IE 411: Operating Systems

Flash-based SSDs

Solid-state Storage Devices (SSDs)

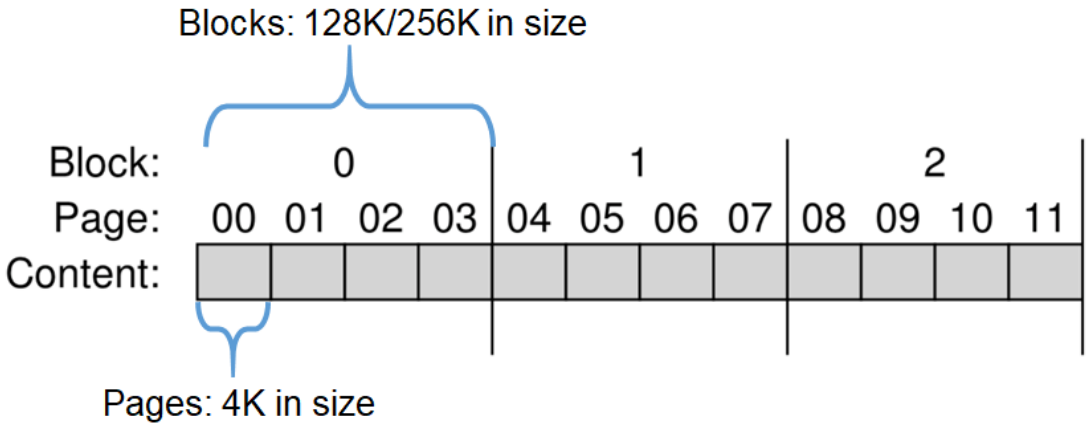
- Unlike hard drives, SSDs have no mechanical parts
- SSDs use transistors (like DRAM), but not quite DRAM either . . .
  - SSD data persists when the power goes out
- Based on a technology known as flash (NAND-based flash)
  - `https://en.wikipedia.org/wiki/Fujio_Masuoka`

Solid-state Storage Devices (SSDs)

- SSDs have a higher \$/bit than hard drives, but better performance (no mechanical delays!)
- SSDs handle writes in a strange way; this has implications for file system design

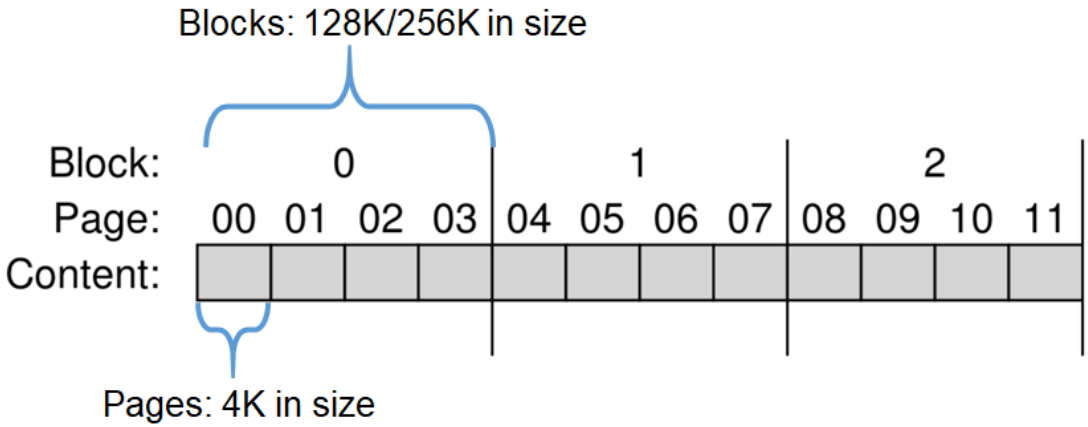
SSD organization

- An SSD contains blocks made of pages



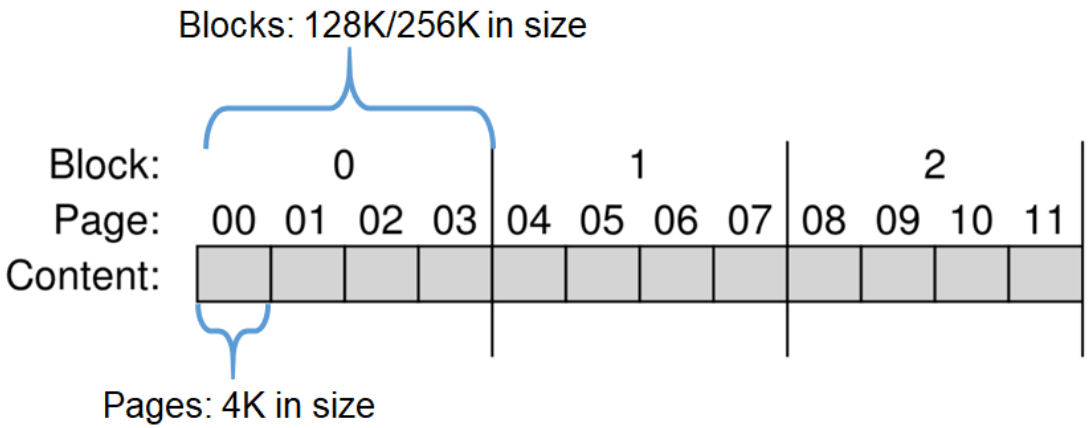
- Read a page
  - Can read any page by specifying the read command and a page number
  - Fast operation: 10s of microseconds
  - Regardless of the location of previous request (random access device)

SSD operations



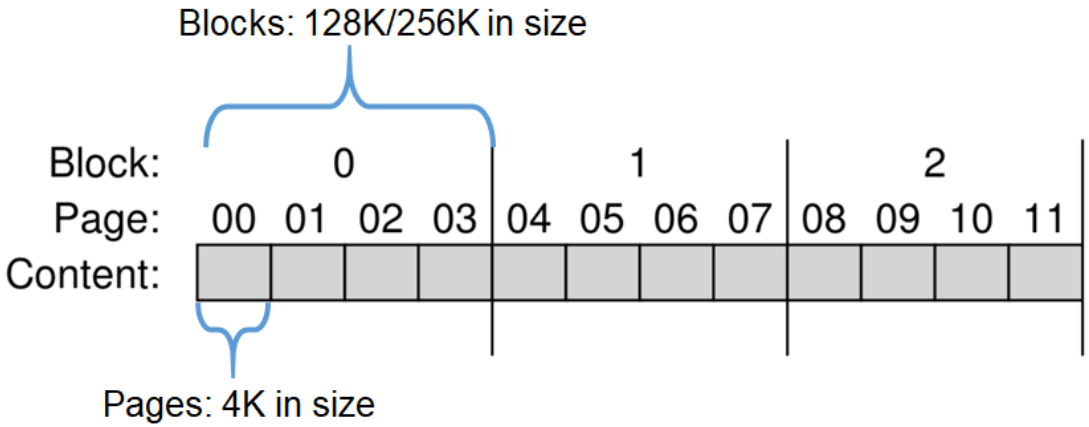
- Erase a block
  - Destroys the contents of the block by setting all bits to the value 1
  - Slow operation: a few milliseconds

SSD operations



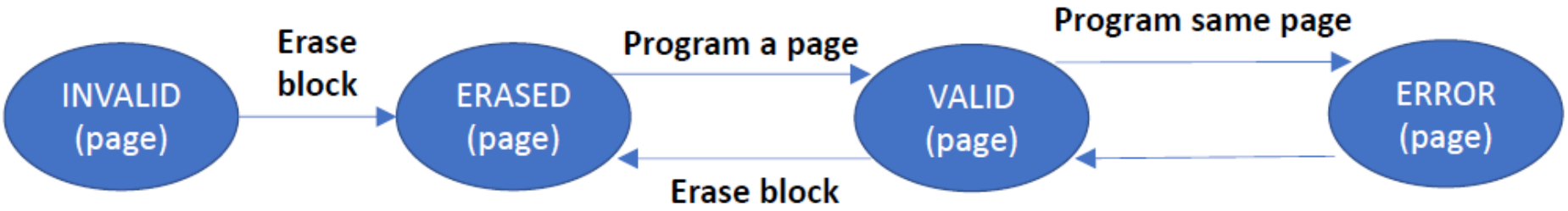
- Erase a block
  - Destroys the contents of the block by setting all bits to the value 1
  - Slow operation: a few milliseconds
  - Before writing to a page within a block, the entire block must be erased!

SSD operations



- Program (i.e., write) a page
  - Writes data to an erased page by changing some of the ones within a page to zeros
  - Less costly than erasing a block, but more costly than reading a page
  - 100s of microseconds

A day in the life of a page



		`iiii`	*Initial: pages in block are* `invalid(i)`
`Erase()`	→	`EEEE`	*State of pages in block are set to* `erased(E)`
`Program(0)`	→	`VEEE`	*Program page 0; state set to* `valid(v)`
`Program(0)`	→	**`error`**	*Cannot re-program page after programming*
`Program(1)`	→	`VVEE`	*Program page 1*
`Erase()`	→	`EEEE`	*Contents erased; all pages programmable*

Example

- Four 8-bit pages within a 4-page block
- Each page is VALID as each has been previously programmed

Page 0	Page 1	Page 2	Page 3
00011000	11001110	00000001	00111111
VALID	VALID	VALID	VALID

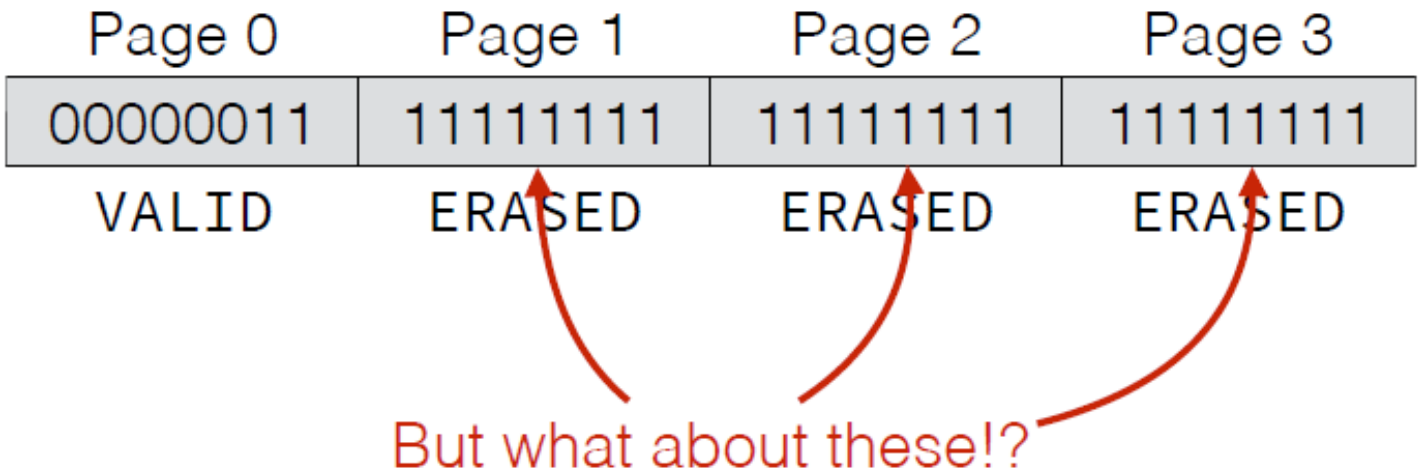
Example

- Say we wish to write to page 0, filling it with new contents
- To write any page, we must first erase the entire block

Page 0	Page 1	Page 2	Page 3
11111111	11111111	11111111	11111111
ERASED	ERASED	ERASED	ERASED

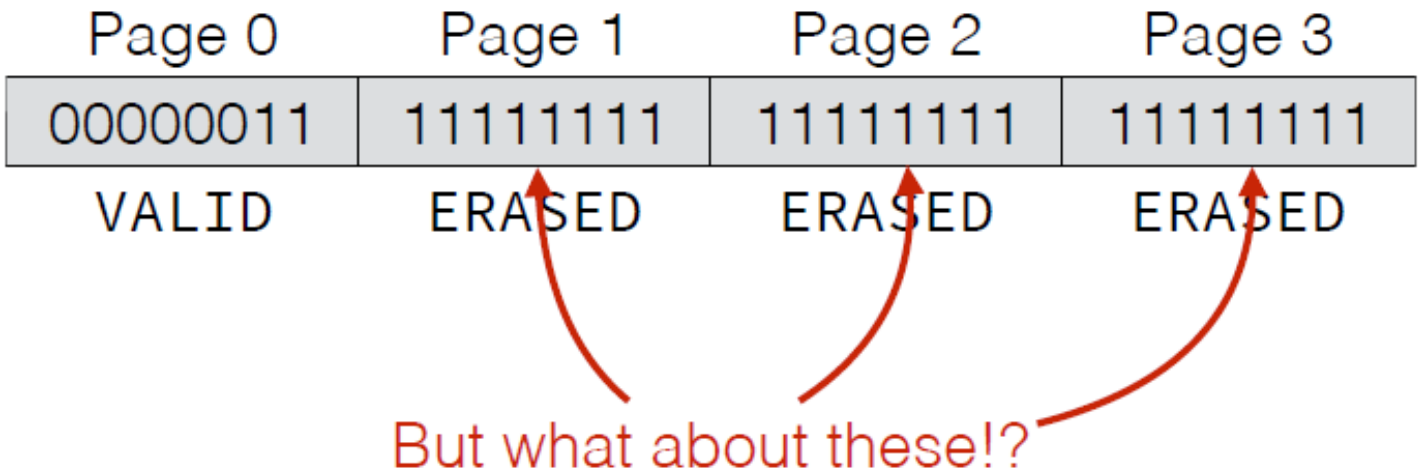
Example

- Now program page 0 with the required content, e.g. 00000011, overwriting the old page 0 (old contents 00011000)



Example

- Now program page 0 with the required content, e.g. 00000011, overwriting the old page 0 (old contents 00011000)



- Before erasing entire block must move other pages to another location (e.g., memory, or elsewhere on the flash)

SSD Reliability

- No need to worry about head crashes, unlike hard disks
- Primary concern: wear-out
  - if you erase a block a lot, it may become unusable

Flash Translation Layer (FTL)

- Typically implemented inside the storage device
- Maps logical blocks (that the OS uses) to physical blocks (that the device uses)
- Performance goals
  - wear leveling – try to spread writes across blocks to minimize wearing out certain blocks
  - garbage collection – consolidates valid data into new blocks, frees blocks with invalid data

FTL Approach 1: Direct Mapping

- Have a 1-1 correspondence between logical pages and physical pages
- Reading a page is straightforward
- Writing a page is trickier
  - Read the entire physical block into memory
  - Update the relevant page in the in-memory block
  - Erase the entire physical block
  - Program the entire physical block using the new block value

FTL Approach 1: Direct Mapping

- Problem 1: Write amplification
  - Writing a single page requires reading and writing an entire block

FTL Approach 1: Direct Mapping

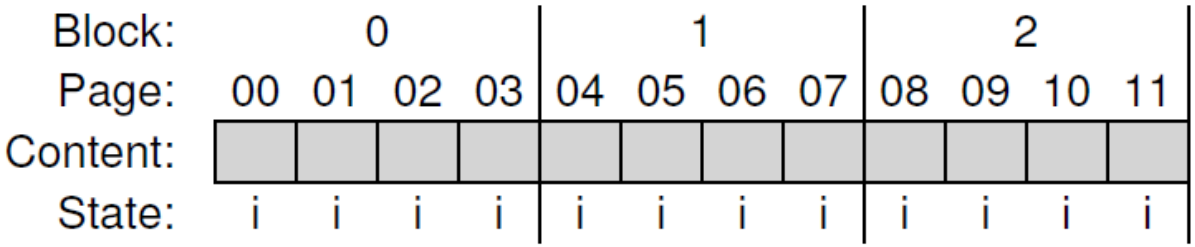
- Problem 1: Write amplification
  - Writing a single page requires reading and writing an entire block
- Problem 2: Poor reliability
  - If the same logical block is repeatedly written, its physical block will quickly fail
  - Particularly unfortunate for logical metadata blocks

FTL Approach 2: Log-based mapping

- Treat the physical blocks like a log
- Send data in each page-to-write to the end of the log
- Maintain a mapping (inside SSD) between logical pages and the corresponding physical pages

Log-structured FTL example

- Assume the client issues the following sequence of operations
  - Write(100) with contents a1
  - Write(101) with contents a2
  - Write(2000) with contents b1
  - Write(2001) with contents b2
- Initial state: all pages are INVALID (i)



Log-structured FTL example

- FTL decided to write logical block 100 to logical page 00
- It must therefore erase physical block 0

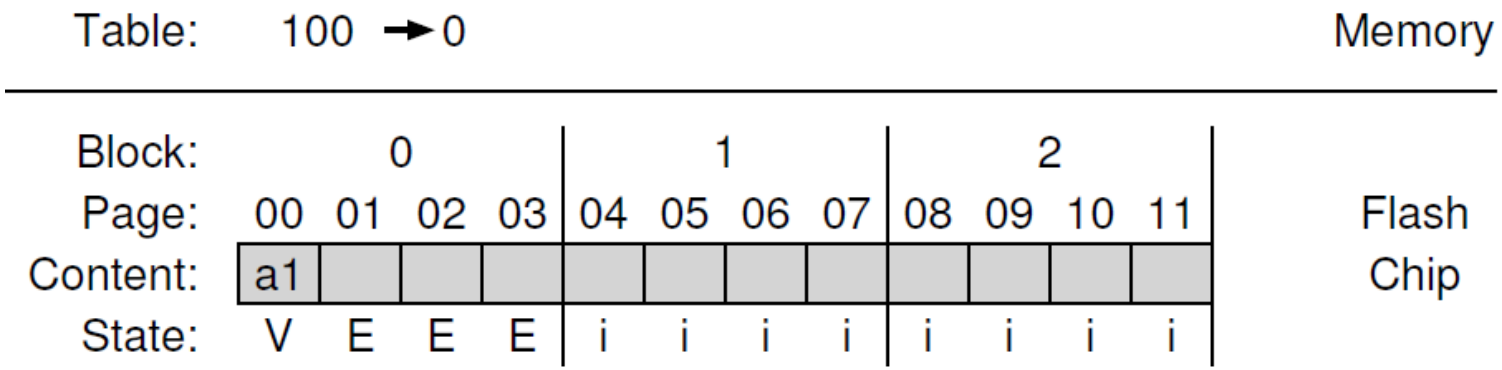
Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:												
State:	E	E	E	E	i	i	i	i	i	i	i	i

- Then it can write the page

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:	a1											
State:	V	E	E	E	i	i	i	i	i	i	i	i

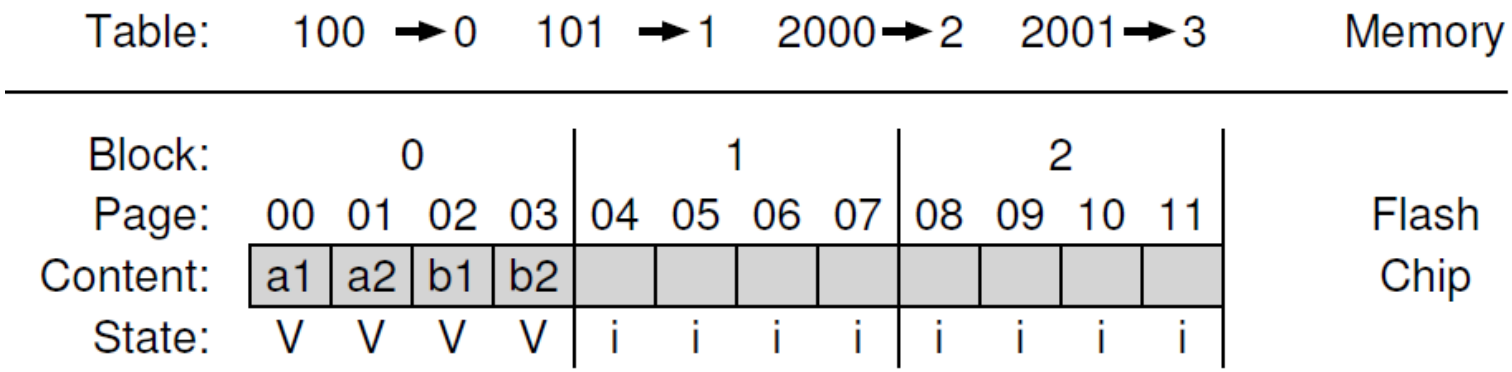
FTL example

- Mapping table tells us how to translated logical blocks to physical blocks



Log-structured FTL example

- We write the remaining blocks into the device

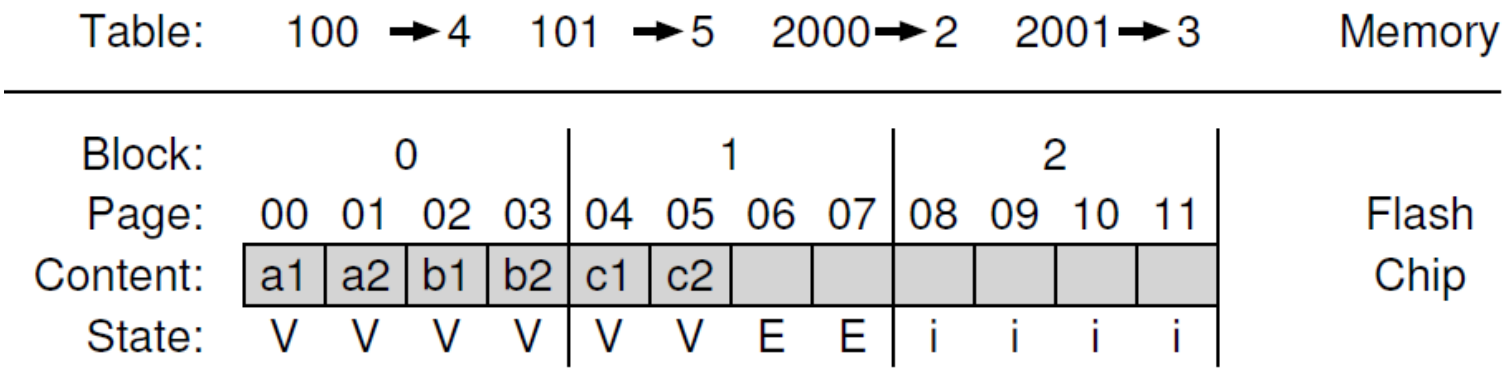


Two major issues

- First, overwrites of logical blocks leads to garbage, i.e., old version of data around the drive and taking up space
  - device has to periodically perform garbage collection (GC) to find said blocks and free space for future writes
  - excessive garbage collection drives up write amplification and lowers performance
- Second, high cost of in-memory mapping tables; the larger the device, the more memory such tables need

Garbage collection (via example)

- Assume that blocks 100 and 101 are written again, with contents c1 and c2
- Writes continue into physical block 1: SSD has to first erase that block and make it ready for programming

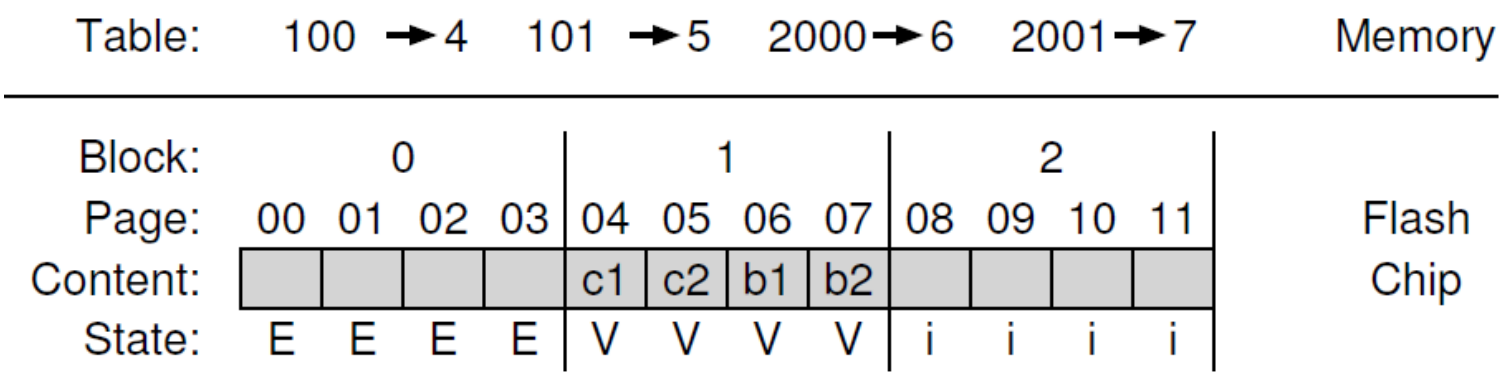


- We have accumulated two pages of garbage

Garbage collection (from time to time)

- Block 0 has two dead blocks (pages 0 and 1) and two live blocks (pages 2 and 3 which contain blocks 2000 and 2001, respectively)
- To garbage collect, the device will
  - read live data (pages 2 and 3) from block 0
  - write live data to end of the log
  - erase block 0 (freeing it for later usages)
- How does garbage collector know which pages are live within each block?
  - check the mapping table!

After garbage collection



- Rather an expensive operation, since blocks need to be erased and writes need to happen