2023 Fall ECE 344: Operating Systems

Lecture 27 1.0.0

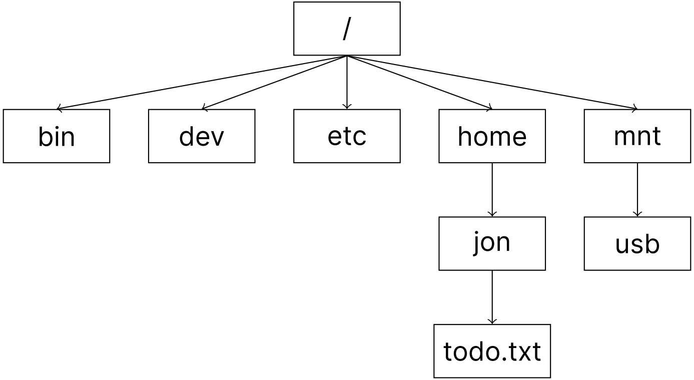
Filesystems

Jon Eyolfson

2023 Fall

Filesystems

Usual layout of a POSIX Filesystem (here: parts of FHS):



Working Directory: /home/jon

What is the absolute and relative path to **todo.txt**? To **usb**?

POSIX Filesystem

todo.txt Relative: ./todo.txt
todo.txt Absolute: /home/jon/todo.txt
usb Relative: ../../mnt/usb
usb Absolute: /mnt/usb

Special symbols:

- . — Current directory
- .. — Parent directory
- ~ — User's home directory ($HOME)

Relative paths are calculated from current working directory ($PWD)

You Can Access Files Sequentially or Randomly

Sequential access
- Each read advances the position inside the file
- Writes are appended and the position set to the end afterwards

Random access
- Records can be read/written to the file in any order
- A specific position is required for each operation

POSIX Filesystem

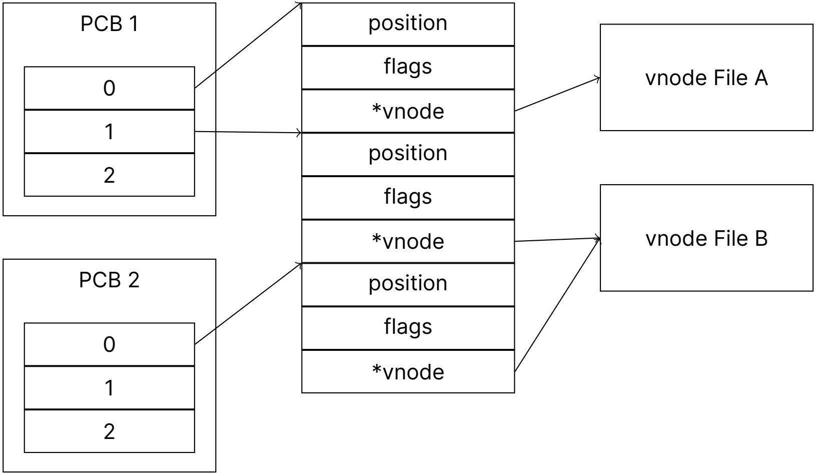
```
int open(const char *pathname, int flags, mode_t mode);

// flags can specify which operations: O_RDWR,O_WRONLY, O_RDWR
// also: O_APPEND moves the position to the end of the file initially

off_t lseek(int fd, off_t offset, int whence);

// lseek changes the position to the offset
// whence can be one of: SEEK_SET, SEEK_CUR, SEEK_END
//   set makes the offset absolute, cur and end are both relative
```

Accessing Directory API

```c
DIR *opendir(char *path); // open directory
struct dirent *readdir(DIR *dir); // get next item
int closedir(DIR *dir); // close directory
```

```c
void print_directory_contents(char *path) {
    DIR *dir = opendir(path);
    struct dirent *item;
    while (item = readdir(dir)) {
        printf("- %s\n", item->d_name);
    }
    closedir(path);
}
```

File Tables Are Stored in the Process Control Block (PCB)



Each Process Contains a File Table in its PCB

A File Descriptor is an index in the table

Each item points to a system-wide *global open file table*

The GOF table holds information about the seek position and flags
It also points to a *VNode* (supports read/write/etc)

A vnode (virtual mode) holds information about the file
vnodes can represent regular files, pipes, network sockets, etc.

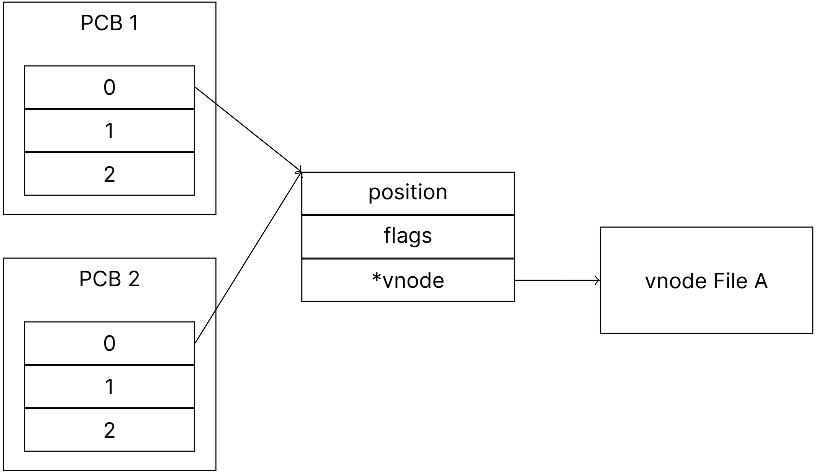
Remember What Happens In A Fork

PCB is copied on fork

Specifically for us, the local open file table gets inherited

Both PCBs point to the same Global Open File Table entry

Both Processes Point to the Same GOF Entry



There Are Some “Gotchas” For This Sharing

Current position in file is shared between both processes

Seek in one process leads to seek in all other processes using the same GOF entry

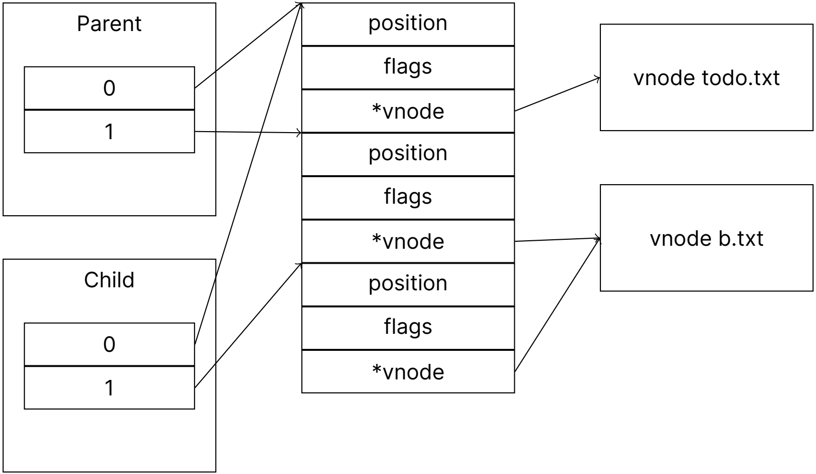
Opening the same file in both processes after forking creates multiple GOF entries

How many LOF and GOF Entries Exist? What is the Relationship?

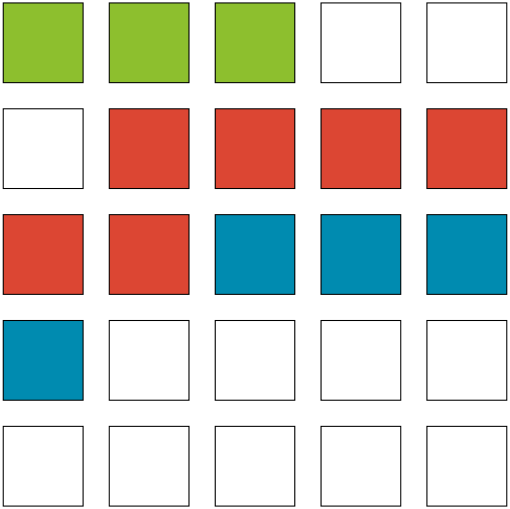
```
open("todo.txt", O_RDONLY);
fork();
open("b.txt", O_RDONLY);
```

Assume there are no previously opened files (not even the standard ones)

There are 2 LOF Entries Each, and 3 GOF Entries



How Do We Store Files? Contiguous Allocation?



Contiguous Allocation Is Fast, If There Are No Modifications

Space efficient: Only start block and # of blocks need to be stored

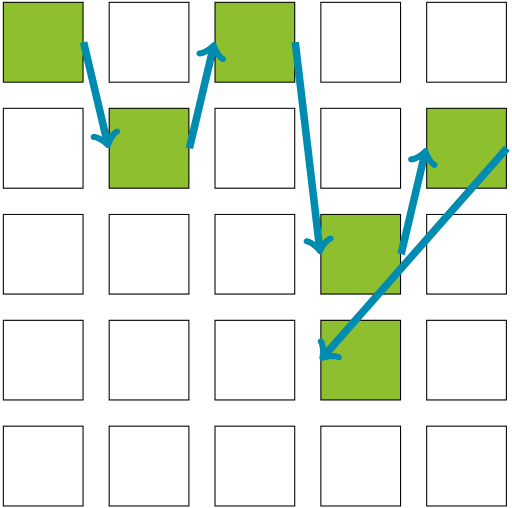
Fast random access: $\mathit{block} = \mathit{floor}(\frac{\mathit{offset}}{\mathit{blocksize}})$

Files can not grow easily

  Internal fragmentation (may not fill a block)

  External fragmentation when files are deleted or truncated

What About Storing Like a Free List of Pages? Linked Allocation



Linked Allocation Has Slow Random Access

Space efficient: Only start block needs to be stored

- Blocks need to store a pointer to the next block (block is slightly smaller)

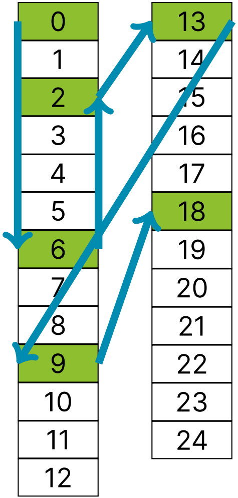
Files can grow/shrink

- No external fragmentation
- Internal fragmentation

How can we increase random access speed? We need to walk each block

- Each block may be located far away (it will never be cached)

File Allocation Table Moves The List to a Separate Table



0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

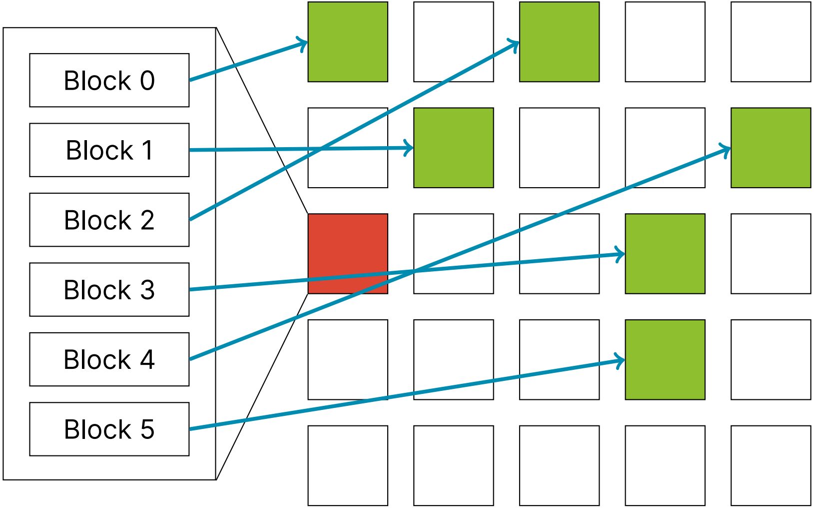
File Allocation Table (FAT) is Similar to Linked Allocation

Files can grow/shrink
- No external fragmentation
- Internal fragmentation

Fast random access: FAT can be held in memory/cache
- FAT size is linear to disk size: can become very large

How can we further increase random access speed?

Indexed Allocation Maps Each Block Directly



For Indexed Allocation, Each File Needs an Index Block

Files can still grow/shrink

- No external fragmentation
- Internal fragmentation

Fast random access

File size limited by the maximum size of the index block (fit it in one block)

Indexed Allocation Problem

Assume this scenario:

- An index block stores pointers to data blocks only (no meta information)
- A disk block is 8 KiB in size
- A pointer to a block is 4 Bytes

What is the maximum size of a file managed by this index block?

Indexed Allocation Solution

Assume this scenario:

- An index block stores pointers to data blocks only (no meta information)
- A disk block is 8 KiB in size
- A pointer to a block is 4 Bytes

# of pointers = $\frac{8\text{KiB}}{4\text{B}} \frac{2^{13}\text{B}}{2^{2}\text{B}} = 2^{11}$
# of addressable blocks = # of pointers
Total of bytes = $2^{11} \times 2^{13} = 2^{24} = 16\text{MiB}$

Filesystems Enable Persistence

They describe how files are stored on disks:

- API-wise you can open files, and change the position to read/write at
- Each process has a local open file and there’s a global open file table
- There’s multiple allocation strategies: contiguous, linked, FAT, indexed