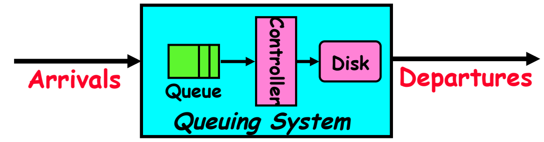
CS162
Operating Systems and Systems Programming
Lecture 18

File Systems, Naming, and Directories

March 30, 2010

Ion Stoica
http://inst.eecs.berkeley.edu/~cs162

Introduction to Queuing Theory



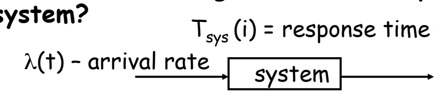
- **Model:**
  - Task arrives at a certain rate, i.e., arrival rate
  - Only one task is processed at a time
  - Tasks waits in FIFO queue to be processed
- **Parameters:**
  - Queueing (waiting) time (T_q): time a task waits in the queue
  - Service time (T_{ser}): time it takes to process the task
  - Response (system) time (T_{sys}): total time a task spends in the system

$$T_{sys} = T_q + T_{ser}$$

- Queuing Theory applies to long term, steady state behavior
  - Typical queuing theory doesn't deal with transient behavior

Little's Theorem

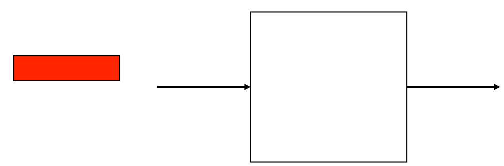
- Apply to virtual any system, e.g., disk, router, network, checkout line in a supermarket
- $\lambda(t)$: arrival rate of requests (tasks)
- T_{sys} (i): system (response) time of request
- What is the average number of requests in the system?



- Note: apply to the number of requests waiting in the queue as well
- Intuition:
  - Assume arrival rate is λ = *1* request per second and the response time of each request is T_{sys} = *4* seconds
  - What is the average number of requests in the system?

Example

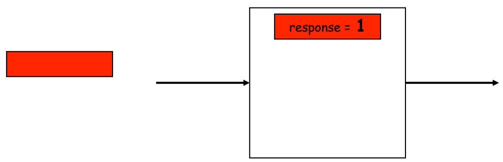
- Arrival rate = 1; response (system) time = 4



Time = 0

Example

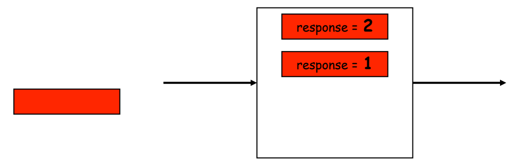
- **Arrival rate = 1; response (system) time = 4**



Time = 1

Example

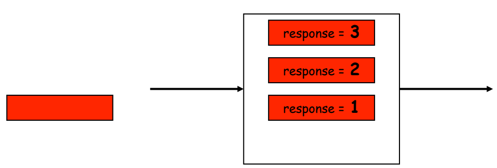
- **Arrival rate = 1; response (system) time = 4**



Time = 2

Example

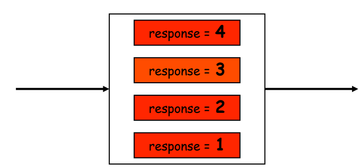
- **Arrival rate = 1; response (system) time = 4**



Time = 3

Example

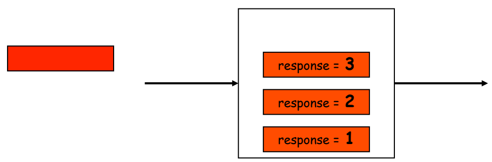
- **Arrival rate = 1; response (system) time = 4**



Time = 4

Example

- Arrival rate = 1; response (system) time = 4



Time = 4

Q: What is the average number of requests in system?

A: number_of_requests_in_system = avg_arrival_rate x avg_response

Little Theorem (cont'd)

- Applies to any arrival time distribution
- Applies to any service time distribution
- Assumptions:
  - Queue large enough: requests are not dropped
  - Steady state system:
    - » Arrival rate and service time distribution do not change
    - » Enough capacity to process all requests: queue does not increase indefinitely

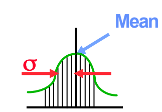
Goals for Today

- Queuing Theory: Continued
- File Systems
  - Structure, Naming, Directories

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from lecture notes by Kubiatowicz.

Random distributions

- Random variable: a variable (x) that takes some value (x_i) with a given probability (p_i)
- Random distribution: set of values and their probabilities
- Server spends variable time with customers
  - x_i: service time for request i
  - p_i: probability service time of a request is x_i
  - Mean (expected value): $\mu = E(x) = \Sigma p_i x_i$
  - Variance: $\sigma^2 = \Sigma p_i (x_i - \mu)^2$



Distribution of service times

$$\sigma^2 = \sum_{i=1}^{n} p_i (x_i - \mu)^2$$

$$= \sum_{i=1}^{n} p_i (x_i^2 - 2\mu x_i + \mu^2) = \sum_{i=1}^{n} p_i x_i^2 - 2\mu \sum_{i=1}^{n} p_i x_i + \mu^2$$

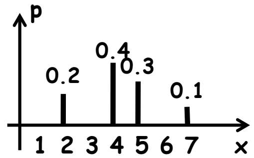
$$= \sum_{i=1}^{n} p_i x_i^2 - 2\mu^2 + \mu^2 \quad = \sum_{i=1}^{n} p_i x_i^2 - \mu^2$$

$$= E(x^2) - E(x)^2$$

Random Distribution (example)

- Consider following distribution

x_i	2	4	5	7
p_i	0.2	0.4	0.3	0.1



- Mean (expected value):

  $\mu = E(x) = \Sigma p_i x_i = 0.2*2 + 0.4*4 + 0.3*5 + 0.1*7 = \mathbf{4.2}$

- Variance:

  $\sigma^2 = \Sigma p_i(x_i - \mu)^2 = 0.2*(2-4.2)^2 + 0.4*(4-4.2)^2 + 0.3*(5-4.2)^2 + 0.1*(7-4.2)^2 = \mathbf{1.96}$

- Variance (2nd method): $\sigma^2 = E(x^2) - E(x)^2$

  $E(x2) = 0.2*2^2 + 0.4*4^2 + 0.3*5^2 + 0.1*7^2 = 19.6$

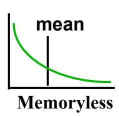
  $E(x^2) - E(x)^2 = 19.6 - 4.2^2 = 19.6 - 17.64 = \mathbf{1.96}$

Administrivia

- Group Evaluations not Optional
  - You will get a zero for project if you don't fill them out!
  - We use these for grading
- Check glookup to make sure that we have right grades
  - Make sure that we don't have errors

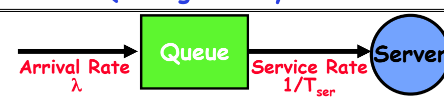
Coefficient of Variation

- Squared coefficient of variance: $C = \sigma^2/\mu^2$
  Aggregate description of the distribution
  - Previous example: $C = 1.96/17.64 = 0.111\ldots$
- Important values of C:
  - No variance or deterministic ⇒ $C=0$
  - "memoryless" or exponential ⇒ $C=1$
    - » Past tells nothing about future
    - » Many complex systems (or aggregates) well described as memoryless
  - Disk response times $C \approx 1.5$ (majority seeks < avg)



- Mean Residual Wait Time, m1(z):
  - Mean time must wait for server to complete current task
  - Can derive $m1(z) = \frac{1}{2}\mu\times(1 + C)$
    - » Not just $\frac{1}{2}\mu$ because doesn't capture variance
  - $C = 0 \Rightarrow m1(z) = \frac{1}{2}\mu$; $C = 1 \Rightarrow m1(z) = \mu$

A Little Queuing Theory: Mean Wait Time



- Parameters that describe our system:
  - λ: mean number of arriving customers/second
  - T_{ser}: mean time to service a customer ("μ")
  - C: squared coefficient of variance = σ^2/μ^2
  - u: server utilization ($0 \le u \le 1$): $u = \lambda/(1/T_{ser}) = \lambda \times T_{ser}$
- Parameters we wish to compute:
  - T_q: Time spent in queue
  - L_q: Length of queue = $\lambda \times T_q$ (by Little's law applied to waiting queue)
- Basic Approach:
  - Customers before us must finish; mean time = $L_q \times T_{ser}$
  - *If something at server*, takes m1(z) to complete on avg
    - » m1(z): mean residual wait time at server= $T_{ser} \times \frac{1}{2}(1+C)$
    - » Chance something at server = u ⇒ mean time is u × m1(z)
- Computation of wait time in queue (T_q):
  - $T_q = L_q \times T_{ser} + u \times m1(z)$

A Little Queuing Theory: M/G/1 and M/M/1

- Computation of wait time in queue (T_q):
  - $T_q = L_q \times T_{ser} + u \times m1(z)$ — Little's Law
  - $T_q = \lambda \times T_q \times T_{ser} + u \times m1(z)$
  - $T_q = u \times T_q + u \times m1(z)$ — Defn of utilization (u)
  - $T_q \times (1 - u) = m1(z) \times u \Rightarrow T_q = m1(z) \times u/(1-u) \Rightarrow$
  - $T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1 - u)$
- Notice that as $u \rightarrow 1$, $T_q \rightarrow \infty$!
- Assumptions so far:
  - System in equilibrium; No limit to the queue: works First-In-First-Out
  - Time between two successive arrivals in line are random and memoryless: (M for C=1 exponentially random)
  - Server can start on next customer immediately after prior finishes
- General service distribution (no restrictions), 1 server:
  - Called M/G/1 queue: $T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1 - u)$
- Memoryless service distribution ($C = 1$):
  - Called M/M/1 queue: $T_q = T_{ser} \times u/(1 - u)$

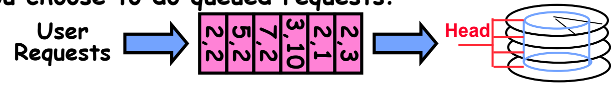
A Little Queuing Theory: An Example

- Example Usage Statistics:
  - User requests 10 × 8KB disk I/Os per second
  - Requests & service exponentially distributed (C=1.0)
  - Avg. service = 20 ms (controller+seek+rot+Xfertime)
- Questions:
  - How utilized is the disk?
    - » Ans: server utilization, $u = \lambda T_{ser}$
  - What is the average time spent in the queue?
    - » Ans: T_q
  - What is the number of requests in the queue?
    - » Ans: $L_q = \lambda T_q$
  - What is the avg response time for disk request?
    - » Ans: $T_{sys} = T_q + T_{ser}$ (Wait in queue, then get served)
- Computation:

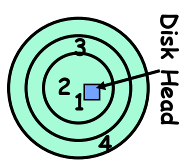
λ (avg # arriving customers/s) = 10/s
T_{ser} (avg time to service customer) = 20 ms (0.02s)
u (server utilization) = $\lambda \times T_{ser}$ = 10/s × .02s = 0.2
T_q (avg time/customer in queue) = $T_{ser} \times u/(1 - u)$
= 20 × 0.2/(1-0.2) = 20 × 0.25 = 5 ms (0 .005s)
L_q (avg length of queue) = $\lambda \times T_q$=10/s × .005s = 0.05
T_{sys} (avg time/customer in system) =$T_q + T_{ser}$= 25 ms

Disk Scheduling

- Disk can do only one request at a time; What order do you choose to do queued requests?



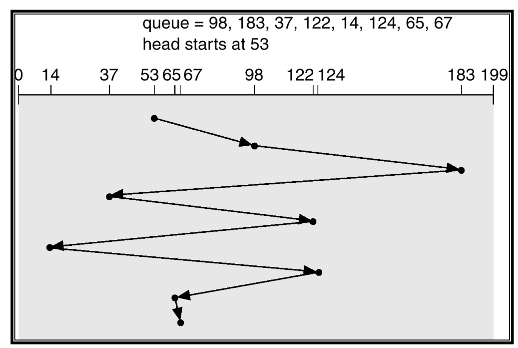
- Each request: [cylinder, sector]
- Scheduling discipline
  - FIFO Order
  - SSTF: Shortest seek time first
  - SCAN
  - C-SCAN: Circular-Scan



- Illustrate with an example:
  - Request list: 98, 183, 37, 122, 14, 124, 65, 67
  - Head starts: 53
  - Ignore sectors

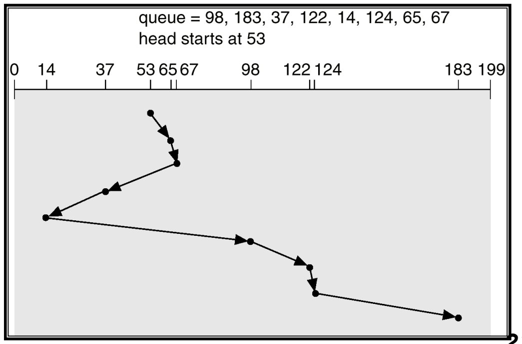
FIFO

- Fair among requesters, but order of arrival may be to random spots on the disk ⇒ Very long seeks
- Head movement of 640 cylinders



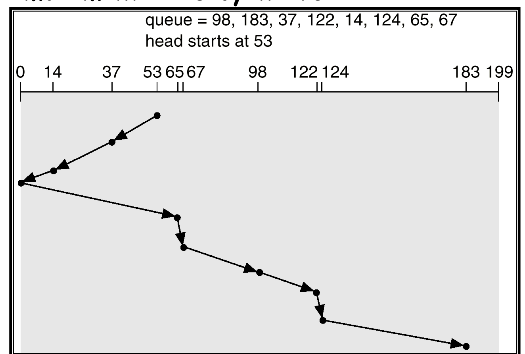
SSTF

- Pick the request that's closest on the disk head
- Con: reduce seeks, but may lead to starvation
- Head movement: 236 cylinders
- Note: need also to include rotational delay in calculation, since rotation can be as long as seek



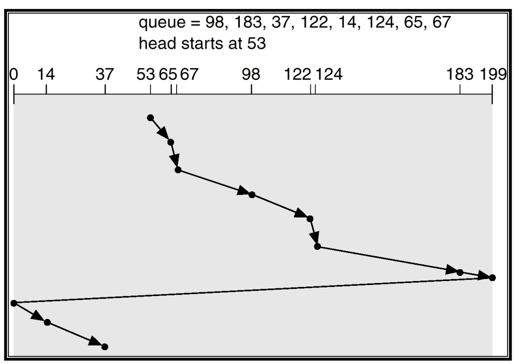
SCAN

- Implements an Elevator Algorithm: take the closest request in the direction of travel
- No starvation, but retains flavor of SSTF
- Head moves to lower cylinders
- Head movement: 208 cylinders



C-SCAN

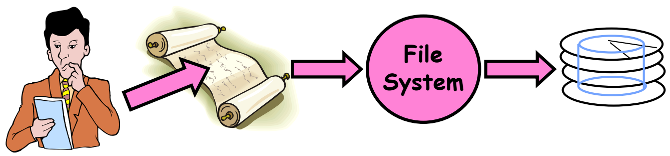
- Skips any requests on the way back
- Fairer than SCAN, not biased towards pages in middle



Building a File System

- **File System:** Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.
- File System Components
  - Disk Management: collecting disk blocks into files
  - Naming: Interface to find files by name, not by blocks
  - Protection: Layers to keep data secure
  - Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc
- User vs. System View of a File
  - User's view:
    - » Durable Data Structures
  - System's view (system call interface):
    - » Collection of Bytes (UNIX)
    - » Doesn't matter to system what kind of data structures you want to store on disk!
  - System's view (inside OS):
    - » Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
    - » Block size ≥ sector size; in UNIX, block size is 4KB

Translating from User to System View



- What happens if user says: give me bytes 2—12?
  - Fetch block corresponding to those bytes
  - Return just the correct portion of the block
- What about: write bytes 2—12?
  - Fetch block
  - Modify portion
  - Write out Block
- Everything inside File System is in whole size blocks
  - For example, `getc()`, `putc()` ⇒ buffers something like 4096 bytes, even if interface is one byte at a time
- From now on, file is a collection of blocks

Disk Management Policies

- Basic entities on a disk:
  - File: user-visible group of blocks arranged sequentially in logical space
  - Directory: user-visible index mapping names to files (next lecture)
- Access disk as linear array of sectors
  - Identify sectors as vectors [cylinder, surface, sector]
  - Logical Block Addressing (LBA). Every sector has integer address from zero up to max number of sectors.
  - Controller translates from address ⇒ physical position
    - » First case: OS/BIOS must deal with bad sectors
    - » Second case: hardware shields OS from structure of disk
- Need way to track free disk blocks
  - Link free blocks together ⇒ too slow today
  - Use bitmap to represent free space on disk
- Need way to structure files: File Header
  - Track which blocks belong at which offsets within the logical file structure
  - Optimize placement of files' disk blocks to match access and usage patterns

Designing the File System: Access Patterns

- How do users access files?
  - Need to know type of access patterns user is likely to throw at system
- Sequential Access: bytes read in order ("give me the next X bytes, then give me next, etc")
  - Almost all file access are of this flavor
- Random Access: read/write element out of middle of array ("give me bytes i—j")
  - Less frequent, but still important. For example, virtual memory backing file: page of memory stored in file
  - Want this to be fast – don't want to have to read all bytes to get to the middle of the file
- Content-based Access: ("find me 100 bytes starting with Berkeley")
  - Example: employee records – once you find the bytes, increase my salary by a factor of 2
  - Many systems don't provide this; instead, databases are built on top of disk access to index content (requires efficient random access)

Designing the File System: Usage Patterns

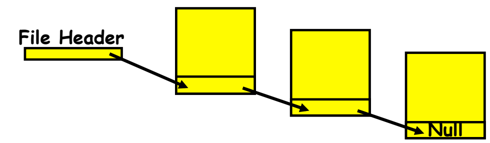
- Most files are small (for example, .login, .c files)
  - A few files are big – nachos, core files, etc.; the nachos executable is as big as all of your .class files combined
  - However, most files are small – .class's, .o's, .c's, etc.
- Large files use up most of the disk space and bandwidth to/from disk
  - May seem contradictory, but a few enormous files are equivalent to an immense # of small files
- Although we will use these observations, beware usage patterns:
  - Good idea to look at usage patterns: beat competitors by optimizing for frequent patterns
  - Except: changes in performance or cost can alter usage patterns. Maybe UNIX has lots of small files because big files are really inefficient?

How to organize files on disk

- Goals:
  - Maximize sequential performance
  - Easy random access to file
  - Easy management of file (growth, truncation, etc)
- First Technique: Continuous Allocation
  - Use continuous range of blocks in logical block space
    - » Analogous to segmentation in virtual memory
    - » User says in advance how big file will be (disadvantage)
  - Search bit-map for space using best fit/first fit
    - » What if not enough contiguous space for new file?
  - File Header Contains:
    - » First block/LBA in file
    - » File size (# of blocks)
  - Pros: Fast Sequential Access, Easy Random access
  - Cons: External Fragmentation/Hard to grow files
    - » Free holes get smaller and smaller
    - » Could compact space, but that would be *really* expensive
- Continuous Allocation used by IBM 360
  - Result of allocation and management cost: People would create a big file, put their file in the middle

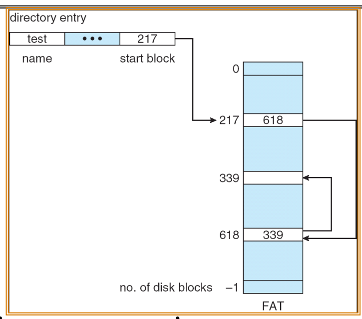
Linked List Allocation

- Second Technique: Linked List Approach
  - Each block, pointer to next on disk



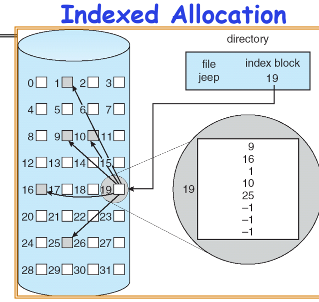
  - Pros: Can grow files dynamically, Free list same as file
  - Cons: Bad Sequential Access (seek between each block), Unreliable (lose block, lose rest of file)
  - Serious Con: Bad random access!!!!
  - Technique originally from Alto (First PC, built at Xerox)
    - » No attempt to allocate contiguous blocks

Linked Allocation: File-Allocation Table (FAT)



- MSDOS links pages together to create a file
  - Links not in pages, but in the File Allocation Table (FAT)
    - » FAT contains an entry for each block on the disk
    - » FAT Entries corresponding to blocks of file linked together
  - Access properties:
    - » Sequential access expensive unless FAT cached in memory
    - » Random access expensive always, but *really* expensive if FAT not cached in memory

Indexed Allocation



- Third Technique: Indexed Files (Nachos, VMS)
  - System Allocates file header block to hold array of pointers big enough to point to all blocks
    - » User pre-declares max file size;
  - Pros: Can easily grow up to space allocated for index Random access is fast
  - Cons: Clumsy to grow file bigger than table size Still lots of seeks: blocks may be spread over disk

Summary

- Queuing Latency:
  - M/M/1 and M/G/1 queues: simplest to analyze
  - As utilization approaches 100%, latency → ∞

$$T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1 - u))$$

- Disk scheduling
  - Minimize seek time while preserving fairness
- File System:
  - Transforms blocks into Files and Directories
  - Optimize for access and usage patterns
  - Maximize sequential access, allow efficient random access