();                                /* Se ejecuta la funcion adecuada */
```

4.4. Object file formats

Es clave para la performance del sistema. Algunos proveen su interacción con el paginado. Se puede utilizar el mismo formato para ejecutables, objetos y bibliotecas. Que dos sistemas operativos tengan el mismo OFF no significa que los programas de uno puedan correr en el otro. Típicamente, los formatos soportan el uso de headers, segmento de texto, segmento de datos, información de relocación, definiciones externas, referencias para el linking e información de linking dinámico.

4.4.1. DOS com

- Se hacen con *exe2bin*
- Se cargan en una dirección fija de memoria
- Datos y código están en el mismo segmento
- Tienen un tamaño máximo de bytes
- Se puede decir que es un "null file format"

4.4.2. DOS exe

- Aparecen en DOS 2.0
- Tiene previsión para relocación de memoria

4.4.3. Common object file format (COFF)

- Se compone de varias secciones separadas por headers
- Se usa para bibliotecas de enlace estático
- Soporta debug de C

4.4.4. Windows portable executable

- Es una adaptación del COFF
- Tiene definido espacio para resources
- Tiene definido tablas para el uso de bibliotecas compartidas
- Hay herramientas de análisis

5. Virtualización

5.1. Condiciones de Popek y Goldberg

- Equivalencia/fidelidad: Un programa corriendo bajo un Virtual Machine Monitor (VMM) debe exhibir el mismo comportamiento que aquel que se obtiene corriendolo directemente en una máquina.
- Control de recursos/seguridad: El VMM debe tener completo control de los recursos virtualizados.
- Eficiencia/performance: Una gran fracción de las instrucciones de máquina deben ser ejecutadas sin intervención del VMM.

5.2. Paravirtualización

La paravirtualización es una técnica de virtualización que presenta una interfaz de software a las máquinas virtuales que es similar pero no idéntica a aquella del software subyacente. La intención de modificar esta interfaz es la de reducir aquellas ejecuciones que son complicadas en ambientes virtualizados. Una paravirtualización efectiva permite al VMM ser más simple.

5.3. Binary translation

La traducción binaria es la emulación de un set de instrucciones por otro mediante la traducción del código. Mediante este método, secuencias de instrucciones son traducidas de la fuente al set de instrucciones destino.

5.4. Hipervisores

Un hipervisor, o "virtual machine monitor" (VMM), es un elemento de software, firmware o hardware que permite crear y correr máquinas virtuales. Una computadora en la que un hipervisor está corriendo una o más máquinas virtuales se denomina máquina anfitrión. Cada máquina virtual es una máquina huesped. El hipervisor le presenta al sistema operativo huesped una plataforma virtual operativa y maneja la ejecución de los sistemas operativos huéspedes.

5.4.1. Tipos de hipervisor

- Tipo 1: El hipervisor corre directamente en el hardware del anfitrión para controlar dicho hardware y administrar los sistemas operativos huéspedes. Los sistemas operativos huéspedes corren en un nivel superior a aquel del hipervisor.
- Tipo 2: El hipervisor corre dentro de un sistema operativo convencional. Con la capa del hipervisor como una capa de software secundaria, el sistema operativo huesped corre como una tercer capa por encima del hardware.

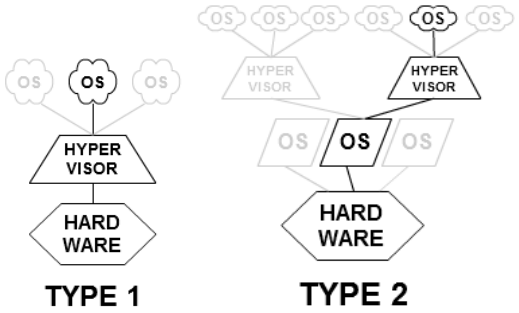

Figura 3: Hipervisores tipo 1 y 2.

6. File Systems

El file system es el método que se utiliza para guardar y organizar archivos y datos dentro de una computadora. El file system provee acceso y almacenamiento a datos y programas. Los dos aspectos fundamentales de un file system son su interfaz de usuario y su implementación. En la interfaz de usuario se exporta la noción de directorios y archivos.

6.1. Files

Son una colección de datos con nombre, una unidad lógica de almacenamiento que abstrae las propiedades físicas del dispositivo de almacenamiento. Los archivos proveen persistencia ante los reinicios, activaciones de programas y fallas de energía.

6.2. Archivos mapeados a memoria

Son un segmento de memoria virtual al que se le asignó una correlación directa byte-a-byte con alguna porción de un archivo o recurso similar a un archivo. Este recurso es, típicamente, un archivo que está físicamente en el disco, pero también puede ser un dispositivo, objeto de memoria compartida u otro recurso que el sistema operativo puede referenciar mediante un file descriptor. Una vez en memoria los archivos mapeados se ven como parte de la memoria. Ejemplo de *map* y *unmap*:

```
#include <sys/mman.h>

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

/* Unmap */
int munmap(void *start, size_t length);

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

    addr = mmap(NULL, length + offset - pa_offset, PROT_READ, MAP_PRIVATE, fd, pa_offset);

    if (addr == MAP_FAILED)
        handle_error(``mmap'');

    s = write(STDOUT_FILENO, addr + offset - pa_offset, length);

    if (s != length) {

        if (s == -1)
            handle_error(``write'');

        fprintf(stderr, ``partial write'');
        exit(EXIT_FAILURE);
    }
}
```

7. Clustered File Systems

7.1. RAID

RAID es una tecnología de almacenamiento que combina múltiples discos en una sola unidad lógica. Los datos son distruibuidos a traves de los discos en una de las varias maneras, denominadas "niveles RAID", dependiendo de la redundancia y performance requerida.

7.2. Disk mirroring

Disk mirroring es la replicación de volúmenes de discos en discos duros separados para asegurar la disponibilidad contínua. La replicación se realiza a través de microcodigo en el controlador del disco o mediante software de servidor. El mirroring, además de asegurar la disponibilidad, sirve como copia de seguridad de los datos ante alguna falla de hardware. Típicamente es una solución propietaria, no compatible entre vendedores diferentes.

7.3. Data striping

Se denomina data striping a la técnica de segmentar datos secuencialmente lógicos, como un archivo, de manera que segmentos consecutivos son almacenados en dispositivos de almacenamiento distintos. Este mecanismo es útil cuando un proceso que accede un dispositivo necesita acceso a datos de manera más veloz de la que este dispositivo puede proveer. Separando los datos en discos distintos, se puede acceder a los datos de manera concurrente.

7.4. RAID parity

Muchos niveles de RAID emplean un sistema protección contra errores llamado "paridad". El bit de paridad, o check bit, es un bit al final de string que indica si la cantidad de unos es par o impar. En la tecnología RAID se utiliza como método de redundancia. Si un disco falla, los datos restantes en otros discos pueden ser combinados con los datos de paridad para reconstruir los datos faltantes. La reconstrucción de datos a través del bit de paridad se puede obtener utilizando la operación XOR. Por ejemplo:

Drive 1: 01101101 Drive 2: 11010100

Datos de paridad:

```
  01101101
XOR 11010100
  ----------
  10111001
```

Los datos de paridad se almacenan en Drive 3. Si alguno de estos discos falla, los datos pueden ser reconstruidos y guardados en un disco de repuesto. Por ejemplo, si falla el Drive 2, se pueden obtener sus datos haciendo:

```
  01101101
XOR 10111001
  ----------
  11010100
```

Drive 2: 11010100

7.5. Implementaciones RAID

La tecnologia RAID puede ser implementada tanto en software como en hardware. En los "Software RAIDs" el procesador debe usar su tiempo para las operaciones RAID. Estas operaciones se realizan en una capa entre el File System y el Device Driver. En los "Hardware RAIDs" se requiere un controlador dedicado por parte de los discos. El controlador debe tener un back end hacia los discos y un front end hacia el host (usando un host adapter).

7.6. Fake RAID

El "Fake RAID" es un controlador de firmware que toma las funciones de RAID durante el boot. Una vez que el kernel de un sistema operativo está cargado, el control pasa al sistema operativo. Esta solución se debe a que Windows no puede bootear desde software RAID.

7.7. Tipo de almacenamiento

7.7.1. NAS

Network-attached storage es un tipo de almacenamiento de datos en el que una red de computadoras proveen el acceso a los datos a un grupo heterogéneo de clientes. NAS provee tanto almacenamiento como un file system. NAS aparece ante un sistema operativo cliente como un servidor. Los servicios que provee el NAS son todos orientados al almacentamiento y, aunque es técnicamente posible que corran otro tipo de software, no están diseñados para funcionar como un servidor multi-propósito. Por esta razón, no es necesario que el un dispositivo NAS tenga un sistema operativo "full-featured".

7.7.2. SAN

Storage area network es una red dedicada que provee acceso a datos por bloques. Se los usa principalmente para generar dispositivos de almacenamiento que son percibidos como locales por un sistema operativo. SAN no provee la abstracción de archivos, solo operaciones en bloques. De esta manera, el sistema operativo ve al SAN como un disco que puede ser formateado con un file system y montado al sistema operativo.