Runtime file data structures

Now that we know how the information can be represented on the disk we need to know about the data structures the OS maintains when we use a file.

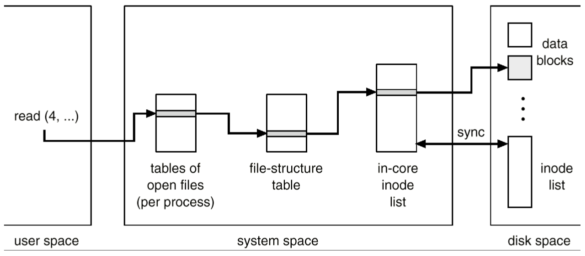
System wide open file table
The system must keep track of all open files.
Information from the on-disk file control block (these must be kept consistent).

Which processes are accessing the file?
How is the file being accessed?

Process open file table
A pointer to the system open file table.
Current file position (for sequential reading or writing).
A pointer to the buffer being used for this file by the process.

The file buffer
Data is read in block (or cluster) amounts.

UNIX runtime file structures



Here we see

• the on disk inodes and data blocks

• the incore copy of the inode, this includes the reference count

• the system wide open file table (file-structure table), this actually stores one entry for every time the file was opened

• the process open file table, just an array of pointers to the file-structure table

What would a fork do?

UNIX fork interaction

```
# twice.py
import os

file = open("temp", "w")
file.write("before the fork") # 15
#file.flush()

if os.fork() == 0: # in the child
    print("child: {}".format(file.tell()))
    print("child: {}".format(file.tell()))
else: # in the parent
    print("parent: {}".format(file.tell()))
    print("parent: {}".format(file.tell()))
file.close()
```

Produces the following output (without flushing):

```
$ python3 twice.py
parent: 15
parent: 30
child: 30
child: 30

$ python3 twice.py
parent: 15
child: 30
parent: 30
child: 30

$ python3 twice.py
parent: 15
parent: 15
child: 30
child: 30
```

Opening and closing files

Most systems require some open call to make the connection between a process and a file.

The open call does several things (not all OSs do all of these):

- searches for the file with that name
- verifies that the process has access rights to use the file in the way specified
  - this means we don’t check after this
  - this can be a security problem, sometimes referred to as the TOCTTOU (time of check to time of use) problem
- records the fact that the file is open (in the system-wide open file table) and which process is using it
- constructs an entry in the process open file table
- allocates a buffer for file data
- returns a pointer (file handle or file descriptor) to be used for future access

Opening a file in UNIX

open(filename, type of open)

e.g.

```
fd = open("OS/test/answers", O_RDWR);
```

convert filename to an inode – this also copies the on-disk inode into memory (if not already there) and locks the inode for exclusive access
if file does not exist or not permitted access return error
allocate system-wide file table entry, points to incore inode, increment the count of open references to the file
fill per-process file table entry with pointer to system-wide file table entry
unlock the inode
return the index into the per-process file table entry (known as the file descriptor)

UNIX write system call

```
write(fd, buffer, count)
```

```
get file table entry from fd
check accessibility
lock inode
while not all written
    if a block doesn't exist for the current position
        allocate one - updates the inode
    if not writing a complete block
        read the block in
    put the data in the block's buffer
    delay write the block (some later time)
    update file offset, amount written
update file size
unlock the inode
```

Delay write

Buffers are shared by the system.
The write doesn't occur until another process is to use the buffer for a different block (LRU replacement) or a daemon process flushes it.

<u>Advantage</u>

if a process wants to access this information it is already/still in memory
e.g. process writes some more and it fits in the same block

<u>Disadvantage</u>

information is not written immediately
usually a daemon process writes data buffers after 30 secs, metadata buffers after 5 secs
sync command forces buffers to write

UNIX append

If the file has been opened in append mode `O_APPEND` then there is a possible race condition.

Before each write the file position pointer is moved to the length of the file.

What if another write changes the length of the file before this write completes?

The file system must guarantee atomicity for the append write operation.

That is why there is an append mode for opening a file.

Before next time

Read from the textbook

22.5.2 – NTFS Recovery

If you want to read about versioning systems

C.A.N. Soules, G.R. Goodson, J.D. Strunk, G.R. Ganger,

Metadata Efficiency in a Comprehensive Versioning System, Technical Report, School of Computer Science, Carnegie Mellon University

You may also want to read about ZFS on Wikipedia.

17.2.1 – Network Operating Systems

17.9.1 – Naming and Transparency

11.5.3 – Consistency Semantics

17.9.2 – Remote File Access