Resumen 2do parcial Sistemas Operativos

Apunte - Implementación de Sistemas de archivos

Concepto de archivo:
Un archivo es una unidad lógica de almacenamiento utilizada por el SO para abstraer las diferencias físicas entre dispositivos de almacenamiento (como discos duros, cintas magnéticas, etc)

- Es una colección con nombres con datos relacionados que se guarda en almacenamiento secundario (persiste entre reinicios).
- Desde el punto de vista del usuario, los datos sólo pueden escribirse en almacenamiento si están dentro de un archivo.
- Pueden contener programas o datos.
- Pueden tener formato libre o estructura fija.

Clave: es una secuencia de datos (bits, bytes, líneas o registros) cuyo significado lo determina quien lo crea o lo usa.

Conceptos claves sobre archivos y sistemas de archivos

- Un archivo es una secuencia de registros lógicos (bytes, líneas, estructuras) definidos por el SO. El SO puede manejar distintos tipos de registros o dejar eso a la aplicación.
- El SO se encarga de mapear los archivos lógicos a los dispositivos físicos como discos.
- Los directorios ayudan a organizar los archivos.
- La estructura del árbol permite subdirectorios, compartición y mayor flexibilidad, aunque complica la administración.

Operaciones del Sistema de Archivos
Estructuras involucradas:
El sistema de archivos usa estructura tanto en disco como en memoria.
En disco:

- Boot control block: indica cómo arrancar el SO
- Volume control block: contiene datos del volumen como cant de bloques, bloques libres, etc. (superblock en Unix)
- Estructura de directorios: org los archivos y sus referencias (inodes o registros)
- FCB: (File Control Block): contiene los metadatos del archivo

En memoria:

- Tabla de montajes: mantiene información sobre los volúmenes montados.
- Caché de directorios: guarda directorios usados recientemente
- Tabla global de archivos abiertos: contiene los FCBs de archivos abiertos
- Buffers: contiene bloques de archivos que están siendo leídos o escritos.

Creación y uso de archivos:
Para CREAR un archivo:

1. El sistema lógico de archivos crea un FCB (contiene permisos, fechas, dueño, grupo, tamaño, bloques o punteros a los bloques)
2. Actualiza el directorio con el nombre y la referencia al FCB
3. Guarda esta información en disco

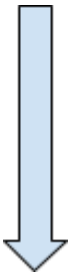
Estructura del Sistema de Archivo
Discos y NUM: los discos son el principal almacenamiento secundario porque permiten:

1. Reescribir bloques en el mismo lugar
2. Acceder directamente a cualquier bloque

El almacenamiento se maneja por bloques (de 512 o 4096 bytes). Esto mejora la eficiencia de las operaciones E/S.

Capas del Sistema de Archivos
Facilita la reutilización del código entre sistemas de archivos pero puede reducir el rendimiento por sobrecarga del SO

1. Control de E/S:
    - Controladores de dispositivos y manejadores de interrupciones
    - Traducen comandos del sistema a instrucciones de hardware
2. Sistema de Archivo Básico:
    - Se encarga de leer / escribir bloques en discos
    - Usa buffers y cachés para optimizar el rendimiento
3. Módulo de organización de archivos:
    - Administra bloques lógicos y el espacio libre
    - Asigna bloques a los archivos
4. Sistemas de Archivos Lógicos:
    - Administra los metadatos
    - Usa estructuras como FCB o inodes
    - Gestiona directorios y protección



programas de aplicación
sistema de archivo lógico
Módulo de organización de archivos
Sistema de archivos básico
control de E/S
dispositivos

Para ABRIR un archivo:

1. Se verifica si está abierto (evitar duplicar datos en memoria)
2. Si no lo está, se busca en el directorio y se copia su FCB a la tabla global
3. Añade una entrada en la tabla del proceso con modo de acceso y posición actual
4. Se devuelve un descriptor de archivo

Para CREAR un archivo:

1. Se elimina la entrada en la tabla del proceso
2. Se reduce el contador de la tabla global
3. Si ningún proceso lo usa, se escriban cambios en disco y se libera su entrada

Implementación de directorios:
Lista lineal
Es la forma más simple de implementar un directorio, una lista con nombres de archivos y punteros a sus datos.

- Ventajas: Fácil de programar
- Desventajas: Búsquedas lentas
- Agregar o eliminar puede requerir reordenar entradas o usar una lista de espacios libres
- Se puede usar una lista ordenada, pero complica inserciones/eliminaciones

Tabla Hash
Combina una lista lineal con una tabla hash para ubicar archivos rápidamente
Ventaja principal: reduce mucho el tiempo de búsqueda
Problemas: Tamaño fijo, colisiones.

Métodos de asignación

- Asignación Contigua: Es una técnica en la que todo el archivo se guarda en bloques contiguos del disco.
    - Ventajas: Acceso rápido secuencial y directo, fácil de implementar
    - Desventajas:
        - Fragmentación externa: al eliminar archivos, el espacio libre queda dividido en “huecos” pequeños no reutilizables fácilmente.
        - Difícil estimar el tamaño del archivo al crearlo
        - Wasted space: para evitar errores, el usuario puede sobreestimar el tamaño, lo que genera fragmentación interna (espacio sin usar dentro del archivo)
    - Compactación de disco (para resolver fragmentación): es lenta y requiere que el sistema esté offline en algunos casos.
- Asignación Enlazada: Cada archivo es una lista enlazada de bloques dispersos en el disco. Cada bloque apunta al siguiente
    - Ventajas: Sín fragmentación externa
        - Tamaño dinámico
        - Sencillo de expandir
    - Desventajas:
        - Acceso sólo secuencial: para llegar al bloque i, se debe recorrer desde el inicio
        - Menor eficiencia en acceso aleatorio
        - Espacio extra
        - Fragilidad
    - Variante: FAT (File Allocation Table)

        - Tabla en inicio del volumen con un índice por bloque
        - Actúa como una lista enlazada en memoria
        - Permite acceso aleatorio eficiente si se cachea la FAT
- Asignación Indexada: Es una técnica donde todos los punteros a bloques se almacenan juntos en un bloque de índice. Este bloque contiene las direcciones de cada bloque en el archivo, lo que permite acceso directo y elimina la fragmentación externa.
    - Ventajas:
        - Permite el acceso directo eficiente a cualquier bloque de archivos
        - No hay fragmentación externa
        - Un archivo puede usar cualquier bloque libre del disco
    - Desventajas:
        - Se desperdicia memoria si el archivo es muy pequeño
        - Los bloques de datos pueden estar dispersos en el disco lo que afecta el rendimiento
        - Requiere estructuras adicionales para archivos grandes

Estructura de las funciones:

*int n_written = write (int fd, char * buf, int n);*
*int n_read = read (int fd, char * buf, int n);* → *devuelve 0 si llega al fin del archivo*

fd → Descriptor del archivo
buf → buffer de memoria para almacenar/leer datos
n → retorna el número de bytes a transferir
Retornan el número de bytes transferidos o -1 error

Comportamiento
- En lectura, puede devolver menos bytes de los solicitados
- En escritura, debe coincidir con los bytes solicitados o indica error

Funciones clave:
1. open() → Abrir archivos existentes

    #include <fcntl.h>
    *int fd = open (char * name, int flags, int permisos);*

    name → string con la ruta del archivo
    flags → modo de apertura (O_RDONLY, O_WRONLY, O_RDWR)

2. creat() → Crear nuevos archivos

    *int fd = creat (char * nombre, int permisos);*
    (crea un nuevo archivo trunca el existente)

3. close() → Cerrar archivos: close (fd)

   Importante: libera el descriptor para reutilizar (límite aprox 20 archivos abiertos)

4. unlink() → Eliminar archivos

   *unlink (char * nombre);* → Equivalente a remove en stdio

5. Ejemplo de copiar un archivo a otro: char buf [BUFSIZ]

   while (n = read (fd, buf, BUFSIZ)) > 0)
   if (write (fd, buf, n) == -1)
   error..

Apunte: Interfaz C con el sistema de archivos UNIX/Linux

File Descriptors
- Todo en Unix (incluyendo dispositivos como teclado y pantalla) se tratan como archivos.
- Una única interfaz maneja comunicación con periféricos

Apertura de archivos
- Antes de leer y escribir se debe abrir un archivo
- El sistema verifica existencia y permisos
  Si es exitoso, devuelve un descriptor de archivo (entero no negativo)

Descriptor de archivo
- Identificador único para acceder al archivo abierto.
- Reemplaza el nombre del archivo en todas las operaciones.
- El sistema gestiona toda la información del archivo; el programa sólo usa el descriptor.
- Por defecto todo programa que se ejecuta desde el shell tiene abierto tres descriptores:
  - 0: entrada estándar (stdin)
  - 1: salida estándar (stdout)
  - 2: error estándar (stderr)

Funciones Básicas:
- read() y write() son las llamadas al sistema para E/S de bajo nivel
- Ambas usan file descriptors en lugar de punteros

Acceso aleatorio con lseek

lseek → permite moverse a posiciones arbitrarias en archivo (acceso aleatorio) en lugar de sólo lectura y escritura secuencial

long lseek (int fd, long offset, int origin) → da la nueva posición del archivo o -1 (error)

- fd: descriptor del archivo
- offset: desplazamiento (bytes)
- origin: punto de referencia
    - SEEK_SET → desde el inicio de archivo
    - SEEK_CUR → desde la posición actual
    - SEEK_END → desde el final del archivo

Ejemplo: función get para lectura aleatoria

```
#include “syscalls.h”

/*get: lee n bytes desde la posición pos */
int get (int fd, long pos, char * buf, int n) {
  if (lseek(fd, pos, 0) > = 0)   /*Mueve a la posicion pos */
    return read (fd, buf, n);
  else
    return -1;
```

Explicación:

1. Primero posiciona el puntero del archivo usando lseek
2. si tiene éxito, realiza la segunda
3. retorna el número de bytes leídos o -1 si hubo error

Listados Directorios
Estructura de Sistemas de Archivos Unix

- Un directorio es un archivo que contiene nombres de archivos y sus inodos (índices a una tabla que almacena metadatos del archivo)
- El formato de los directorios varía entre sistemas, por lo que se usa una capa de abstracción para manejar esta portabilidad (Direct, DIR)

Implementación de un Asignador de Memoria (malloc y free)

MALLOC

- Calcula las unidades necesarias (redondeado)
- Busca en la lista de bloques libres (estrategia “first fit”)
- Si encuentra el bloque exacto lo remueve de la lista
- Si encuentra un bloque mayor lo divide
- Si no hay memoria llama a more core

MORE CORE : Solicita más memoria al SO

- Usa sbrk() para pedir memoria al SO
- Pide bloques grandes para minimizar llamadas al sistema
- La memoria nueva se integra a la lista libre mediante free()

FREE: Libera memoria
- Inserta un bloque liberado en una lista ordenada
- Fusiona bloques adyacentes para evitar fragmentación
- Mantiene free apuntando al último bloque examinado (apt para búsquedas futuras)

Apunte: Segmentos de Memoria de un proceso - malloc()

Conceptos fundamentales del espacio de direcciones en Linux
Cada proceso en Linux tiene un espacio de direcciones dividido entre segmentos principales:

1) Segmento de Texto:
    - Contiene el código ejecutable del programa (instrucciones máquina)
    - Es generado por el compilador / ensamblador
    - Es solo lectura y no cambia (los prog auto-modificables ya no se usan)
    - Puede ser compartido entre procesos

2) Segmento de Datos
    - Almacena variables, arreglos, cadenas y demás datos del programa
    - Se divide en datos inicializados y no inicializados (BSS)
    - Para ahorrar espacio, los datos no se almacenan en el archivo ejecutable; el sistema los crea al cargar el programa
    - Linux usa una “página cero” protegida para representar datos no inicializados, y la reemplaza por una página real en el primer intento de escritura
    - Este segmento puede crecer / disminuir dinámicamente
    - (a través de llamadas como brk) funciones como malloc() usan esta capacidad
    - La zona dinámica se conoce como heap

3) Segmento de Pila (stack)
    - Usado para llamadas a funciones, variables locales, etc.
    - Crece hacia abajo (hacia 0) desde el tope de espacios de direcciones
    - Contiene al inicio las variables de entorno y la línea de comandos
    - Si crece más allá del límite el SO extiende el stack automáticamente

Otras características importantes:
- Segmentos de texto pueden ser compartidos por otros procesos (optimiza memoria)
- Segmentos de datos y pila no se comparten, salvo justo después de un fork() y sólo si no se modifican (copy-on write)
- Mapeo de archivos en mem permite que los procesos accedan a los archivos como si fueran arreglos en RAM.

- Facilita el acceso aleatorio y alta velocidad
- Es la base del uso de librerías compartidas
- Permite compartir mem entre procesos mediante archivos temporales

Llamadas al Sistema para manejo de memoria en Linux
El estándar POSIX no define llamadas al sistema para gestión de memoria, porque se considera demasiado dependiendo del hardware. En su lugar, se sugiere usar malloc de C. Sin embargo, Linux si ofrece syscalls específicas para manejar memoria

Principales llamadas al sistema en Linux

1. brk - Control del segmento de datos

   int brk (void *addr)

   Función: ajusta el tamaño del segmento de datos del proceso
   Parámetro:
   - addr: dirección del primer byte después del nuevo límite del segmento
2. mmap - Mapeo de archivos en memoria

   void * mmap (void *addr, size t len, int prot, int flags, int fd, offset);

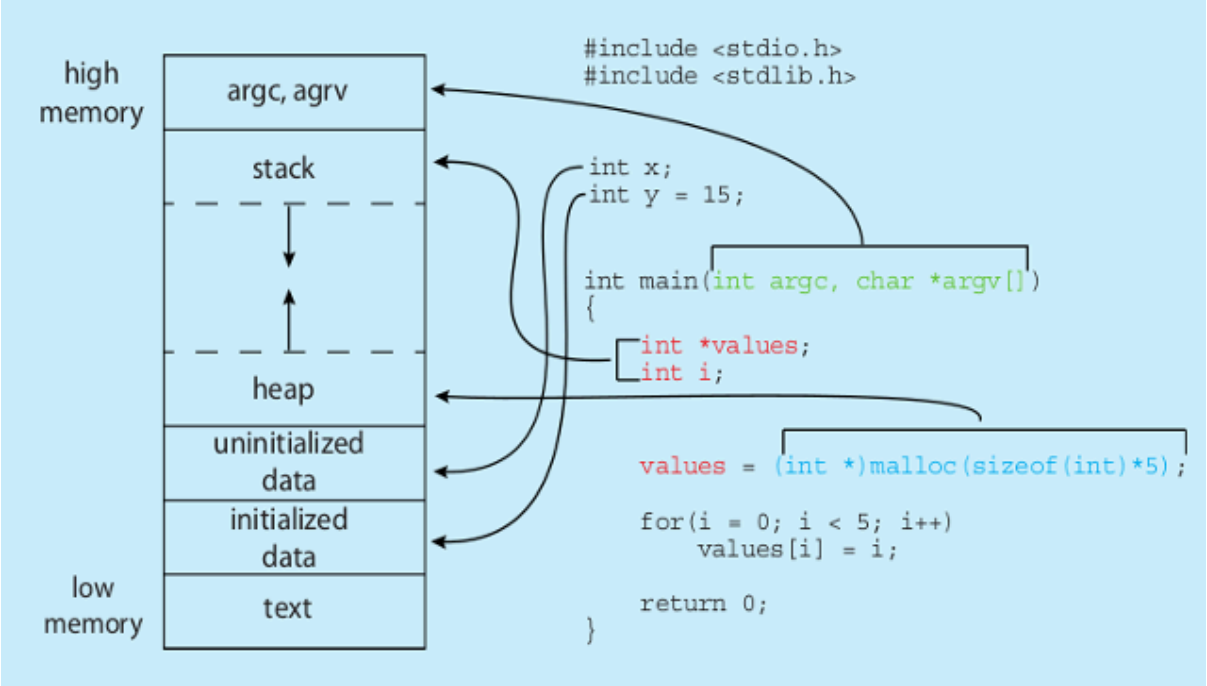
   Parámetros clave:
   - addr: dirección de mapeo (o para que el sistema decida)
   - len: bytes a mapear (múltiplo del tamaño de la página)
   - prot: protección (PROT_READ / PROT_WRITE / PROT_EXEC)
   - flags: MAP_PRIVATE (cambios privados) / MAP_SHARED (cambios visibles a otros procesos)
   - fd: descriptor del archivo abierto
   - offset: punto de inicio en el archivo (múltiplo de página)

   Ventajas:
   - Acceso directo a los archivos como si fueran memoria
   - Compartición eficiente entre procesos
   - Usado intensamente por bibliotecas compartidas

3. munmap - liberar mapeos

   int munmap (void * addr, size_t len);

   - Función: elimina mapeos de memoria
   - Parcial: puede desmapear sólo una porción del archivo mapeado


high memory
argc, agrv
stack
heap
uninitialized data
initialized data
low memory
text
#include <stdio.h>
#include <stdlib.h>
int x;
int y = 15;
int main(int argc, char *argv[])
{
int *values;
int i;
values = (int *)malloc(sizeof(int)*5);
for(i = 0; i < 5; i++)
values[i] = i;
return 0;
}