Universidade Estadual de Campinas
Instituto de Computação

MC504 Sistemas Operacionais

T27 Hard Disk Scheduling

Referência principal
Ch. 37 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 29 de outubro de 2018

Arthur João Catto, PhD

2º semestre de 2018

- To use disk drives efficiently the OS must find
    - Fast access time
    - Large disk bandwidth
        - Bandwidth is the quotient of the total number of bytes transferred by the total time between the first request for service and the completion of the last transfer.
- Access time has two major components
    - Seek time
    - Rotational latency

To read or write, the disk head must be positioned at the desired track and at the beginning of the desired sector.

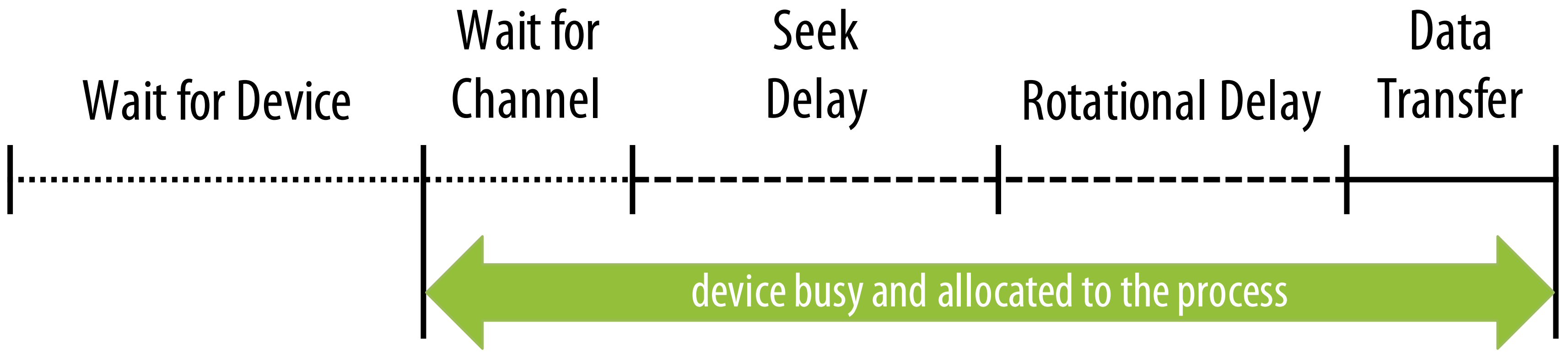
- Seek time
  - Time taken to position the head at the desired track
- Rotational delay or rotational latency
  - Time until the beginning of the desired sector reaches the head

Data transfer occurs as the sector moves under the head.

Seek time is the main reason for differences in performance.

For a single disk there will be a number of pending I/O requests.

If requests are selected randomly, we will achieve poor performance.



There are many types of disk scheduling algorithms

- Sequential
- Priority-based
- ...

In the following examples we will consider

- A 200-cylinder disk
- Read/write head initially on cylinder 53
- The following request queue (required track numbers)
  - 98, 183, 37, 122, 14, 124, 65, 67

First-in, first-out (FIFO)

Processes requests sequentially

Fair to all processes

Approaches random scheduling in performance if there are many processes

First-in, first-out (FIFO)

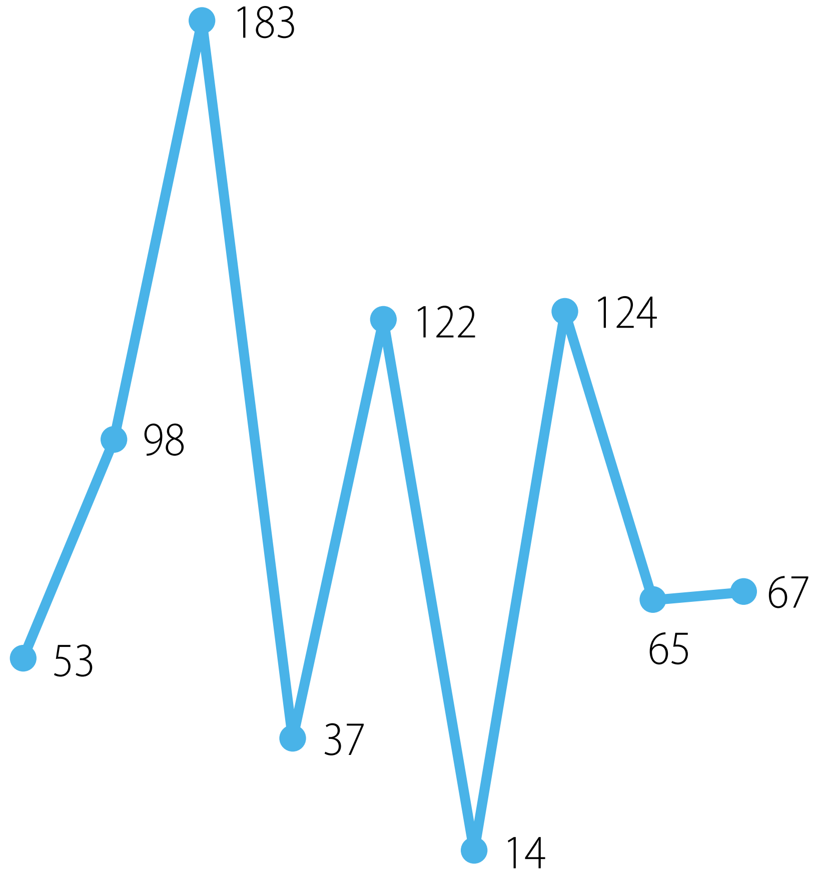
Queue = 98, 183, 37, 122, 14, 124, 65, 67

Head initially parked at cylinder 53

Total seek length

640 cylinders

Track



Last-in, first-out

Good for transaction processing systems

- The device is given to the most recent user so there should be little arm movement

Possibility of starvation since a job may never regain the head of the line

Priority-based scheduling

Goal is not to optimize disk use but to meet other objectives

Short batch jobs may have higher priority

Can provide good interactive response time

Shortest Service Time First

Select the disk I/O request that requires the least movement of the disk arm from its current position

Always chooses the minimum seek time

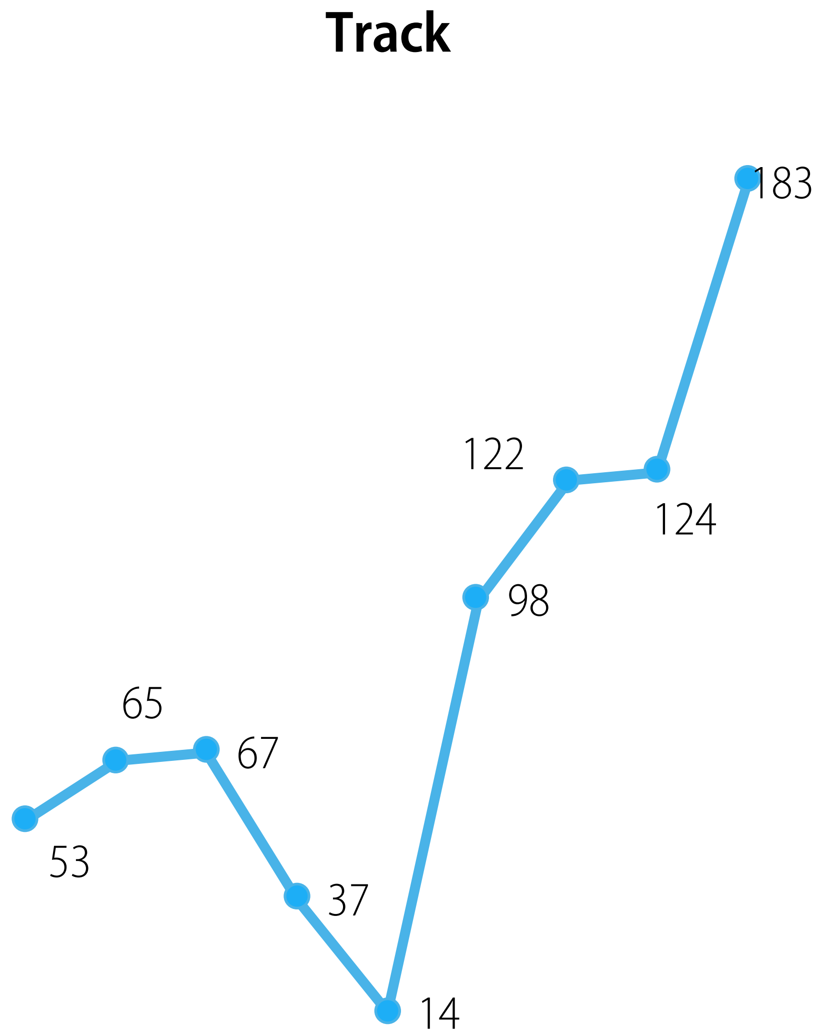
Shortest Service Time First

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

Head initially parked at cylinder 53

Total seek length

236 cylinders



SSTF is not a panacea

Problem 1: The drive geometry is not available to the host OS

- Solution: OS can simply implement ***nearest-block-first*** (*NBF*)

Problem 2: Starvation

- If there were a steady stream of requests to the inner track, requests to other tracks would then be ignored completely.

SCAN

Arm moves in one direction only, satisfying all outstanding requests until it reaches the last track in that direction

Direction is then reversed

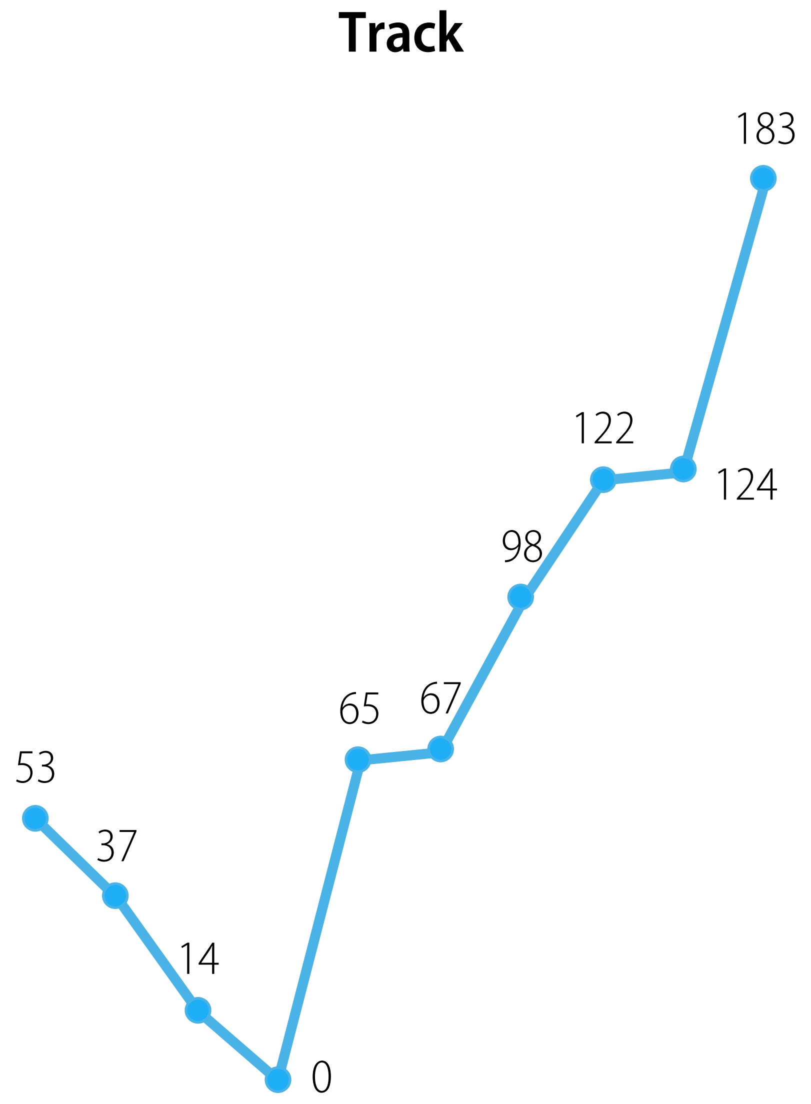
Disk Scheduling Policies

SCAN

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

Head initially parked at cylinder 53

Total seek length
208 cylinders



C-SCAN

Restricts scanning to one direction only

When the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again

Disk Scheduling Policies

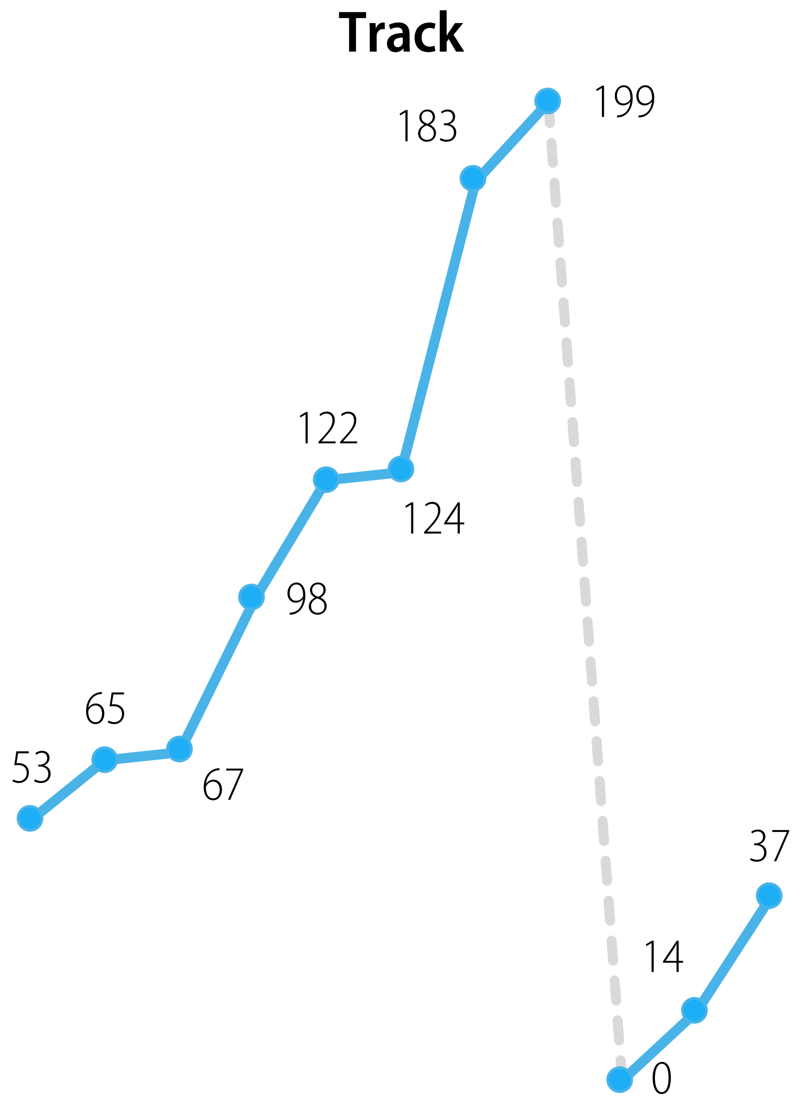
C-SCAN

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

Head initially at cylinder 53, moving forward

Total seek length

197 cylinders + retraction



C-LOOK

Similar to C-SCAN

The arm moves only up to the last request in each direction

After that it changes direction without moving up to the disk border

Disk Scheduling Policies

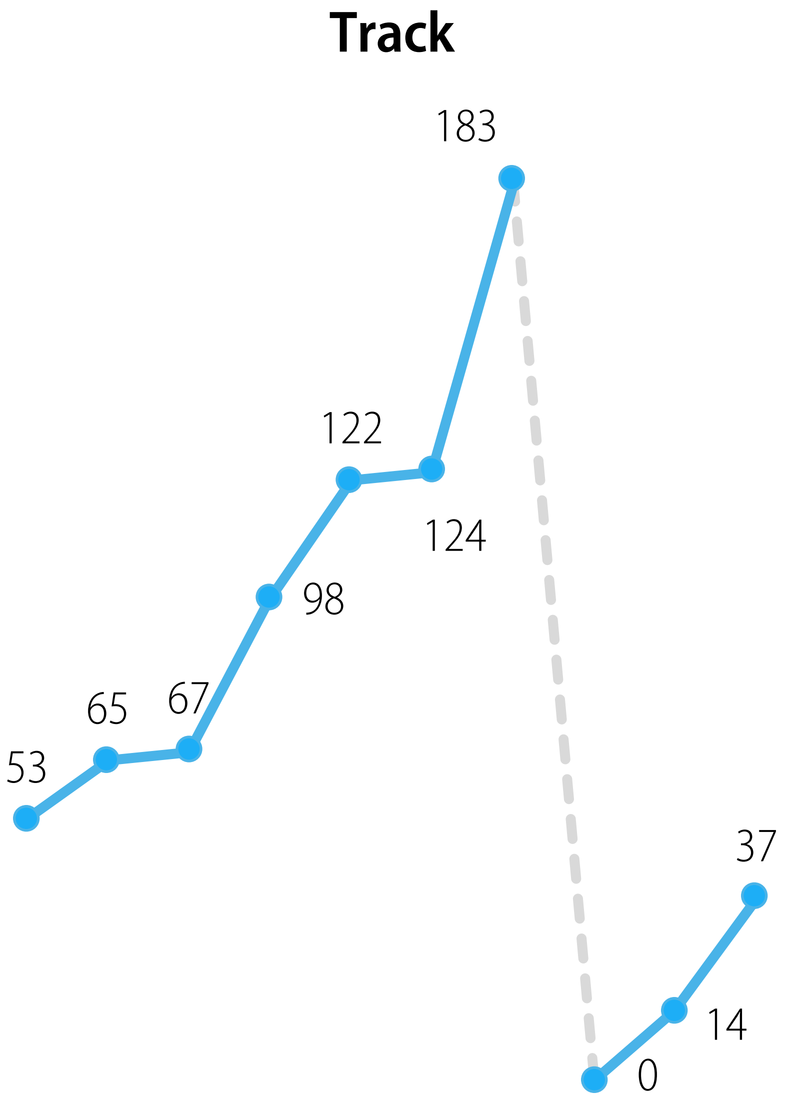
C-LOOK

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

Head initially parked at cylinder 53

Total seek length

167 cylinders + retraction



Disk Scheduling Policies

N-step-SCAN and FSCAN

N-step-SCAN

- Segments the disk request queue into sub-queues of length N
- Sub-queues are processed one at a time, using SCAN
- New requests added to other queue when queue is processed

FSCAN

- Two queues
- One queue is empty for new requests

Choosing a disk scheduling algorithm...

SSTF is common and faster than FCFS.

SCAN and C-SCAN work better with heavy disk load.

Performance depends on number and types of requests.

Service requests are affected by file allocation policies.

Disk scheduling may be written as a separate (easily replaceble) OS module.

SSTF and LOOK are suitable choices for the default algorithm.

Swap-Space Management

Usage of swap-space depends on the memory management algorithm adopted.

- Entire process images.
- Pages that have been pushed out of memory.

Swapping severely impacts system performance.

It is safer to overestimate the need for swap space.

- Lack may lead to process abortions and system crashes.
- Some systems allow the use of multiple swap areas, usually on separate disks.

Swap-Space Management

Swap space may reside

- Within the normal file system
  - Implementation is easy but inneficient.
  - To improve performance OSs attempt to make the swap space contiguous and unmovable.
- In a separate (raw) disk partition
  - Algorithms are designed for speed rather than for storage efficiency.

Disk Cache

Buffer in main memory for disk blocks

Contains a copy of some of the blocks on the disk

Requires a policy for block replacement

Disk cache replacement policies

Least Recently Used

The block that has been in the cache the longest with no reference to it is replaced

The cache consists of a stack of blocks

Most recently referenced block is on the top of the stack

When a block is referenced or brought into the cache, it is placed on the top of the stack

The block on the bottom of the stack is removed when a new block is brought in

Blocks don’t actually move around in main memory

A stack of pointers is used

Disk cache replacement policies

Least Frequently Used

The block that has experienced the fewest references is replaced

A counter is associated with each block

Counter is initialized to 1 when a block is brought in and incremented each time the block is accessed

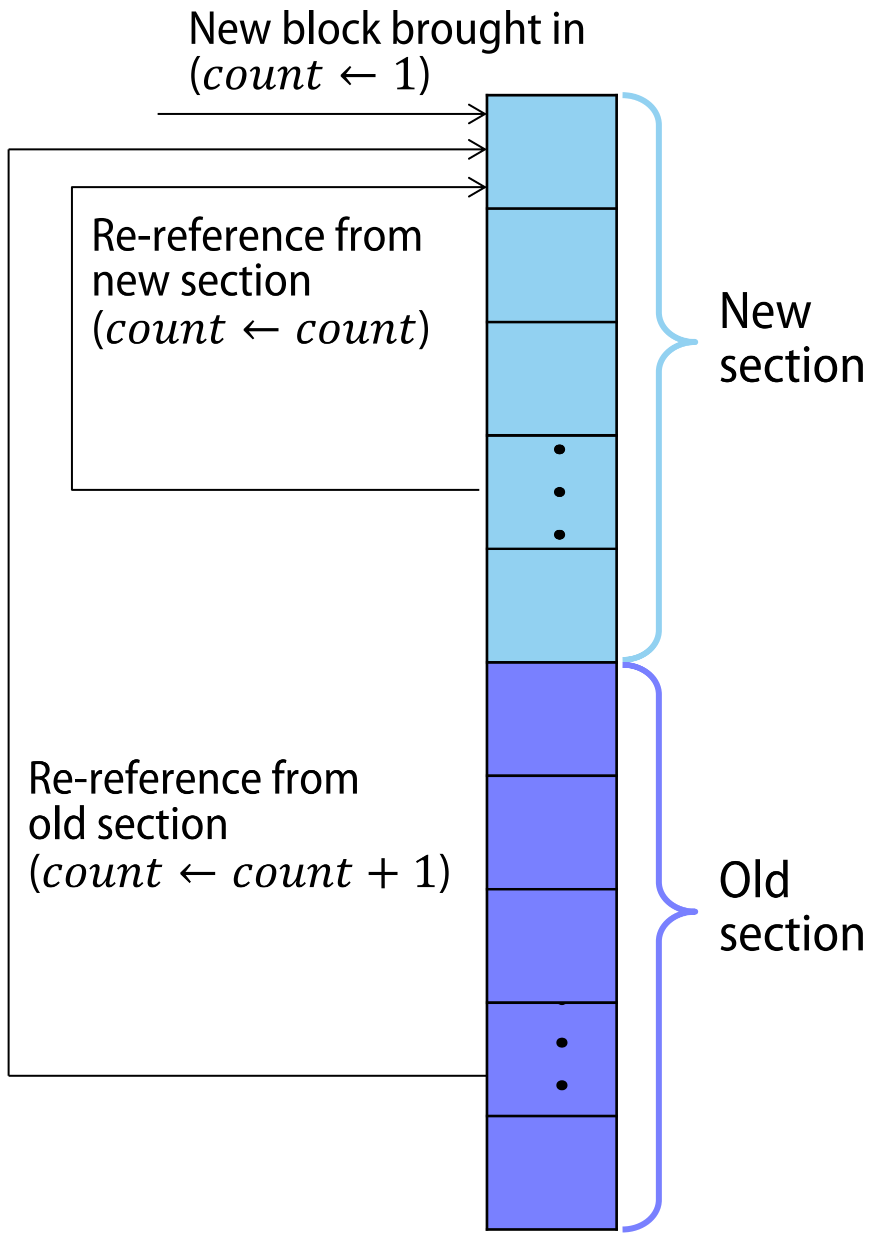
When needed, the block with smallest count is selected for replacement

Because of locality some blocks may be referenced many times in a short period of time but infrequently overall

- Thus the reference count can be misleading

Frequency-Based Replacement 1

- Block with smallest count in old section is replaced
    - In case of draw, the block with the oldest reference (the one closer to the bottom of the stack) is replaced
- Performance slightly better than LRU or LFU


New block brought in
$(count \leftarrow 1)$
Re-reference from new section
$(count \leftarrow count)$
Re-reference from old section
$(count \leftarrow count + 1)$
New section
Old section

Disk cache replacement policies

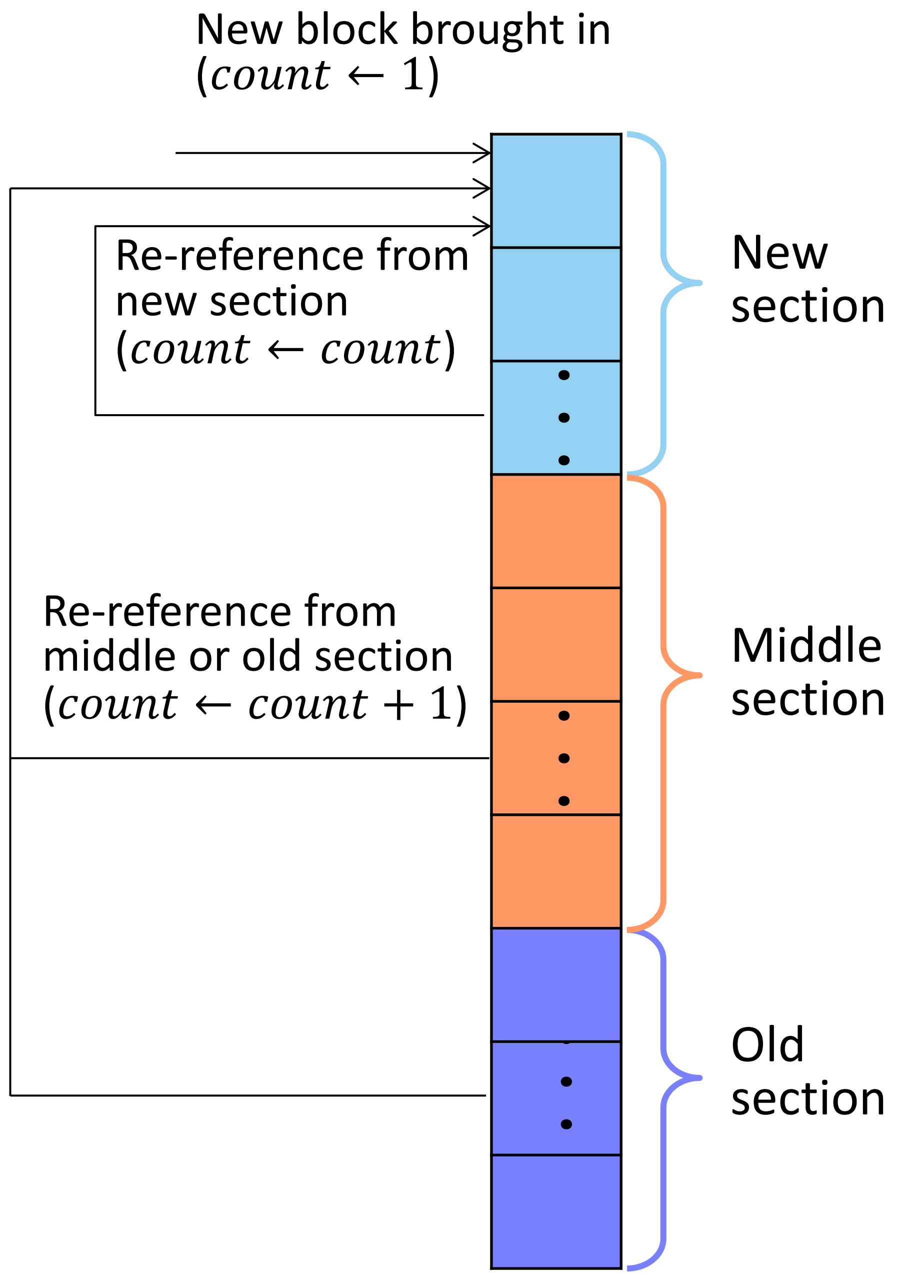
Frequency-Based Replacement 2

Block with smallest count in old section is replaced

- In case of draw, the block with the oldest reference (the one closer to the bottom of the stack) is replaced

Blocks in middle section have a chance to improve count

Performance better than LRU and LFU


New block brought in ($count \leftarrow 1$)
Re-reference from new section ($count \leftarrow count$)
Re-reference from middle or old section ($count \leftarrow count + 1$)
New section
Middle section
Old section

Hamming Code

Code for detecting a single error

- Consider m data bits $d_m \dots d_3 d_2 d_1$
- Adding 1 parity bit we create a code which uses $m + 1$ bits and can detect **one** single error.

Detecting and correcting a single error

- In order to detect and correct any **one** error the code must also indicate the position of such error
  - Assume that for that purpose k extra bits are required, so that the code now has $n = m + k$ bits
  - The k extra bits generate 2^k codes which must be able to indicate
    - the absence of an error or
    - the position of an error in any of the n bit positions
  - So, we need at least $n + 1$ different codes

$$\therefore 2^k \geq n + 1 \therefore 2^{n-m} \geq n + 1 \therefore \frac{2^n}{n+1} \geq 2^m$$

Hamming code for detecting and correcting a single error

- Let
  - $b_n b_{n-1} \dots b_2 b_1$ be the bits in the full code (m of which are the data bits)
  - $p_k \dots p_2 p_1 (\in b_n b_{n-1} \dots b_2 b_1)$ be the associated parity bits
- We want $p_k \dots p_2 p_1$ (read as a binary number) to give the position of the error or 0 if the data is correct.
- Where should the parity bits be placed?
  - At the front of the sequence?
  - At the end?
  - Let's try to find out.

Hamming code for detecting and correcting a single error

- We want $p_k \ldots p_2 p_1$ (read as a binary number) to give the position of the error or 0 if the data is correct.

Error in	p_k	…	p_3	p_2	p_1
b_1	0		0	0	1
b_2	0		0	1	0
b_3	0		0	1	1
b_4	0		1	0	0
b_5	0		1	0	1
b_6	0		1	1	0
b_7	0		1	1	1
…					

Hamming code for detecting and correcting a single error

- Now analyze the contribution of each bit b_i to the code bits p_j.
  - Every b_i contributes to several p_j with the exception of $b_1, b_2, b_4, \ldots$ which contribute to exactly only one.
- So, if we use positions $1, 2, 4, \ldots$ for the parity bits, they will be independent of each other.

Hamming code for detecting and correcting a single error

- So, we will have the full code

b_n	b_{n-1}	…	b_7	b_6	b_5	b_4	b_3	b_2	b_1
		…	d_4	d_3	d_2	p_3	d_1	p_2	p_1

corresponding to

- Reexamining the contribution table, we see that the parity bits can be generated from the data bits as
  - $p_1 = b_3 \oplus b_5 \oplus b_7 \oplus \cdots$
  - $p_2 = b_3 \oplus b_6 \oplus b_7 \oplus \cdots$
  - $p_3 = b_5 \oplus b_6 \oplus b_7 \oplus \cdots$
  - $\ldots$

Error in	p_k	…	p_3	p_2	p_1
b_1	0		0	0	1
b_2	0		0	1	0
b_3	0		0	1	1
b_4	0		1	0	0
b_5	0		1	0	1
b_6	0		1	1	0
b_7	0		1	1	1
…					

Hamming code for detecting and correcting a single error

- Given a full code $b_n b_{n-1} \dots b_2 b_1$, verification is done calculating
  - $x_1 = b_1 \oplus b_3 \oplus b_5 \oplus b_7 \oplus \cdots$
  - $x_2 = b_2 \oplus b_3 \oplus b_6 \oplus b_7 \oplus \cdots$
  - $x_3 = b_4 \oplus b_5 \oplus b_6 \oplus b_7 \oplus \cdots$
  - $\dots$
- Since, for instance, $b_1 = b_3 \oplus b_5 \oplus b_7 \oplus \cdots$
  - If all bits are correct, x_1 must be 0.
  - If any single bit is wrong, x_1 will be 1.
- The same applies to $x_2, x_3, \dots$
- If we look at the contribution table again, it is not difficult to conclude that the value $\dots x_3\ x_2\ x_1$ indicates the correctness of the code or the position of the error.

Hamming code for detecting and correcting a single error

- **Example 1:** Given a data code, generate the corresponding full code
  - Assume $m = 4$ and $d_4\ d_3\ d_2\ d_1 = 1\ 1\ 0\ 0$
  - We must find the smallest $n > m$ such that $\frac{2^n}{n+1} \geq 2^m$, which is 7.
  - The full code $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1$ will correspond to $1_4 1_3 0_2\ p_3\ 0_1\ p_2\ p_1$
  - The parity bits are calculated as
    - $p_1 = b_3 \oplus b_5 \oplus b_7 = 0 \oplus 0 \oplus 1 = 1$
    - $p_2 = b_3 \oplus b_6 \oplus b_7 = 0 \oplus 1 \oplus 1 = 0$
    - $p_3 = b_5 \oplus b_6 \oplus b_7 = 0 \oplus 1 \oplus 1 = 0$
  - Thus, the full code will be $1\ 1\ 0\ 0\ 0\ 0\ 1$

Hamming code for detecting and correcting a single error

- **Example 2:** In example 1, find an error in a data bit
  - The full code was $1\ 1\ 0\ 0\ 0\ 0\ 1$
  - Suppose that we receive
    $1\ 0\ 0\ 0\ 0\ 0\ 1$, with a wrong value in bit b_6 (which corresponds to data bit d_3)
  - Code checking is done by calculating
    - $x_1 = b_1 \oplus b_3 \oplus b_5 \oplus b_7 = 1 \oplus 0 \oplus 0 \oplus 1 = 0$
    - $x_2 = b_2 \oplus b_3 \oplus b_6 \oplus b_7 = 0 \oplus 0 \oplus 0 \oplus 1 = 1$
    - $x_3 = b_4 \oplus b_5 \oplus b_6 \oplus b_7 = 0 \oplus 0 \oplus 0 \oplus 1 = 1$
  - There is an error (since $x_3\ x_2\ x_1$ is not 0) and the wrong bit is given by $x_3\ x_2\ x_1 = \ 1_3\ 1_2\ 0_1 = 6$.

Hamming code for detecting and correcting a single error

- **Example 3:** In example 1, find an error in a parity bit
  - The full code was $1\ 1\ 0\ 0\ 0\ 0\ 1$
  - Suppose that we receive
    $1\ 1\ 0\ 0\ 0\ 1\ 1$, with a wrong value in bit b_2 (which corresponds to parity bit p_2)
  - Code checking is done by calculating
    - $x_1 = b_1 \oplus b_3 \oplus b_5 \oplus b_7 = 1 \oplus 0 \oplus 0 \oplus 1 = 0$
    - $x_2 = b_2 \oplus b_3 \oplus b_6 \oplus b_7 = 1 \oplus 0 \oplus 1 \oplus 1 = 1$
    - $x_3 = b_4 \oplus b_5 \oplus b_6 \oplus b_7 = 0 \oplus 0 \oplus 1 \oplus 1 = 0$
  - There is an error (since $x_3\ x_2\ x_1$ is not 0) and the wrong bit is given by $x_3\ x_2\ x_1 = 0_3\ 1_2\ 0_1 = 2$.