Today

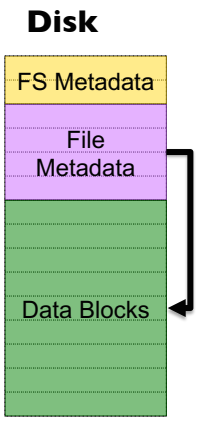
- File System Performance
  - Disk scheduling
- File System Reliability
  - RAID

Review

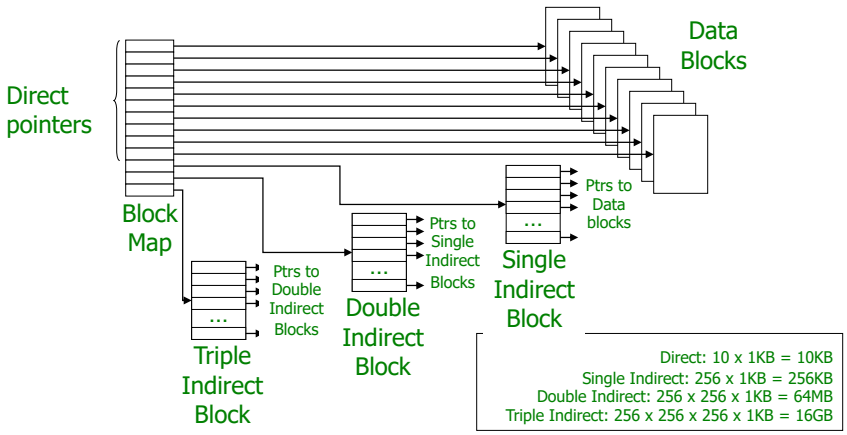
- A disk is a bunch of blocks in which to store data
  - How does FS give order and structure to those blocks?
- How do inodes handle the wide variety of file sizes?
  - What are the benefits of this design?
- How are files within the file system structured?
  - How are they structured in our OS project? On most modern OSs?
- How does the user interact with the file system?
  - What does the FS do in response?
- How can an OS handle multiple file systems within one name space?
  - What are the tradeoffs of this approach?

Review: Disk

- File System Metadata
  - Format, size of blocks
  - Stored in *superblock*
  - Replicated
- File Metadata
  - Inode table
- Data Blocks

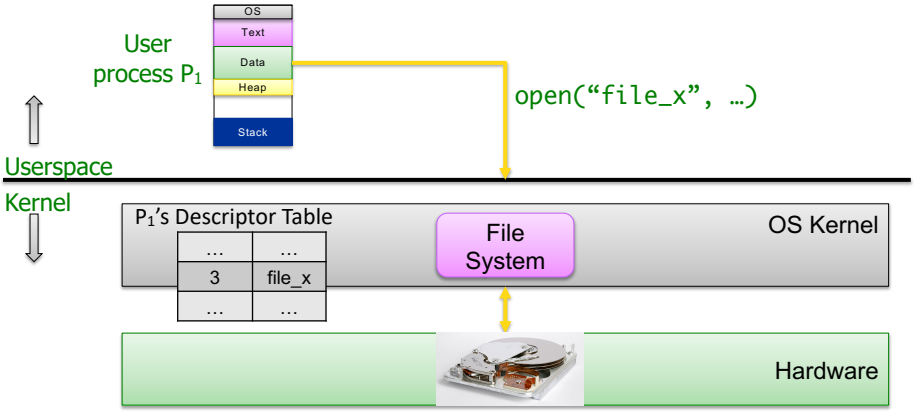


Review: Block Pointers: Multi-Level Table



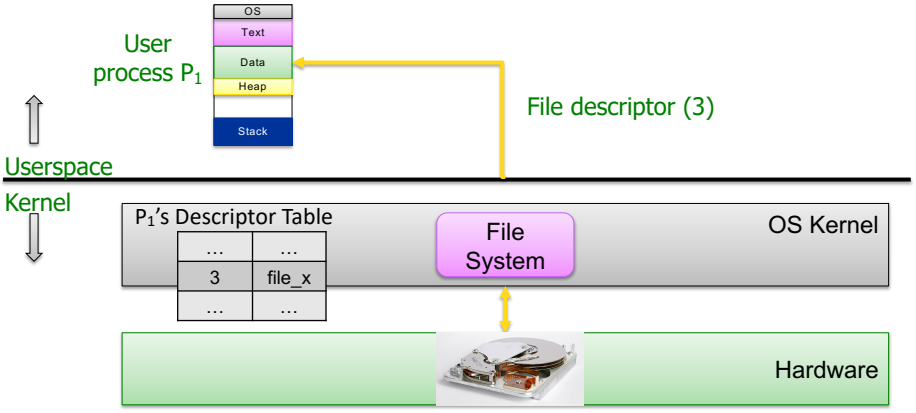
Review: Userspace Perspective

- Userspace processes make system calls to interact with files:



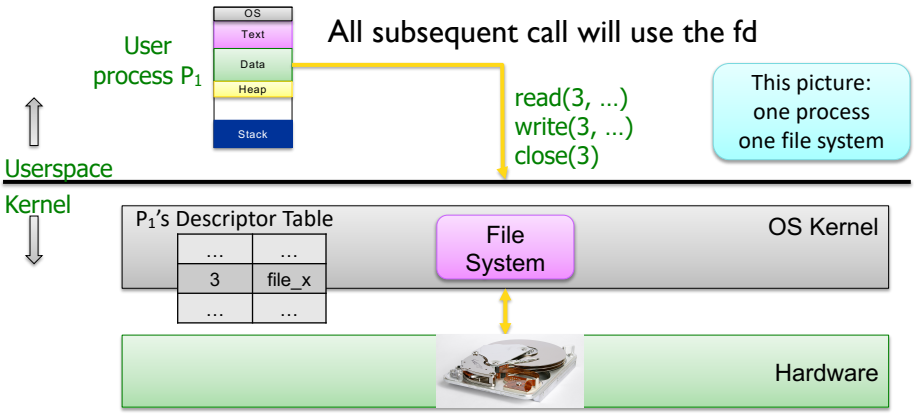
Review: Userspace Perspective

- Userspace processes make system calls to interact with files:



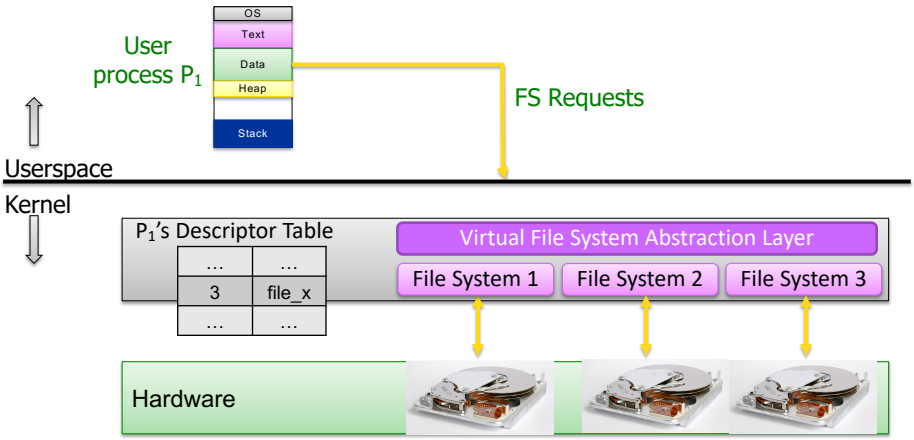
Review: Userspace Perspective

- Userspace processes make system calls to interact with files:



Review: Virtual File System (VFS) Layer

- Userspace processes make system calls to interact with files:



VFS Layer

- Unifies the file name space and paths
  - Paths all start from common root (/) and are passed to VFS layer
  - VFS layer records which paths correspond to which FS
- VFS translates application requests to appropriate low-level FS calls

Analyzing VFS Layer

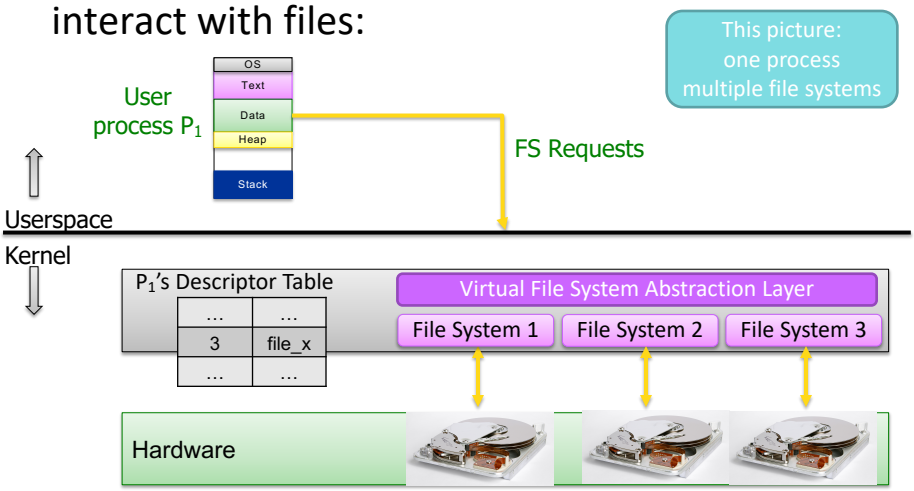
- Benefits
  - user doesn't need to know the details about file systems
  - easy expansion, removal of disks/file systems
- Drawback: layer adds overhead – could slow down performance

Analyzing VFS Layer

- How can we mitigate that performance hit?
  - Caching!
  - Inode Cache
    - Store recently accessed inodes (file/directory info)
  - Directory Cache
    - Full directory path → inode id

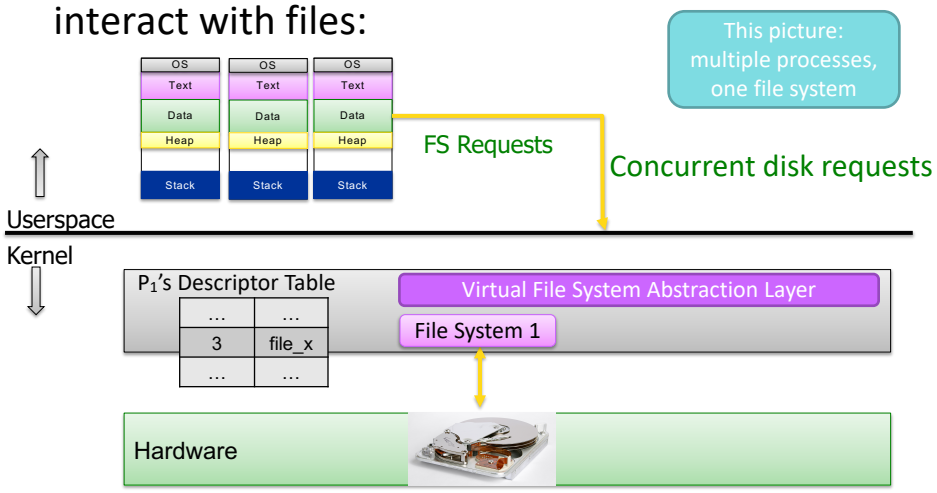
Virtual File System (VFS) Layer

- Userspace processes make system calls to interact with files:



Multiple Concurrent Disk Requests

- Userspace processes make system calls to interact with files:



Disk Scheduling

- Many sources of disk I/O requests
  - OS
  - System processes
  - User processes
- I/O request includes input or output mode, disk address, memory address, number of sectors to transfer
- OS maintains queue of requests, per disk or device
  - Idle disk can immediately work on I/O request
  - Busy disk means request must queue

Disk Scheduling

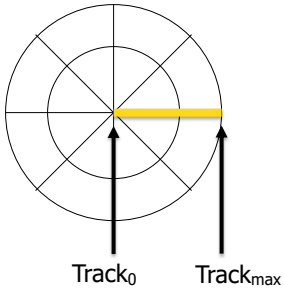
- Unlike CPU scheduling, disk characteristics vary significantly
  - CPUs have different ISAs, but they *mostly* behave the same
- For certain types of disks (solid state), FIFO might make a lot of sense (when targeting throughput):
  - The disk has no moving parts, so the fastest thing to do is just issue requests immediately as they come in
  - Maybe merge adjacent requests
- For traditional spinning disks?

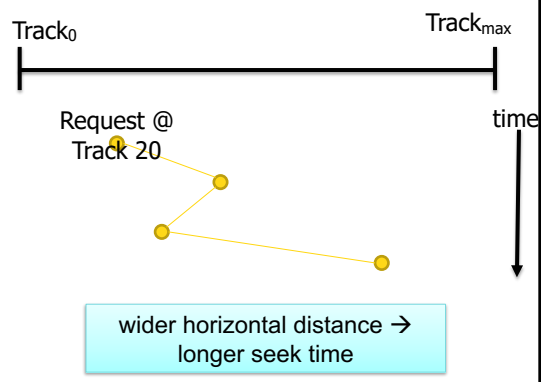
Disk Scheduling

- The operating system is responsible for using hardware *efficiently*
  - For the disk drives: having a fast access time and disk bandwidth
- Minimize seek time
  - Seek time ≈ seek distance
- **Disk bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer

Disk Arm

- Assume the disk arm can move back and forth from left to right and right to left.





Optimizing Disk Scheduling

- Goal: optimize performance
  - First: disk bandwidth
  - Any other concerns?

Disk Scheduling

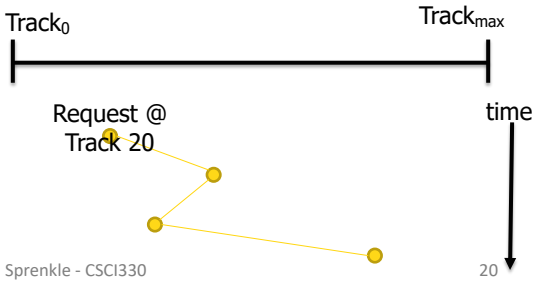
- Like CPU scheduling, disk scheduling is a policy decision
  - What should happen if multiple processes all want to access disk?
- Like CPU scheduling, the choice of metric influences the policy decision:
  - Priority: if a process is important, give execute its requests first
  - Wait time/latency, from request to completion
  - Throughput: maximize the data transfer from the disk
  - Fairness: give each process the same opportunity to access the disk

In some cases, trade-off between last two.

Optimizing Disk Scheduling

- How can we optimize disk scheduling?
- What are possible algorithms?
  - What are their tradeoffs?
  - For each algorithm, write down pros and cons
- What concerns/questions do we have in picking an algorithm?

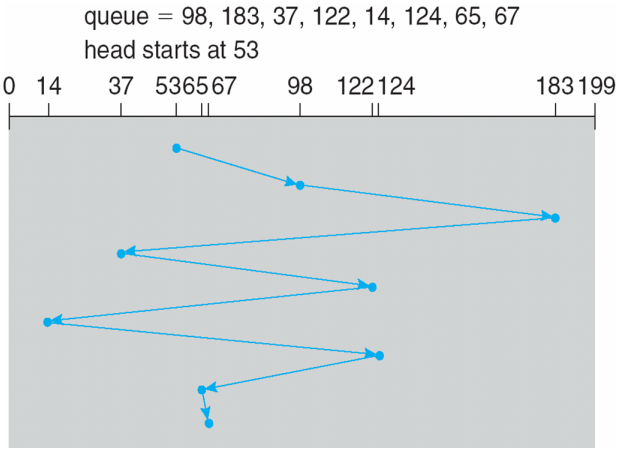
What would your algorithms look like in this form?



Disk Scheduling

- Several algorithms exist to schedule the servicing of disk I/O requests
- The analysis is true for one or many platters
- Consider a request queue
  - 98, 183, 37, 122, 14, 124, 65, 67
  - Head pointer is at 53

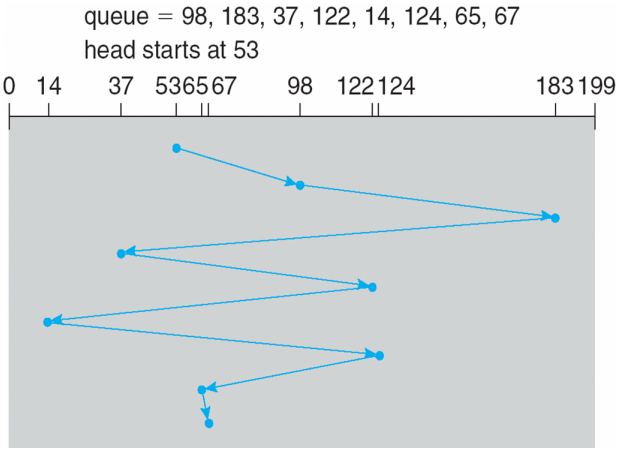
FCFS: First Come First Serve



Total head movement of 640 cylinders

Analyze the performance of this algorithm, specifically here and in general.

FCFS: First Come First Serve

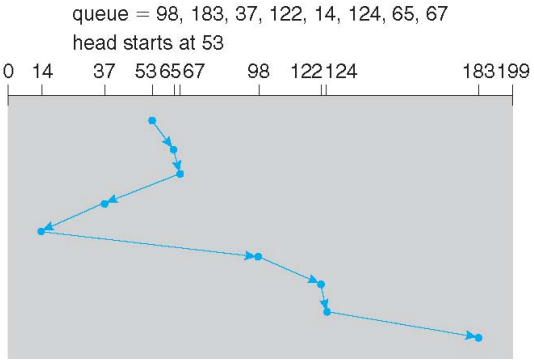


Total head movement of 640 cylinders

Performance is highly variable:
depends on the order and track location of requests

Shortest Seek Time First (SSTF)

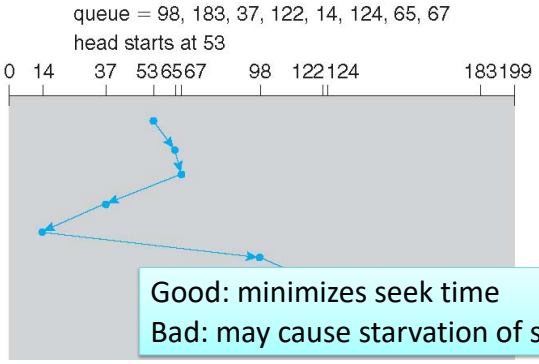
- SSTF selects request with the minimum seek time from the current head position
- SSTF scheduling is a form of SJF scheduling



Total head movement of 236 cylinders

Shortest Seek Time First (SSTF)

- SSTF selects request with the minimum seek time from the current head position
- SSTF scheduling is a form of SJF scheduling



Good: minimizes seek time
Bad: may cause starvation of some requests

Total head movement of 236 cylinders

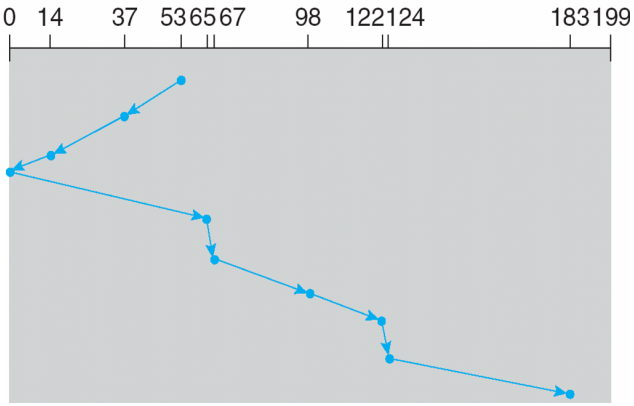
SCAN Algorithm

- Sometimes called the elevator algorithm
- The disk arm starts at one end of the disk, and moves toward the other end
  - services requests until it gets to the other end of the disk
  - head movement is reversed and servicing continues

SCAN Algorithm

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

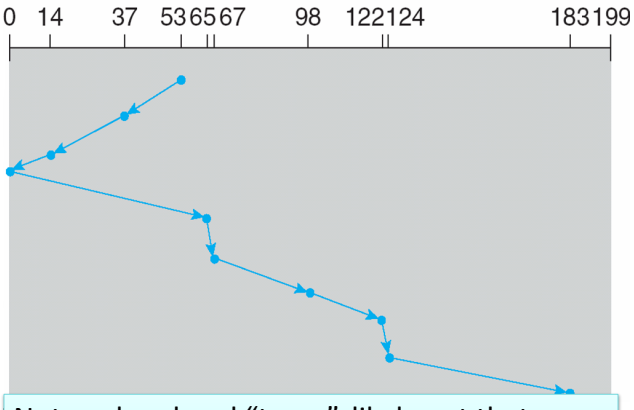


Total head movement of 208 cylinders

SCAN Algorithm

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

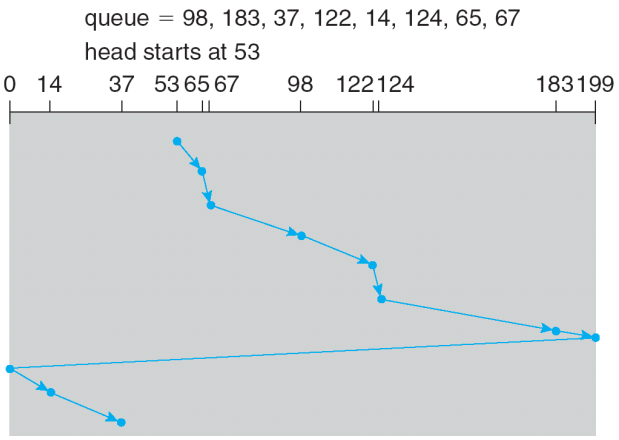


Note: when head "turns", likely not that many requests are right there. The largest density is at other end of disk and wait the longest. So…

C-SCAN

- Provides a more uniform wait time than SCAN
- Head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a *circular* list that wraps around from the last cylinder to the first one

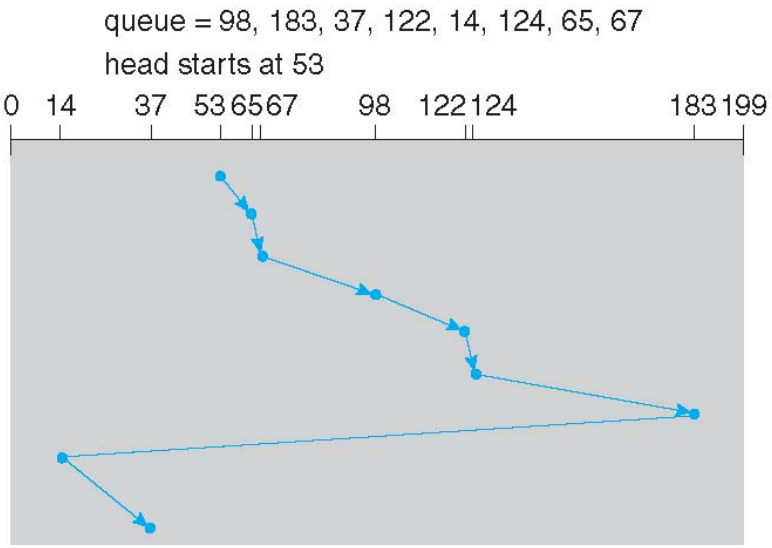
C-SCAN



C-LOOK

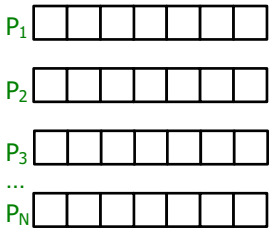
- LOOK a version of SCAN, C-LOOK a version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk

C-LOOK

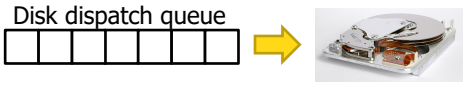


Linux's Default Scheduler: CFQ

- Completely Fair Queueing (CFQ)
  - Not to be confused with the "completely fair scheduler (CFS)" for the CPU…



- Keep a disk request queue for each process
- Move requests from process queues to dispatch queue in round-robin fashion (if equal priority)



Linux's Default Scheduler: CFQ

- Completely Fair Queueing (CFQ)
  - Not to be confused with the "completely fair scheduler (CFS)" for the CPU…

- Keep a disk request queue for each process
- Move requests from process queues to dispatch queue in round-robin fashion (if equal priority)

Can reorder these requests to improve disk performance within some constraints

Selecting a Disk-Scheduling Algorithm

- D-S algorithm should be written as a separate module of the operating system
  - Can be replaced with a different algorithm if necessary
- Performance depends on the number and types of requests
  - If only one request in queue – becomes FCFS
- SCAN is common
  - Less computational cost
- SCAN and C-SCAN perform better for systems with heavy load on the disk
  - Less starvation
- Add a deadline scheduler
  - if a request hasn't been fulfilled within some period of time, service it

How Persistent Storage Affects OS Design

Goal	Physical Characteristics	Design Implication
High Performance	• Large cost to initiate I/O	• Organize storage to access data in large sequential units • Use caching
Named Data	• Large capacity • Survives crashes • Shared across programs	• Support files and directories with meaningful names
Controlled Sharing	• Device may store data from many users	• Include metadata for access control
Reliability	• Crash can occur during updates • Storage devices can fail • Flash memory wears out	• Use transactions • Use redundancy to detect and correct failures • Migrate data to even the wear

David Patterson Garth Gibson Randy Katz

Redundant Array of Inexpensive Disks

RAID

Idea: Replace Small Number of Large Disks with Large Number of Small Disks! (1988 Disks)

	Big, Expensive IBM 3390K	Small, Cheap IBM 3.5" 0061	Small, Cheap x70	
Capacity	20 GBytes	320 MBytes	23 GBytes	
Volume	97 cu. ft.	0.1 cu. ft.	11 cu. ft.	9X
Power	3 KW	11 W	1 KW	3X
Data Rate	15 MB/s	1.5 MB/s	120 MB/s	8X
I/O Rate	600 I/Os/s	55 I/Os/s	3900 IOs/s	6X
MTTF	250 KHrs	50 KHrs	??? Hrs	
Cost	$250K	$2K	$150K	

Disk Arrays have potential for large data and I/O rates, high MB per cu. ft., high MB per KW

But what about reliability?

Array Reliability

- Reliability of N disks = Reliability of 1 Disk ÷ N
  - 50,000 Hours ÷ 70 disks = 700 hours
  - Disk system MTTF: drops from 6 years→1 month!
- Arrays (without redundancy) too unreliable to be useful!

Hot spares support reconstruction in parallel with access: very high media availability can be achieved

Looking Ahead

- Project 4 due Monday after Thanksgiving