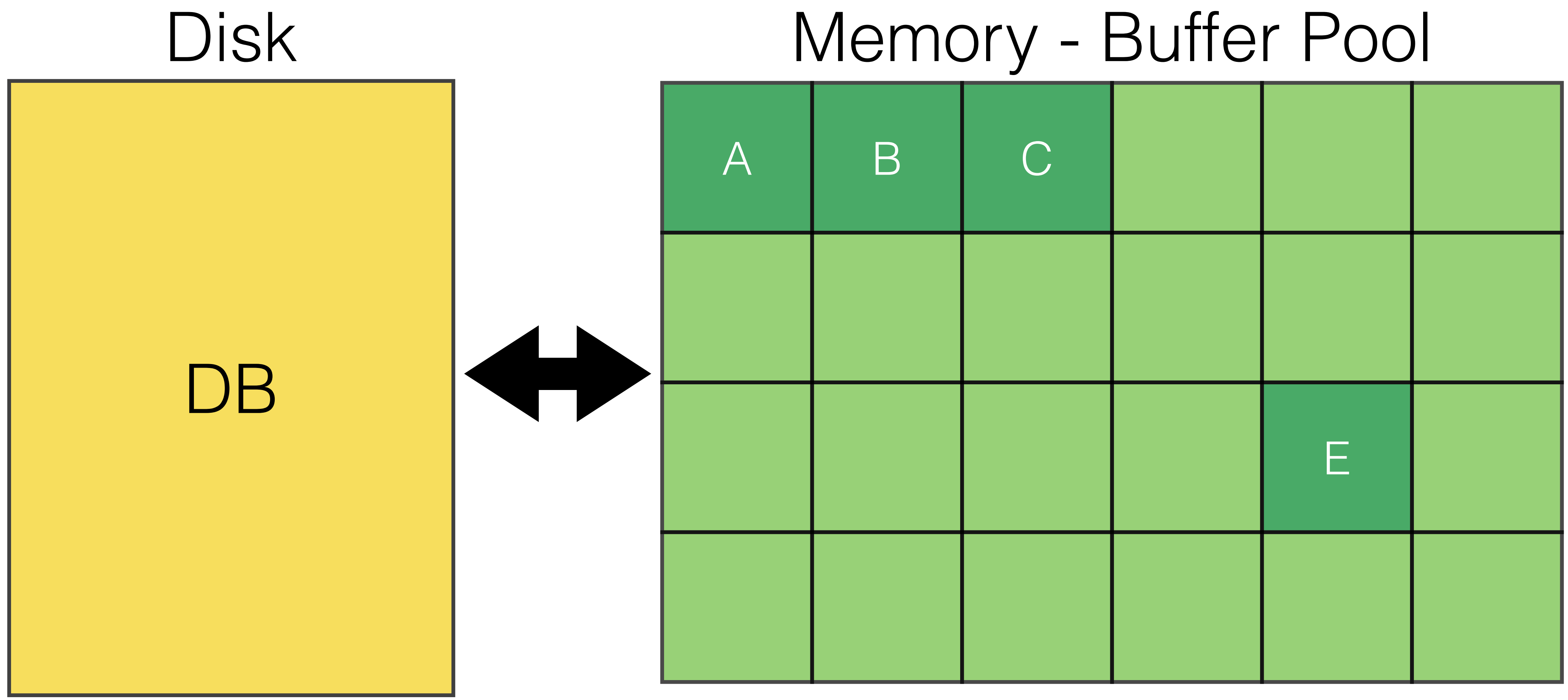
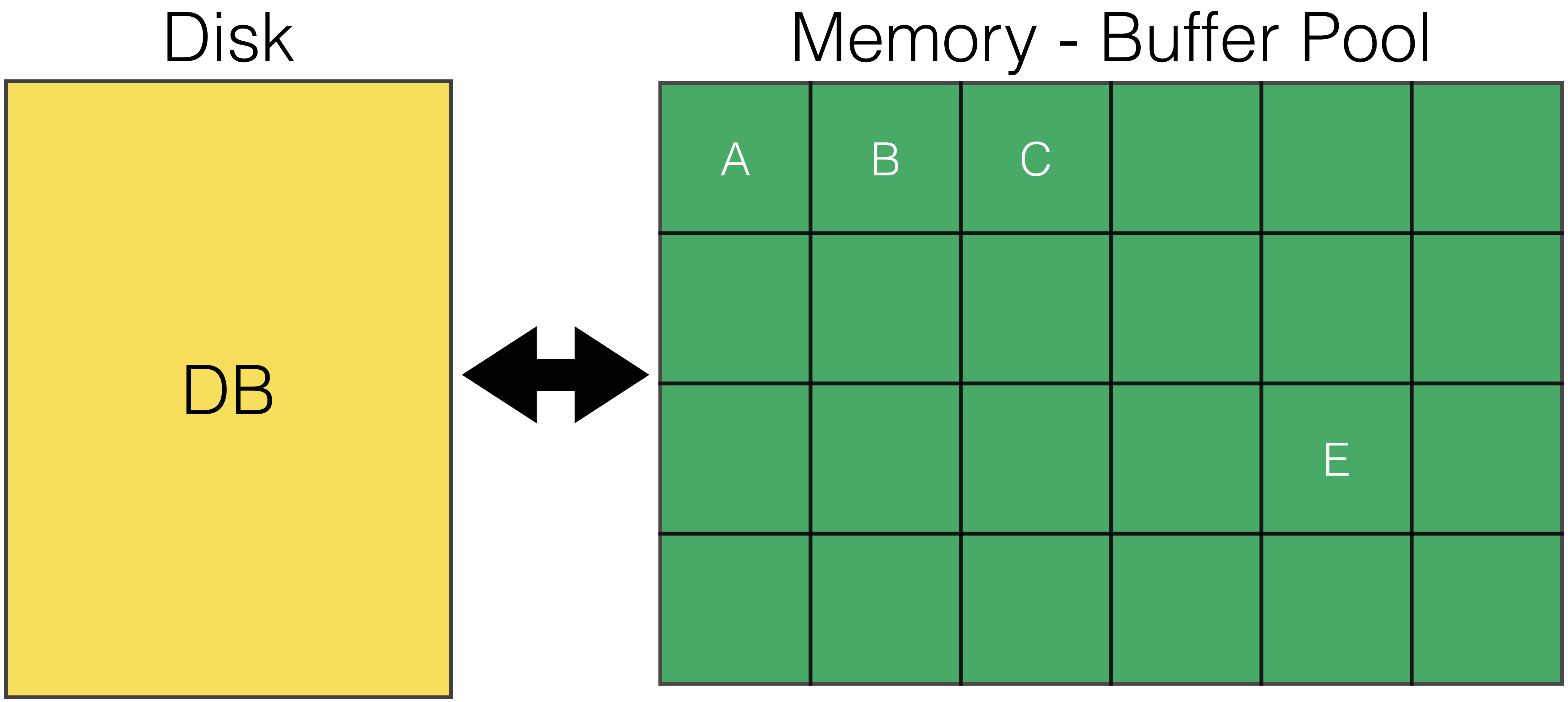
CS186 Discussion #3

(Buffer Management, File Organization)

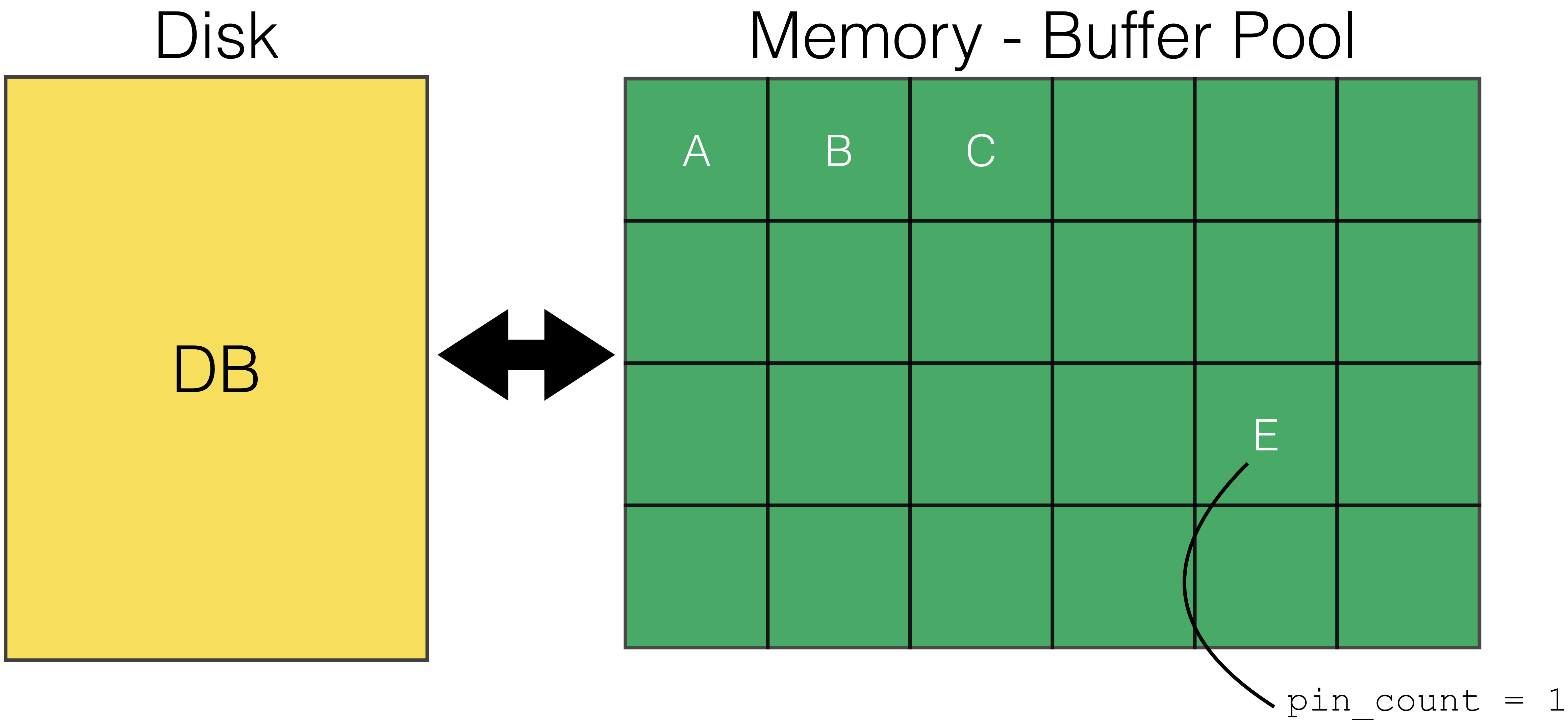
Midterm 1: 10/5


Disk
DB
Memory - Buffer Pool
A
B
C
E

What happens when our buffer pool is full? Which pages can we replace?



"Pin" a page (`pin_count++`) when page is requested. Only replace if `pin_count == 0`.



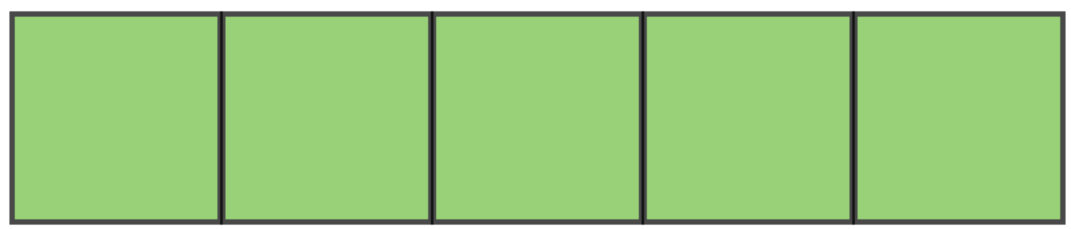
Buffer Replacement Policy

- Frame chosen for replacement using replacement policy (LRU, MRU, Clock, etc.)
- Policy can have a big impact on I/O’s

Least Recently Used (LRU)

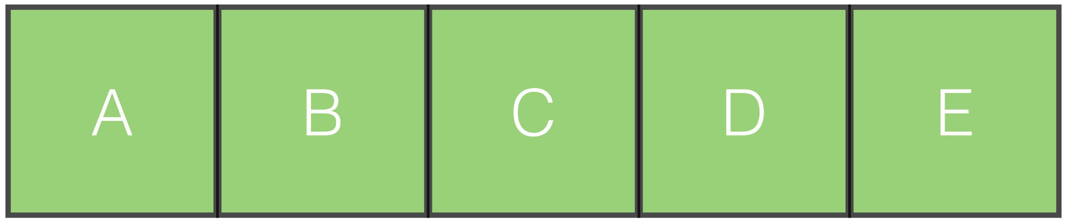
- Replace page unused for the longest amount of time
  - Assumes pages used recently will be used again
- Keep track of last time page was used/pinned
- Prone to sequential flooding
  - Reading all pages in a file multiple times
  - # buffer pages < # pages in file

LRU - Sequential Flooding



A, B, C, D, E, F, A, B, C, D

LRU - Sequential Flooding



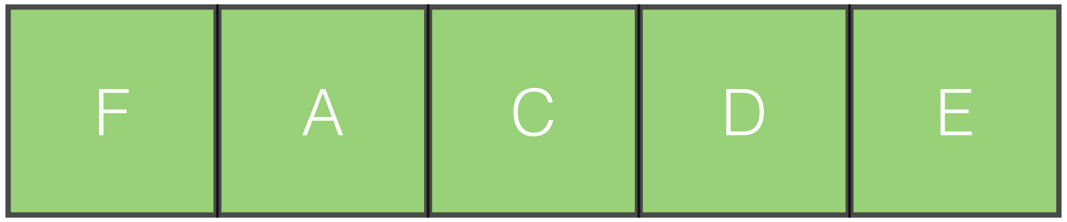
F, A, B, C, D

LRU - Sequential Flooding

F	B	C	D	E

A, B, C, D

LRU - Sequential Flooding

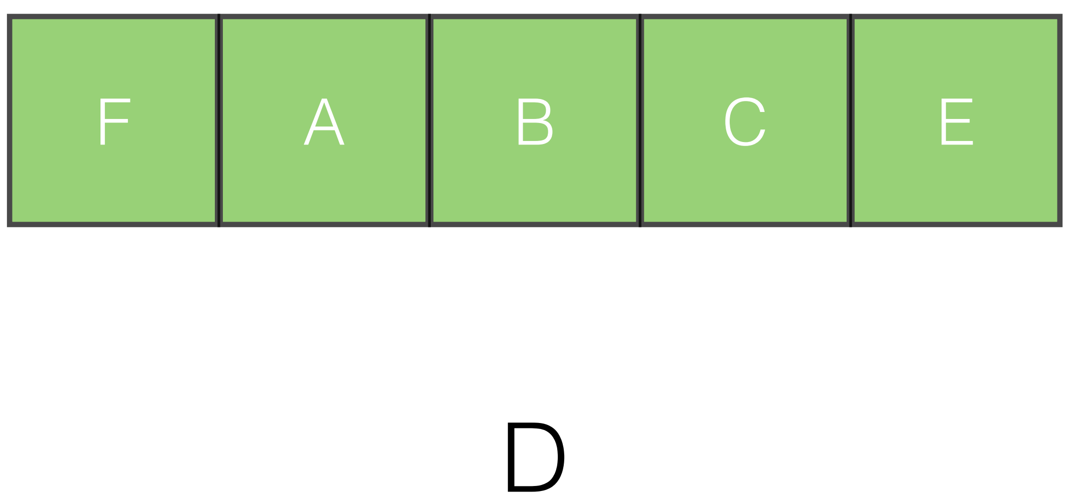


B, C, D

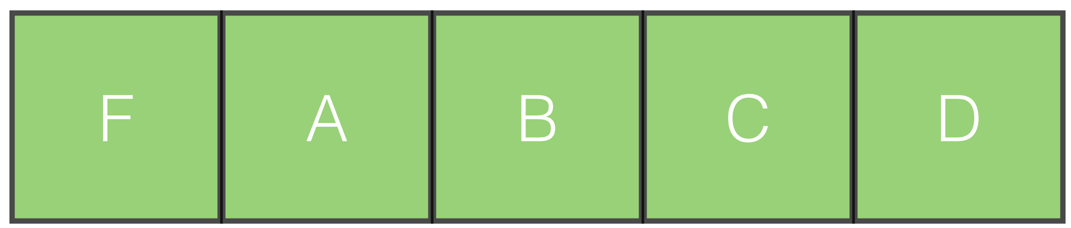
LRU - Sequential Flooding

C, D

LRU - Sequential Flooding



LRU - Sequential Flooding

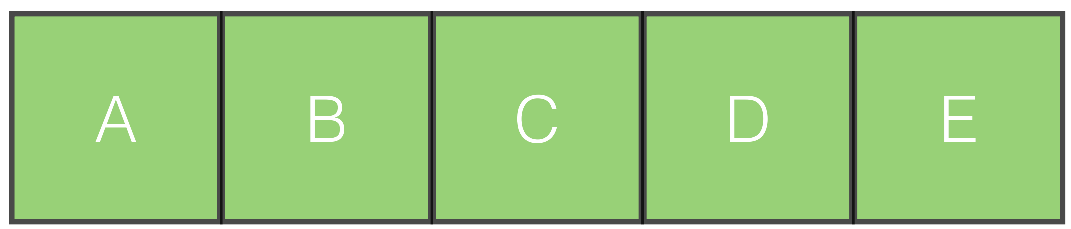


Every page request results in a cache miss!

Most Recently Used (MRU)

- Replace page that has just been used
- Fixes sequential flooding

MRU - Sequential Flooding



F, A, B, C, D

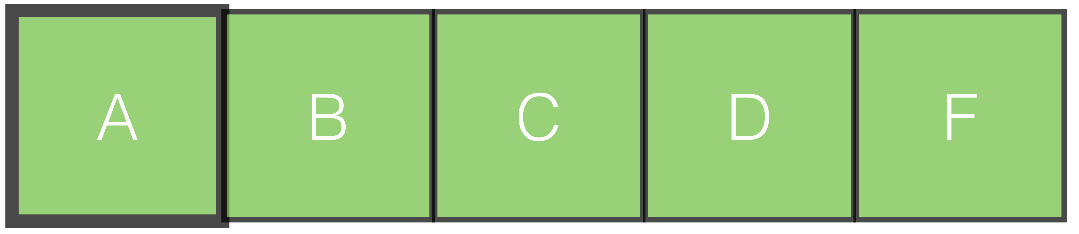
MRU - Sequential Flooding

A	B	C	D	F

A, B, C, D

MRU - Sequential Flooding

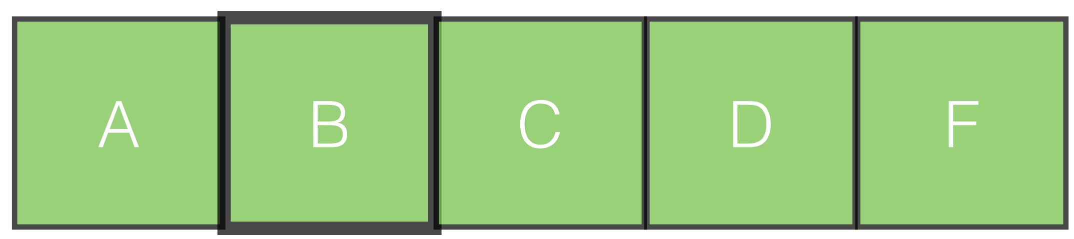
Cache hit!



B, C, D

MRU - Sequential Flooding

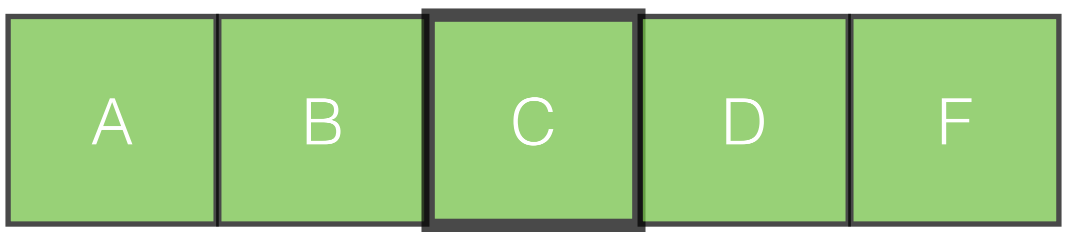
Cache hit!



C, D

MRU - Sequential Flooding

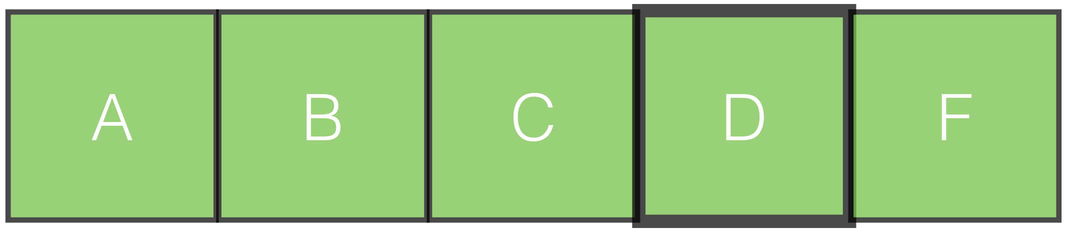
Cache hit!



D

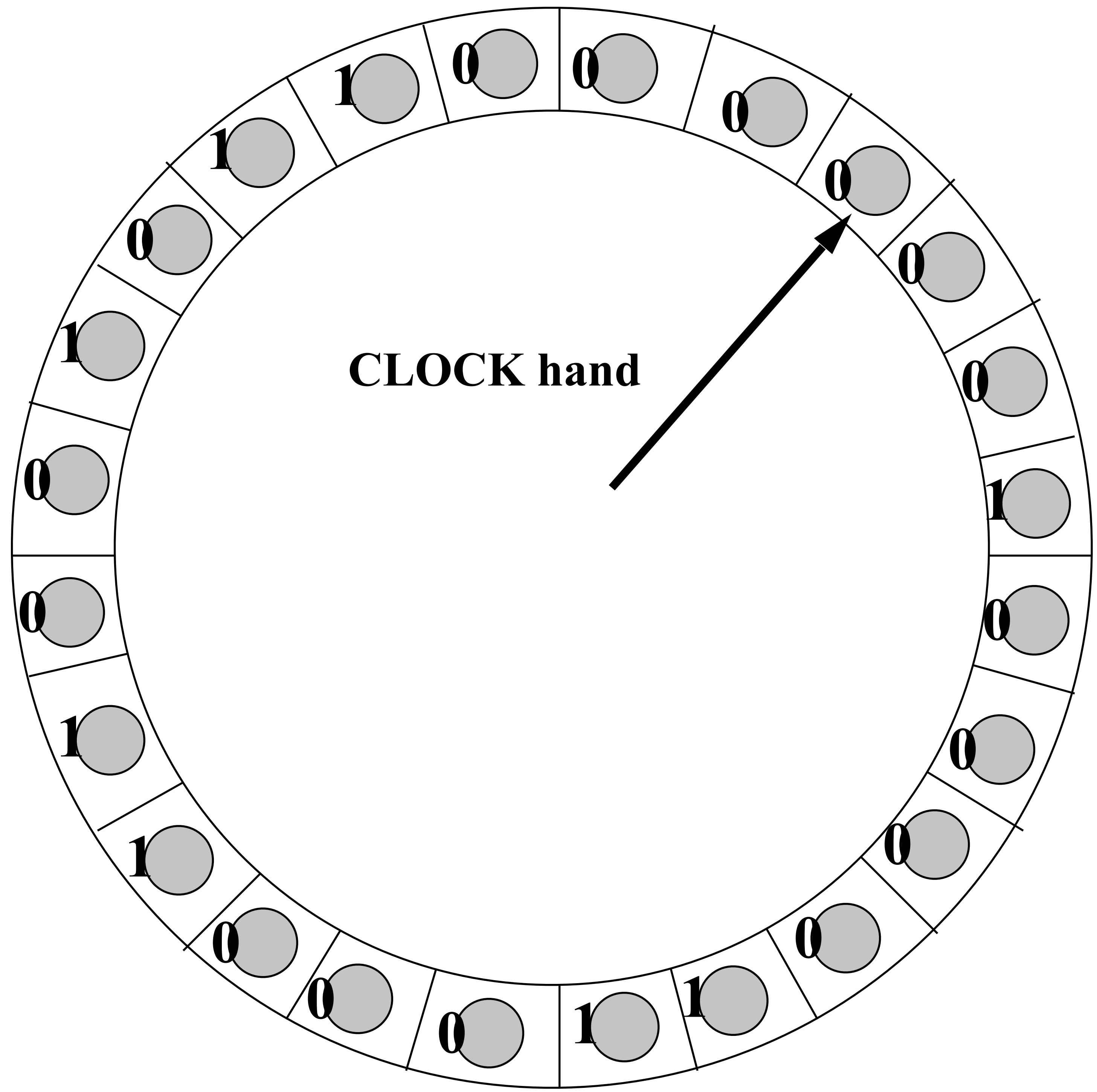
MRU - Sequential Flooding

Cache hit!



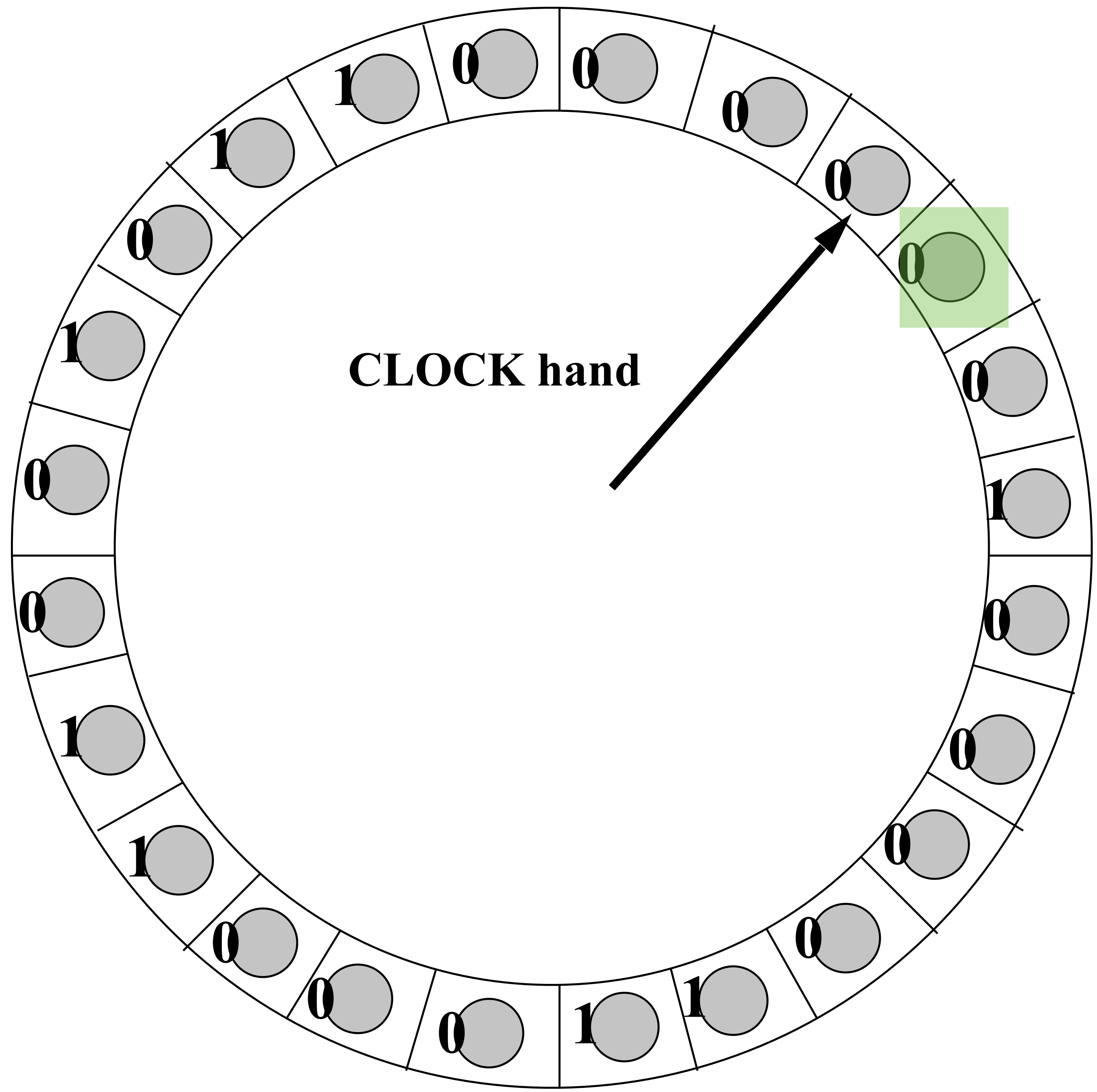
Clock Replacement

- All pages placed in a circular list.
- Each page has reference bit ("second-chance" bit) indicating if page has been accessed.



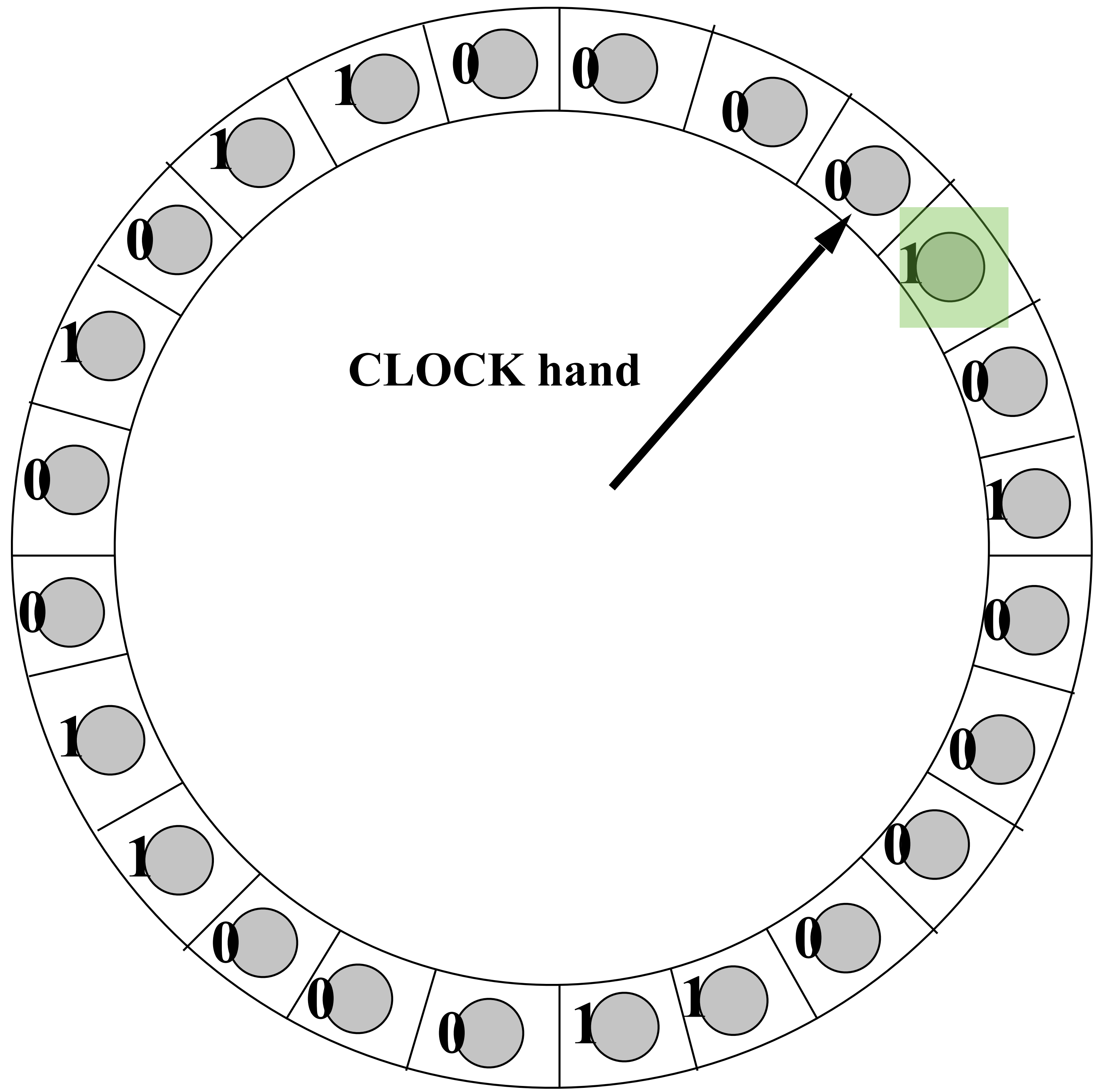
Clock Replacement

- On a HIT, set reference bit to 1.



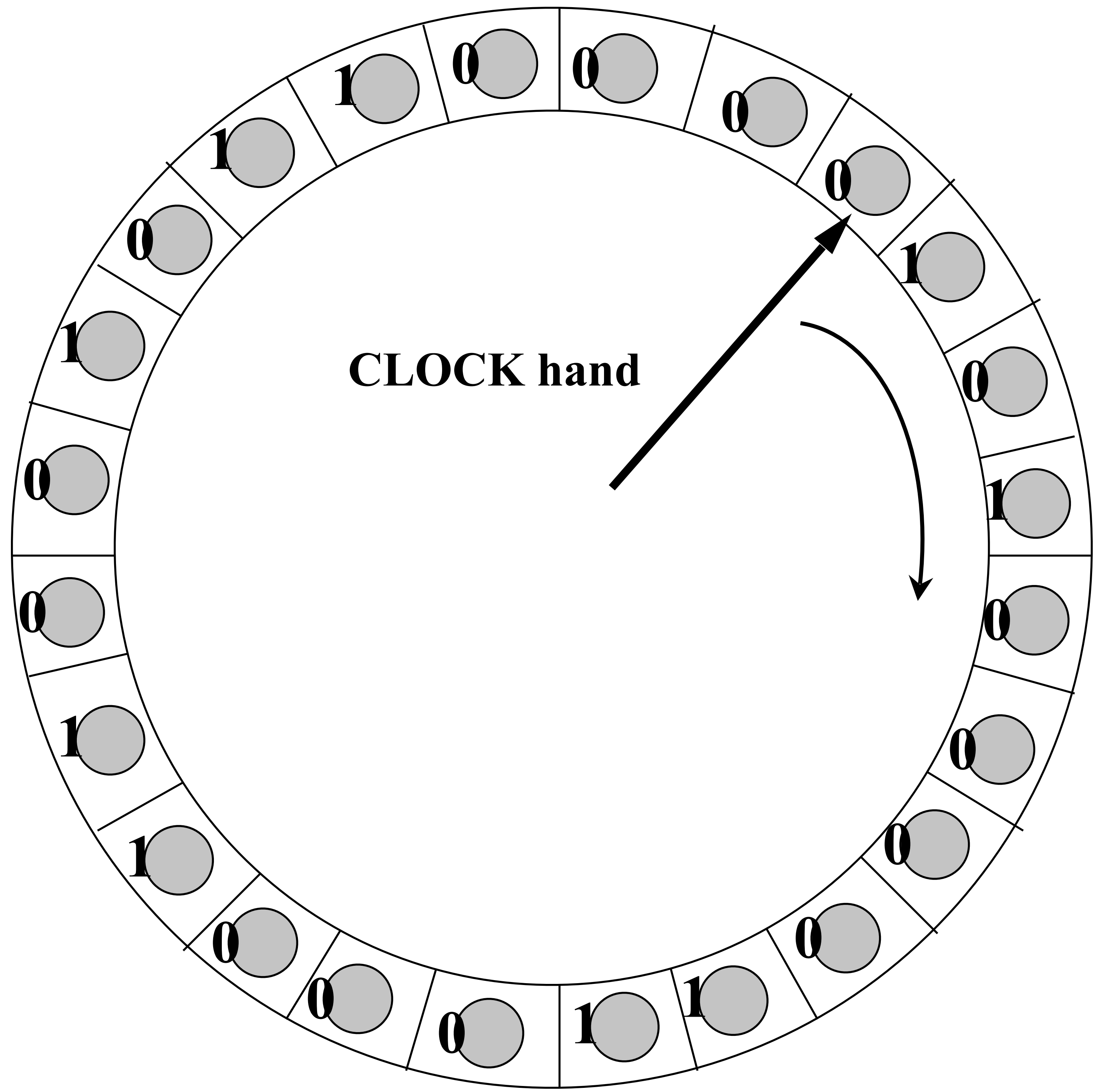
Clock Replacement

- On a HIT, set reference bit to 1.



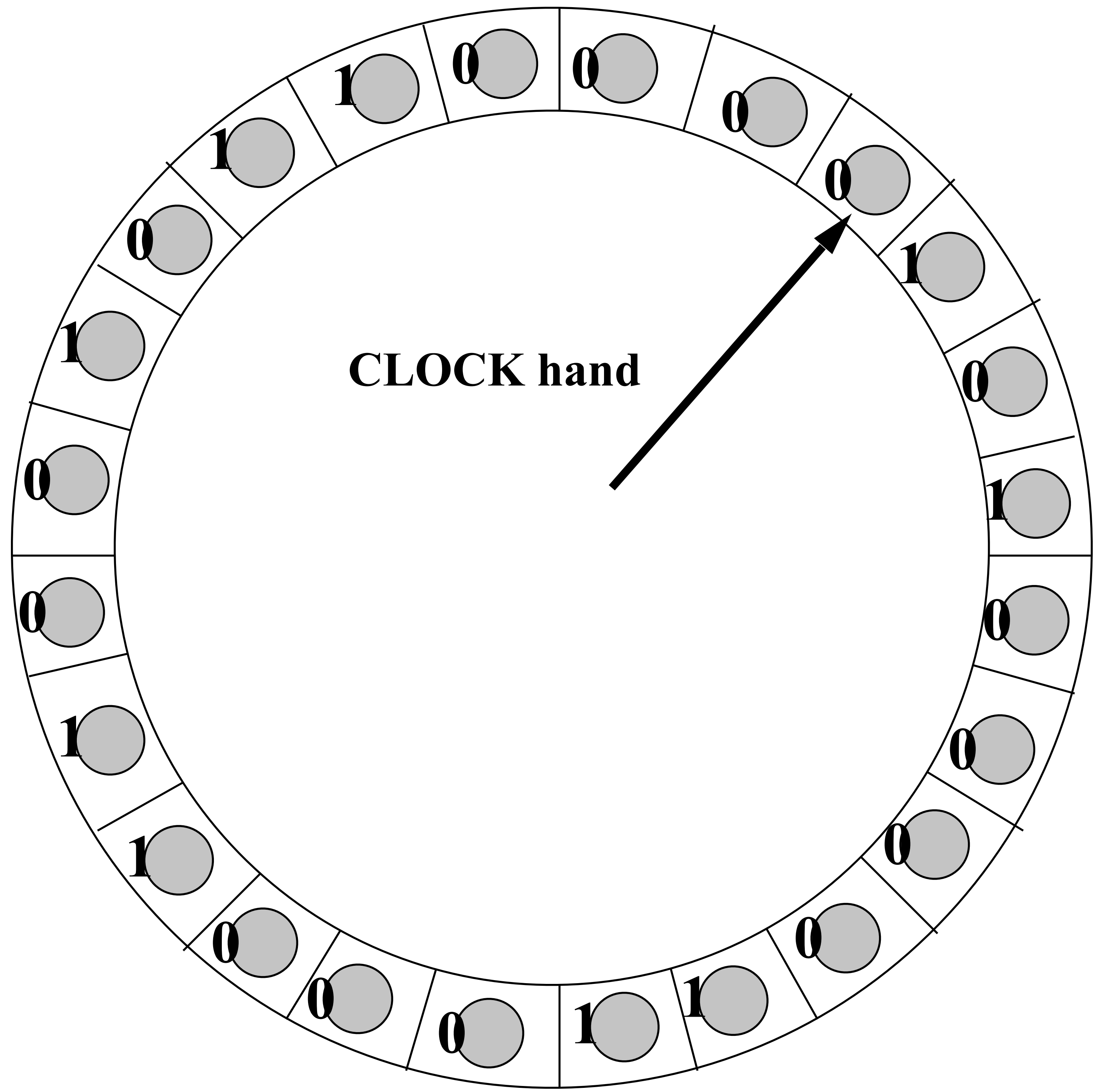
Clock Replacement

- On a MISS, move clock hand until reaches a page with “0” bit.
- Gives “1” bit pages a second chance and does not evict, but resets “1” to “0”.



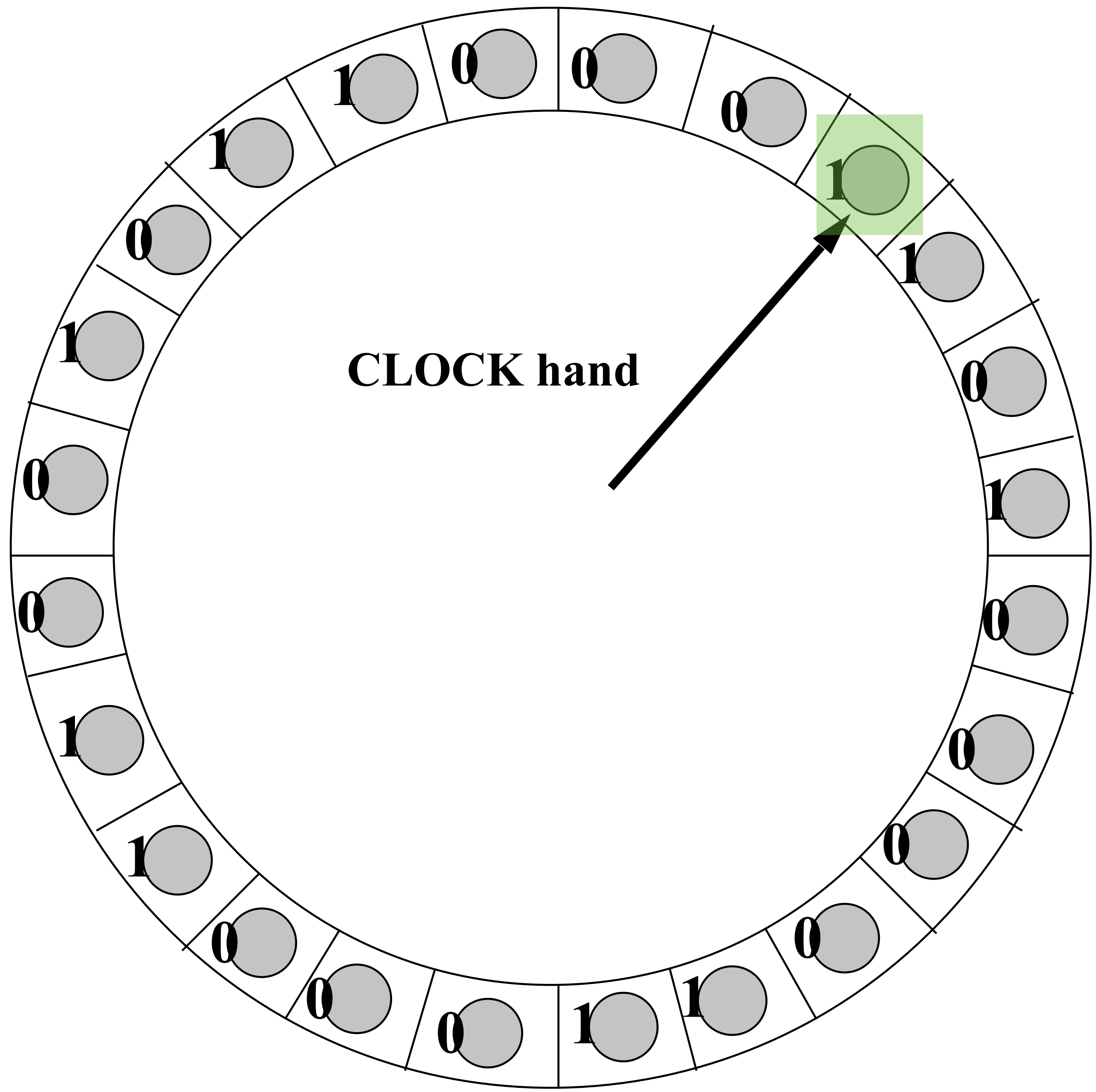
Clock Replacement

- 1 MISS



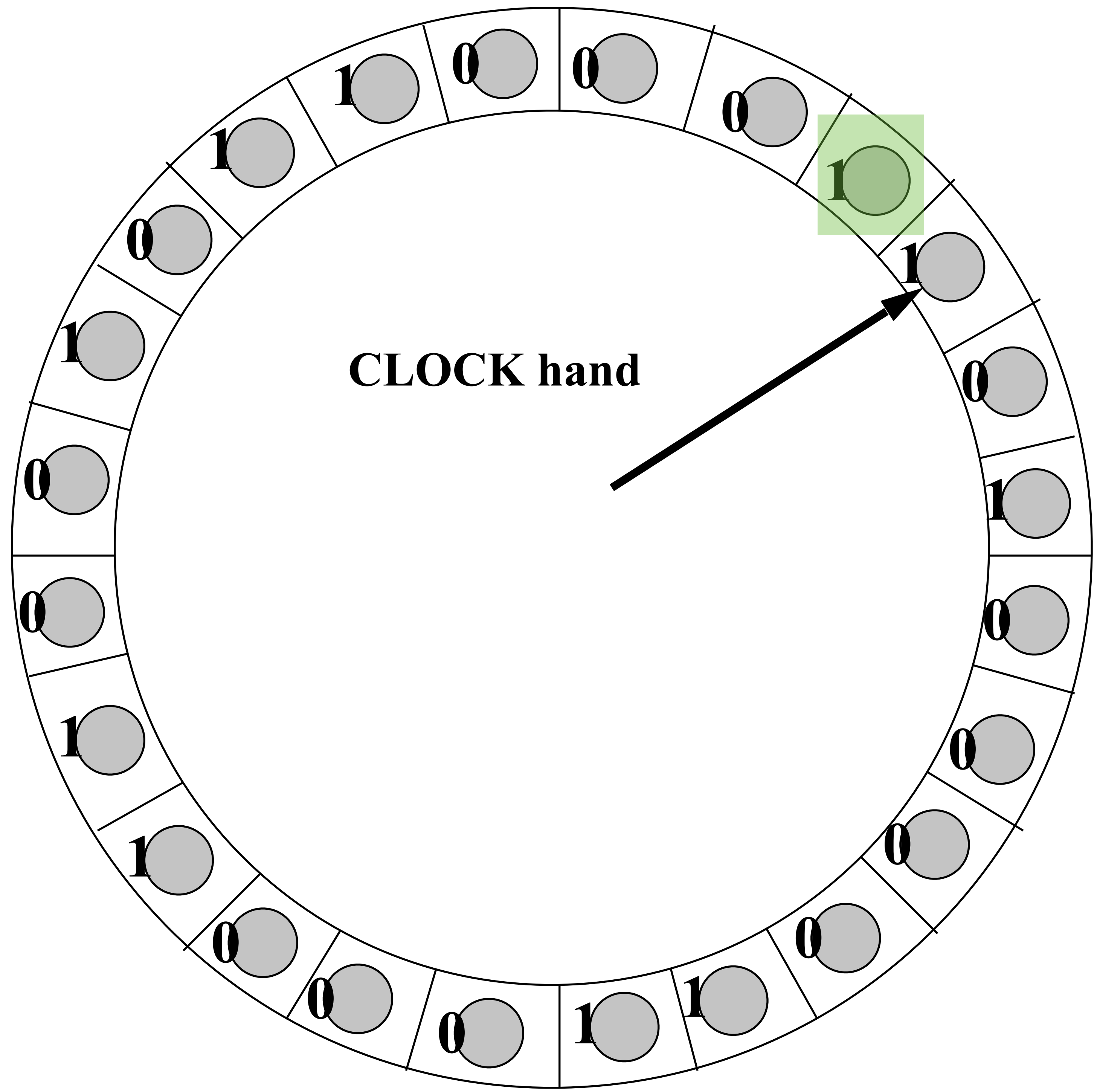
Clock Replacement

- 1 MISS



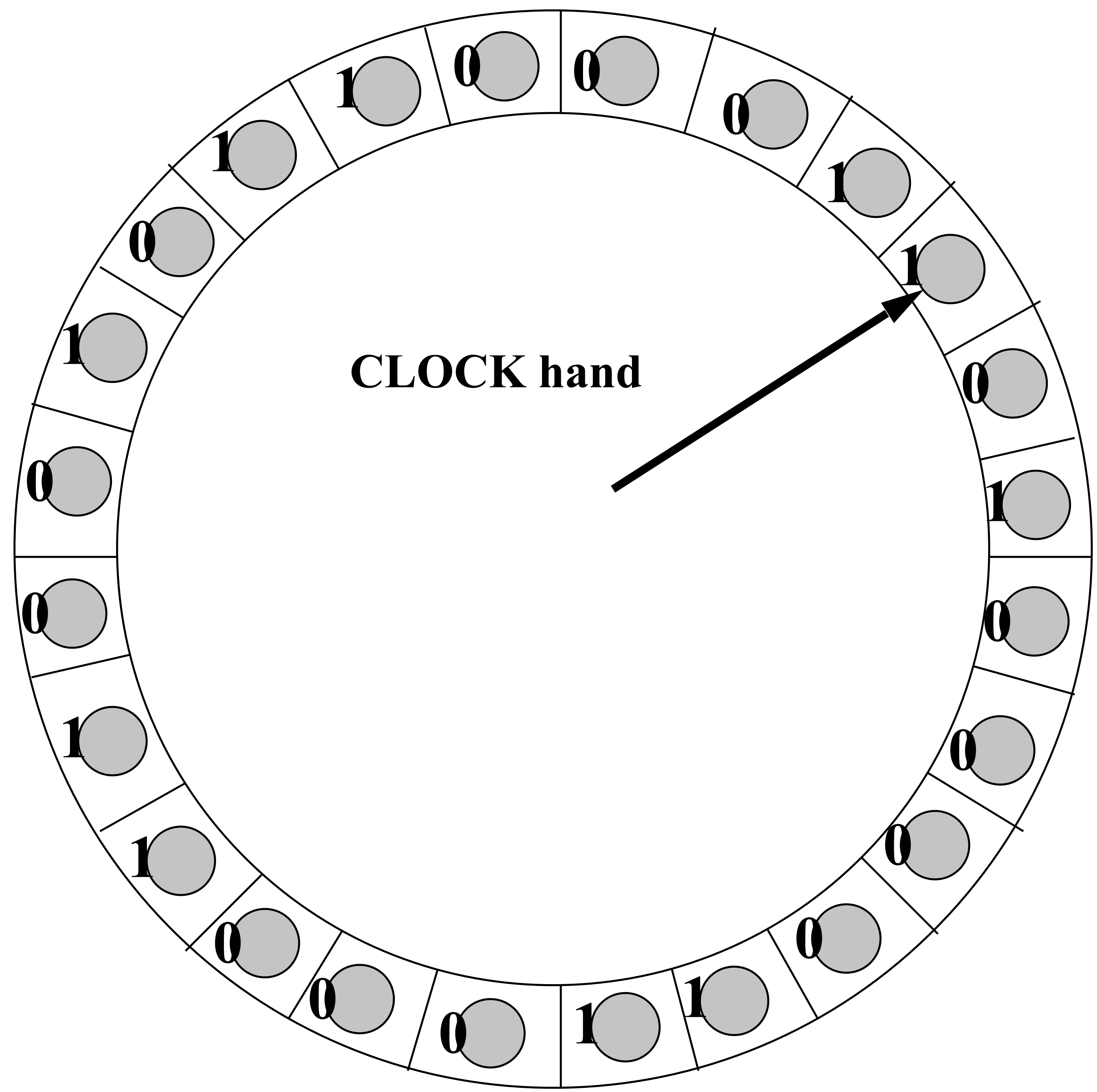
Clock Replacement

- 1 MISS



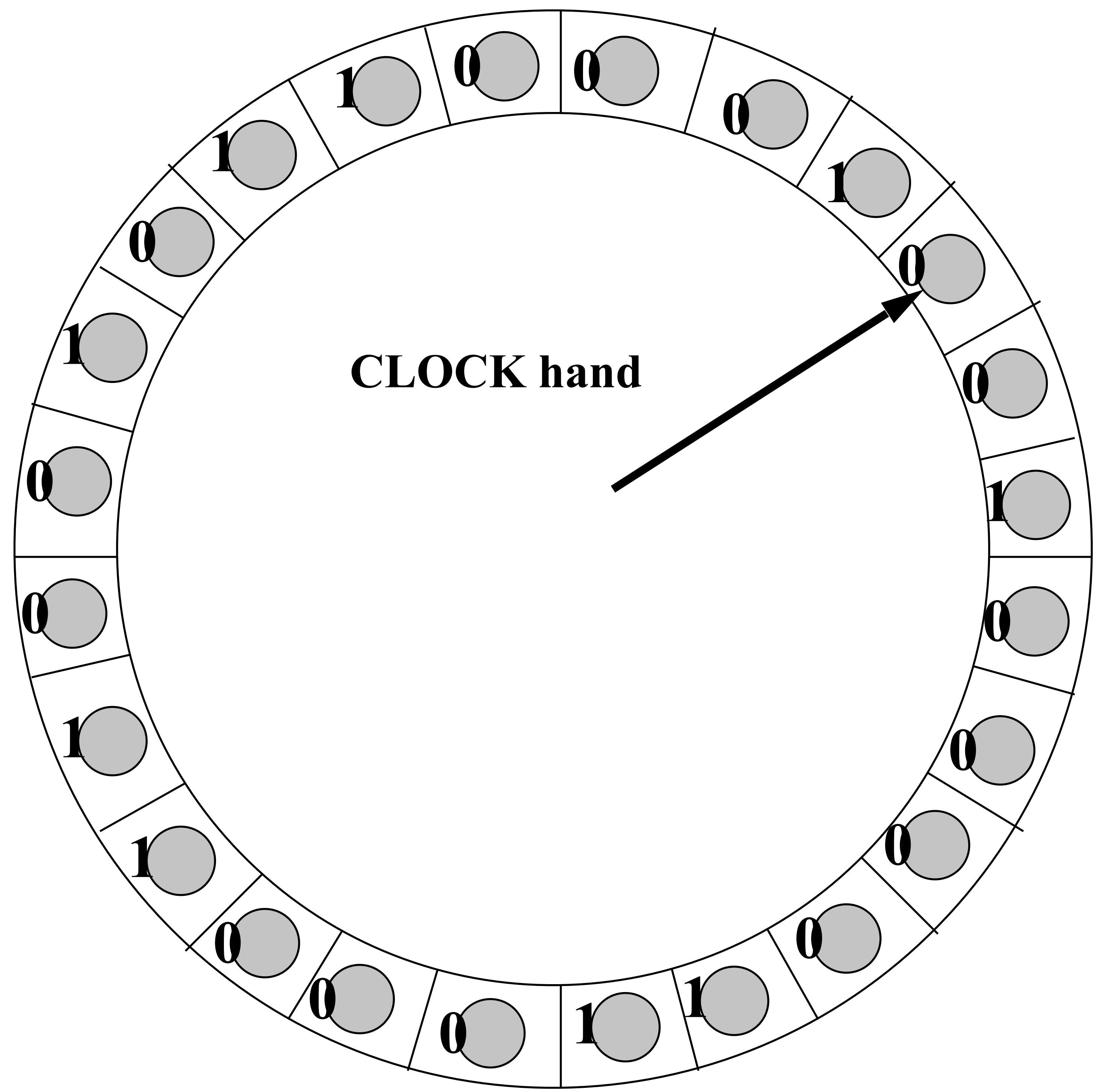
Clock Replacement

- Another MISS



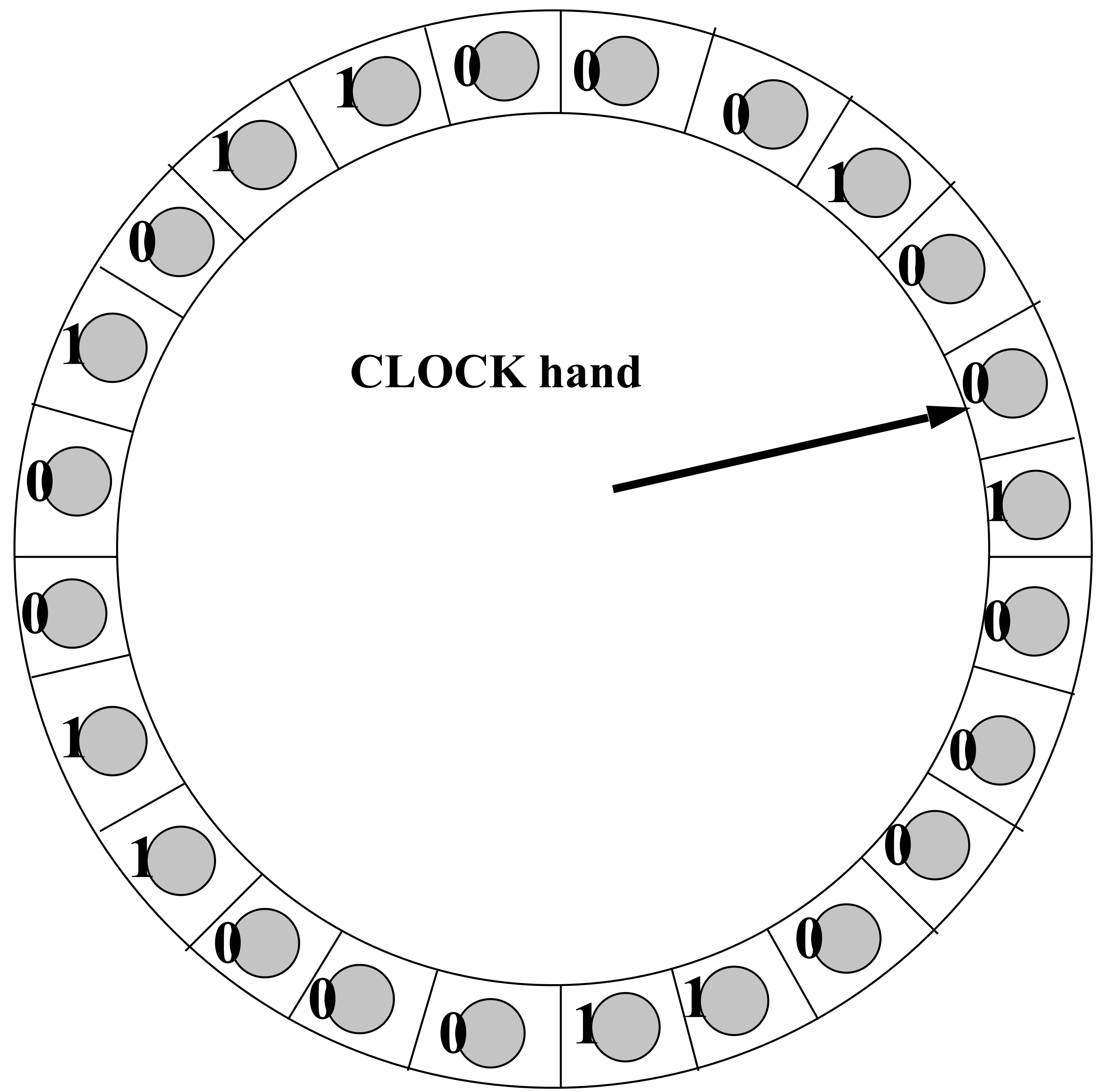
Clock Replacement

- Another MISS



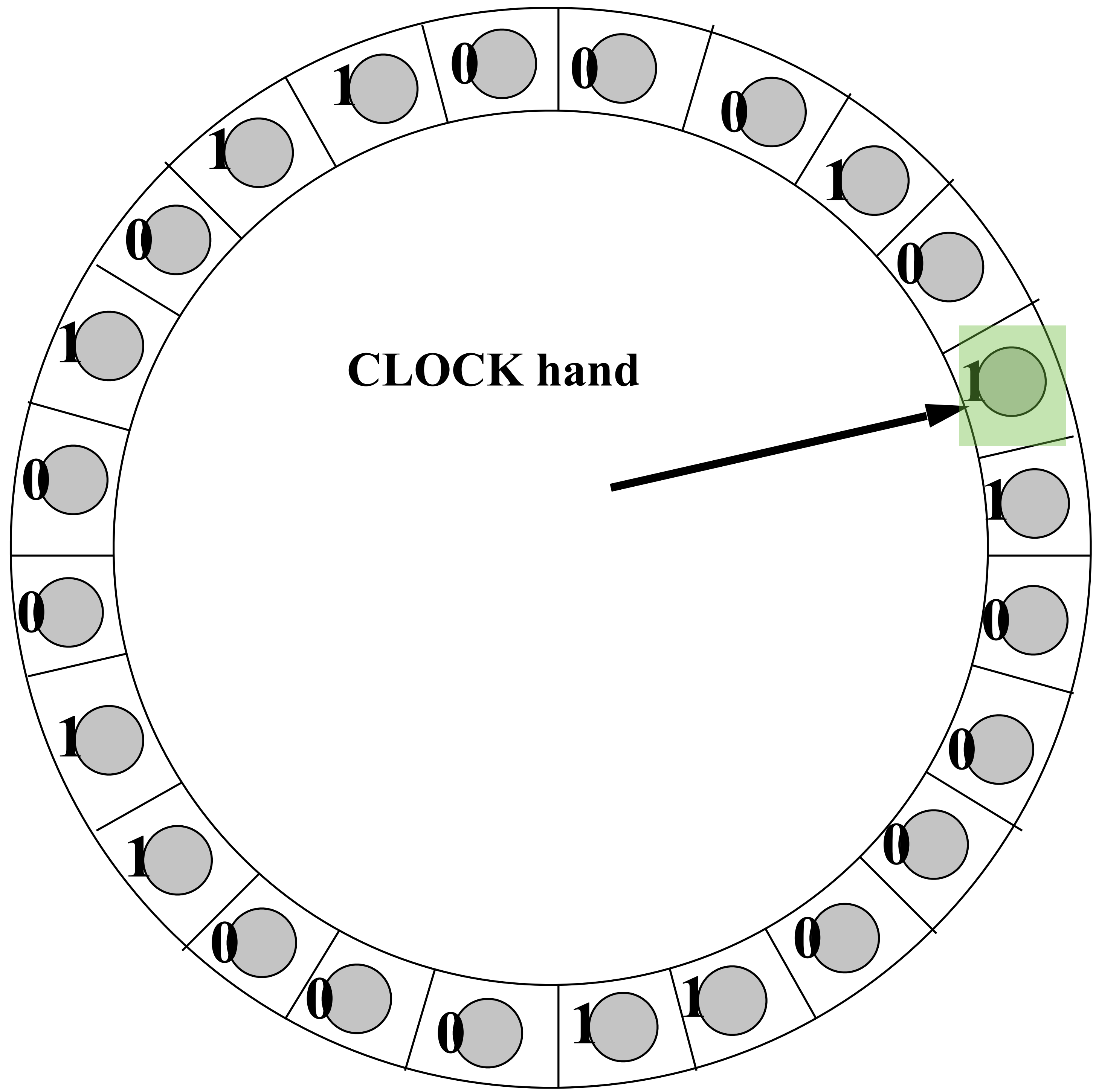
Clock Replacement

- Another MISS



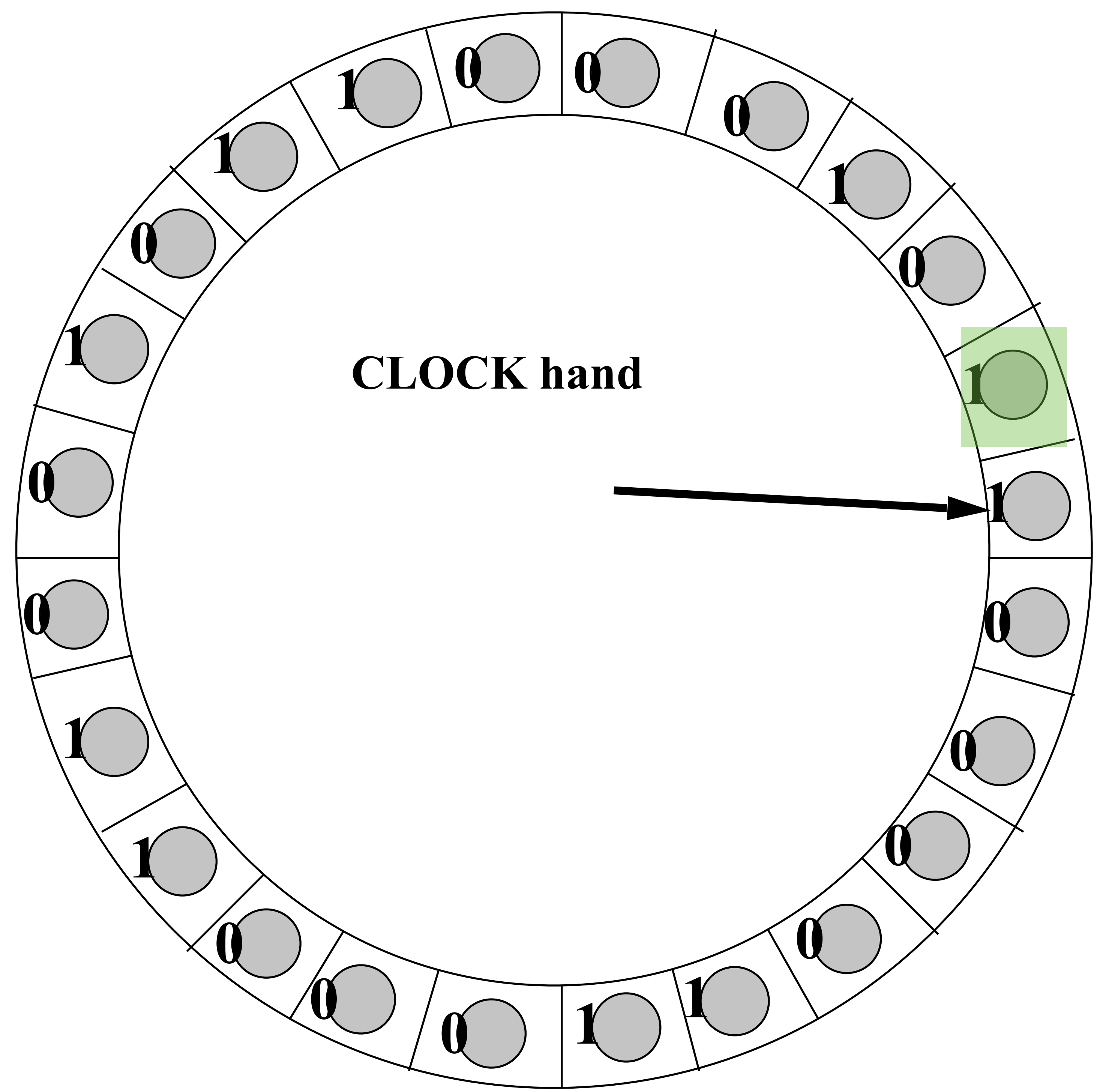
Clock Replacement

- Another MISS



Clock Replacement

- Another MISS



Worksheet: Buffer Replacement Policies

	A	B	C	D	A	F	A	D	G	D	G	E	D	F	
1	A														Hit Rate
2		B													
3			C												
4				D											

LRU

Access Pattern: A B C D A F A D G D G E D F

1	A				✓		✓							F	Hit Rate
2		B				F						E			6/14
3			C						G		✓				
4				D				✓		✓			✓		

MRU

Access Pattern: A B C D A F A D G D G E D F

1	A				✓	F	A								Hit Rate
2		B													2/14
3			C												
4				D				✓	G	D	G	E	D	F	

Clock Replacement

Access Pattern: A B C D A F A D G D G E D F

1	A			(1)	✓	F (1)	(1)	(1)	(1)	(1)	(1)	(0)	D(1)	(0)	Hit Rate
2		B		(1)		(0)	A (1)	(1)	(1)	(1)	(1)	(0)	(0)	F(1)	
3			C	(1)		(0)	(0)	(0)	G (1)	(1)	✓(1)	(0)	(0)	(0)	4/14
4				D(1)		(0)	(0)	✓(1)	(1)	✓(1)	(1)	E(1)	(0)	(0)	

When is LRU the worst replacement policy?

When is LRU the worst replacement policy?

Sequential scans

Why would we use a clock replacement policy over LRU?

Why would we use a clock replacement policy over LRU?

Efficiency (approximation of LRU)

Heap Files

(Page Formats)

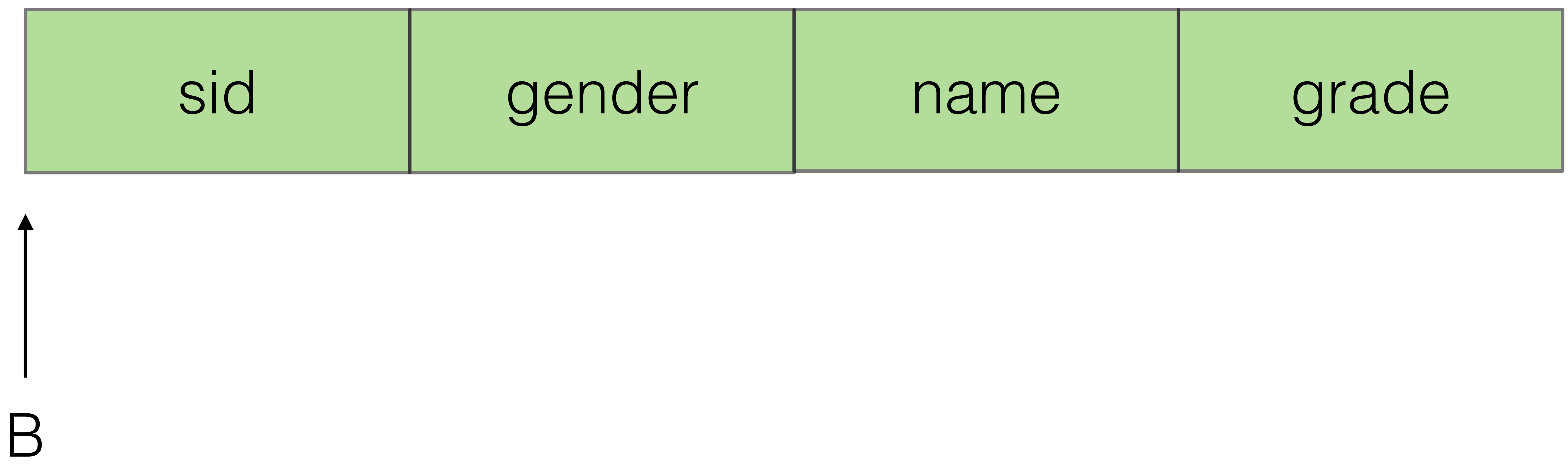
Fixed-Length Records

sid	gender	name	grade

Each field same length

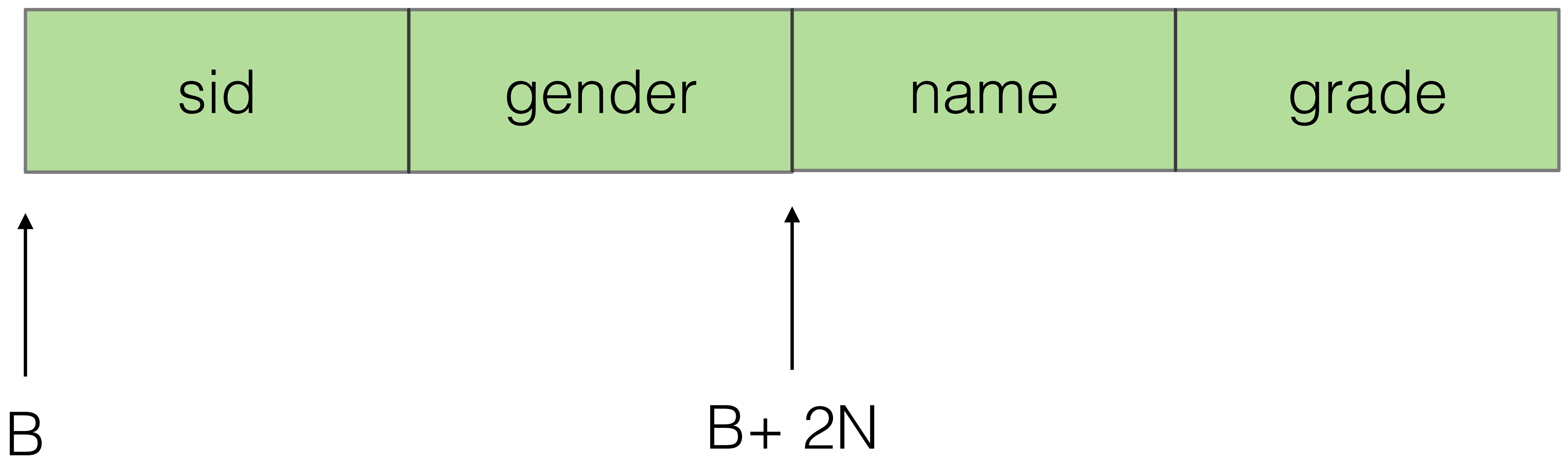
Fixed-Length Records

Each field of size N

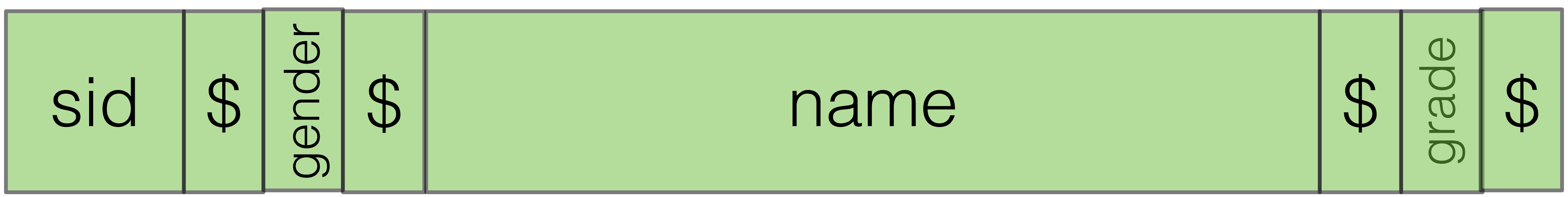


Fixed-Length Records

Each field of size N

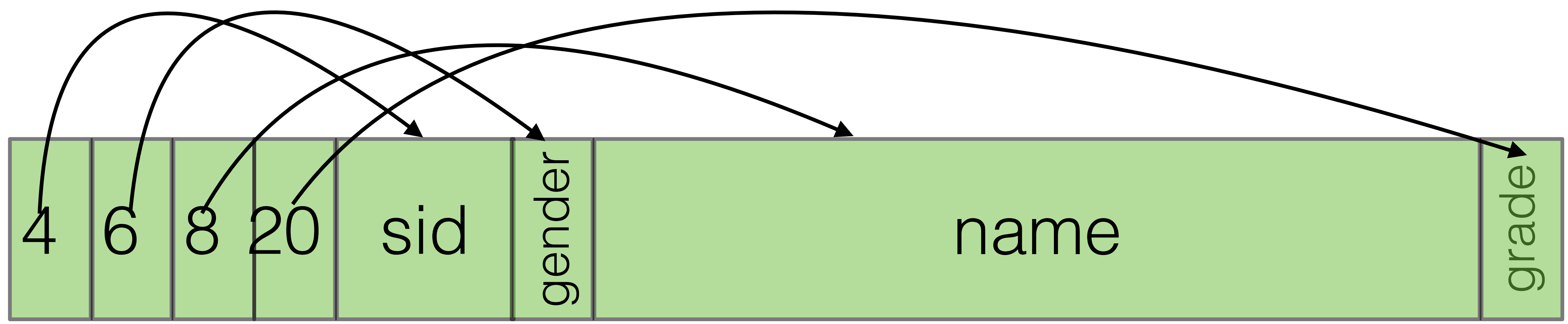


Variable-Length Records



Delimiters

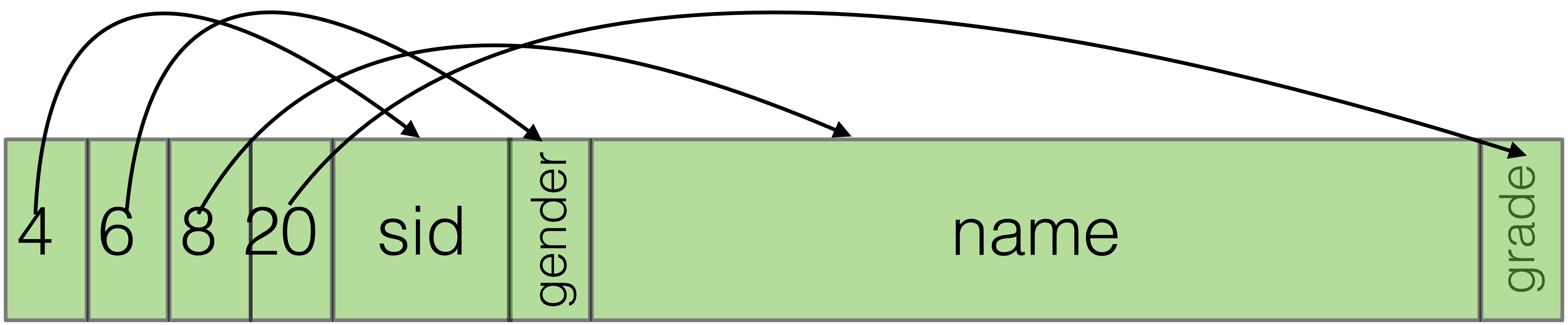
Variable-Length Records



Array of Field Offsets

Benefits?

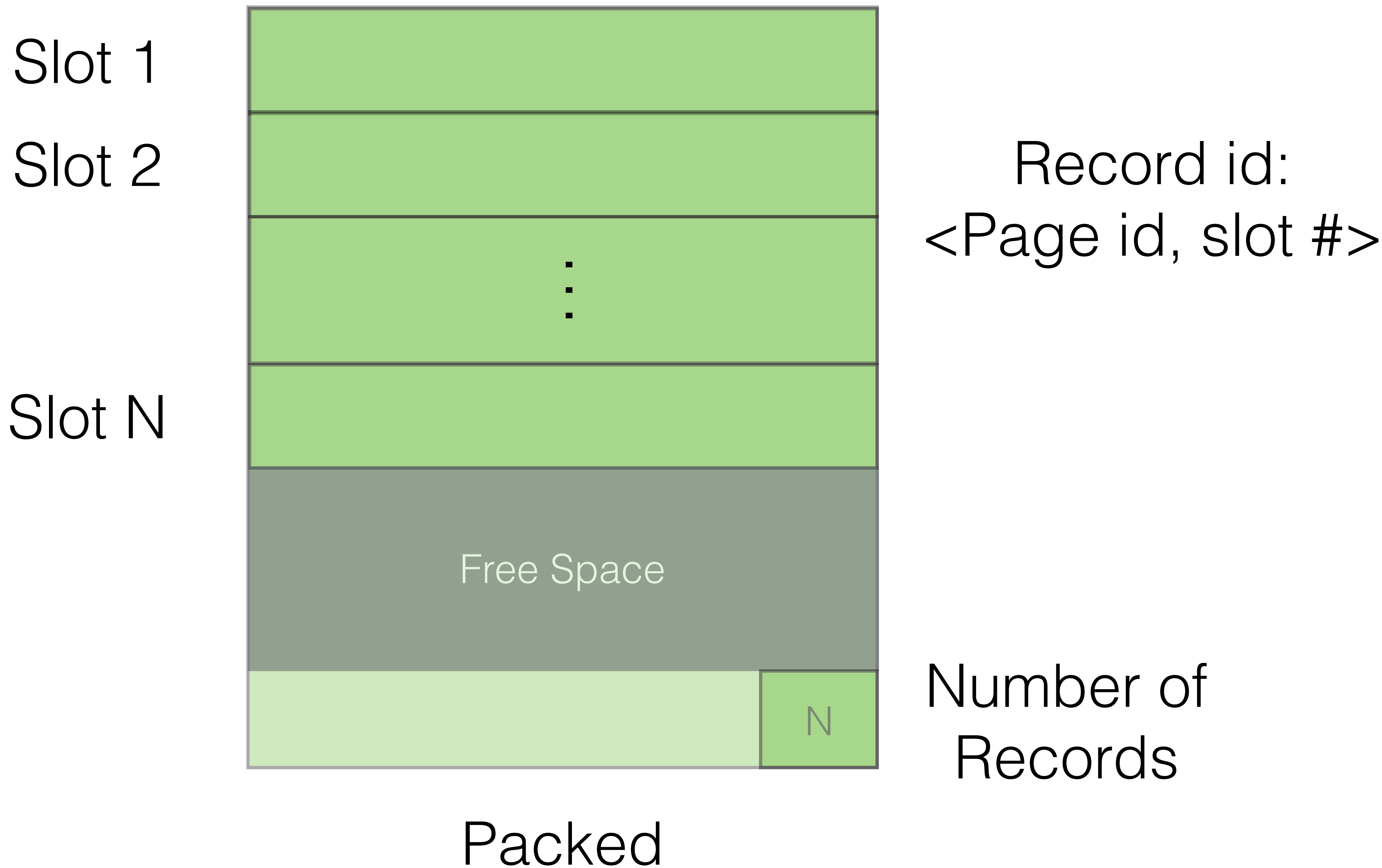
Variable-Length Records



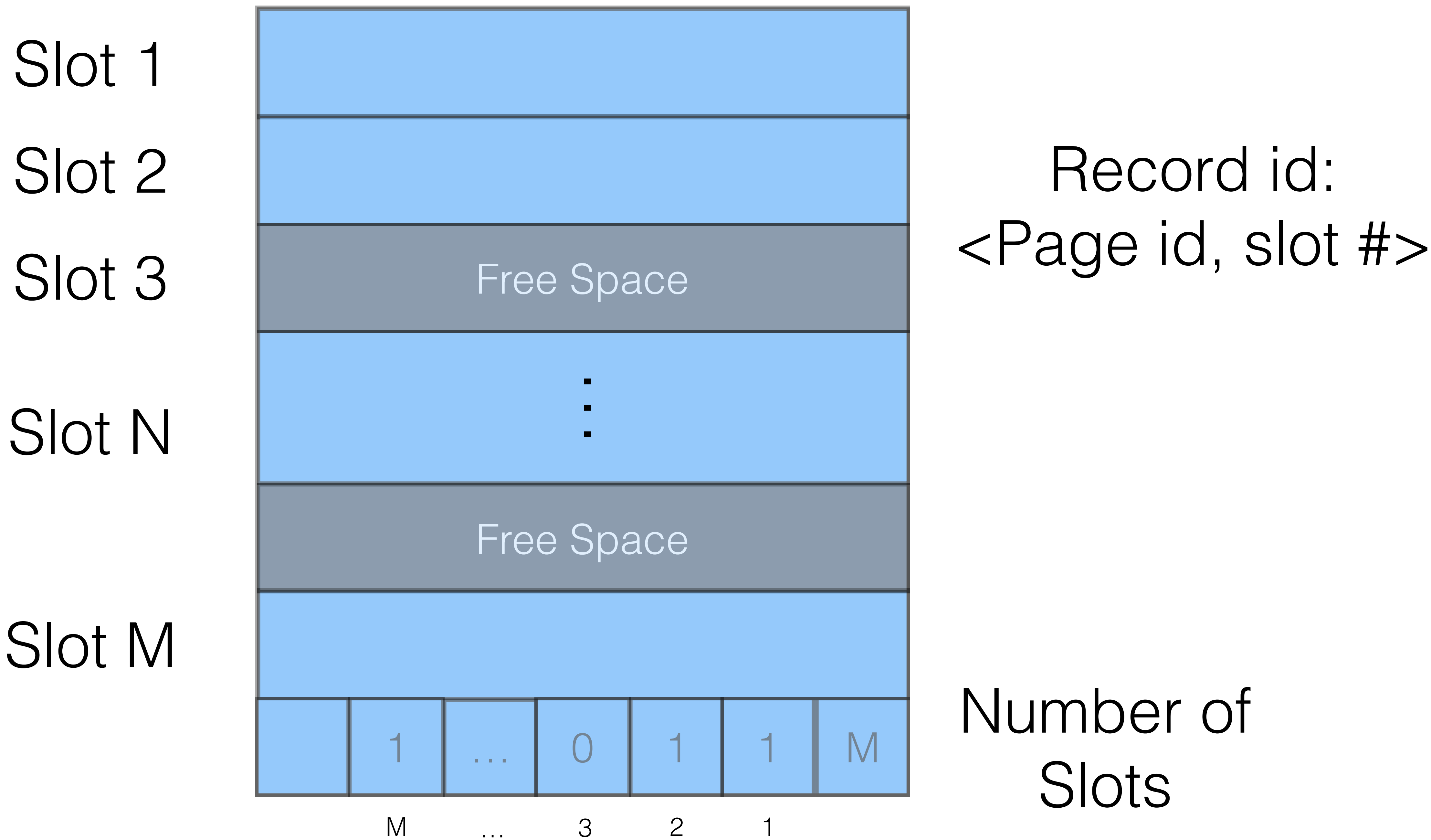
Array of Field Offsets

Benefits?
Efficient storage of nulls

Fixed-Length Records

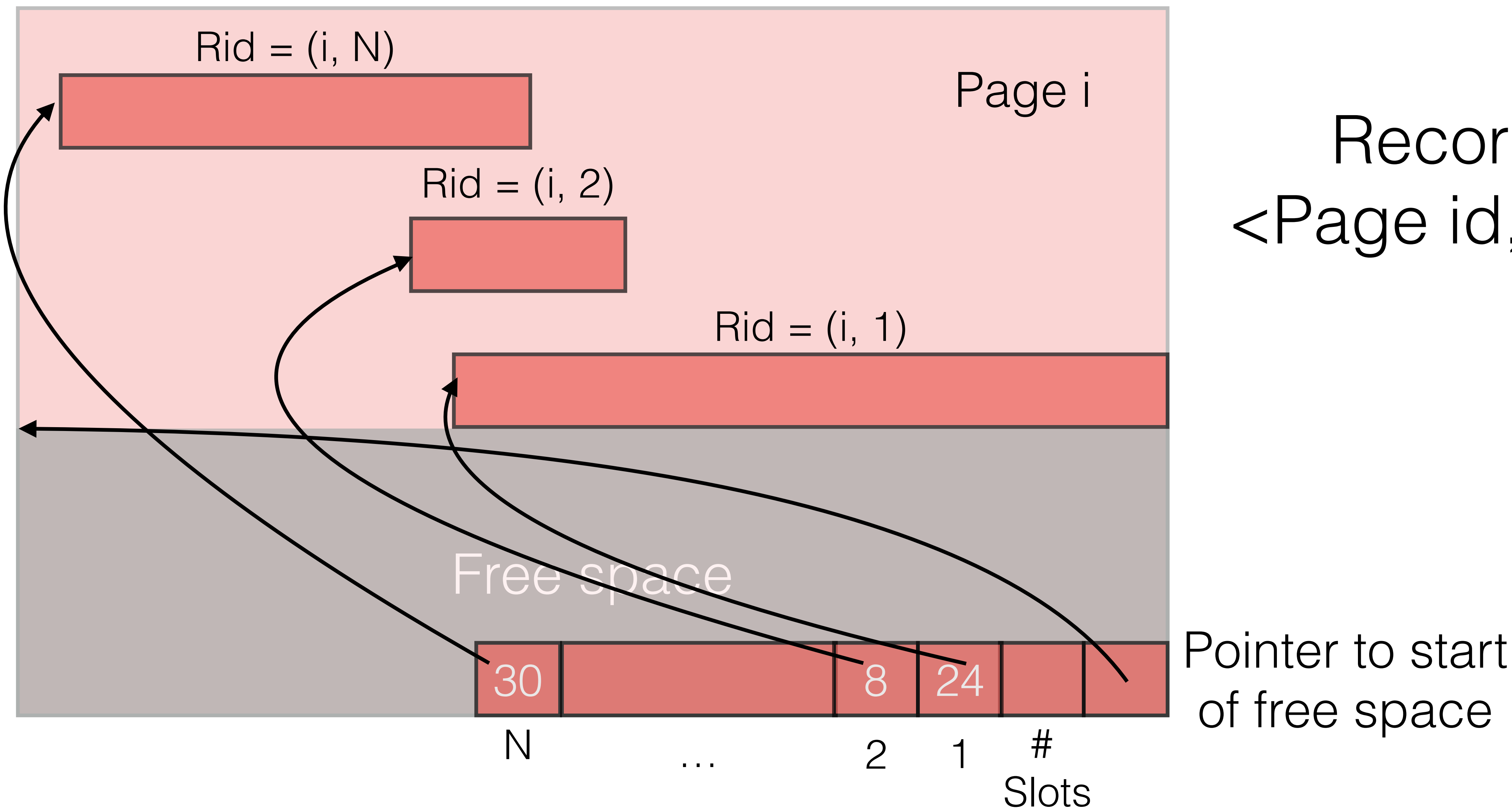


Fixed-Length Records



Unpacked (Bitmap)

Variable Length Records



Record id:
<Page id, slot #>

Slotted Page

Worksheet: Heap Files

What can we do to support variable length records (over fixed length records)?

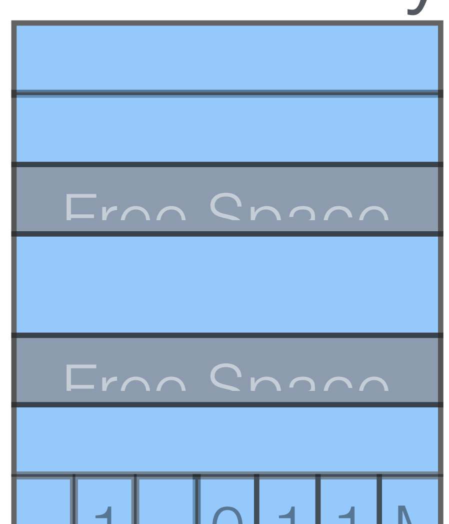
What can we do to support variable length records (over fixed length records)?

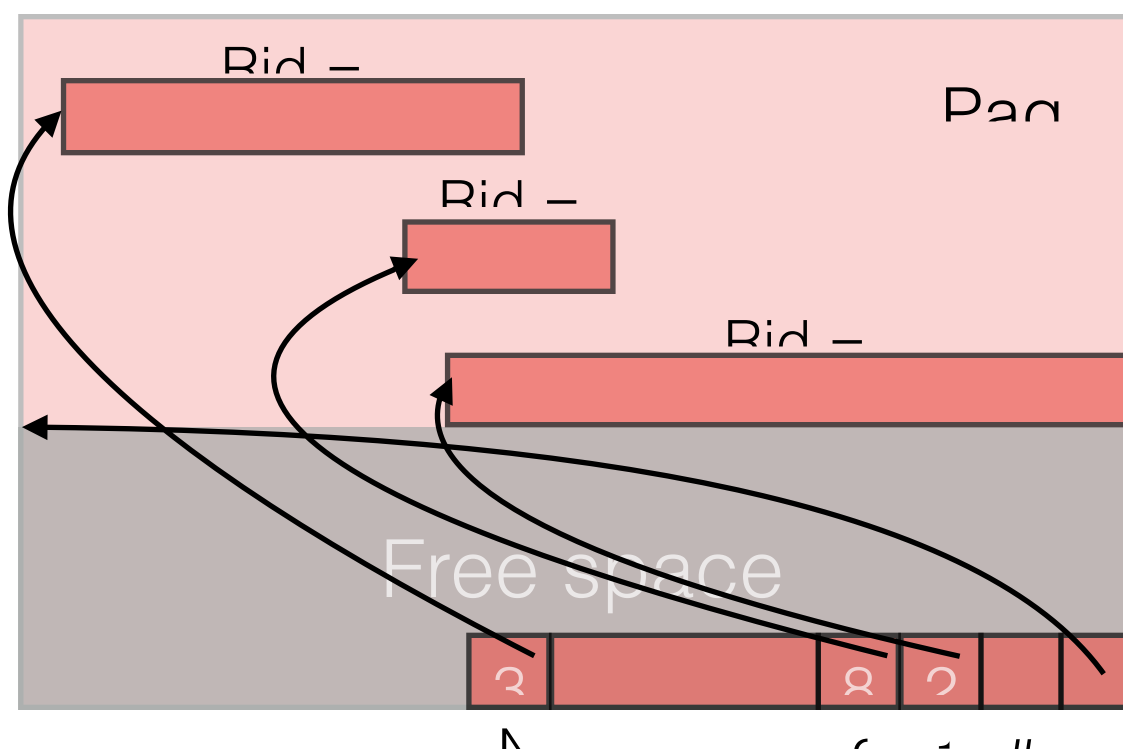
- Delimit with special symbols or use an array of field offsets

What are the advantages and disadvantages of using slotted pages or bitmaps over just tightly packing records together?

What are the advantages and disadvantages of using slotted pages or bitmaps over just tightly packing records together?

- Allow movement of records without changing record ID
- Slotted pages support variable-length records
- BUT: Needs page directory





You have a slotted page with 80 bytes of free space, and it costs 4 bytes to store a directory entry.

What's the size of the largest record you can insert?

You have a slotted page with 80 bytes of free space, and it costs 4 bytes to store a directory entry.

What's the size of the largest record you can insert?

Need 4 bytes for the entry, so (80 - 4) = 76 bytes

You have a slotted page with 80 bytes of free space, and it costs 4 bytes to store a directory entry.

At most, how many 1-byte large records can you insert?

You have a slotted page with 80 bytes of free space, and it costs 4 bytes to store a directory entry.

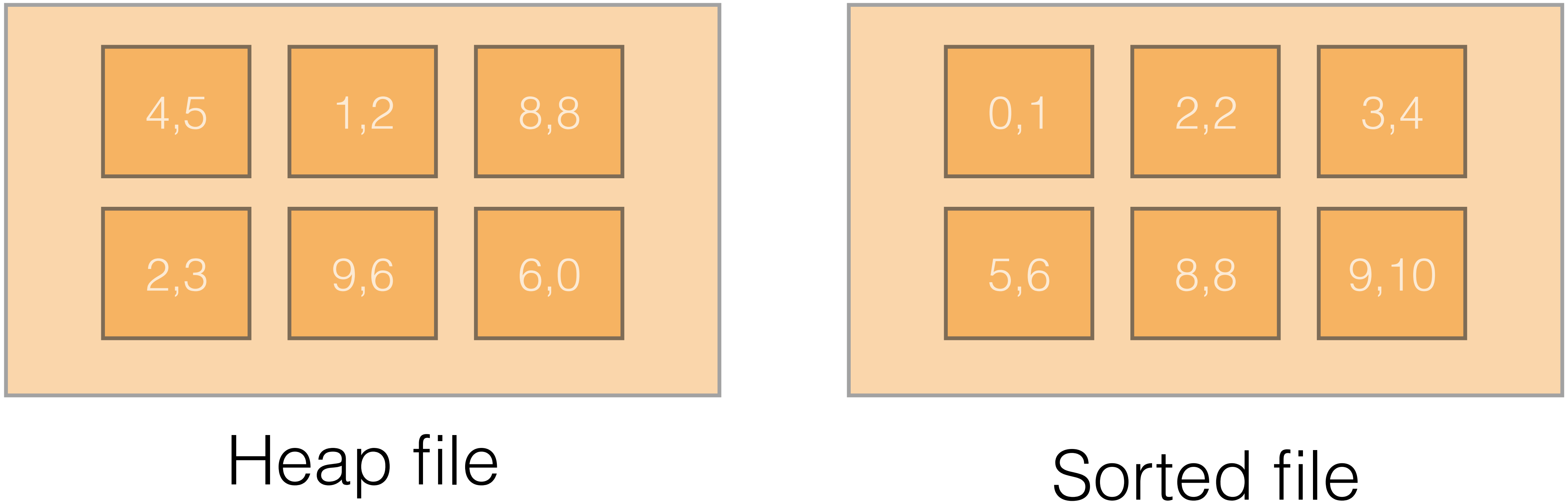
At most, how many 1-byte large records can you insert?

- Amount of space taken up by x 1-byte records
= (1 byte for record + 4 for directory entry)
= (5 bytes / record)

- Free space / (amount of space per record)
= 80 / 5 = 16 records

File Organization

- Heap files: unordered set of records
- Sorted file: ordered set of records



Average I/O Costs

B = pages in file

Operation	Heap File	Sorted File
Scan all records		
Equality Search		
Range Search		
Insert		
Delete		

Average I/O Costs

B = pages in file

Operation	Heap File	Sorted File
Scan all records	B	B
Equality Search		
Range Search		
Insert		
Delete		

Average I/O Costs

B = pages in file

Operation	Heap File	Sorted File
Scan all records	B	B
Equality Search	0.5B	
Range Search		
Insert		
Delete		

Average I/O Costs

B = pages in file

Operation	Heap File	Sorted File
Scan all records	B	B
Equality Search	0.5B	log_2(B)
Range Search		
Insert		
Delete		

Average I/O Costs

B = pages in file

Operation	Heap File	Sorted File
Scan all records	B	B
Equality Search	0.5B	log_2(B)
Range Search	B	
Insert		
Delete		

Average I/O Costs

B = pages in file

Operation	Heap File	Sorted File
Scan all records	B	B
Equality Search	0.5B	log_2(B)
Range Search	B	log_2(B) + # pages matched
Insert		
Delete		

Average I/O Costs

B = pages in file

Operation	Heap File	Sorted File
Scan all records	B	B
Equality Search	0.5B	log_2(B)
Range Search	B	log_2(B) + # pages matched
Insert	2	
Delete		

Average I/O Costs

B = pages in file

Operation	Heap File	Sorted File
Scan all records	B	B
Equality Search	0.5B	log_2(B)
Range Search	B	log_2(B) + # pages matched
Insert	2	log_2(B)+ (B/2) *2
Delete		

Average I/O Costs

B = pages in file

Operation	Heap File	Sorted File
Scan all records	B	B
Equality Search	0.5B	log_2(B)
Range Search	B	log_2(B) + # pages matched
Insert	2	log_2(B)+ (B/2) *2
Delete	0.5B+1	

Average I/O Costs

B = pages in file

Operation	Heap File	Sorted File
Scan all records	B	B
Equality Search	0.5B	log_2(B)
Range Search	B	log_2(B) + # pages matched
Insert	2	log_2(B)+ (B/2) *2
Delete	0.5B+1	log_2(B)+ (B/2) *2

Worksheet: File Organization

Consider the table Enrolled(sid, course, grade) with 500 pages, 6,000 tuples and the query `SELECT * FROM Enrolled where sid > 4500`.
Assume SIDs are unique and range from 0 to 6000.

- How many I/Os would this query take if the table was stored in a heap file?

Consider the table Enrolled(sid, course, grade) with 500 pages, 6,000 tuples and the query `SELECT * FROM Enrolled where sid > 4500.`
Assume SIDs are unique and range from 0 to 6000.

- How many I/Os would this query take if the table was stored in a heap file?

B = 500

Consider the table Enrolled(sid, course, grade) with 500 pages, 6,000 tuples and the query `SELECT * FROM Enrolled where sid > 4500.`

Assume SIDs are unique and range from 0 to 6000.

- How many I/Os would this take if the table was stored in a sorted file sorted by grade?

Consider the table Enrolled(sid, course, grade) with 500 pages, 6,000 tuples and the query `SELECT * FROM Enrolled where sid > 4500.`
Assume SIDs are unique and range from 0 to 6000.

- How many I/Os would this take if the table was stored in a sorted file sorted by grade?

B = 500

Consider the table Enrolled(sid, course, grade) with 500 pages, 6,000 tuples and the query `SELECT * FROM Enrolled where sid > 4500.`

Assume SIDs are unique and range from 0 to 6000.

- How many I/Os would this take if the table was stored in a sorted file sorted by SID?

Consider the table Enrolled(sid, course, grade) with 500 pages, 6,000 tuples and the query `SELECT * FROM Enrolled where sid > 4500.`
Assume SIDs are unique and range from 0 to 6000.

- How many I/Os would this take if the table was stored in a sorted file sorted by SID?

$$\log_2(500) + (6000-4500)/6000 * 500$$

$$= \log_2(500) + \tfrac{1}{4} * 500$$

Given 6 buffer pages and an access pattern of pages: A, R, B, T, P, H, A, C, N, M, O, A, A, D, E, A, B, C, B, E, A, F, G, H, A, B, C, B, C, E, I, R, H, T, A, C, D, A, O, B, N, C, T, D, F, E, A, G, I, A, I, L, Which pages are in the buffer pool at the end if we used an LRU cache policy?

Given 6 buffer pages and an access pattern of pages: A, R, B, T, P, H, A, C, N, M, O, A, A, D, E, A, B, C, B, E, A, F, G, H, A, B, C, B, C, E, I, R, H, T, A, C, D, A, O, B, N, C, T, D, F, E, A, G, I, A, I, L, Which pages are in the buffer pool at the end if we used an LRU cache policy?

LIAGEF

Given this variable length record with fields delimited by the special character '\0'. For simplicity, every field is a variable length string of 1 byte characters and each bucket is a 1 byte bucket.

'j'	'o'	'e'	'\0'	's'	'm'	'i'	't'	'h'	'\0'	'2'	'9'	'1'	'\0'	'3'	'.'	'2'	'\0'

Show how the record would be organized using the array of field offsets representation of a variable length record (assume the first byte of the record is byte 0 and we use one byte for each offset value)?

Given this variable length record with fields delimited by the special character '\0'. For simplicity, every field is a variable length string of 1 byte characters and each bucket is a 1 byte bucket.

'j'	'o'	'e'	'\0'	's'	'm'	'i'	't'	'h'	'\0'	'2'	'9'	'1'	'\0'	'3'	'.'	'2'	'\0'

Show how the record would be organized using the array of field offsets representation of a variable length record (assume the first byte of the record is byte 0 and we use one byte for each offset value)?

4	7	12	15	'j'	'o'	'e'	's'	'm'	'i'	't'	'h'	'2'	'9'	'1'	'3'	'.'	'2'	