CS 134:
Operating Systems
File System Implementation

2013-05-17

CS34

CS 134:
Operating Systems
File System Implementation

Overview

Overview

Implementation Issues
Caching
Failures
Disk Scheduling

API
What Goes in an API?
Two Strange APIs
Consistency
Other File Operations

Disk Caching—Class Exercise

If OS caches blocks of a file in memory,

- How should it track what it's caching?
- How should it decide what to cache?

Disk Caching—Class Exercise
If OS caches blocks of a file in memory,
How should it track what it's caching?
How should it decide what to cache?

There's a close relationship to paging algorithms.

Disk Buffering

Allow writes to return immediately

- Copy data into a buffer
- Write out to the disk

Buffers vs. Caches

Which do we actually *need*?

Disk Buffering
Allow writes to return immediately
Copy data into a buffer
Write out to the disk
Buffers vs. Caches
Which do we actually *need*?

We *need* some kind of buffer (the write may not be the same size as the block).

We don’t need the cache; it just improves performance.

Buffer Cache

Store disk blocks waiting to write in *buffer cache*

- "Free" buffers used as cache for blocks recently read
- Dirty buffers will eventually be written to disk
- Allow buffer cache to use any free memory in the machine
- Ensure that a certain number of buffers are always available

Class Exercises & Reminders

When should we write the dirty buffers?

Per-process or system-wide?

Remind you of anything?

Buffer Cache

Store disk blocks waiting to write in *buffer cache*

- "Free" buffers used as cache for blocks recently read
- Dirty buffers will eventually be written to disk
- Allow buffer cache to use any free memory in the machine
- Ensure that a certain number of buffers are always available

Class Exercises & Reminders

When should we write the dirty buffers?

Per-process or system-wide?

Remind you of anything?

Buffer Cache (cont.)

One buffer cache for all processes and all block devices

- Local disks
- Remote disks
- Tapes, CD-ROMs, etc.

Buffer Cache (cont.)
One buffer cache for all processes and all block devices
- Local disks
- Remote disks
- Tapes, CD-ROMs, etc.

Dealing with System Failure—Class Exercise

Extending the file by one block. . .

In what order should we update the structures on disk to cause minimum damage if the system crashes?

(Assume disk will never leave a block half-written. . .)

What about

- File creation?
- File deletion?

Dealing with System Failure—Class Exercise

Extending the file by one block. . .

In what order should we update the structures on disk to cause minimum damage if the system crashes?

(Assume disk will never leave a block half-written. . .)

What about

- File creation?
- File deletion?

Recovering from System Failure

Before system failure. . .

- Backups!

After system failure. . .

- Run consistency checker—compares data in directory structure with data blocks on disk, tries to fix inconsistencies.
- Recover lost files (or disk) by restoring data from backup

Recovering from System Failure

Before system failure. . .
- Backups!

After system failure. . .
- Run consistency checker—compares data in directory structure with data blocks on disk, tries to fix inconsistencies.
- Recover lost files (or disk) by restoring data from backup

Disk Scheduling

OS needs to use all I/O devices efficiently:

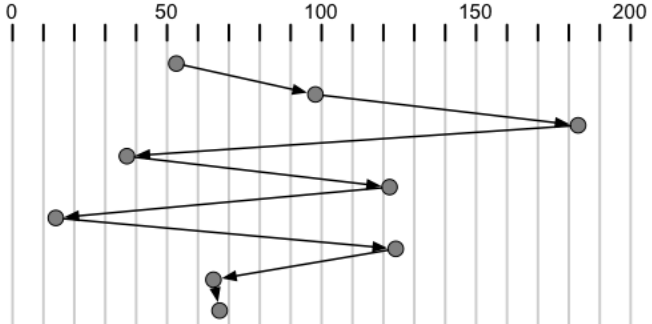
- Minimize access time—composed of
  - *Seek time*
  - *Rotational latency*
- Maximize disk bandwidth
  - Bandwidth = Total Bytes Transferred / Total Time Taken
- Usually, disk requests can be re-ordered
- Several algorithms exist to schedule disk I/O requests
  - Consider, e.g., cylinders 98, 183, 37, 122, 14, 124, 65, 67 and an initial head position of 53

Disk Scheduling

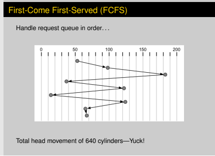
OS needs to use all I/O devices efficiently:
- Minimize access time—composed of
  - *Seek time*
  - *Rotational latency*
- Maximize disk bandwidth
  - Bandwidth = Total Bytes Transferred / Total Time Taken
- Usually, disk requests can be re-ordered
- Several algorithms exist to schedule disk I/O requests
  - Consider, e.g., cylinders 98, 183, 37, 122, 14, 124, 65, 67 and an initial head position of 53

First-Come First-Served (FCFS)

Handle request queue in order. . .

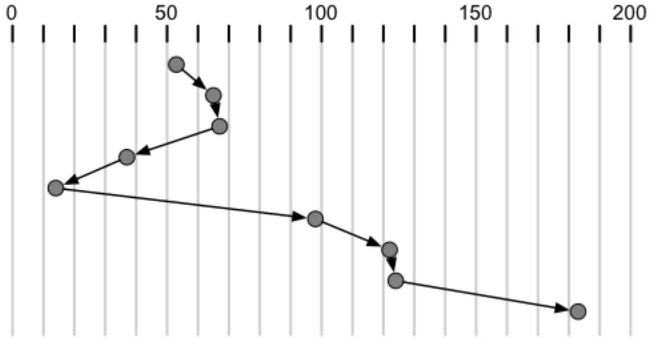


Total head movement of 640 cylinders—Yuck!

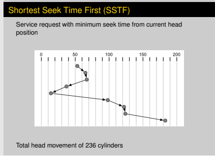


Shortest Seek Time First (SSTF)

Service request with minimum seek time from current head position



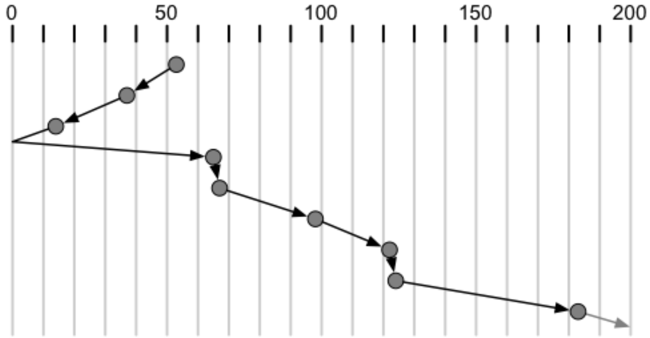
Total head movement of 236 cylinders



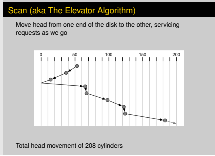
SSTF may starve requests (why?).

Scan (aka The Elevator Algorithm)

Move head from one end of the disk to the other, servicing requests as we go

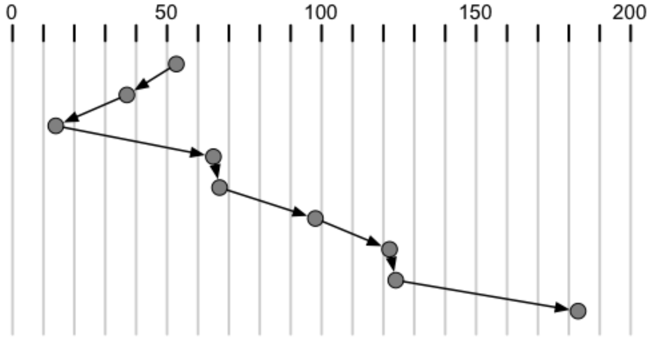


Total head movement of 208 cylinders

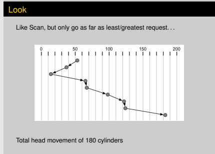


Look

Like Scan, but only go as far as least/greatest request. . .

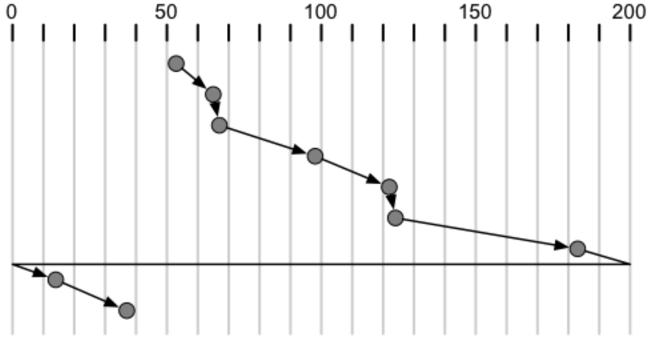


Total head movement of 180 cylinders

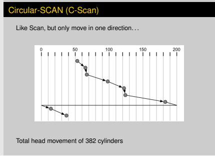


Circular-SCAN (C-Scan)

Like Scan, but only move in one direction...



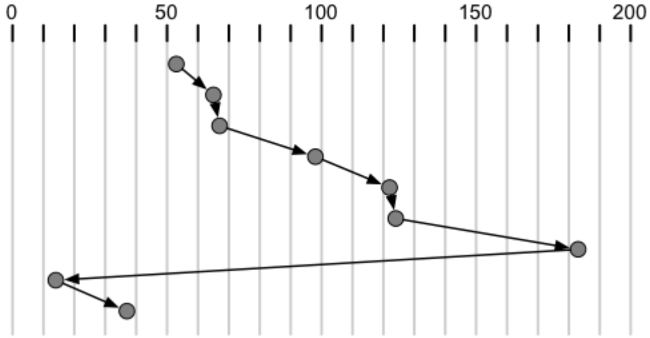
Total head movement of 382 cylinders



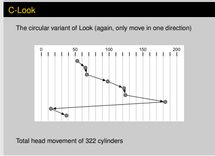
Why do this? It wastes head movement...

C-Look

The circular variant of Look (again, only move in one direction)



Total head movement of 322 cylinders



Fairness

What if we had two processes, each producing the following requests

- 0, 5, 10, . . . , 75
- 125, 130, 135, . . . , 200

How fair would these techniques be?

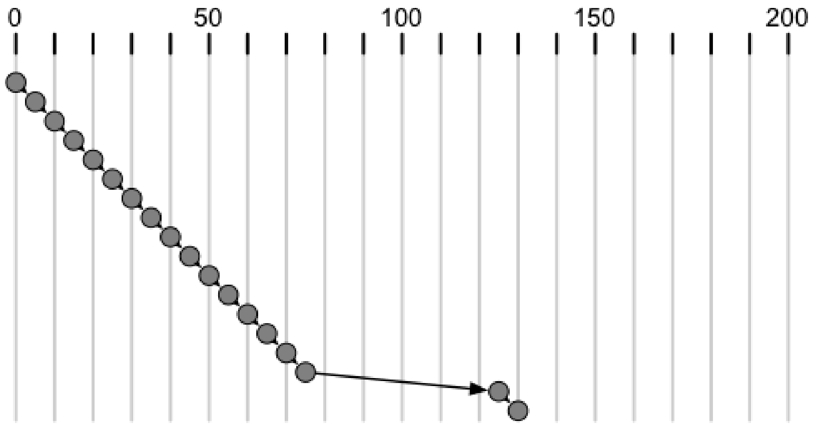
Fairness

What if we had two processes, each producing the following requests

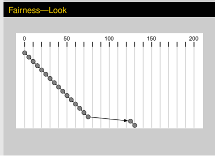
- 0, 5, 10, . . . , 75
- 125, 130, 135, . . . , 200

How fair would these techniques be?

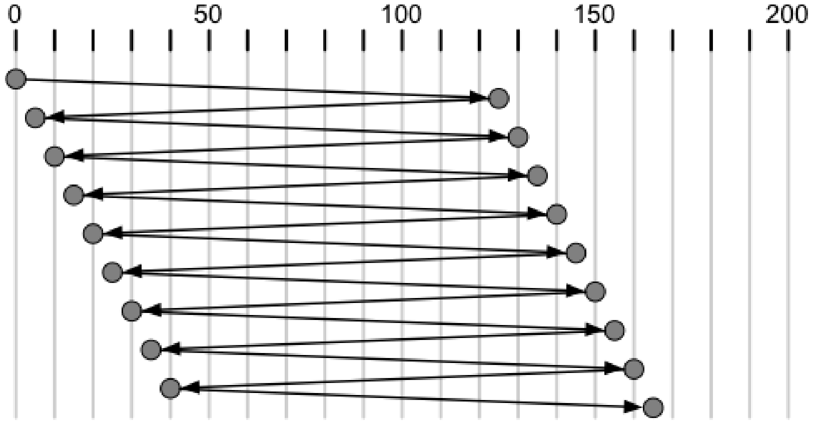
Fairness—Look

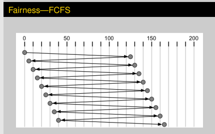


2013-05-17
CS34
Implementation Issues
Disk Scheduling
Fairness—Look

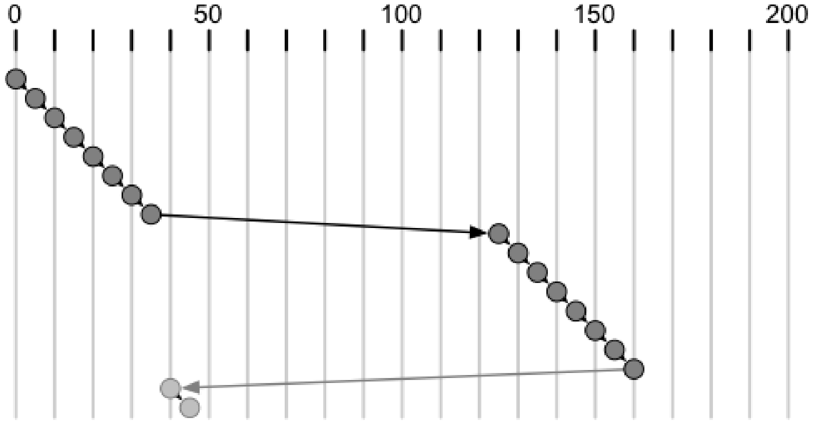


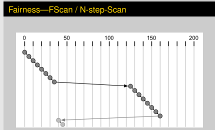
Fairness—FCFS





Fairness—FScan / N-step-Scan





Selecting a Disk-Scheduling Algorithm

Comparing algorithms, we find that:

- FCFS can be okay **if** disk controller manages the scheduling (many do these days)
- SSTF is common and has natural appeal
- LOOK works well (lowest amount of head moment in our test)
- LOOK and C-LOOK perform better for systems under heavy load

Selecting a Disk-Scheduling Algorithm

Comparing algorithms, we find that:
- FCFS can be okay **if** disk controller manages the scheduling (many do these days)
- SSTF is common and has natural appeal
- LOOK works well (lowest amount of head moment in our test)
- LOOK and C-LOOK perform better for systems under heavy load

Modern Disk Geometry

Modern disks maximize utilization by varying the number of sectors per cylinder

- Logical block→(Sector, Track/Cylinder)?
  - Complex or impossible for operating system to calculate!

Sometimes tracks are even laid out in a zig-zag pattern

Consequences?

Modern Disk Geometry
Modern disks maximize utilization by varying the number of sectors per cylinder
Logical block→(Sector, Track/Cylinder)?
Complex or impossible for operating system to calculate!
Sometimes tracks are even laid out in a zig-zag pattern
Consequences?

Disk scheduling becomes impossible to perform perfectly. But approximations work reasonably well. Or we can just hand it off to the disk (modern disks are *smart*).

But careful placement of vital data across platters may not work out.

File-Access API

Class Exercise

What operations should we provide for accessing files?

Describe & Develop

- Basic requirements for a file access API
- A stateful interface that satisfies those requirements
- A *stateless* interface that satisfies them

File-Access API
Class Exercise
What operations should we provide for accessing files?
Describe & Develop
- Basic requirements for a file access API
- A stateful interface that satisfies those requirements
- A *stateless* interface that satisfies them

This is a lengthy exercise; they should divide into groups and do it on paper.

Stateful File Access

OS maintains some "context" for each open file...

File access

- *handle* = `open` (*filename, mode*)
- `read` (*handle, length, buffer*)
- `write` (*handle, length, buffer*)
- `truncate` (*handle, length*)
- `seek` (*handle, position*)
- `close` (*handle*)

File management

- `info` (*name, info*)
- `delete` (*name*)
- `change_directory` (*dirname*)
- `create_dir` (*dirname*)
- `move` (*name, name*)

Stateful File Access

OS maintains some "context" for each open file...

File access
- handle = open (filename, mode)
- read (handle, length, buffer)
- write (handle, length, buffer)
- truncate (handle, length)
- seek (handle, position)
- close (handle)

File management
- info (name, info)
- delete (name)
- change_directory (dirname)
- create_dir (dirname)
- move (name, name)

Stateless File Access

No (apparent) internal state—each system call fully describes the desired operation

File access

- `create` (*filename*)
- `read` (*filename, pos, length, buffer*)
- `write` (*filename, pos, length, buffer*)
- `truncate` (*filename, length*)

File management

- `info` (*name, info*)
- `delete` (*name*)
- `create_dir` (*dirname*)
- `move` (*name, name*)

Class Exercise

Contrast these two approaches. . .

Stateless File Access
No (apparent) internal state—each system call fully describes the desired operation
File access
create (filename)
read (filename, pos, length, buffer)
write (filename, pos, length, buffer)
truncate (filename, length)
File management
info (name, info)
delete (name)
create_dir (dirname)
move (name, name)
Class Exercise
Contrast these two approaches. . .

Completeness: Each method can simulate the other
Stateless operation:

- Simple
- Works well in a multi-threaded program
- Basis for NFS

Stateful operation:

- Assumes file-locality and sequential access is common
- Provides the operating system with more information about which files are being used
- Maps well to other kinds of device besides files (e.g., read/write to a terminal)
- May add arbitrary limitations (e.g, maximum open files)

Michigan Terminal System (60's)

- Files divided into variable-length "lines"
  - Even binary files made up of lines
  - Each line numbered (fixed-point, 6 fractional decimal digits)
  - Could read by line number or by "next line"
  - Could write by line number (inserting in middle if appropriate) or (?) just append at end
- No user access to devices
  - E.g., print by creating file, then handing to OS
  - Slightly problematic when terminals introduced…

Michigan Terminal System (60's)
- Files divided into variable-length "lines"
  - Even binary files made up of lines
  - Each line numbered (fixed-point, 6 fractional decimal digits)
  - Could read by line number or by "next line"
  - Could write by line number (inserting in middle if appropriate) or (?) just append at end
- No user access to devices
  - E.g., print by creating file, then handing to OS
  - Slightly problematic when terminals introduced…

RSX-11M/VMS (70's)

- "inodes" exposed to application
- Directories identified by OS but accessed with file API
- OS responsible for allocating blocks upon request
- Complex read/write interface (asynchronous, fixed/variable records, devices treated separately)
- Much of file system in library

RSX-11M/VMS (70's)

- "inodes" exposed to application
- Directories identified by OS but accessed with file API
- OS responsible for allocating blocks upon request
- Complex read/write interface (asynchronous, fixed/variable records, devices treated separately)
- Much of file system in library

Consistency Model

Additional complications:

- Multiple processes can access files
  —Two (or more) processes could read and write same file
- Asynchronous writes + errors = ?
- What if file is moved/renamed/deleted while a process is using it?

What should the rules be?

Consistency Model

Additional complications:

- Multiple processes can access files
  —Two (or more) processes could read and write same file
- Asynchronous writes + errors = ?
- What if file is moved/renamed/deleted while a process is using it?

What should the rules be?

Class Exercise

Develop and justify a consistency model for file operations.

Class Exercise

Develop and justify a consistency model for file operations.

Class Exercise

Develop and justify a consistency model for file operations.

Develop another one.

Class Exercise

Develop and justify a consistency model for file operations.

Develop another one.

Unix Consistency Model

Unix uses the following rules:

- All file operations are globally atomic
- File is only deleted when its link (name) count is zero—an `open` counts as a link
- `write` in one process is globally visible immediately afterwards
- `write`s are asynchronous—I/O errors may not be discovered until the file is `close`d

Unix Consistency Model

Unix uses the following rules:

- All file operations are globally atomic
- File is only deleted when its link (name) count is zero—an `open` counts as a link
- `write` in one process is globally visible immediately afterwards
- `write`s are asynchronous—I/O errors may not be discovered until the file is `close`d

These rules do not map well onto a stateless I/O interface.

NFSv2 Consistency Model

Classic NFS uses the following rules:

- All file operations are globally atomic and stateless
- `move`/`rename`/`delete` can disrupt accesses performed by other processes
- `write` in one process is globally visible immediately afterwards
- `write`s are synchronous—I/O errors are discovered immediately

NFSv2 Consistency Model
Classic NFS uses the following rules:
- All file operations are globally atomic and stateless
- move/rename/delete can disrupt accesses performed by other processes
- write in one process is globally visible immediately afterwards
- writes are synchronous—I/O errors are discovered immediately

These rules map well onto a stateless I/O interface, but are not entirely consistent with the POSIX file model.

File Locking

What if we want to lock sections of a file?

File Locking

What if we want to lock sections of a file?

File Access—Leveraging the VM System

Use virtual memory system to provide file access

- “Map” file into memory
- Page faults retrieve file’s data from disk
- Provide copy-on-write or writeback semantics

Add the following system calls:

- `map_file` (*handle, size, mode, address*)
- `unmap_file` (*address*)

(This mechanism doesn’t allow extending or truncating the file.)

File Access—Leveraging the VM System

Use virtual memory system to provide file access
- “Map” file into memory
- Page faults retrieve file’s data from disk
- Provide copy-on-write or writeback semantics

Add the following system calls:
- `map_file` (*handle, size, mode, address*)
- `unmap_file` (*address*)

(This mechanism doesn’t allow extending or truncating the file.)

More File Access

Besides calls for protection, there are other system calls we might want:

- Control owner, permissions, etc.
- Control file caching
- Find (remaining) capacity of the disk
- Eject disk
- Discover whether any reads/writes have failed
- . . .

More File Access

Besides calls for protection, there are other system calls we might want:

- Control owner, permissions, etc.
- Control file caching
- Find (remaining) capacity of the disk
- Eject disk
- Discover whether any reads/writes have failed
- . . .

Generic Mechanisms

Two approaches

- `ioctl(`*handle, request, buffer*`)`
- Pseudo-files and pseudo-filesystems

Generic Mechanisms

Two approaches
- ioctl(handle, request, buffer)
- Pseudo-files and pseudo-filesystems

Ioctl is clumsy, hard to use, prone to inconsistency. Pseudo-files aren't good for adding one new feature (such as locking).