

## Aims of filesystems

## Lecture 5

### Filesystem interface

We will see how filesystem is used to organize information for the OS.

#### References:

- [OSC] 11.1–11.5 (Filesystem Interface), 13.1–13.3, 13.4–13.5 (I/O Systems).
- [ULK] pp. 33–34 (Device Drivers), 328–334 (VFS), 378–384 (Associating Files with I/O Devices, Device Drivers), 391–397 (Character/Block Device Handling), 415–417, 431–433 (Disk Caches).
- Manpages: `proc(5)`, `symlink(2)`, `dup(2)`, `getdents(2)`, `opendir(3)`, `read-dir(3)`, `stat(2)`, `fcntl(2)`, `select(2)`, `poll(2)`, `alarm(2)`.

POS(0230A)

POS(0230A)-5.1

## Filesystem structure

Filesystem is formed by **files** and **directory entries**.

- Early days: **tree**. A directory can contain other directories, but each directory and file has exactly one directory as parent.
- Unix: **acyclic graph**. Like trees, but directories and files may have multiple parents ("**hard links**"), as long as no cycle is formed.  
No cycles, to allow reference counting to tell when to deallocate a file. Linux: cycles are prevented by disallowing directories with multiple parents.

Hard links is handy: with it, the same file object can be referred to in multiple ways. (A copy that doesn't use extra space.)

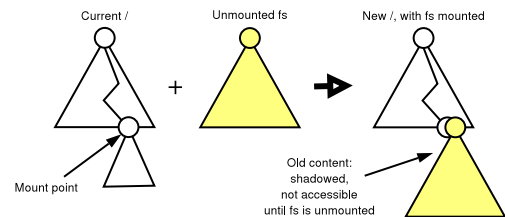
Another, less efficient way to achieve similar effect: **soft (symbolic) links**. Similar to a separate file, soft links have its own content. But the content is treated as a filename which the OS "forwards" the operations to.

This does not have cycle restriction, but do no reference counting.

POS(0230A)-5.2

## Filesystem mounting

A filesystem must be **mounted** before it is used. We can treat mounting as adding a filesystem to the namespace of the system.



Again, different OS implement this differently.

- In Unix, we have a single tree having all the systems mounted.
- In Windows, each mounted filesystem is given a drive letter.

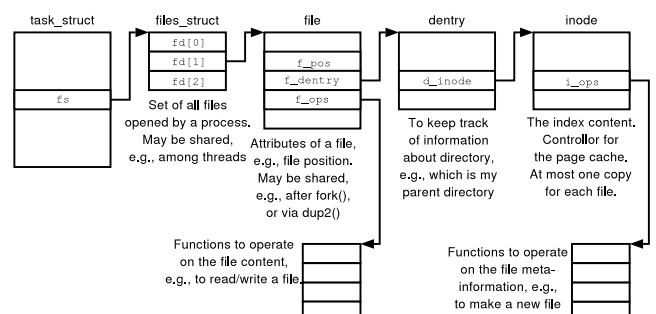
POS(0230A)-5.3

## From open() to file descriptor

- A process calls `open()`, given a path, to use a file.
- The OS traverses the path to find the directory object containing the file, and use the directory object to **lookup** (create or reuse) the **directory entry** (dentry) within the directory containing that file.  
The directory object itself is an "inode".
- When a dentry is created, an **inode** is also created to hold the **meta-information** about the file. The dentry holds a pointer to it.  
Meta-information includes size, ownership, permission modes, etc.
- The inode will have information about **what functions to use** when the file object is read, written, etc.  
E.g., for a device, it will probably read/write the I/O ports.
- A **file object** is created, having a pointer to the dentry. The file object is added to the PCB of the process, and its index is the fd.

POS(0230A)-5.4

## Illustration: kernel filesystem structures in Linux



At the bottom are two sets of functions, which depends on the type of file objects...

POS(0230A)-5.5

### Strategy functions

- It should now be clear what is **really** meant by file descriptors: it is an **index** to the file table of the process.
- A file descriptor may point to **any type** of file objects. It might be a disk file, a nameless pipe, a FIFO, a device file, etc.
- When a system call (e.g., `read()`) uses the fd, the file object is found, and the `f_ops` member of it is extracted.
- `f_ops` points to a **strategy function table**, which store functions that actually implement read, write, etc. (Similar for `i_ops`.)
- **Different types of objects** have different `f_ops`, and hence use **different strategy functions** to perform read/write/other operations.
- E.g., the read function of disk files will load the file from disk, that of a pipe will get the data from the memory buffer, and that of a device driver will read data from the physical device.

POS(0230A)-5.6

### Working on dentries and inodes from processes

- Sometimes processes want to access the kernel data structures.
- In particular, to **list a directory**, the processes must be able to get a list of dentries within a directory.
- The kernel exposes that in-kernel data structure by a system call, which copies the kernel dentries to the process virtual memory.  
Non-portable: Different system uses different function, although in both Linux and Solaris they are `getdents()`. No one actually use them. But if you really want to see it done (it's ugly), see `getdents.c` in the source of this lecture.
- **C library** functions `opendir()` and `readdir()` is normally used instead, which eventually makes the system calls.  
Making the program more portable.
- The processes may also want the **information in the inode** (e.g., UID, permission, size, etc), which the user can obtain by using the `stat()` / `lstat()` / `fstat()` system call.

POS(0230A)-5.7

### Code example

Simple code to list file sizes and modification dates:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

int main() {
    DIR *d = opendir(".");
    struct dirent *ent = readdir(d);
    while (ent) {
        struct stat buf;
        stat(ent->d_name, &buf);
        printf("%s: size = %ld, last modified: %s",
            ent->d_name, buf.st_size, ctime(&buf.st_mtime));
        ent = readdir(d);
    }
    return 0;
}
```

POS(0230A)-5.8

### Doing I/O using file semantics

Since lesson 2 we know that we can **perform I/O by reading/writing a device file**. Now it should be easy to understand how it is done:

- Whenever a file is read, a **read function** is extracted from the `f_ops` member of the `file`, and that function is called to perform I/O.
- For regular files, a **filesystem read function** is extracted, which locates the file in the disk (or cache, see later), and copy it to our buffer.
- For device files, a **device driver read function** is extracted, which get data from the I/O device and copy it to our buffer.
- Similar things happen for write and other operations.  
Try to understand why the filesystem is such an important interface: it is flexible. Give a new set of functions and it operates in a new way.
- But... which device driver to call? It depends on what's in the `f_ops`, which is assigned when the file is **opened**.

POS(0230A)-5.9

### Device files in Unix

What is assigned to the `f_ops` is determined by 2 entities:

- The **filesystem** containing the file. (The `/proc` directory can do special things even to normal files because it is in a special filesystem.)
- The **file type** of the file.

There are two types of files allocated to devices:

- **Block** device files, for storage devices.
- **Character** device files, for all other devices.

When such a file is opened...

- The OS finds the device driver handling the device by looking for the **major device number** of the device file, determining the `f_ops`.
- The device driver use the **minor device number** to find which device the file actually refers to, which might further change `f_ops`.

POS(0230A)-5.10

### Example: Linux major numbers

Device number	Character device	Block device
1	Memory devices (e.g., mem, null, zero, full, random)	RAM disk
2	Terminal devices (master side)	Floppy
3	Terminal devices (slave side)	IDE
4	Virtual consoles, Serial port	
6	Parallel port	
8		SCSI disks
10	Miscellaneous devices	
11	Raw keyboard	SCSI CDs
13	Joystick etc	IDE
14	Sound	
65-71		SCSI disks

POS(0230A)-5.11

### Implementation issue: short interrupt

Within interrupt handlers, **interrupts are typically disabled**.  
Reduce kernel stack usage, avoid many race conditions.

**Hardware interrupts must be handled quickly.** E.g., if the CPU cannot read the old data of serial port before new data come, the old data is lost.  
i.e., Interrupt handlers must be **quick**.

Typical handlers have two parts: (1) read the device to **free the device**, and (2) move the process to ready queue, deal with the buffers and cache, etc.  
**Only the first part need interrupt disabled.** Two strategies:

- **Solaris:** the second part is **done in a high priority kernel thread** instead of the interrupt handler.
- **Linux:** the second part is done only **before switching back to user-mode**, with interrupts enabled.  
i.e., the interrupt handler will only record the fact that something needs to be done.

POS(0230A)-5.12

### Character device handling

For polling:

1. **Setup** the I/O by writing to the control registers.
2. while control register indicate I/O is **pending**:
  - **Schedule** other processes to run, and wait until the scheduler choose the process again.
3. **Perform** the I/O by reading or writing the data.
4. If necessary, **reset** the I/O by writing to the control register.

For interrupt driven I/O:

1. **Setup** I/O, interrupt and DMA by writing to control registers.
  2. Continue, or put the process into wait queue.
- Interrupt handler**
3. **Find** the completed operation.
  4. **Perform** the I/O by reading or writing the data register.
  5. **Reset** the I/O by writing control registers.
  6. Put associated processes back to **ready**.

POS(0230A)-5.13

### Sidetrack: Don't want the program to stop when doing I/O?

Unlike reading and writing files ,reading and writing a device usually needs to wait for a long time. We say the I/O **blocks** until completion.

Many programs **cannot afford to suspend all other activities**. E.g., many GUI programs must continue responding to the user during I/O.

**Method 1:** create a **"I/O" thread** for the operation. The I/O thread is blocked, but the main thread continues to work.  
So it turns the I/O problem into a IPC problem. We will look at IPC later.

**Method 2:** Set the file to **non-blocking**, using `fcntl()`. Now if an I/O operation needs to wait, the operation **returns an error** instead of waiting.

*In simple words:* "If the fd has input (or can do output immediately), do it. Otherwise, tell me there is none (or it is not ready)."

POS(0230A)-5.14

### Sidetrack: Non-blocking example: clear the input

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
void clear_input() {
    long flag = fcntl(0, F_GETFL);
    char c;
    fcntl(0, F_SETFL, flag | O_NONBLOCK); // Nonblocking I/O now
    while (read(0, &c, 1) != -1); // Read all available characters
    fcntl(0, F_SETFL, flag); // Reset I/O to blocking
}
int main() {
    char line[80];
    setvbuf(stdin, NULL, _IONBF, 0); // unbuffered I/O
    sleep(8); // supposed to be doing other things
    clear_input();
    printf("Type a line: ");
    fgets(line, 80, stdin);
    printf("Got %s", line);
    return 0;
}
```

POS(0230A)-5.15

### Sidetrack: I/O without stopping continued

Problem: The following wastes the CPU:

```
char ch;
fcntl(fd1, F_SETFL, O_NONBLOCK); // set to non-blocking
fcntl(fd2, F_SETFL, O_NONBLOCK);
for (;;) {
    if (read(fd1, &ch, 1) != -1)
        /* work for fd1 */;
    if (read(fd2, &ch, 1) != -1)
        /* work for fd2 */;
}
fcntl(fd1, F_SETFL, 0);
fcntl(fd2, F_SETFL, 0);
```

**Method 3:** Use a system call (`select()`, `poll()`) which **waits until at least one file is ready for read**.

*In simple words:* "Here are all things I'm interested in. If none of them are ready for I/O, please give the CPU to other threads—I don't have anything to do until then."

POS(0230A)-5.16

### Sidetrack: Example: use poll()

Here is a very simple example that waits only for `stdin`. However, the program works equally well if you wait for more file descriptors.

```
#include <sys/poll.h>
int main() {
    struct pollfd events[1]; // use larger array if more events
    events[0].fd = 0; // wait stdin
    events[0].events = POLLIN; // for input
    // Can put more events to wait here
    printf("Waiting\n");
    poll(events, 1, 2000); // 1 event to wait, 2s timeout
    if (events[0].revents & POLLIN)
        printf("Ready\n");
    else
        printf("Not ready for 2 seconds\n");
}
```

POS(0230A)-5.17

## Sidetrack: I/O without stopping, continued

### Method 4: Use signals.

There are two ways to exploit signals in our program doing I/O:

- Perform the **blocking** I/O anyway, but setup the other events to **interrupt** the blocking I/O. (By sending a signal.)  
It is okay to block as long as nothing else requires my attention.
- Setup the file descriptor so that it will **generate a signal when I/O is ready**. Then do the I/O in the signal handler.  
I have other CPU-intensive things to do, so do those things. When I/O can be done, tell me so that I interrupt the CPU-intensive things and work on the I/O first.

POS(0230A)-5.18

## Sidetrack: Interrupting input example: timed input

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void do_nothing(int signo) {}
int main() {
    char str[80];
    struct sigaction act, oact; // Stop SIGALRM from killing process
    printf("Give me a line in 10 seconds, or die. ");
    act.sa_handler = do_nothing;
    act.sa_flags &= ~SA_RESTART; // Don't restart system calls
    sigaction(SIGALRM, &act, &oact);
    alarm(10); // Generate SIGALRM in 10 seconds
    if (fgets(str, 80, stdin) == NULL)
        printf("You're dead now. BANG!\n");
    sigaction(SIGALRM, &oact, 0);
    return 0;
}
```

POS(0230A)-5.19

## Block devices

Some storage devices (hard disks, floppies, CDROM drives) have special characteristics that the OS kernel exploit to provide better performance...

- These devices transfer data not in single **bytes**, but instead in multiples of fixed size **blocks**, which fits perfectly in memory **frames**.  
Most disk uses 512 byte blocks, so 8 blocks is exactly 1 frame.
- The blocks in the devices are identified by **addresses**. If we read the same address a hundred times without a write in between, we are guaranteed to read the same thing.
- Most programs access a few blocks of **consecutive addresses** one after another.

They are the mysterious **block** devices that we have come across a few times. They are the basis used to implement filesystems.

Do you see a strange loop here? We need filesystem to access block device files, and block device files are used to implement filesystems...

POS(0230A)-5.20

## The block layer

Most OS have facilities so that **the filesystem does not directly access** the block devices. Instead, there is a middle layer, the **block layer**. It's functionalities:

- **Receive requests** from the filesystems.
- **Maintain buffers** for I/O that are not yet completed.
- Keep a **cache** of recently accessed disk blocks, to reduce the amount of read and writes required. The cache can be shrunk upon request, when memory is tight.
- **Schedule** the reads and writes, so that nearby blocks in the disk are read together to reduce time needed to access all the required blocks.  
Again, more about it in the last lecture.
- **Make requests** to device drivers to read blocks when they are not in cache, or write blocks when cache is shrunk.

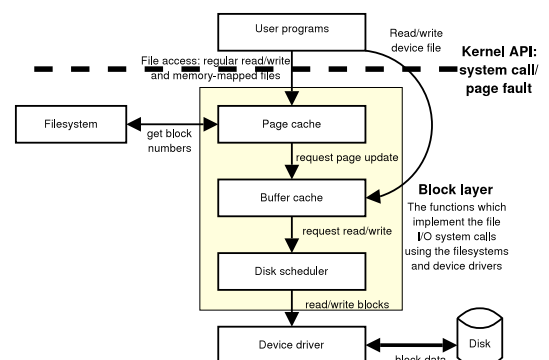
POS(0230A)-5.21

## From file descriptors to disks

- The user process calls `read()`. The **read strategy** function of the file object, which points to a **block layer** function, is extracted and called.
- The strategy function uses the **current pointer** of the file object to find **which page is needed**, and see if it is already in the page cache.  
The file location is counted in pages, to share information with `mmap()`.
- If there is a cache miss, the block layer asks the **filesystem** to find the **block numbers** corresponding to those pages.  
Filesystems deal with disk organization, and thus are responsible for the task.
- The block layer asks the **block cache** to return the content of the blocks. Normally, the **disk scheduler** is then invoked to transfer the blocks.  
In rare circumstances, the block cache may already have the block. This happens if the block device is read directly, skipping the filesystem.
- The disk scheduler rearrange the order of the requests, and asks the **device driver** to read the needed blocks.

POS(0230A)-5.22

## Together with mmap and meta-information



So block device drivers typically don't need to deal with paging. The block layer does all the tedious repetitive tasks.

POS(0230A)-5.23

## Cache management

- Note that in our description, **both mmaped files and read/write uses the page cache.**
- This is called **unified cache**, which implements the Unix consistency semantics in a way that memory-mapped I/O are treated the same.  
The alternative is to use a cache for memory-mapped files and another for read/write. That would allow file read/write to cache in size different from page size, but then mmaped files will not have the same content as that read/write.
- The page cache **uses frames to store the page contents.** When more and more files are used, the cache will grow larger.
- How to **get back the frames** when we are low on frames? It is the same mechanism as the one when we learn **swapping**, so let's delay the discussion until then.