Lecture #21: File system naming + location – putting it all together

****
Review -- 1 min
****

fs data layout – how to find blocks of a file given its header
- ♦ trees, linked lists, etc
- ♦ how to get good sequential layout

Transactions –
ACID
Logging (redo, undo, commit, rollback)
LFS

****
Outline - 1 min
****

Kernel data structures
Open/close/read/write v. mmap
project overview
Scheduling

****
Preview - 1 min
****

M – midterm
Then, memory systems, protection

****
Lecture - 20 min
****

1. Disk scheduling

Disk can only do 1 request at a time; what order do you choose to do the requests

if 0 or 1 request queued, easy
>1 – try to arrange requests in some order that reduces seek time

1.1 FIFO

QUESTION: how will this work?
Fair among requestors, but order of arrivals may be random → long seeks

1.2 SSTF – shortest seek time first

pick the request that is closest on disk (although called SSTF, today include rotational delay b/c rotation can be as long as seek)

QUESTION: how will this work

good at reducing seeks
can cause starvation

Is it optimal?

1.3 SCAN

SCAN implements elevator algorithm – take the closest request in the direction of travel
No starvation, but retains flavor of SSTF

1.4 CSCAN

2. API and caching/ Kernel data structures for file system

2.1 Read/write interface

Kernel maintains per-process **open file table --**
each entry -- pointer OpenFile object stored in kernel memory

system call (user)	kernel action
open("path")	→ put a pointer to right file in FD table; return index

close(fd) → drop entry from fd table

read(fd, buffer, length) → user refers to open files with index
write(fd, buffer, length) of file descriptor table

What needs to be in OpenFile object to support read/write?

- Inumber (or, if caching, pointer to in-memory FileHeader object)
- per-open-file data (e.g., file position, …)

Why have a separate fd table

- why not just give user pointer to FileHeader object in kernel?
  - o how does kernel know when it can free object?
  - o convenience: per-open-file data (file position, …)
- why not just use path for all operations (e.g., read(path, offset, …))
  - efficiency – string operations, protection checks

2.2 Caching

Read and write end up calling disk block read/disk block writes

We've stated several times that we need good caching for file systems to work well. How does this work?

Simple answer: block cache

Replace all uses of

```
ReadDisk(blockNum, buffer)
```

With

```
ReadDiskCache(blockNum, buffer){
    ptr = cache.get(blockNum); // just a hash table
    if(ptr){
        copy BLKSIZE bytes from ptr to buffer
    }
    else{
        newBuf = malloc(BLKSIZE);
        ReadDisk(blockNum, newBuf);
        cache.insert(blockNum, newBuf);
        copy(blockNum, buffer, BLKSIZE);
    }
```

}

Advantage: simple – write all FS code as if always reading from disk and insert the cache at the lowest level

Issues: replacement policy --> in a few weeks when we talk about memory systems

Disadvantage: copy overhead – each read copies block into a new buffer

For in-kernel use, we could return a pointer to cached version

- More complex: need to deal with reference counting, etc., but we could make it work…

What about avoiding copies to user space?

2.3 Mmap interface

void *mmap(int fd, size length, …)

map the specified open file into a region of my virtual memory, and return a pointer to that region

How might we implement this?
How would we update your page table?
How do I read a file?
How do I write a file?
What happens if a page is evicted from the cache?
What happens if a page is brought back into the cache?

****

Admin - 3 min

****

Midterm postmortem
Guest lecture thursday
Project 4 out. Start early.

****
Lecture - 23 min
****

3. RAIDS and availability

moved to S7.doc

****
Summary - 1 min
****