

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 table entry (these must be kept consistent).

Which process is accessing the file.

How is the file being accessed.

How many processes are accessing the file.

Process open file table

A pointer to the system open file table.

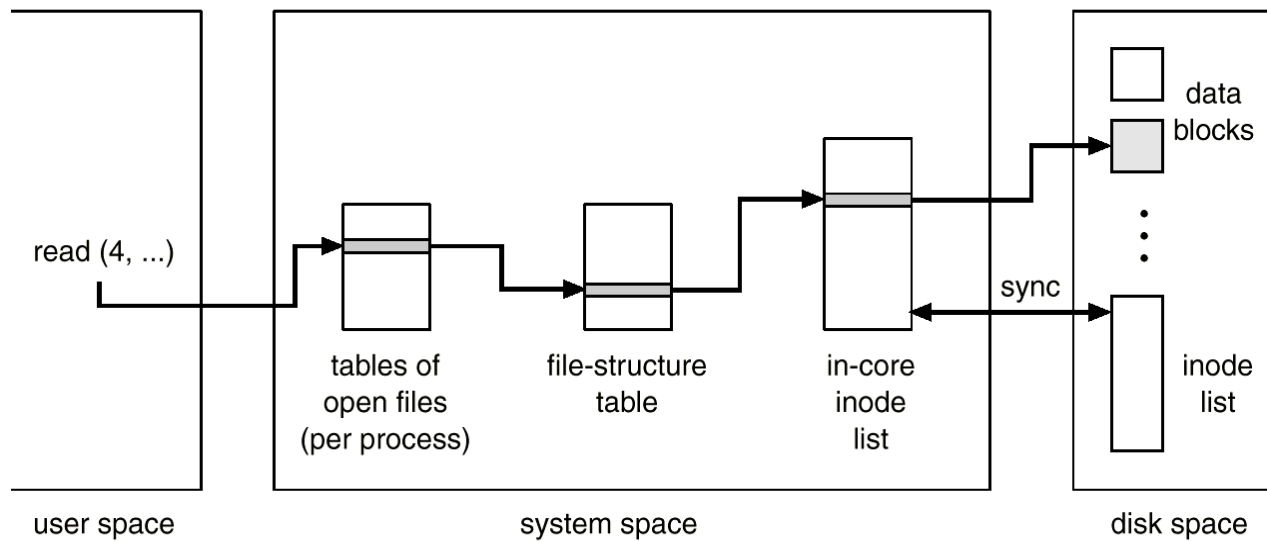
Current file position (for sequential reading or writing).

A pointer to the buffer being used for this file by the process.

The file buffer

Data is read in block 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

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 and locks
the inode for exclusive access

if file does not exist or not permitted access
return error

allocate file table entry for inode, initialize the
count of processes using the file

fill user file table entry with pointer to system
wide file table entry

unlock the inode

return the index into the user 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

if a block doesn't exist for the current position

 allocate one - updates the inode

while not all written

 if not writing a complete block

 read the block in

 put the data in the block's buffer

 delay write the block

 update file offset, amount written

update file size

unlock the inode

Delay write

Buffers are shared by 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.

Advantage

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

Disadvantage

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

File versioning systems

It can be very useful to keep earlier versions of files.

- We can recover from mistakes.
- We can restore damaged files.
- We can compare versions to see the changes made.
 - Useful for security purposes (self-securing storage)
 - Also useful for other purposes – e.g. working on a project with others and you want to see the changes they made.

Similar to code management services like CVS.

- Sometimes we want to use earlier versions and still hold on to the recent versions.

Can be done in a variety of ways but all require extra disk space.

Methods of versioning

A new version could be created when the file is closed.

A new version could be created after every modification – known as comprehensive versioning.

Obviously a lot more versions.

Either way we can -

- keep complete copies of all previous versions
 - very space intensive
 - but fast to retrieve/recover
- keep a journal (or log) of changes
 - the journal keeps a record of the changes between two versions
 - retrieving requires work to reproduce earlier versions
- keep a tree with all data
 - finding any version takes the same amount of time
 - can be slow for current version if the tree is big

Example

e.g.

a file with the contents:

“Dear Mum, I hope you are well.”

gets modified and saved as:

“Dear Mum, I am doing really well in my
Operating System course. I hope you are
well.”

then as:

“Dear Mum, I thought I was doing really well
in my Operating System course until I sat
the test. I hope you are well.”

Log version

Original version was

Dear Mum, I hope you are well.

changes to get to version 2

11i54 (54 chars inserted at position 11)

changes to get to version 3

13d (a deletion at position 13)

am (the deleted data, this must be kept)

13i13

74i21

The current version is always stored.

Dear Mum, I **thought I was** doing really well in my Operating System course **until I sat the test**. I hope you are well.

To get previous versions have to go backwards through the log.

If we want to be able to roll forward from a checkpoint we need the new data in lines like 13i13.

Multiversion B-tree

A1: Dear Mum, I _

A2: hope you are well.

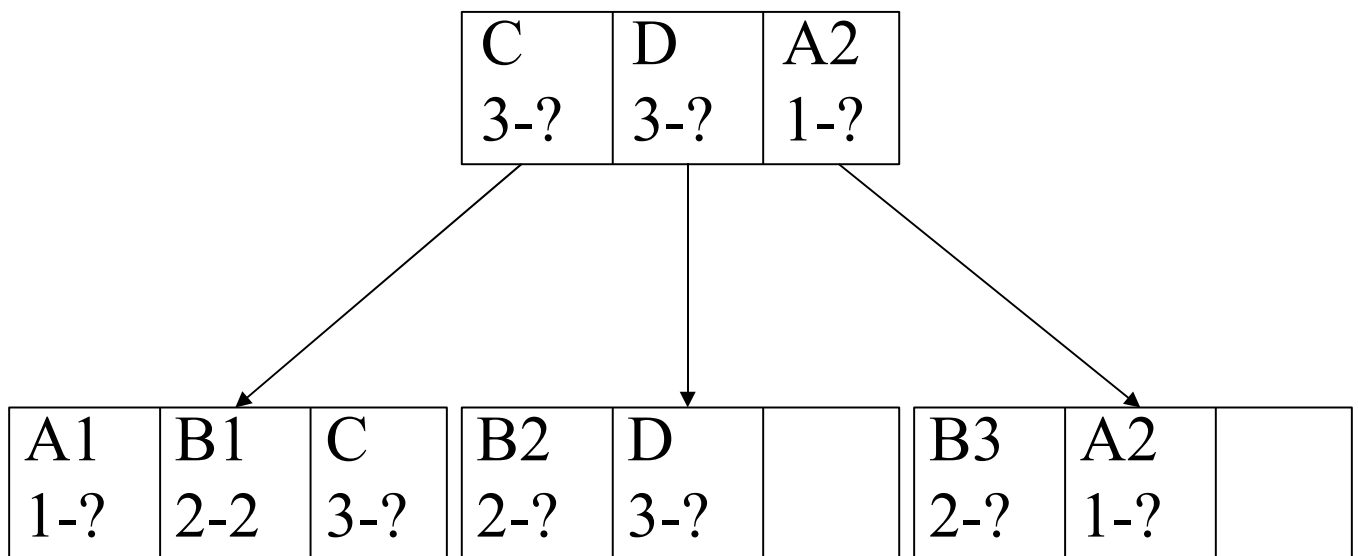
B1: am

B2: _doing really well in my Operating
System course

B3: . _I_

C: thought I was

D: _until I sat the test



The version ranges (2-?) show which version the leaf is valid for.

? means up to the present.

Advantages and disadvantages

Log system

- very compact
- access to the current version is the same as without versioning
- slow to revert to previous versions especially if there are many versions
- can use checkpoints to improve this, but this adds considerably to the space requirement

Tree system

- very compact
- quick to revert to any previous version
- if there are lots of versions the tree can be big and then access to the current version will be a little slow
- can keep a separate copy of the current version, this also adds to the space requirement

Neither method works well if the data between versions is completely different. We are stuck with having to keep complete versions.

Pruning

All conventional versioning systems use pruning to keep the amount of data stored under control.

Different heuristics

- a fixed number of versions
- treat some changes as more important than others
- “observe” user behaviour, e.g., most often accessed
- the user has to explicitly request a version be held
 - snapshot systems – keep versions of files at particular times
- only keep versions for a small number of files

Self-securing storage

All metadata, directories and critical files (OS files) are kept on a versioning system.

Any intrusion (that uses files) can be tracked because the intruder cannot erase changes they have made to the system.

We need to maintain all versions between checks for intrusion.

This is sometimes referred to as a detection window.

If the system is unable to keep enough versions we signal an alarm.

Either something has gone wrong, in the sense of not enough space allocated for a normal amount of usage.

Or someone is trying to force a pruning to hide their tracks.

VMS versions

When a file is closed VMS checks the number of versions. If the number is greater than the maximum number then the oldest version is discarded.

Windows XP

Even Windows XP does some versioning.

It takes a checkpoint (restore points) of important system files on a regular basis.

daily

on installation of new drivers and applications

NTFS maintains a log of all changes to metadata, along with redo,undo information and whether the change was committed.

So it can recover all metadata to a consistent state after a crash (but not all data).

Before next time

Read from the textbook

21.5.2 – NTFS Recovery

If you want to read more 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