CSE462/562: Database Systems (Fall 24)

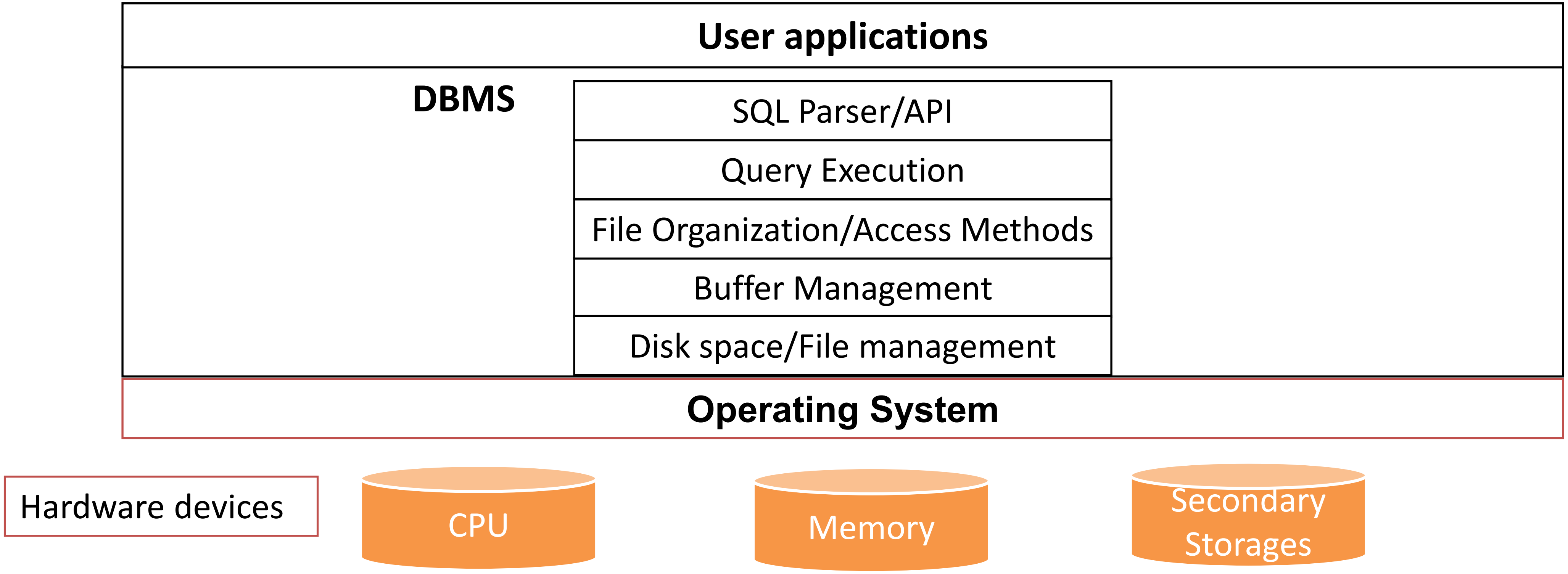
Lecture 3: Physical Storage, POSIX I/O Interface and Buffer Management

9/3/2024

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

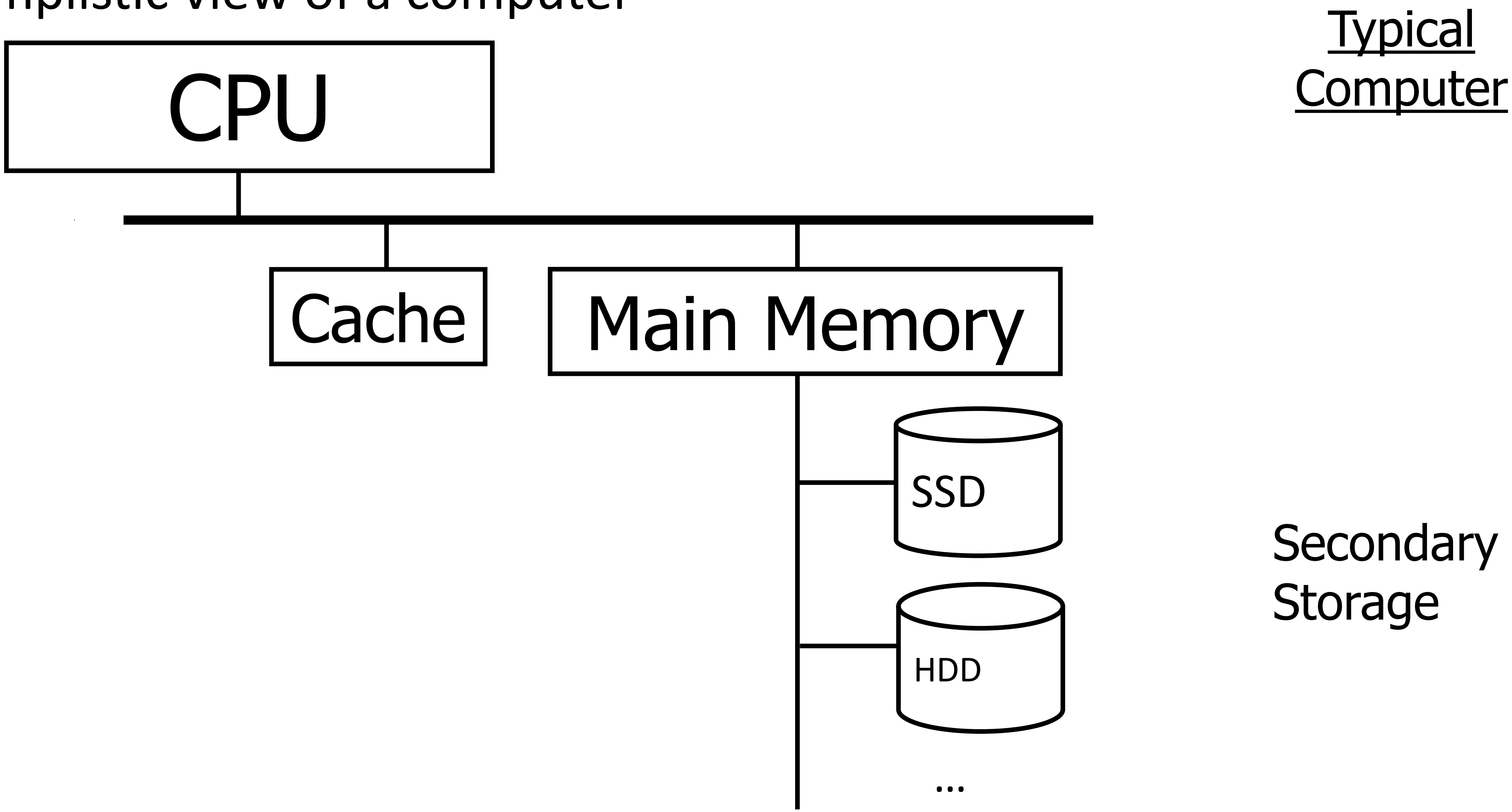
Last Update: 8/30/24, 1:00 PM

Big Picture

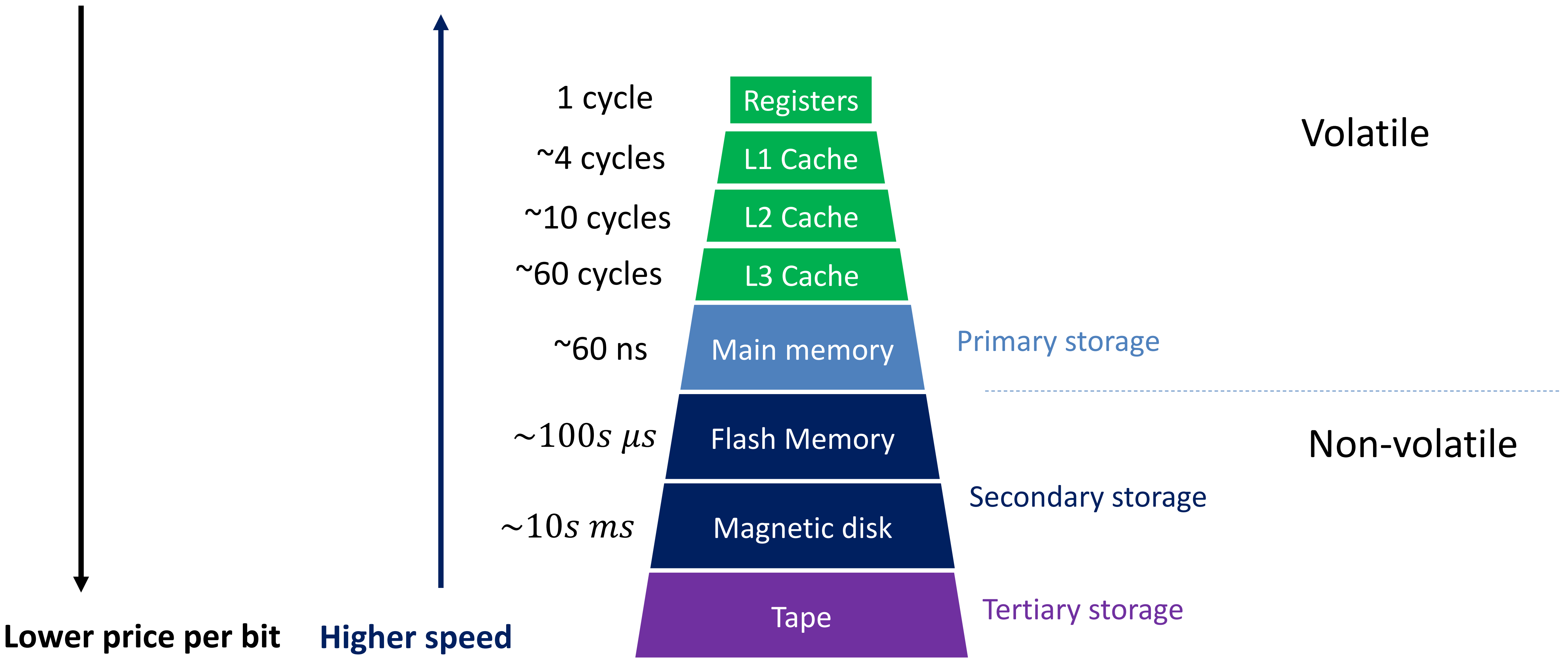

User applications
DBMS
SQL Parser/API
Query Execution
File Organization/Access Methods
Buffer Management
Disk space/File management
Operating System
Hardware devices
CPU
Memory
Secondary Storages

Typical (& oversimplified) computer architecture

- A simplistic view of a computer



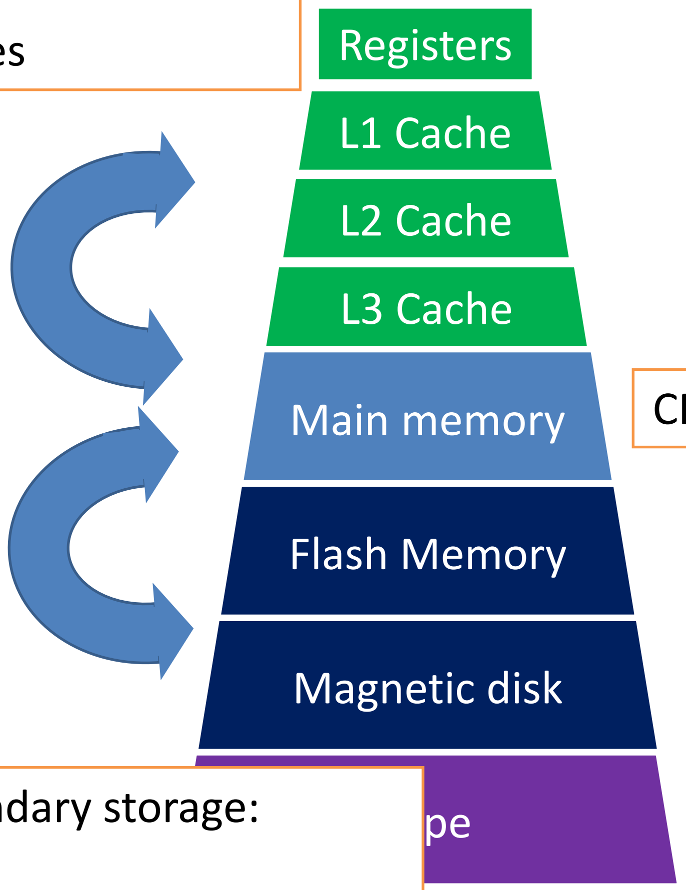
Storage Hierarchy



Data Transfers

Between cache and main memory:
hardware/OS controlled
usually in small units of cache lines

Volatile



CPU operates on main memory (byte addressable)

Non-volatile

Between main memory and secondary storage:
DBMS controlled (read/write)
usually with large block I/O

Non-volatile storage

- Common non-volatile (secondary) storage
  - Flash memory (e.g., SSD)
  - Magnetic disk

- Advantages
  - Cheaper -- can store much more data than memory with the same cost
  - Non-volatile – data are saved in server shutdown/power failure

- Disadvantages
  - Block device: read/write in the units of sectors (usually 512B/4096B)
  - Higher latency: usually >= 1 – 2 orders of magnitude slower than main memory

- Tertiary storage: tape (sequential I/O only)
  - Very slow but inexpensive; good for archiving data

Closer look at non-volatile storage

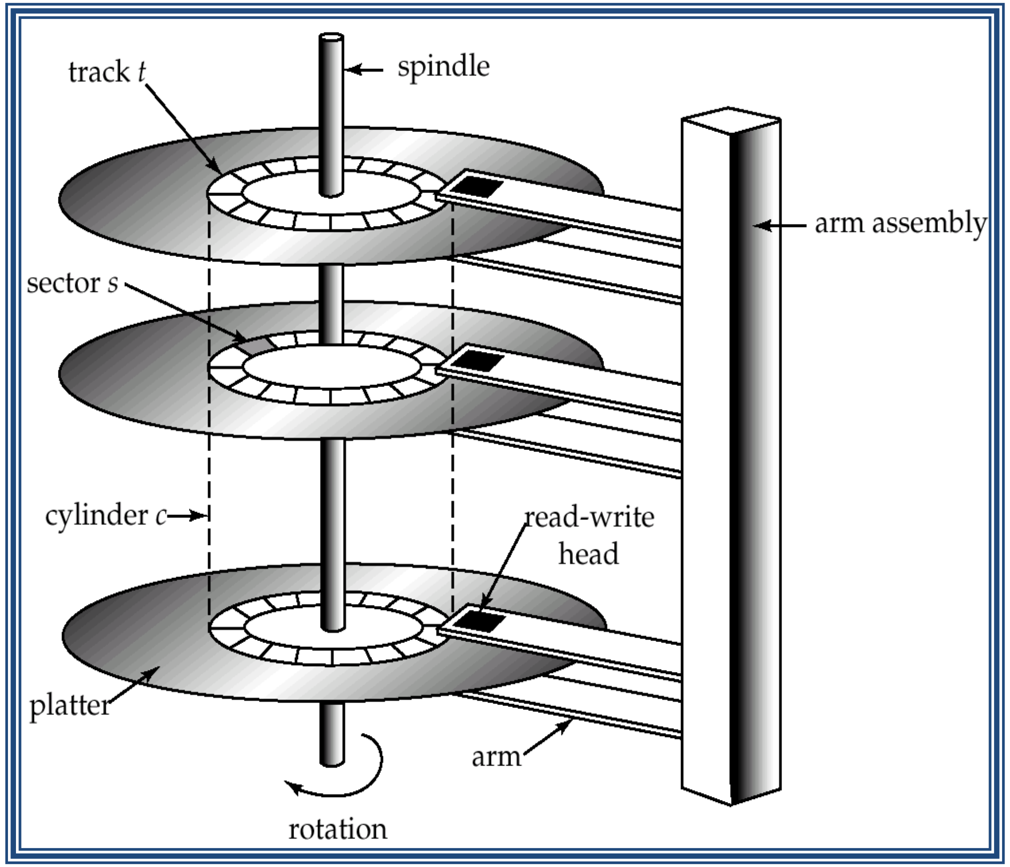
- We need to know the performance characteristics of non-volatile storage
  - to optimize database storage design

Magnetic disk (HDD)

Solid State Drive (SSD)

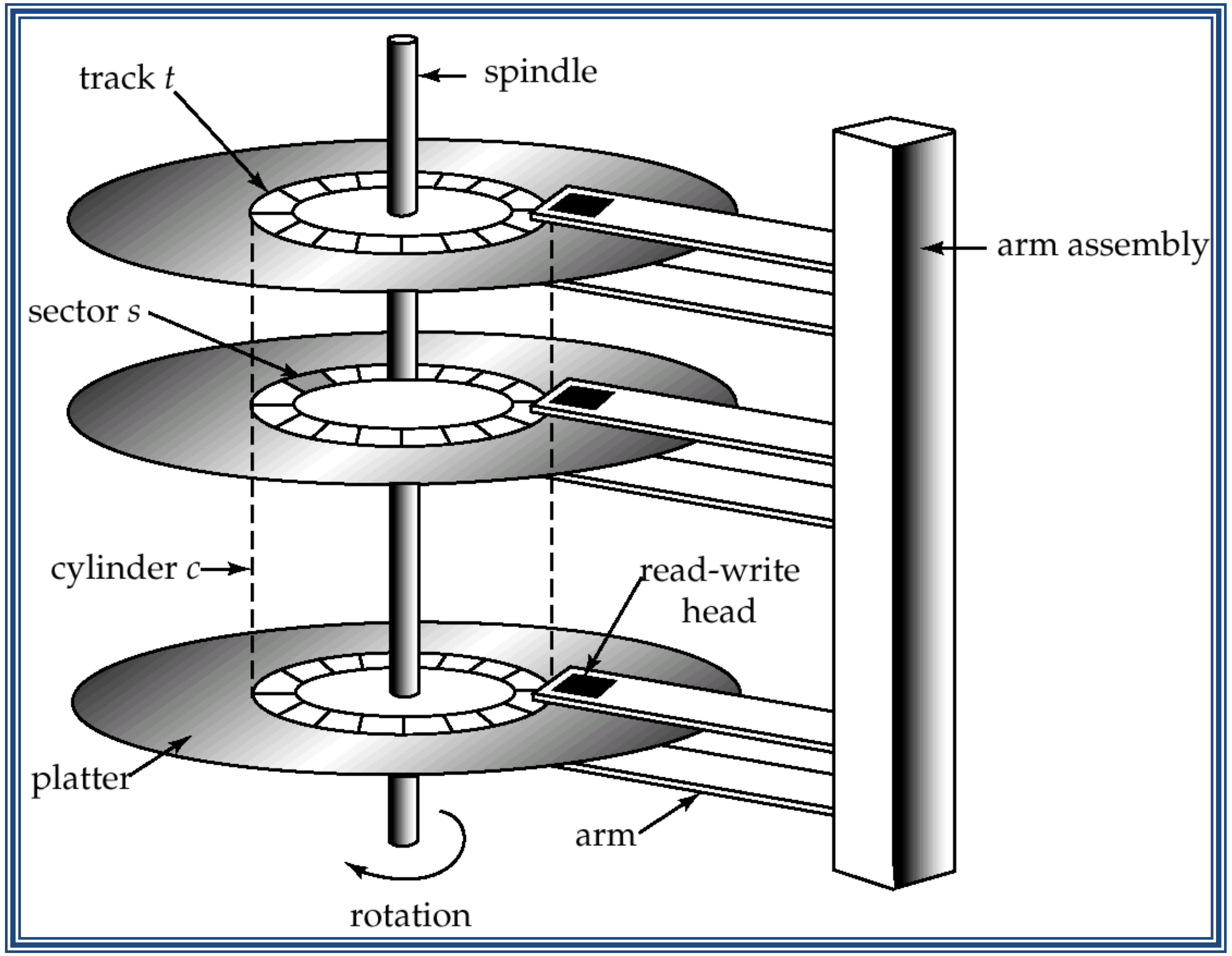
Magnetic disk organization

- Multiple platters
  - Each platter has *two* surfaces for data storage
  - Platters spin at the *same* rate (e.g., 7200 rpm)
  - A ring on a surface is called a track
    - A track is divided into many sectors of fixed size (512 B)
    - A sector is the *smallest* unit of I/O
- A single arm assembly with multiple disk heads
  - Can only move inward/outward *together*
  - The vertical stack of tracks is called a cylinder
    - Disk heads can be over the tracks of the *same* cylinder at the *same* time
  - Usually one read/writes at the same time
- Address of a sector: cylinder - head - sector
  - (0, 0, 0) : first sector; (0, 0, 1): second sector, …
    (0, 1, 0) : the S^{th} sector, (1, 0, 0) the $(SH)^{th}$
    where S is the max # of sectors/track and H is the # of heads
  - Reality: today's disks use logical block addressing (linear block #)
    - Translated to the actual geometry by disk controller
    - Nevertheless, this is still a good model for understanding HDD performance.



Magnetic disk I/O latency

- File systems perform I/O in units of multiple sector (page)
  - 4KB~16KB are most common
- Break-down of I/O latency of a page
  - Seek time: moving arms to the cylinder
    - 2 ~ 20 ms per seek
    - 4 ~ 10 ms on average
  - Rotation delay:
    wait for the sector to be under a head
    - Depending on rotation speed (5400 rpm - 15000 rpm)
    - E.g, 7200 rpm = 120 rotations/second
      => 1/120 = 8.33 ms / rotation
      on average it needs a half rotation
      => 8.33 / 2 = 4.17 ms on average
  - Transfer time: time for reading/writing data
    - Data transfer rate: 50 - 200 MB/s
    - ⇔ 0.02 ~ 0.08 ms for 4KB pages
- Average access time
  - 4KB page, 7200 rpm: roughly 8 ~ 15 ms



Impact of I/O pattern on magnetic disk

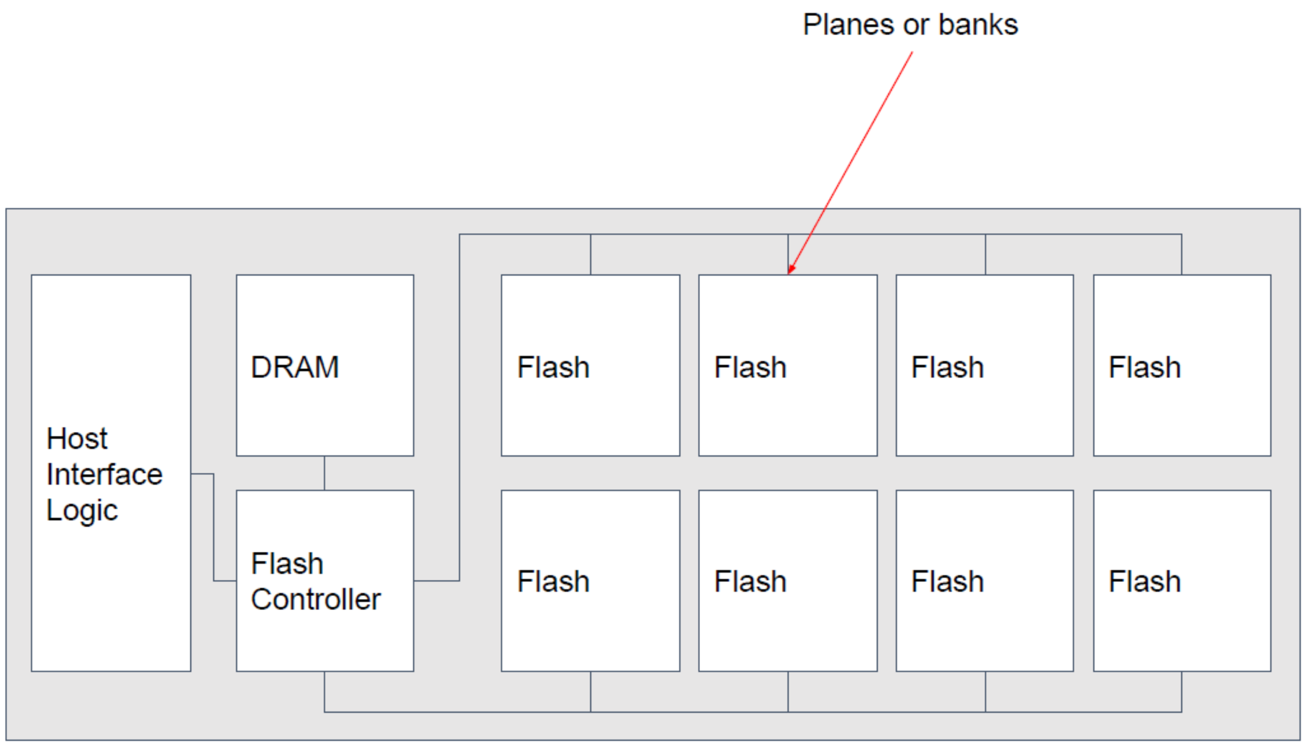
- I/O pattern has a huge impact on I/O performance
  - E.g., 4KB page size
    - Sequential read/write: usually 100 ~ 200+ MB/s
    - Random read/write: 50 ~ 200 IOPS ⇔ 200 KB ~ 800 KB /s

    - > 2 orders of magnitude difference in terms of data transfer rate

  - Rule of thumb:
    - Random I/O: very slow; avoid reading a lot of data from random location
    - Sequential I/O: better for accessing a lot of data

Flash memory / solid state drive

- NAND Flash is the most common storage media for solid state drives
- No mechanical parts (magnetic disk can have head crash => data corruption/loss)
  - More reliable; less likely to fail due to physical shocks
- Faster than magnetic disk



Flash memory / solid state drive

- NAND SSD has asymmetric read/write performance
  - 4KB page, typical SSD internal performance numbers
    - Read latency: 20 to 100 μs ; throughput: > 500 MB/s
    - Write latency: 200 μs; throughput: > 500 MB/s
    - Erase latency: ~2 ms

  - Three ops: read/write/erase
    - Read/write works on pages (usually 4KB)
      - Write can only change some bits from 1 to 0 (not the other way around!)
      - Muse erase before write a page.
    - Erase works on blocks (e.g., 256 KB)
      - Resets all bits in a block to 1
      - Flash translation layer: indirection of page numbers to physical pages
        - Solves two problems: slow erase and flash wear
  - Actual performance also often bound by peripheral bus's bandwidth and IOPS

Flash memory / solid state drive

- NAND SSD has asymmetric read/write performance
  - The performance from DB stand of view?
    - No single answer depending on how you use it
      - I/O queue depth, I/O api, access pattern, page size, peripheral bus type and etc.

    - In a typical case:
      - Sequential I/O is still preferred, although random I/O isn't as bad as in HDD
      - SSDs have much better random I/O performance than magnetic disk
        - 10k - 1M IOPS
      - and higher bandwidth as well
        - up to 7GB/s on PCIe 4.0, ~500MB/s on SATA

File System Interface

- POSIX I/O interface
  - A standard synchronous I/O interface
  - Agnostic to the underlying storage device/file system

A *file descriptor* is a reference to an *open file description*, an entry in the system-wide table of open files that records file offsets and file status flags.

open(2): open and possibly create a file -> *file descriptor* (int)

```
int fd = open("/data/a.dat", O_RDONLY | O_CREAT, 0644);
```

opens the file at path `/data/a.dat`

1. read-only access
2. create the file if it does not exist

The permission bits if the file is created. 0644 = rw allowed for user (file owner); read only for group & others.

Case 1: fd >= 0 on success.
Case 2: fd == -1 if an error occurred -- check `errno` for reasons; also see `strerror(3)`

File System Interface

- POSIX I/O interface
  - A standard synchronous I/O interface
  - Agnostic to the underlying storage device/file system

A *file descriptor* is a reference to an *open file description*, an entry in the system-wide table of open files that records file offsets and file status flags.

open(2): open and possibly create a file -> *file descriptor* (int)

```
int fd = open("/data/a.dat", O_RDONLY | O_CREAT, 0644);
```

pread(2), pwrite(2): read from or write to a file descriptor at a given offset

```
char buf[4096];
ssize_t sz = pread(fd, buf, 4096, 1048576);
if (sz == 4096) /* success */; else /* error */;
```

reading 4096 bytes at file offset 1048576 = 4096 * 256 (i.e., reading page 255 from a file assuming 4KB pages)

File System Interface

- POSIX I/O interface
  - A standard synchronous I/O interface
  - Agnostic to the underlying storage device/file system

A *file descriptor* is a reference to an *open file description*, an entry in the system-wide table of open files that records file offsets and file status flags.

open(2): open and possibly create a file -> *file descriptor* (int)

```
int fd = open("/data/a.dat", O_RDONLY | O_CREAT, 0644);
```

pread(2), pwrite(2): read from or write to a file descriptor at a given offset

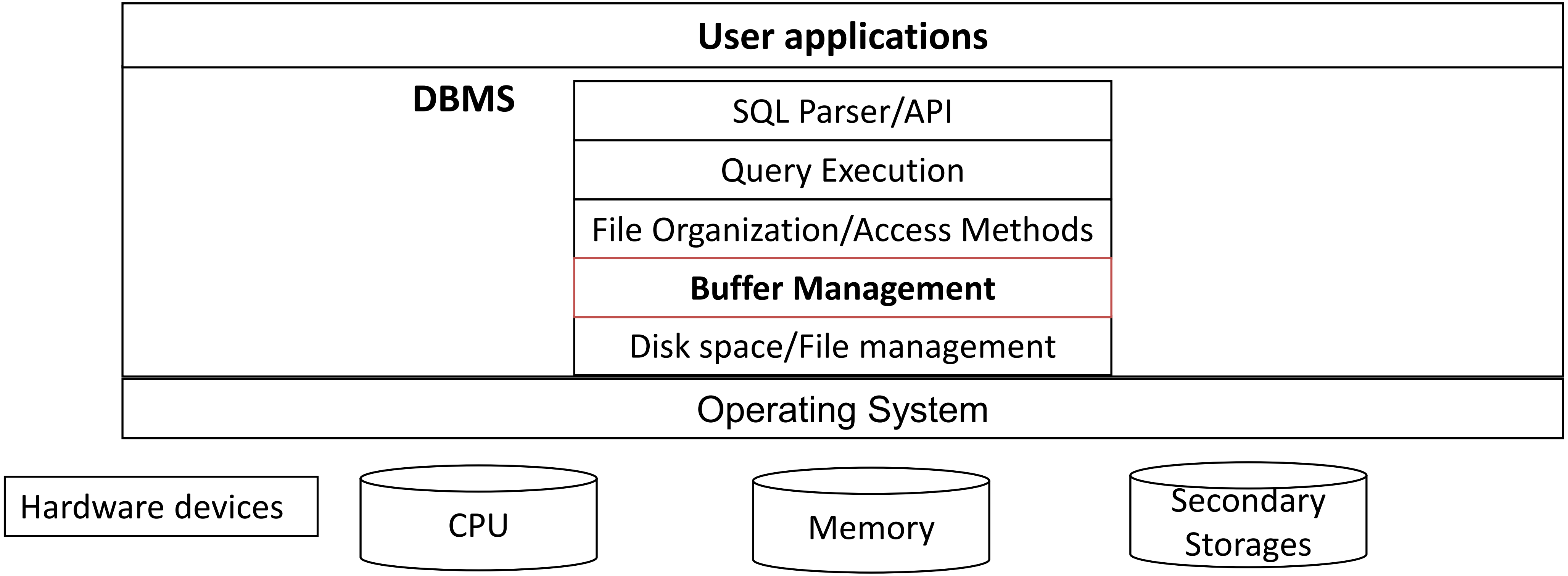
posix_fallocate(3), fallocate(2)

fsync(2), fdatasync(2),

close(2)

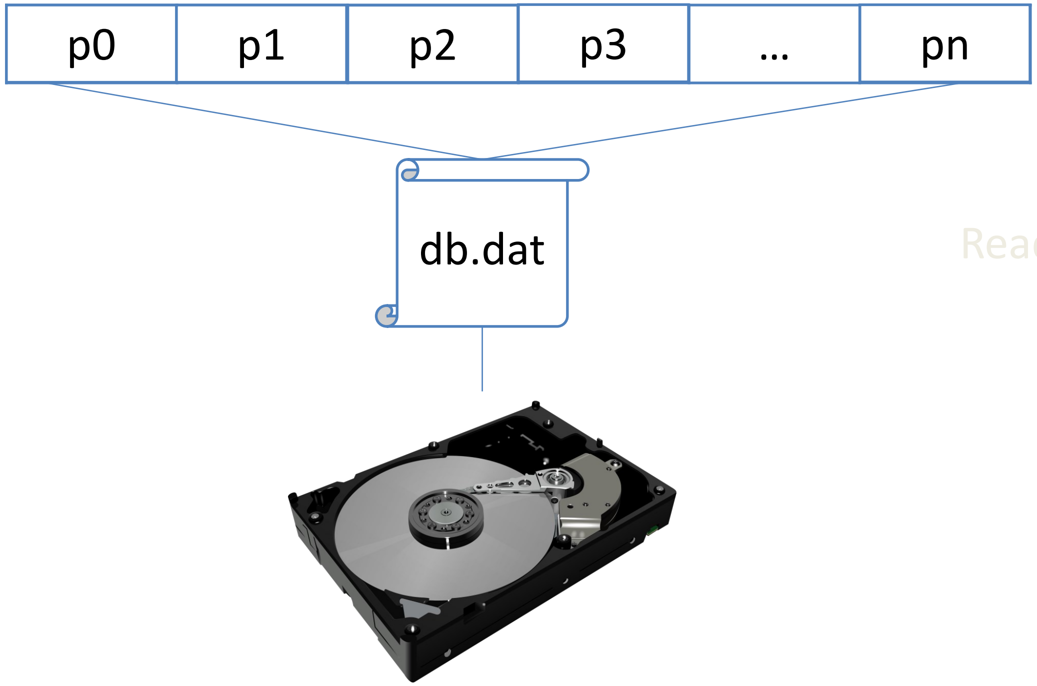
Check man pages for more details.

Big Picture


User applications
DBMS
SQL Parser/API
Query Execution
File Organization/Access Methods
Buffer Management
Disk space/File management
Operating System
Hardware devices
CPU
Memory
Secondary
Storages

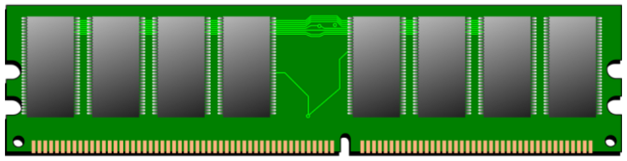
How does database access data pages?

- Data pages are stored in disk file
  - suppose we want to count how many rows there are in a database table
    - need to scan all pages
  - page must be loaded into memory before any computation happens



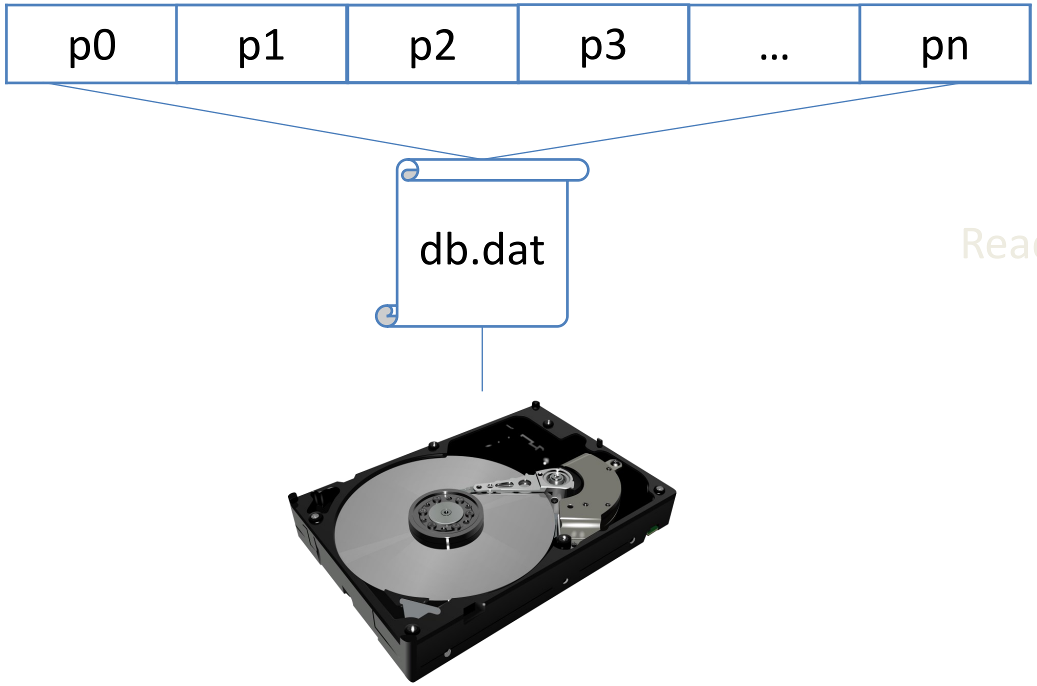
Read: ~10 ms

Computation: < 1 μs



How does database access data pages?

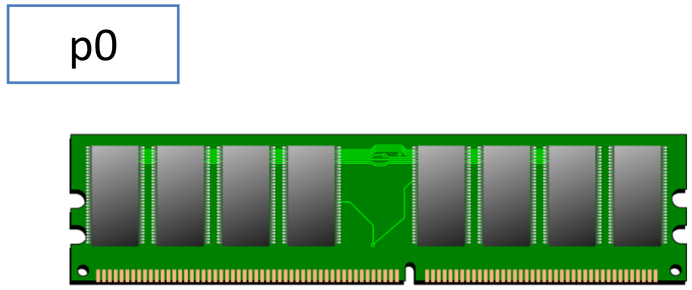
- Data pages are stored in disk file
  - suppose we want to count how many rows there are in a database table
    - need to scan all pages
  - page must be loaded into memory before any computation happens



- Repeat for all the n pages
- Execution time dominated by I/O

Read: ~10 ms

Computation: < 1 μs



How does database access data pages?

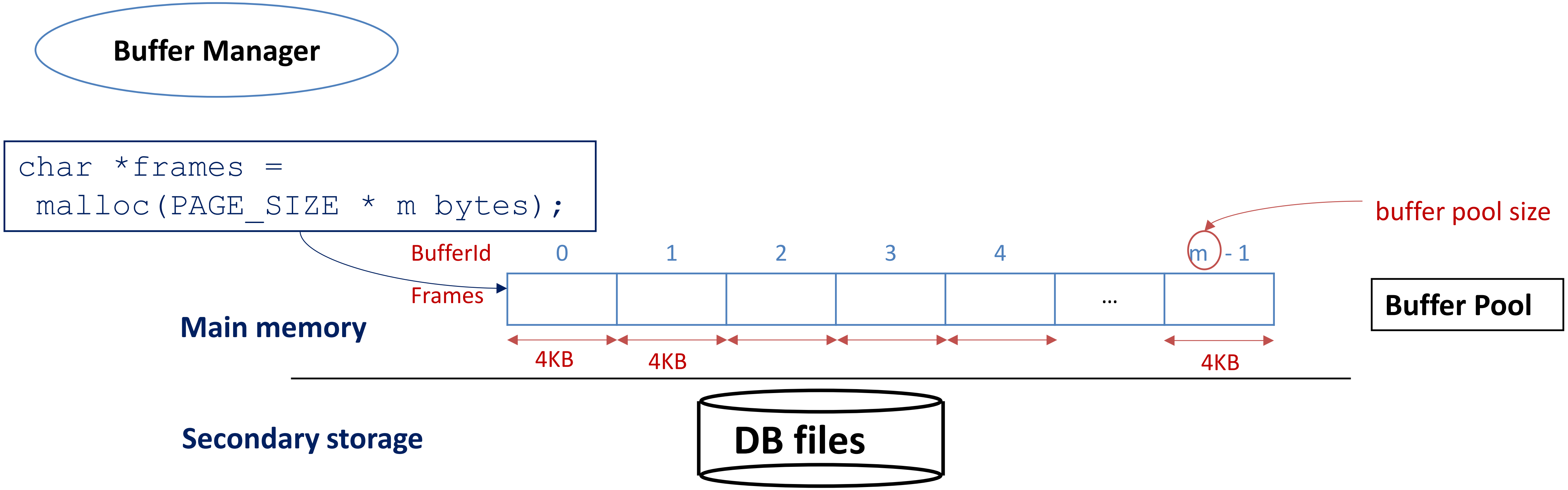
- Data pages are stored in disk file
  - suppose we want to count how many rows there are in a database table
    - need to scan all pages
  - page must be loaded into memory before any computation happens

- What if we want to scan the data file for multiple passes?
  - Option 1: read/write the entire page on demand before reading/writing the integer <- very slow
  - Option 2: read all data pages into memory at the beginning <- not scalable
    - May not fit in memory
    - What to do on modify?
      - Immediately write back? Or Flush when program shutsdown?
      - Data persistence?

- Solution: buffer pool

Buffer management in DBMS

- Buffer manager manages a fixed-size pool of in-memory page frames which
  - are of the same size as the data pages (e.g., 4KB)



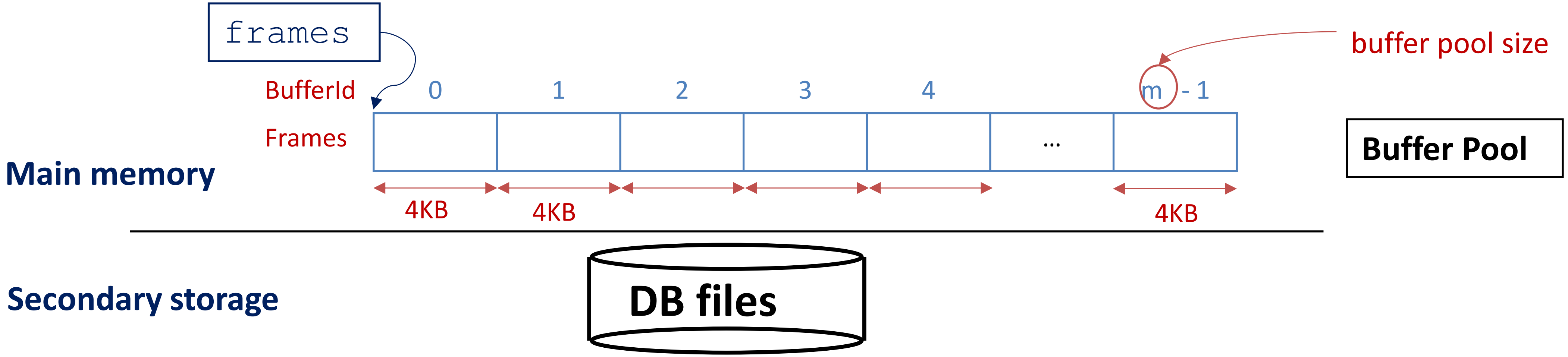
Handling a page request (buffer miss)

- Handling page request
  - Suppose we want to read/write a page in the file with page number = 100

Upper level components

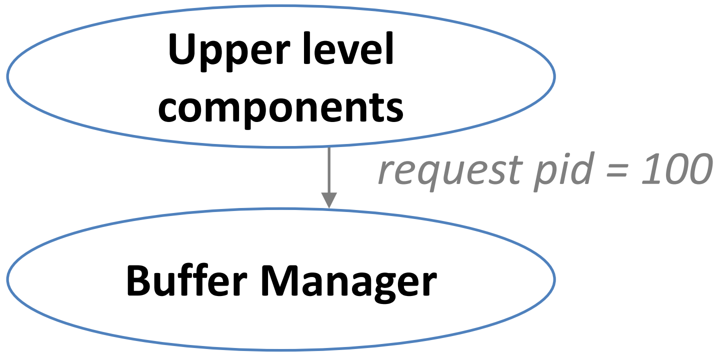
Buffer Manager

```
HandlePageRequest(pid):
 if pid exists in some buffer frame i:
   return &frames[i]
 else:
   find a free frame i
   ReadPage(pid, &frames[i])
   return &frames[i]
```

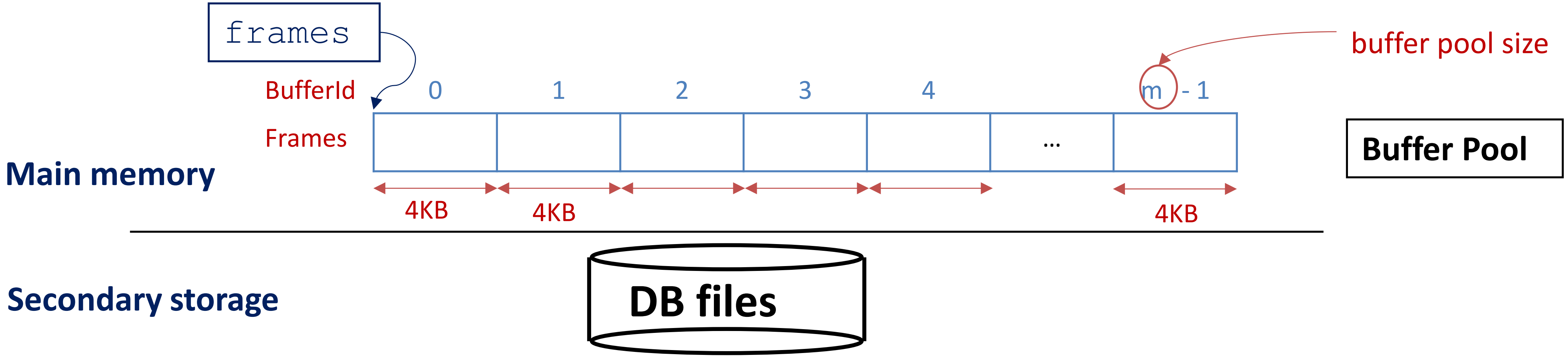


Handling a page request (buffer miss)

- Handling page request
  - Suppose we want to read/write a page in the file with page number = 100

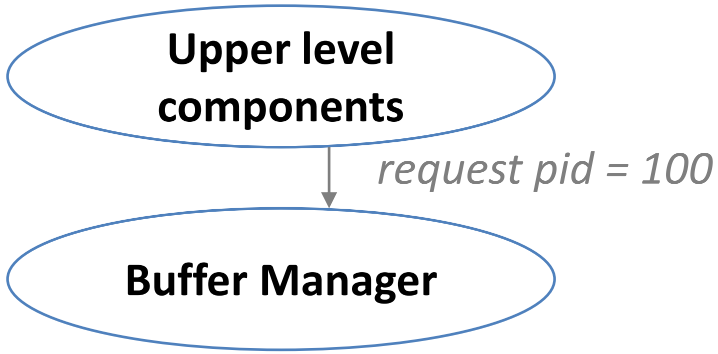



```
HandlePageRequest(pid):   // pid = 100
 if pid exists in some buffer frame i:
   return &frames[i]
 else:
   find a free frame i
   ReadPage(pid, &frames[i])
   return &frames[i]
```

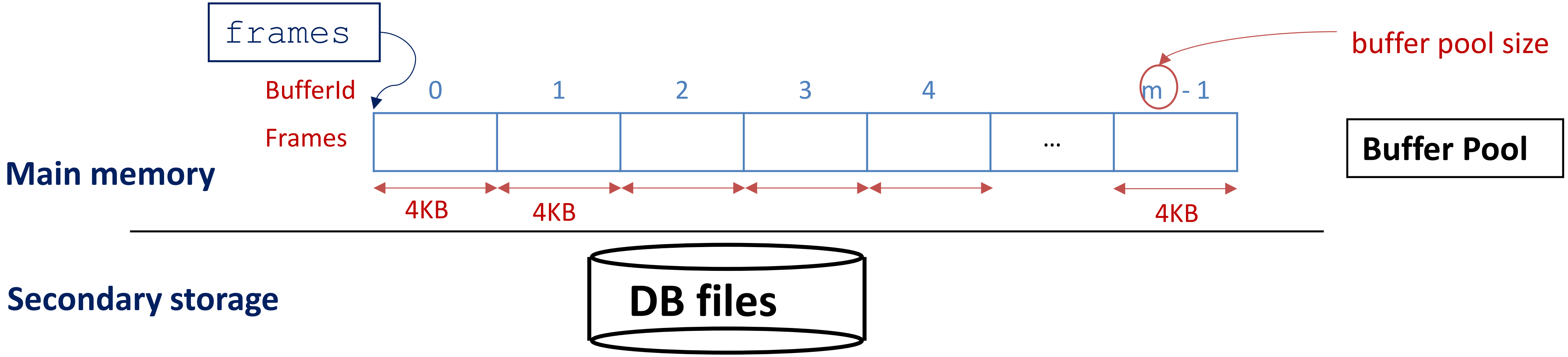


Handling a page request (buffer miss)

- Handling page request
  - Suppose we want to read/write a page in the file with page number = 100

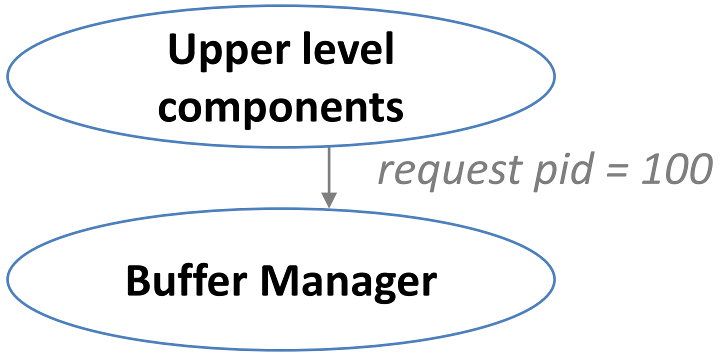



```
HandlePageRequest(pid):   // pid = 100
 if pid exists in some buffer frame i:
   return &frames[i]
 else:
   find a free frame i
   ReadPage(pid, &frames[i])
   return &frames[i]
```

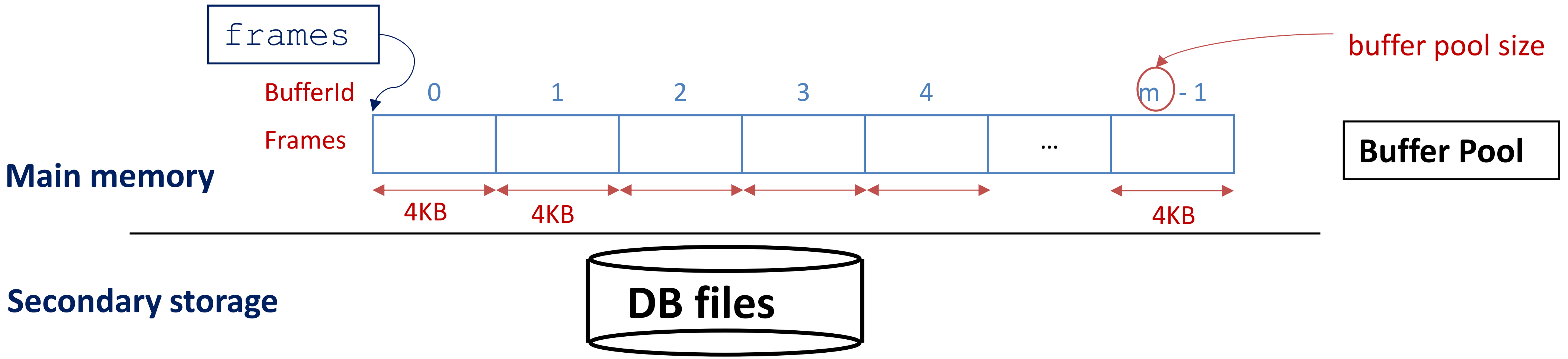


Handling a page request (buffer miss)

- Handling page request
  - Suppose we want to read/write a page in the file with page number = 100

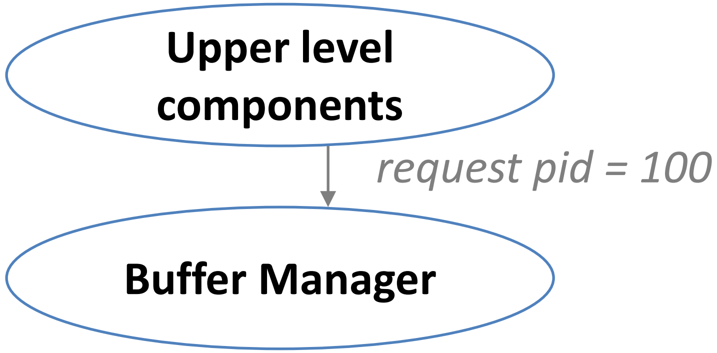



```
HandlePageRequest(pid):   // pid = 100
 if pid exists in some buffer frame i:
   return &frames[i]
 else:
   find a free frame i     // i = 0
   ReadPage(pid, &frames[i])
   return &frames[i]
```

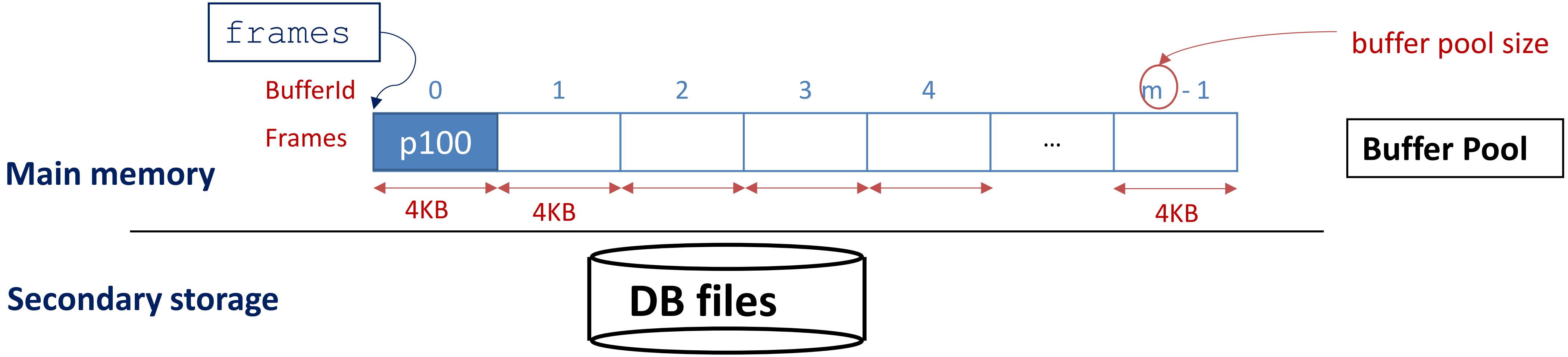


Handling a page request (buffer miss)

- Handling page request
  - Suppose we want to read/write a page in the file with page number = 100

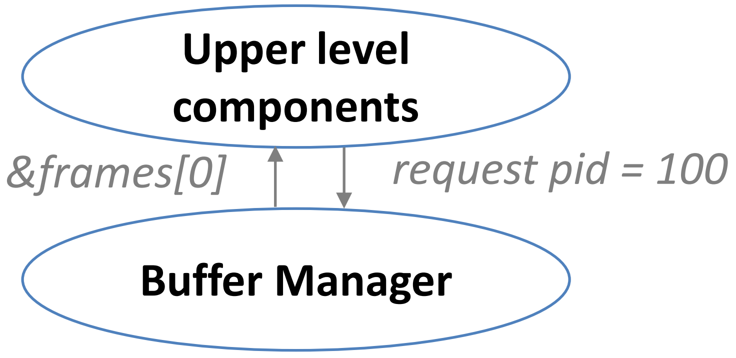



```
HandlePageRequest(pid):   // pid = 100
 if pid exists in some buffer frame i:
   return &frames[i]
 else:
   find a free frame i     // i = 0
   ReadPage(pid, &frames[i])
   return &frames[i]
```

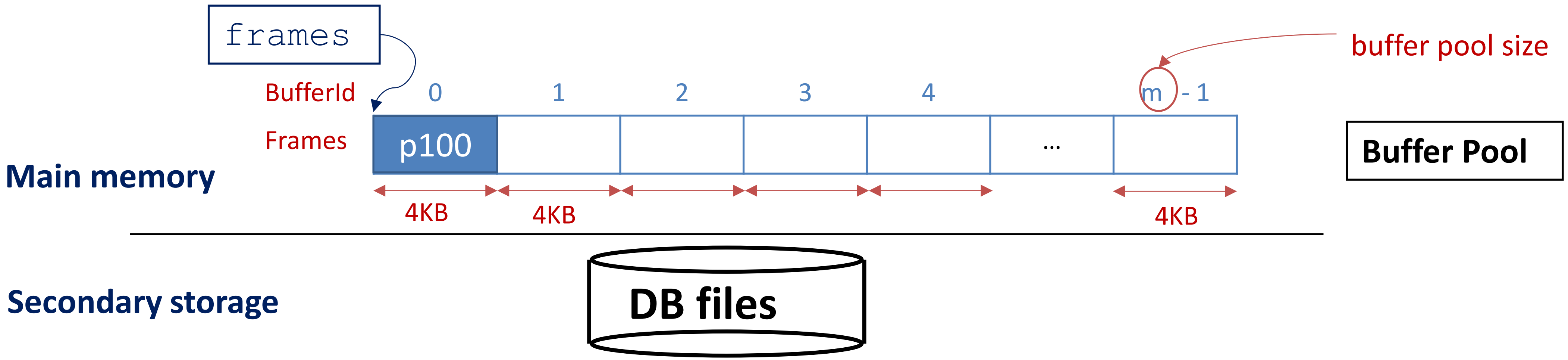


Handling a page request (buffer miss)

- Handling page request
  - Suppose we want to read/write a page in the file with page number = 100 *Cost: 1 I/O*

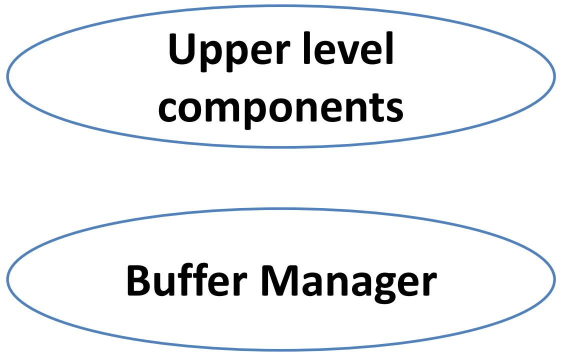



```
HandlePageRequest(pid):   // pid = 100
 if pid exists in some buffer frame i:
   return &frames[i]
 else:
   find a free frame i     // i = 0
   ReadPage(pid, &frames[i])
   return &frames[i]
```

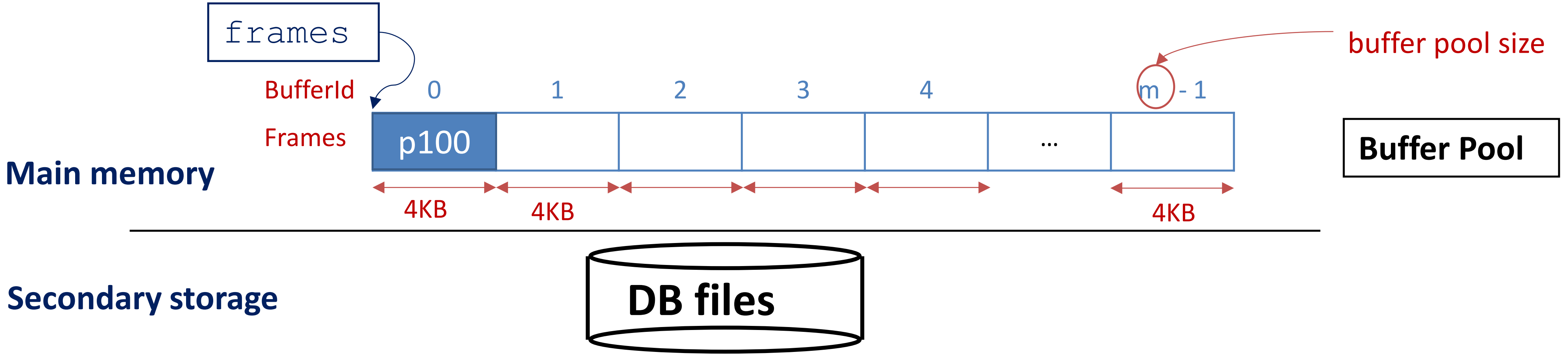


Handling a page request (buffer hit)

- Handling page request
  - Suppose we want to read the same page again (pid = 100)

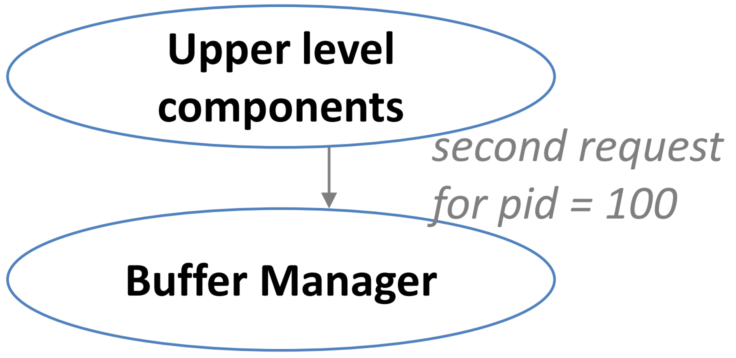



```
HandlePageRequest(pid):
 if pid exists in some buffer frame i:
   return &frames[i]
 else:
   find a free frame i
   ReadPage(pid, &frames[i])
   return &frames[i]
```

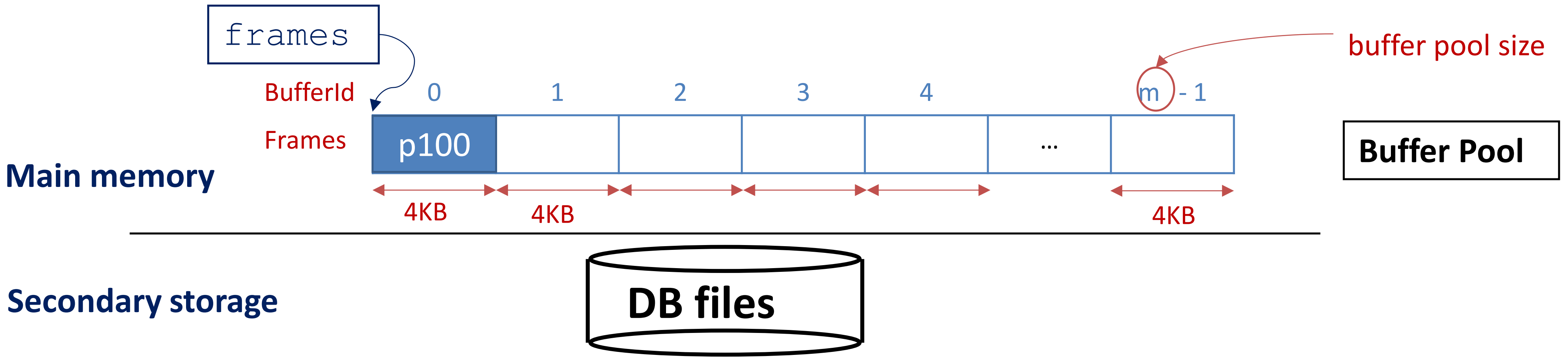


Handling a page request (buffer hit)

- Handling page request
  - Suppose we want to read the same page again (pid = 100)



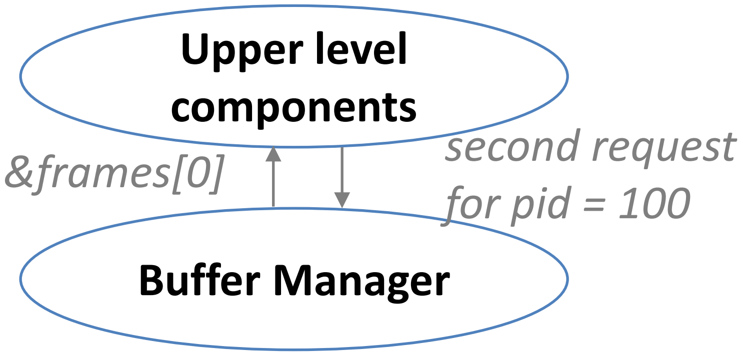

```
HandlePageRequest(pid):   // pid = 100
 if pid exists in some buffer frame i:
   return &frames[i]
 else:
   find a free frame i
   ReadPage(pid, &frames[i])
   return &frames[i]
```



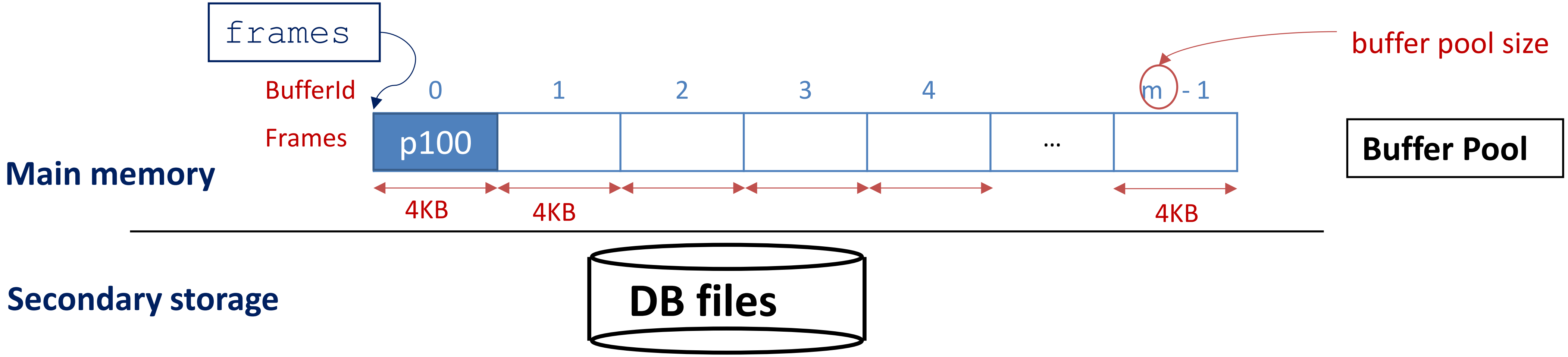
Handling a page request (buffer hit)

- Handling page request
  - Suppose we want to read the same page again (pid = 100)

Cost: 0 I/O




```
HandlePageRequest(pid):   // pid = 100
 if pid exists in some buffer frame i:
   return &frames[i]   // i = 0
 else:
   find a free frame i
   ReadPage(pid, &frames[i])
   return &frames[i]
```

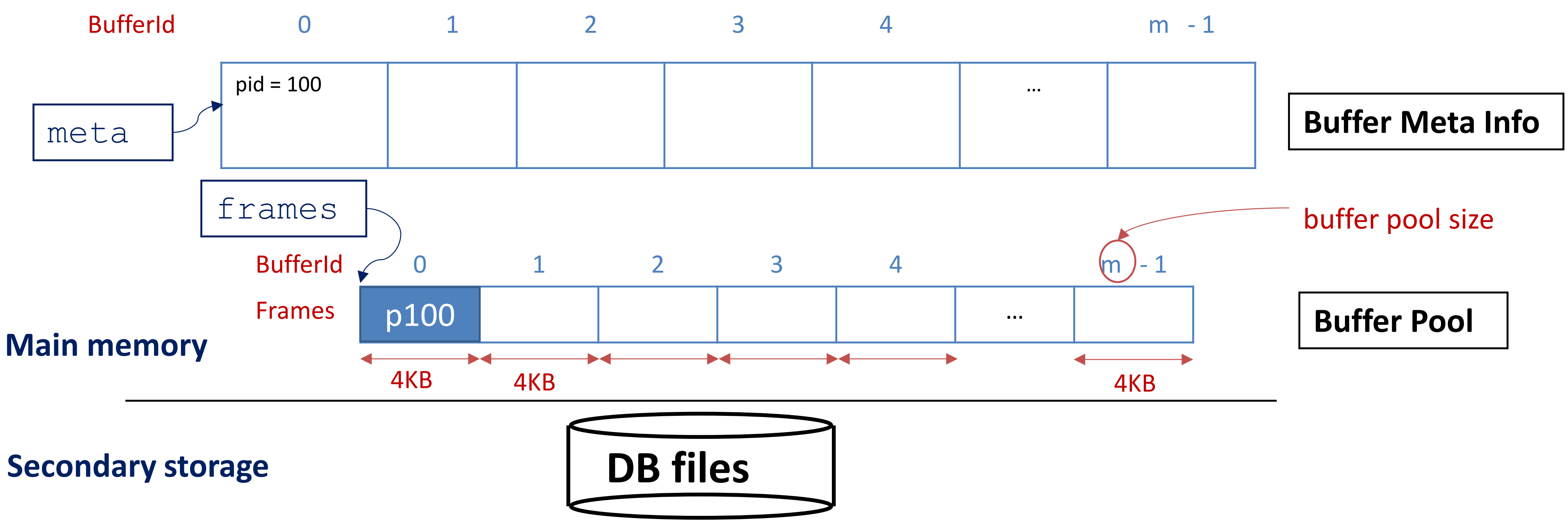


Map page numbers to buffer frames

- How to implement line 2?

```
2  if pid exists in some buffer frame i:
```

  - Need to store the page numbers, but where?
    - For each buffer frame, we maintain a metadata structure which includes **pid**.



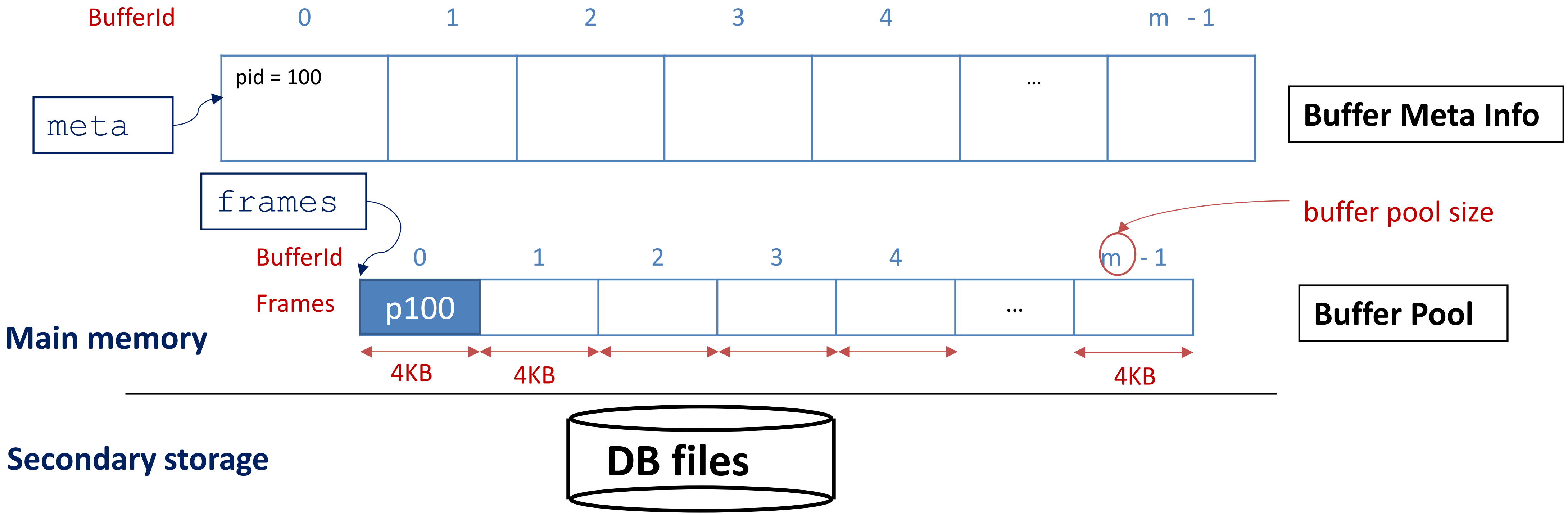
Map page numbers to buffer frames

- How to implement line 2?

```
2  if pid exists in some buffer frame i:
```

```
for (BufferId i = 0; i < m; ++i) {
  if (meta[i].pid == 100)
    return i;
}
return InvalidBufferId;
```

O(m) time -- slow!



Map page numbers to buffer frames

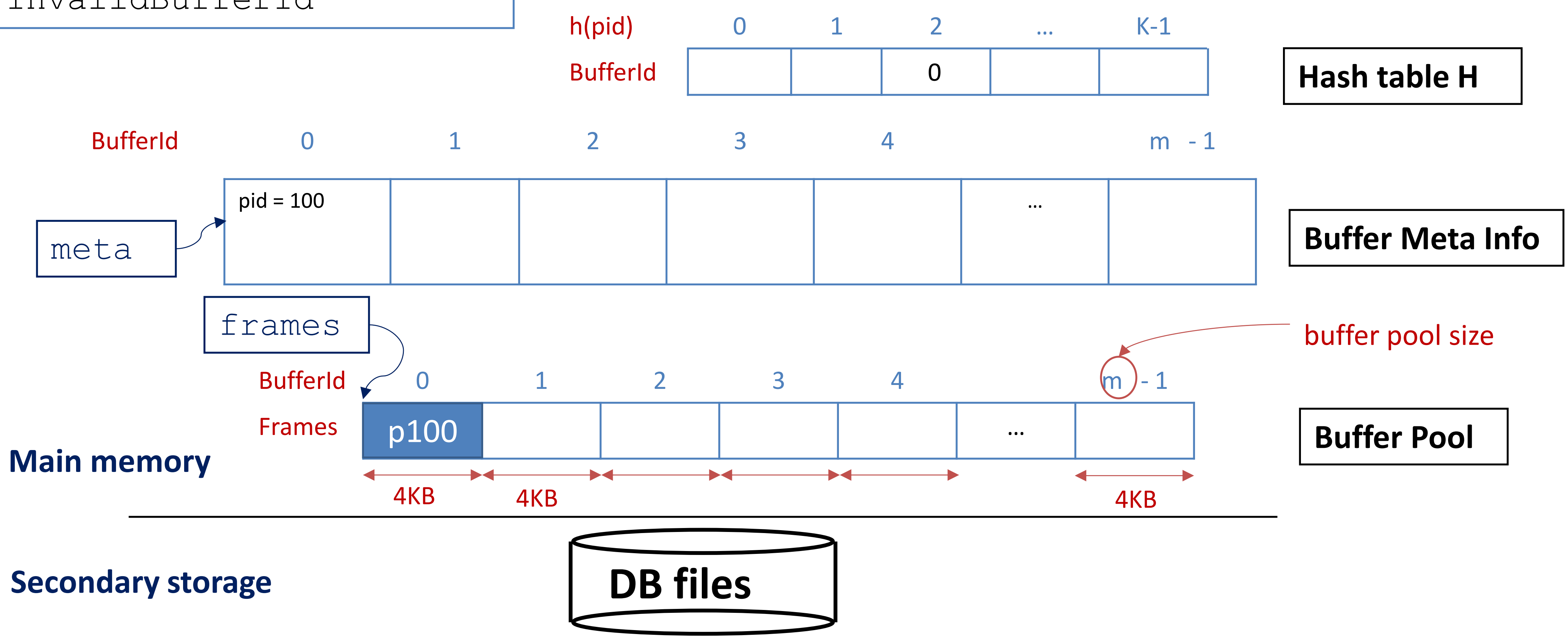
- How to implement line 2?

```
if (H.find(100) != H.end())
  return H[100];
return InvalidBufferId
```

```
2  if pid exists in some buffer frame i:
```

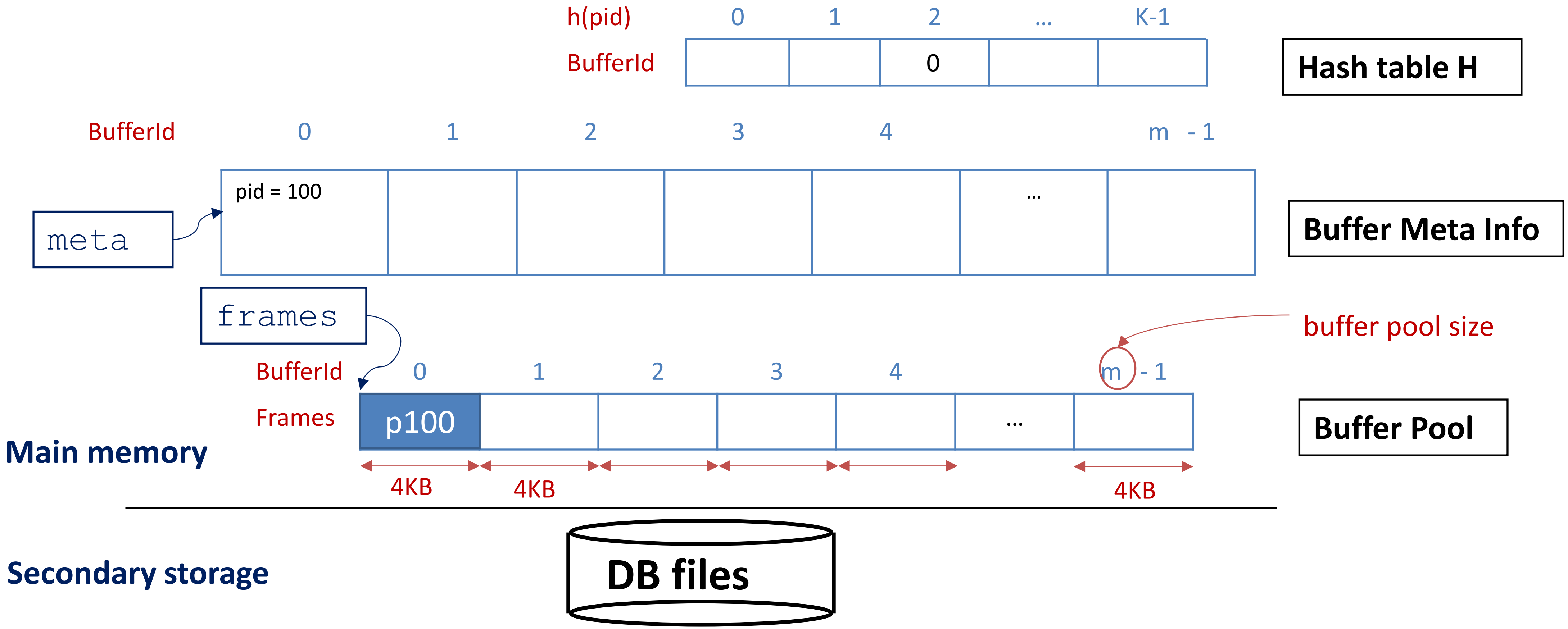
O(1) time in expectation

suppose h(100) == 2



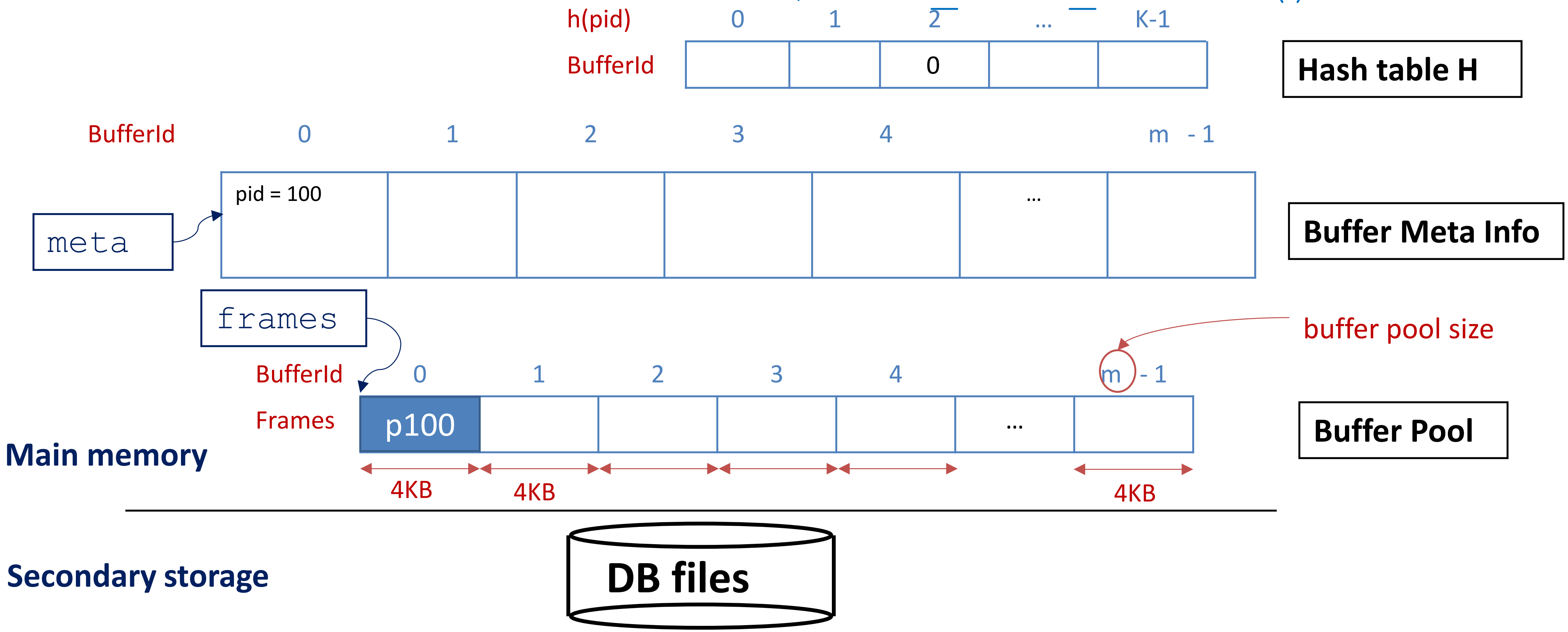
Map page numbers to buffer frames

- Practical consideration for hash tables
  - DBMS usually has its own hash tables implementation for buffer manager -- why?
    - memory constraints, efficiency, concurrency control, …



Map page numbers to buffer frames

- Practical consideration for hash tables
  - For Project 2: feel free to use libraries (e.g., `absl::flat_hash_map`)
    - Tips for time and memory efficiency: avoid rehashing
      - Set the initial bucket count `K >= m / max_load_factor()`



Buffer eviction

- What if we run out of buffer frames?
  - e.g., we are scanning a table with N = 100 pages, but buffer pool size m = 10

Buffer eviction

- What if we run out of buffer frames?
  - Buffer eviction: choose a victim to remove from the buffer pool
    - Several possible policies (more on this later)

Buffer eviction

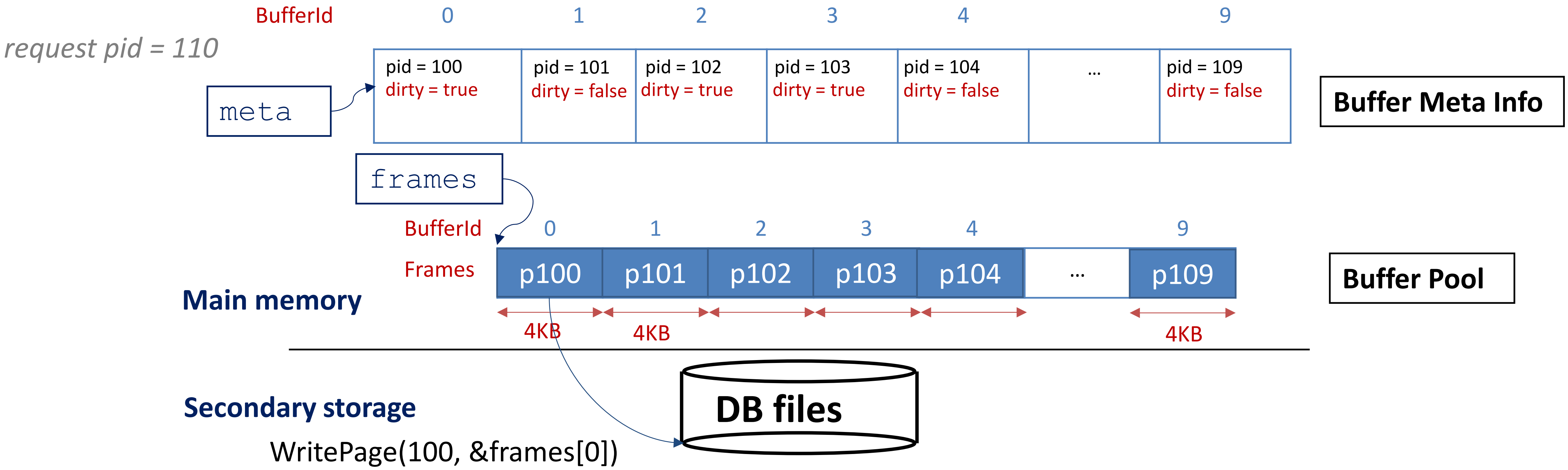
- What if we run out of buffer frames?
  - Buffer eviction: choose a victim to remove from the buffer pool
    - Several possible policies (more on this later)

Buffer eviction

- What if we run out of buffer frames?
  - Buffer eviction: choose a victim to remove from the buffer pool
    - Several possible policies (more on this later)

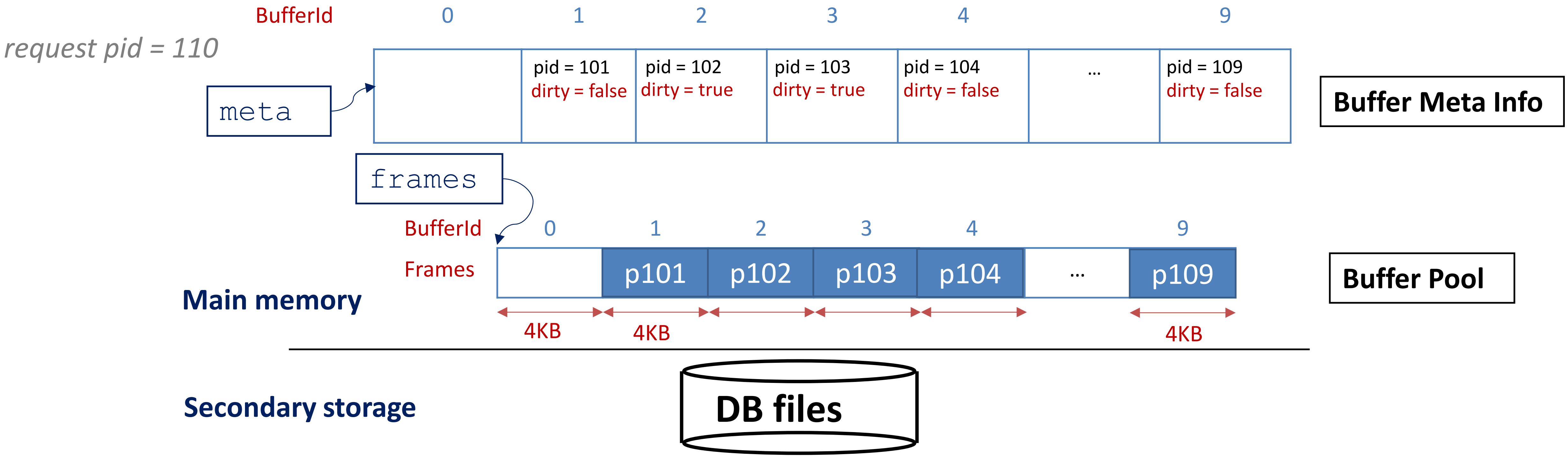
Page requested for writes

- Potential problem with page eviction?
  - What if the evicted page is modified? (e.g., `UPDATE A SET x = x + 10 WHERE …`)
    - We must write modified page back before eviction



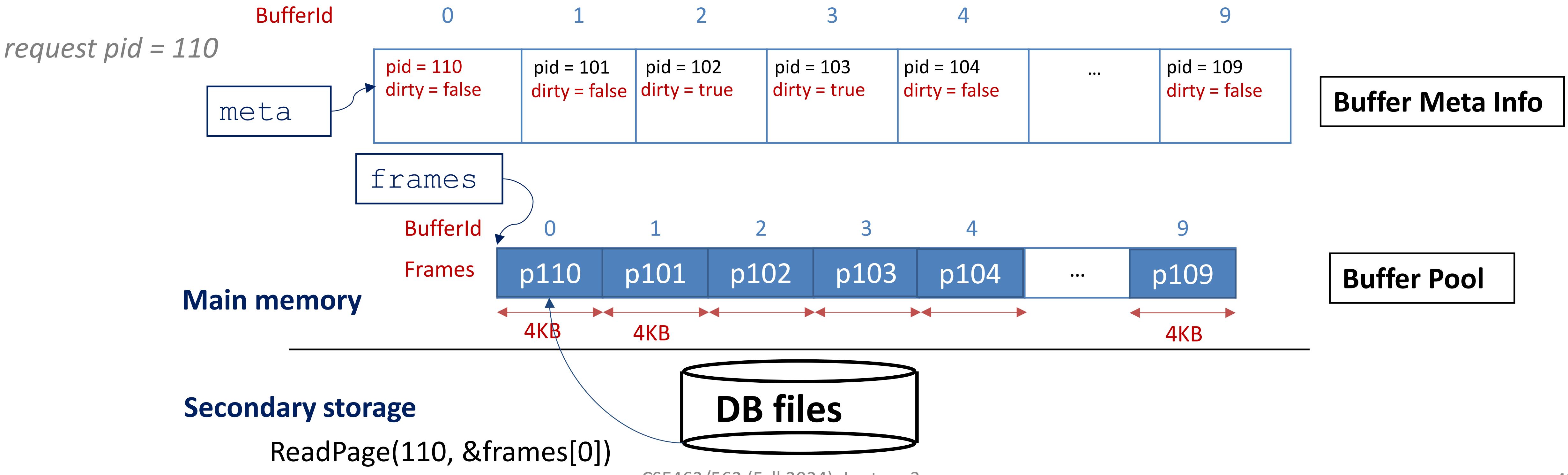
Page requested for writes

- Potential problem with page eviction?
  - What if the evicted page is modified? (e.g., `UPDATE A SET x = x + 10 WHERE …`)
    - We must write modified page back before eviction



Page requested for writes

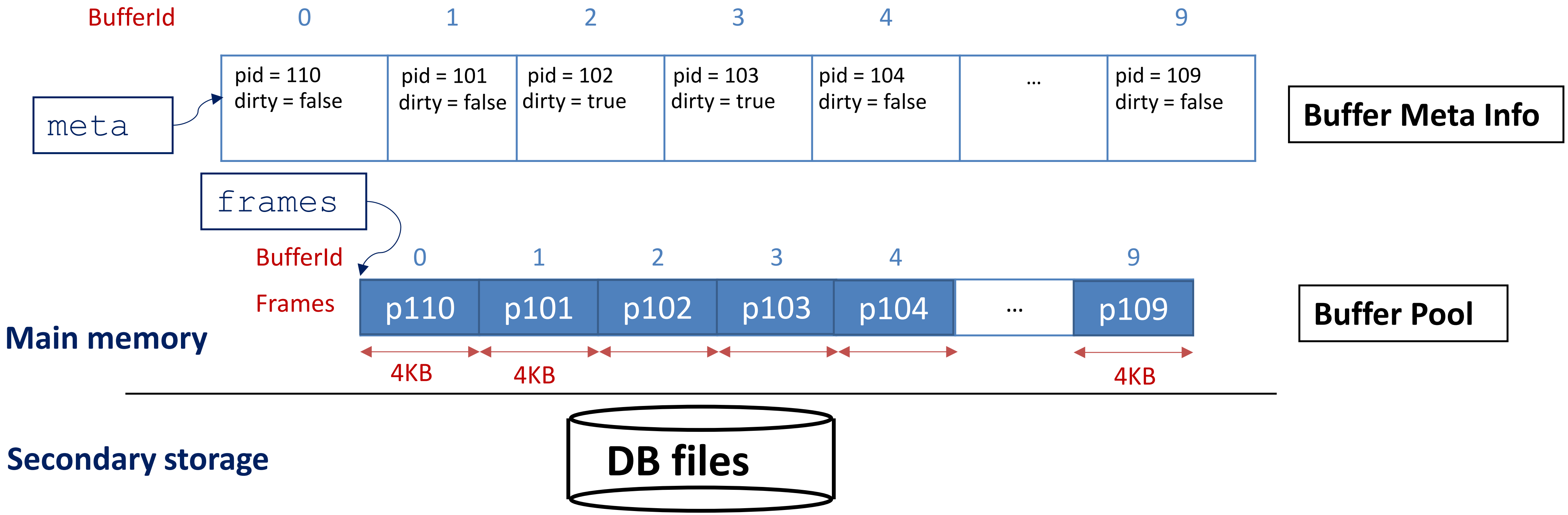
- Potential problem with page eviction?
  - What if the evicted page is modified? (e.g., `UPDATE A SET x = x + 10 WHERE …`)
    - We must write modified page back before eviction



Buffer pins

- Problems with concurrency
  - One thread reading a block while the other tries to evict it

T1: char * frame = BufMgr.HandlePageRequest(110)   // &frames[0]

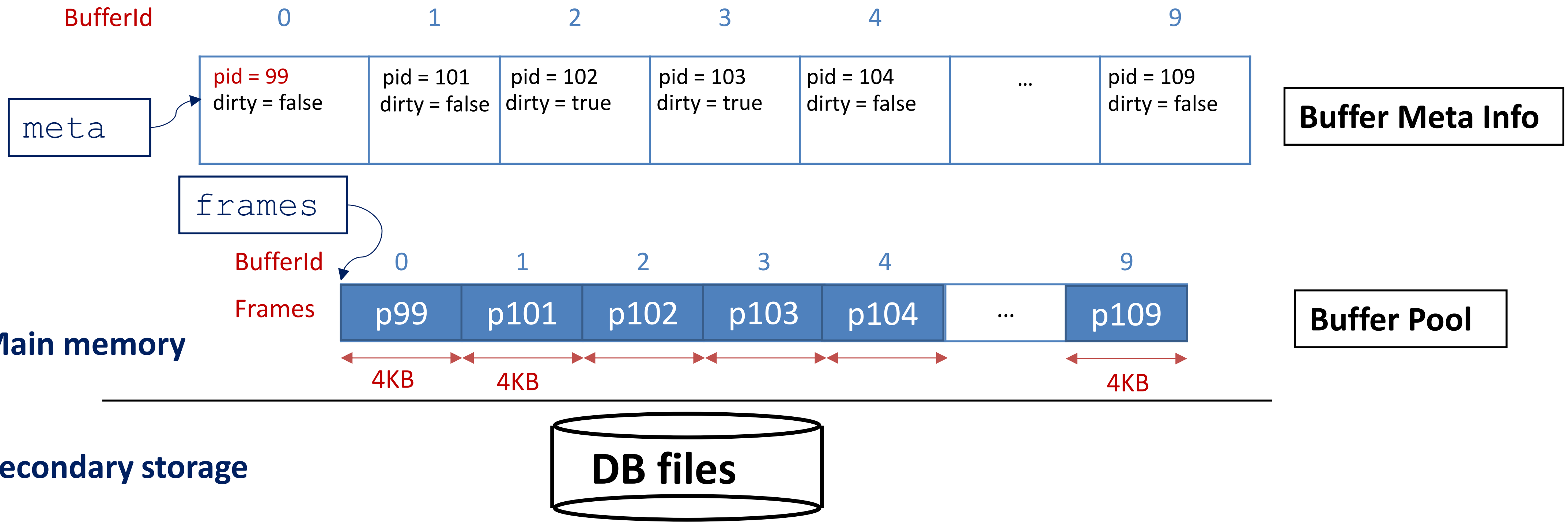


Buffer pins

- Problems with concurrency
  - One thread reading a block while the other tries to evict it

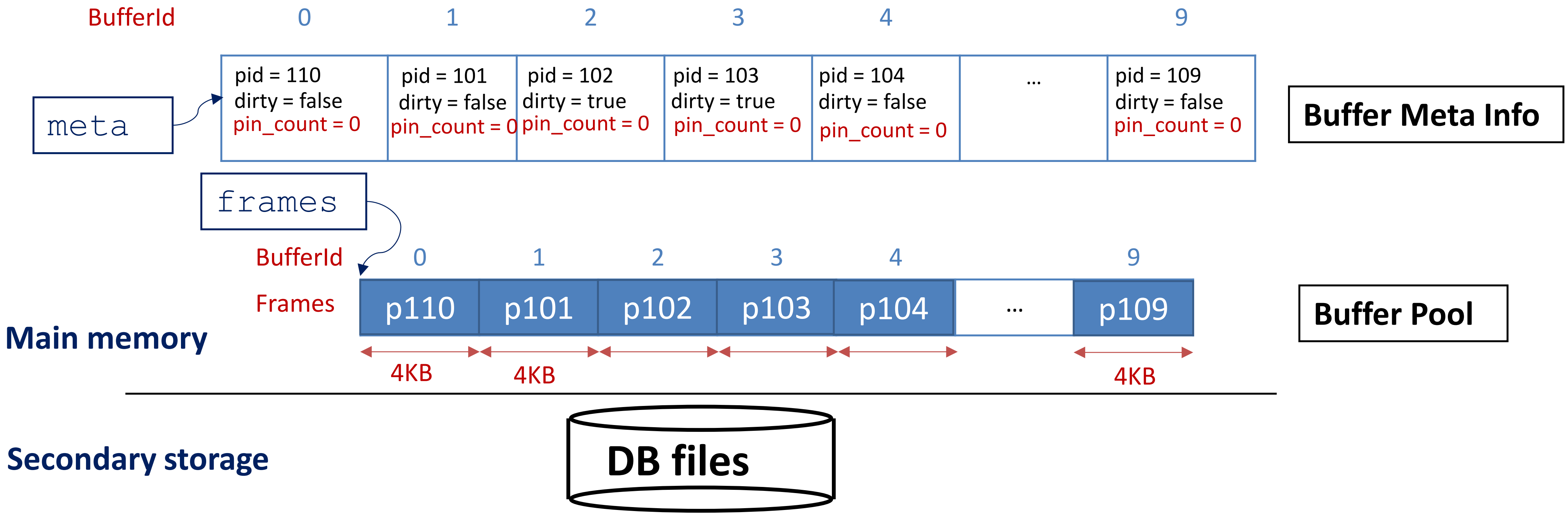
T1: char * f1 = BufMgr.HandlePageRequest(110) // &frames[0] *f1 now contains a wrong page for T1*

T2: char * f2 = BufMgr.HandlePageRequest(99) // &frames[0]



Buffer pins

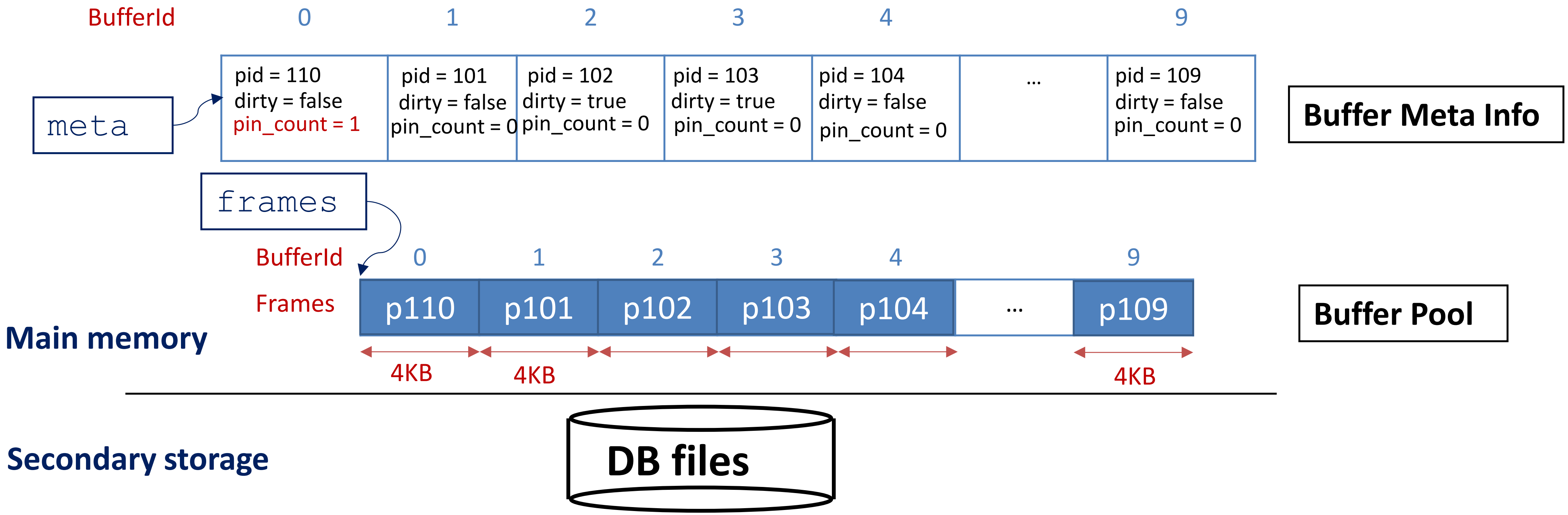
- Solution: introducing a buffer pin count per buffer frame
  - Upon page request, pin count++
  - Upon page release, pin count--
  - Never evict a page with pin count > 0



Buffer pins

- Solution: introducing a buffer pin count per buffer frame
  - Upon page request, pin count++
  - Upon page release, pin count--
  - Never evict a page with pin count > 0

T1: BufferId b1 = BufMgr.PinPage(110, &f1)   // b1 = 0

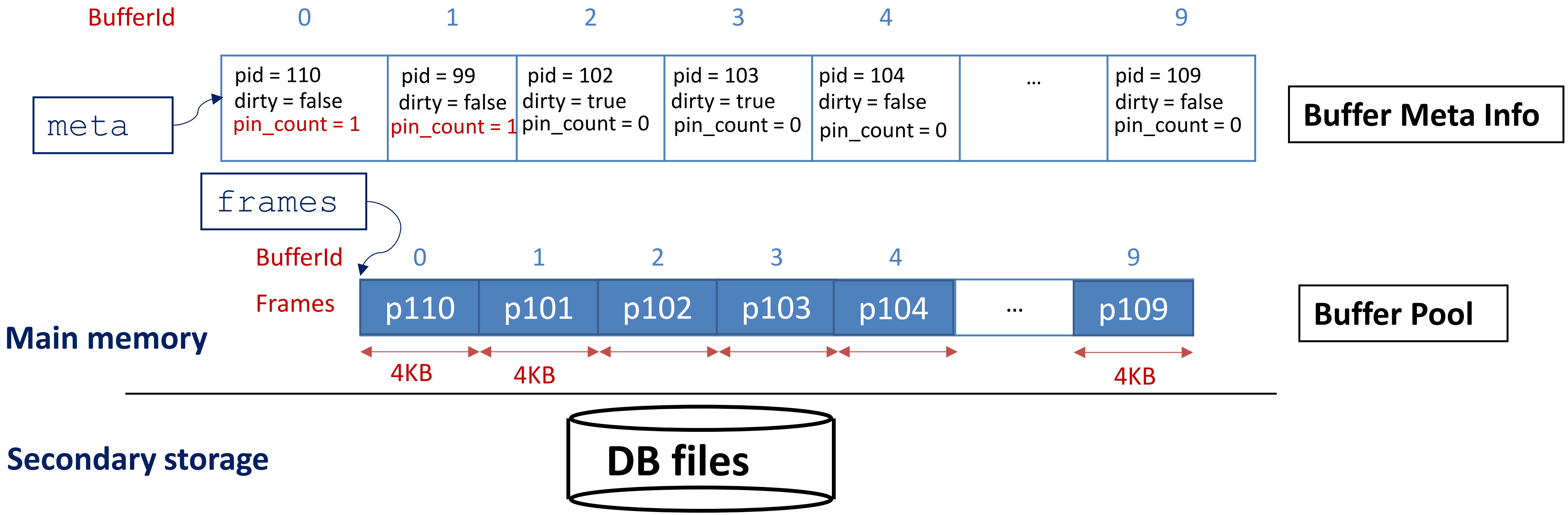


Buffer pins

- Solution: introducing a buffer pin count per buffer frame
  - Upon page request, pin count++
  - Upon page release, pin count--
  - Never evict a page with pin count > 0

T1: BufferId b1 = BufMgr.PinPage(110, &f1) // b1 = 0

T2: BufferId b2 = BufMgr.PinPage(99, &f2) // b2 = 1



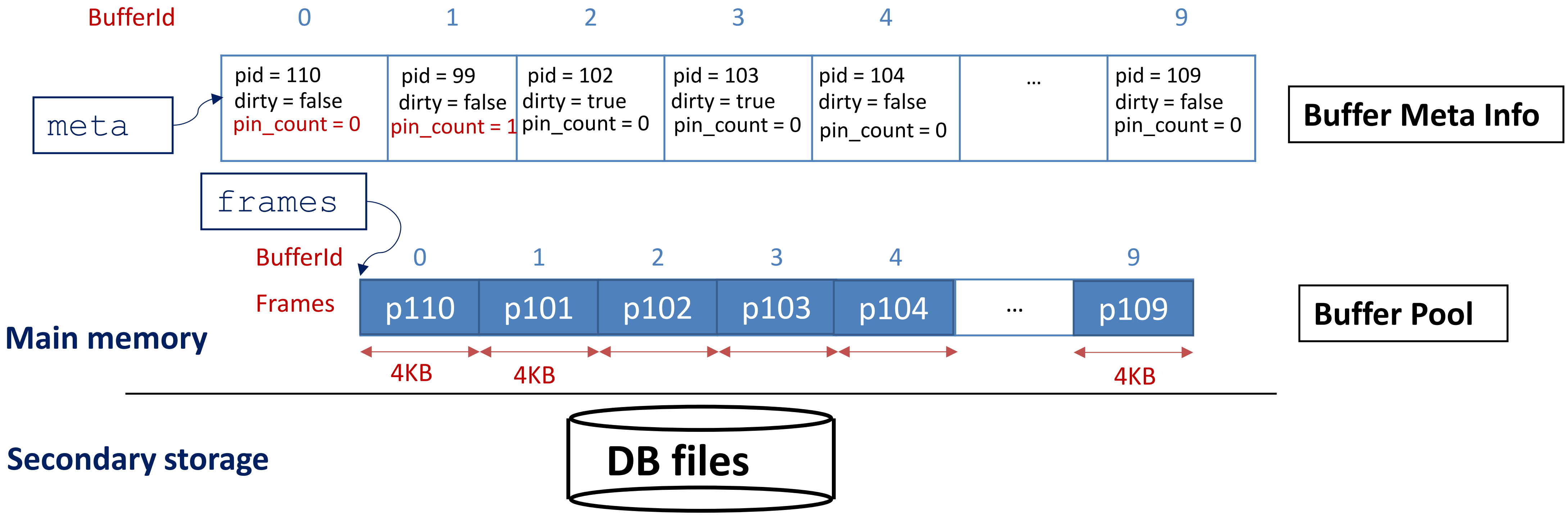
Buffer pins

- Solution: introducing a buffer pin count per buffer frame
  - Upon page request, pin count++
  - Upon page release, pin count--
  - Never evict a page with pin count > 0

T1: BufferId b1 = BufMgr.PinPage(110, &f1) // b1 = 0

T2: BufferId b2 = BufMgr.PinPage(99, &f2) // b2 = 1

T1: BufMgr.UnpinPage(b1)



Buffer pins

- Solution: introducing a buffer pin count per buffer frame
  - Upon page request, pin count++
  - Upon page release, pin count--
  - Never evict a page with pin count > 0

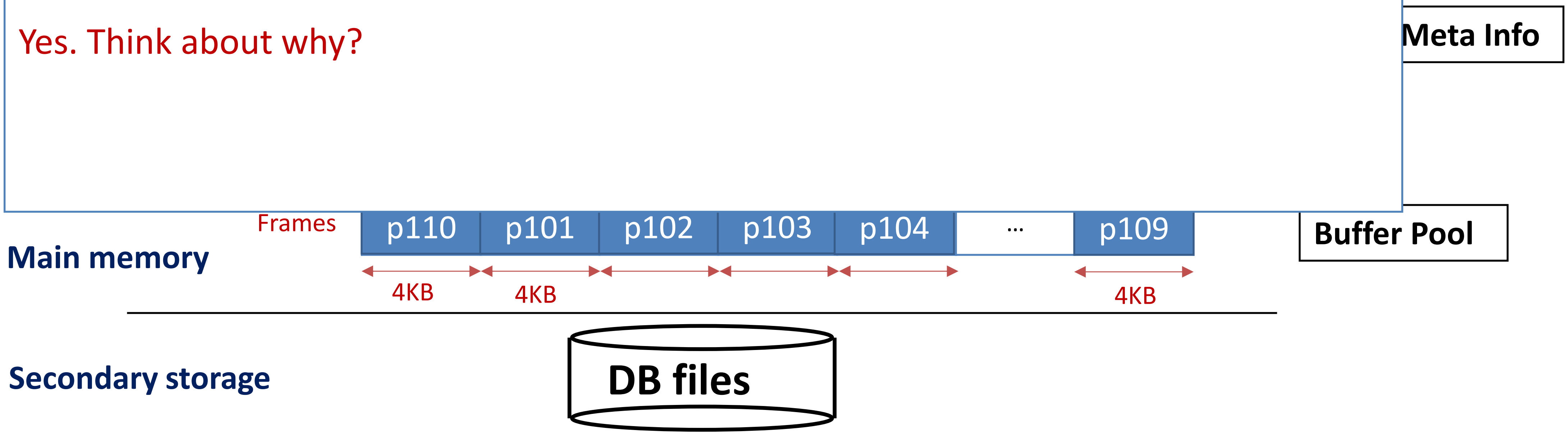
T1: BufferId b1 = BufMgr.PinPage(110, &f1) // b1 = 0

T2: BufferId b2 = BufMgr.PinPage(99, &f2) // b2 = 1

T1: BufMgr.UnpinPage(b1)

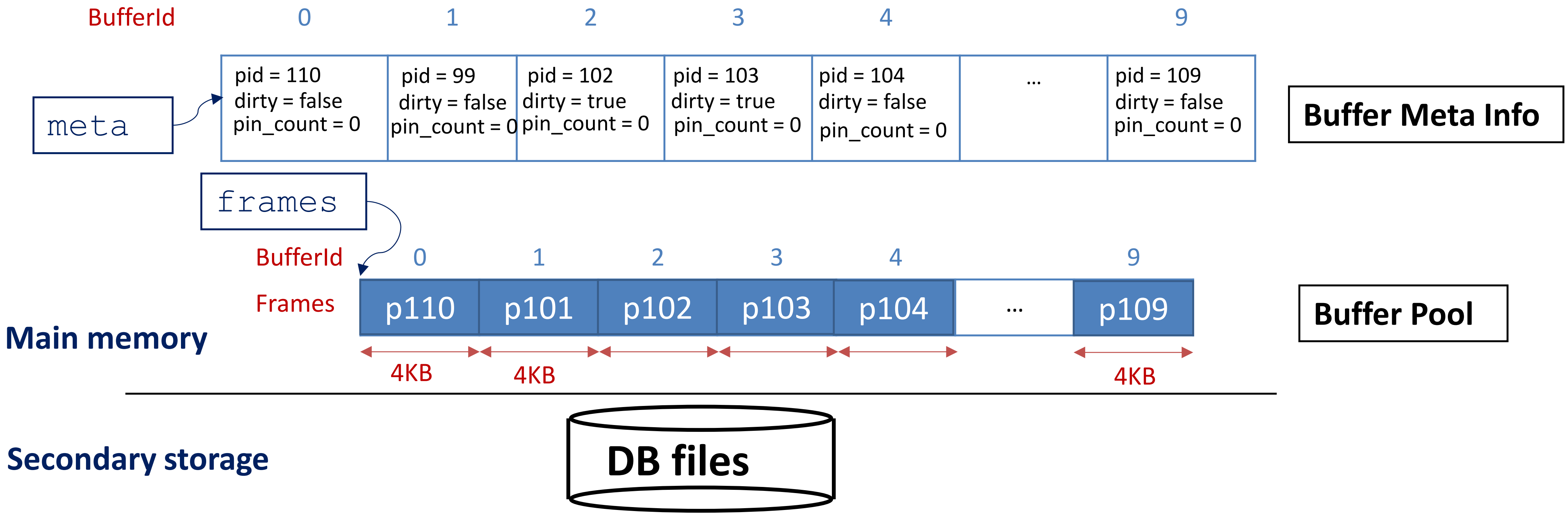
Question: are buffer pins necessary when the DBMS is single-threaded?

Yes. Think about why?



Eviction policy

- How do we choose a victim for eviction?
  - Randomly? The one with the lowest buffer ID that is not pinned? (Inefficient!)



Eviction policy

- Eviction policy (aka replacement policy)
  - An algorithm for choosing unpinned frames when there’s no free frame
    - It can have huge impacts on the # of I/Os, depending on the access pattern

  - Many common choices:
    - Least recently used (LRU)
    - Most recently used (MRU)
    - Clock
    - Database workload specific policies
    - …

Least Recently Used (LRU) policy

- Least Recently Used (LRU)
  - for each page in buffer pool, the order of the pages *were last unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages -> typical transactional workload

- Example: P stands for PinPage, and U stands for UnpinPage, m = 3
  - P(1), P(2), P(3), U(2), U(3), P(4), U(1), P(3), U(3)

BufferId	0	1	2
Frames			
pincount			

Least Recently Used (LRU) policy

- Least Recently Used (LRU)
  - for each page in buffer pool, the order of the pages *were last unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages -> typical transactional workload

- Example: P stands for PinPage, and U stands for UnpinPage, m = 3
  - P(1), P(2), P(3), U(2), U(3), P(4), U(1), P(3), U(3)

BufferId	0	1	2
Frames	p1	p2	p3
pincount	1	1	1

Least Recently Used (LRU) policy

- Least Recently Used (LRU)
  - for each page in buffer pool, the order of the pages *were last unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages -> typical transactional workload

- Example: P stands for PinPage, and U stands for UnpinPage, m = 3
  - P(1), P(2), P(3), U(2), U(3), P(4), U(1), P(3), U(3)

BufferId	0	1	2
Frames	p1	p2	p3
pincount	1	0	1

LRU list:

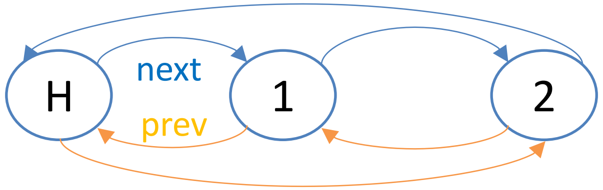
Least Recently Used (LRU) policy

- Least Recently Used (LRU)
  - for each page in buffer pool, the order of the pages *were last unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages -> typical transactional workload

- Example: P stands for PinPage, and U stands for UnpinPage, m = 3
  - P(1), P(2), P(3), U(2), U(3), P(4), U(1), P(3), U(3)

BufferId	0	1	2
Frames	p1	p2	p3
pincount	1	0	0

LRU list:



How to implement in practice?

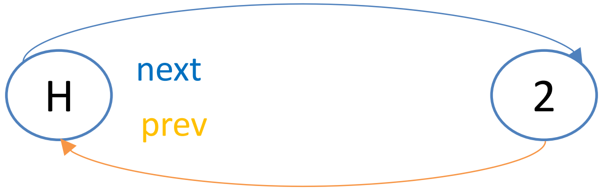
Exercise: how to remove a node in the middle of LRU list when there’s a buffer hit?

Least Recently Used (LRU) policy

- Least Recently Used (LRU)
  - for each page in buffer pool, the order of the pages *were last unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages -> typical transactional workload

- Example: P stands for PinPage, and U stands for UnpinPage, m = 3
  - P(1), P(2), P(3), U(2), U(3), P(4), U(1), P(3), U(3)

LRU list:



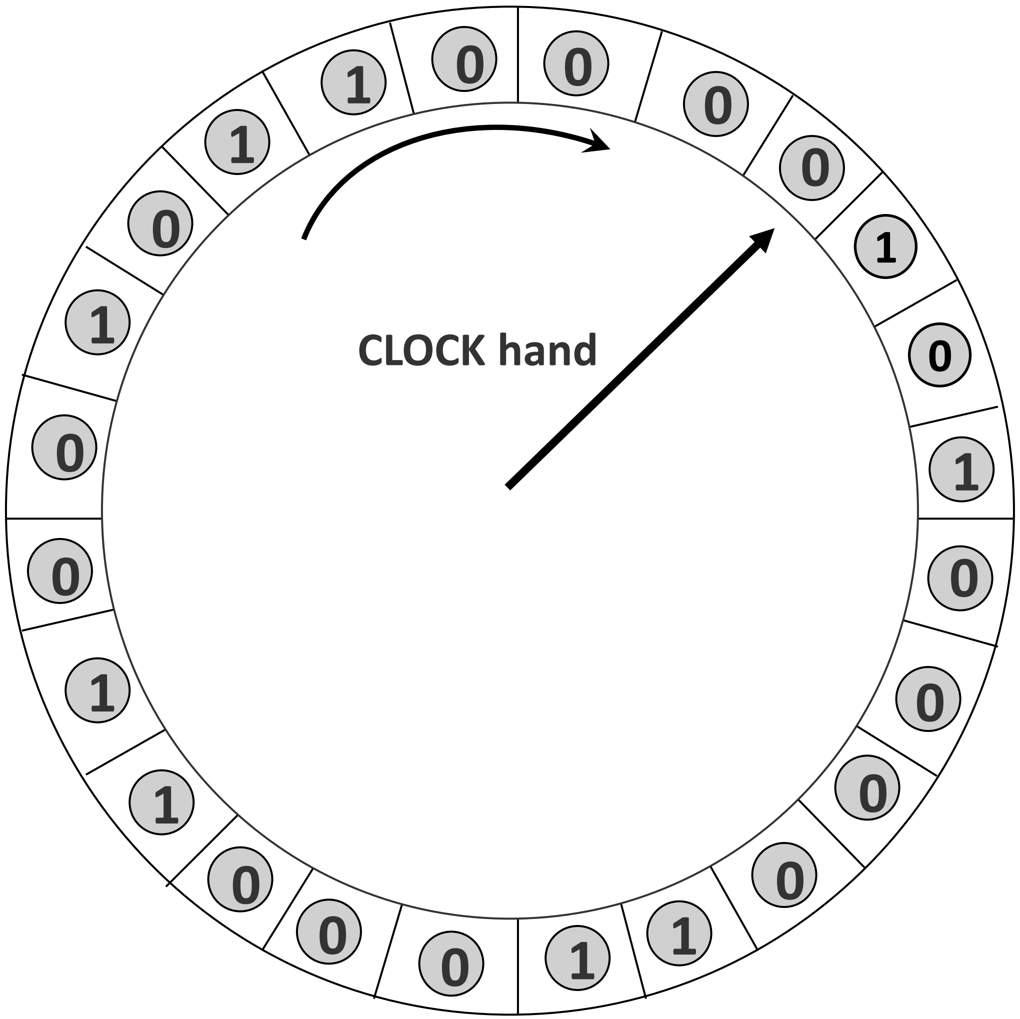
BufferId	0	1	2
Frames	p1	p4	p3
pincount	1	1	0

Least Recently Used (LRU) policy

- Least Recently Used (LRU)
  - for each page in buffer pool, the order of the pages that *were last unpinned*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
    - Works well for repeated accesses to popular pages -> typical transactional workload
- Problems?
  - Sequential flooding:
    - # buffer frames < # pages in file means every existing page in the buffer gets evicted
    - Prevents buffer hit for other transactions working on other files
- DB may know the access pattern before hand so that it can adapt its replacement policies
  - e.g., using a small ring buffer for sequential scan to avoid flooding the entire buffer pool

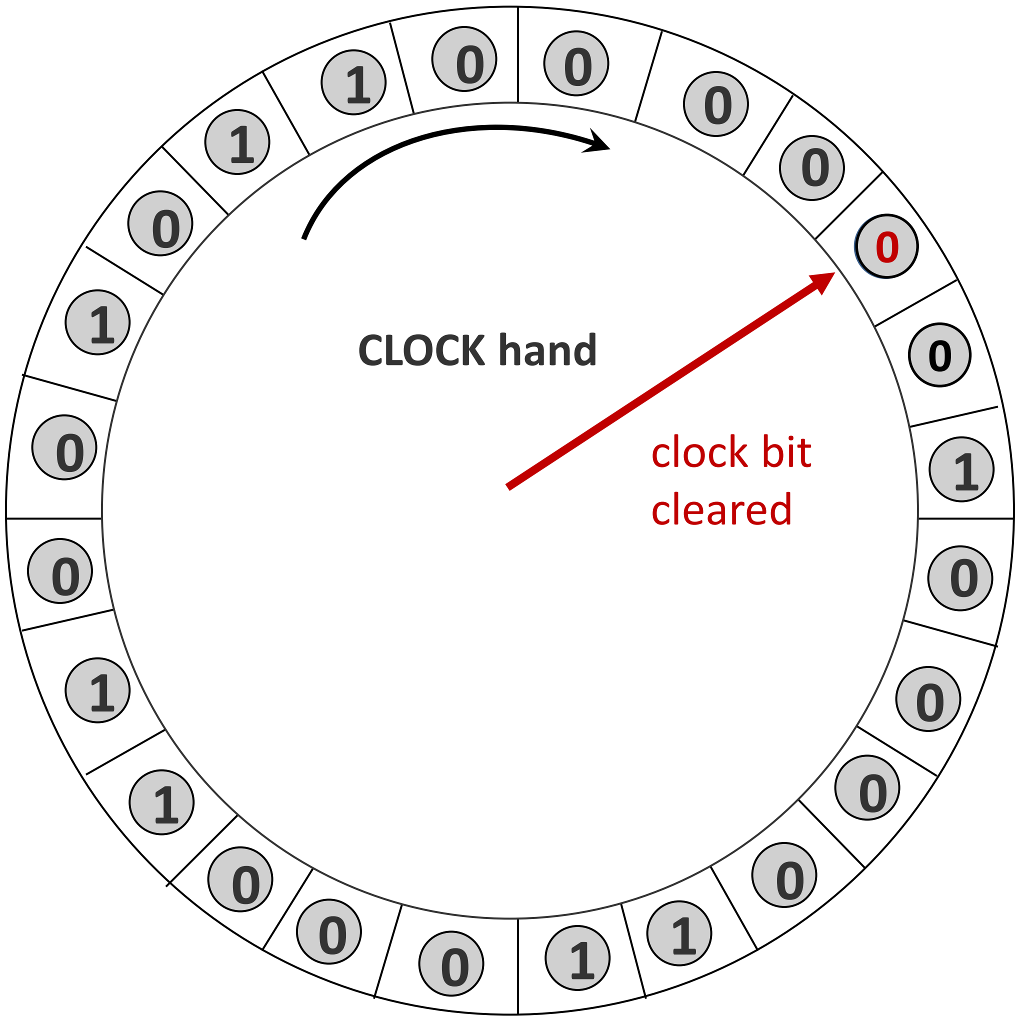
Clock policy

- Approximate LRU
- Each buffer frame has a clock bit
  - Set upon page pinned or unpinned (why?)
- When we need an eviction, move the clock hand
  - Ignore any page that is still pinned
  - Otherwise
    - If bit is set, clear it
    - If bit is clear, evict it
    - i.e., second chance



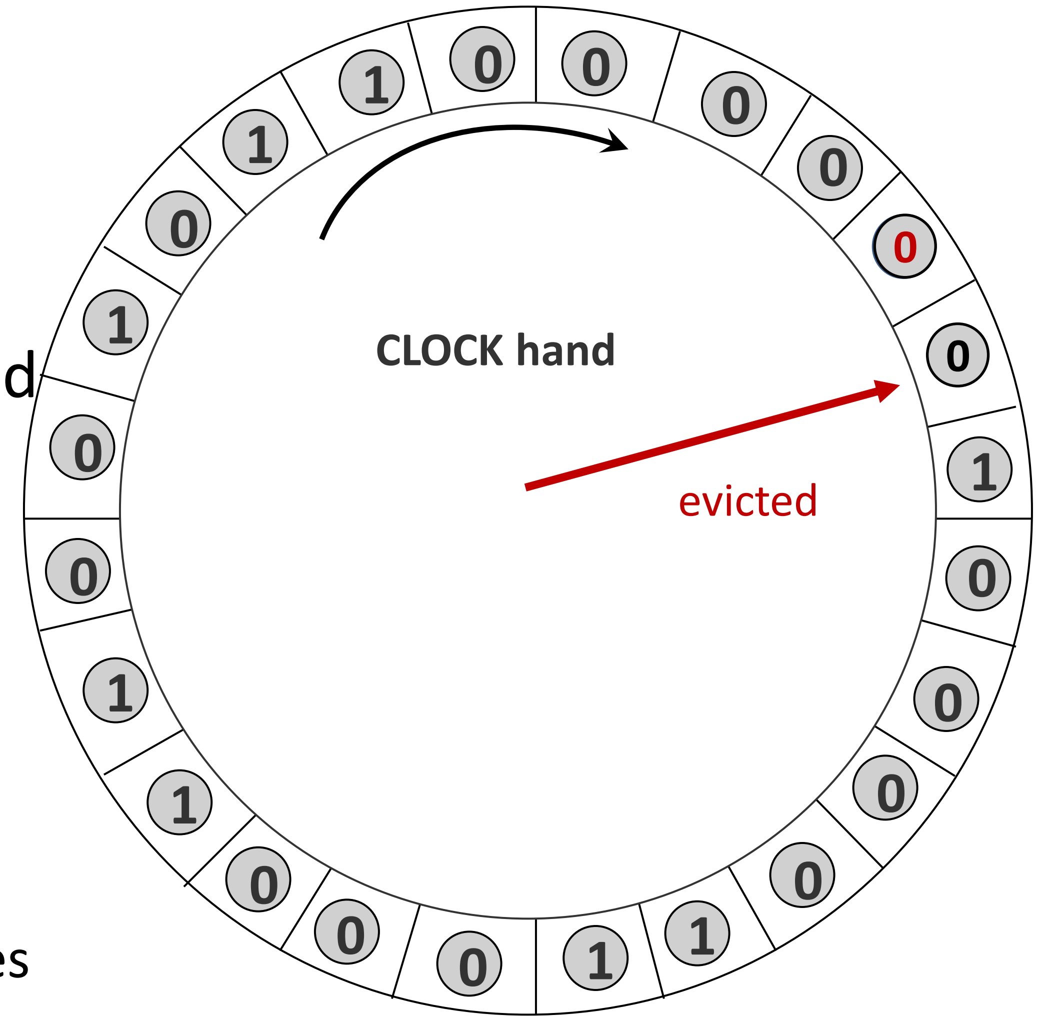
Clock policy

- Approximate LRU
- Each buffer frame has a clock bit
  - Set upon page pinned or unpinned (why?)
- When we need an eviction, move the clock hand
  - Ignore any page that is still pinned
  - Otherwise
    - If bit is set, clear it
    - If bit is clear, evict it
    - i.e., second chance



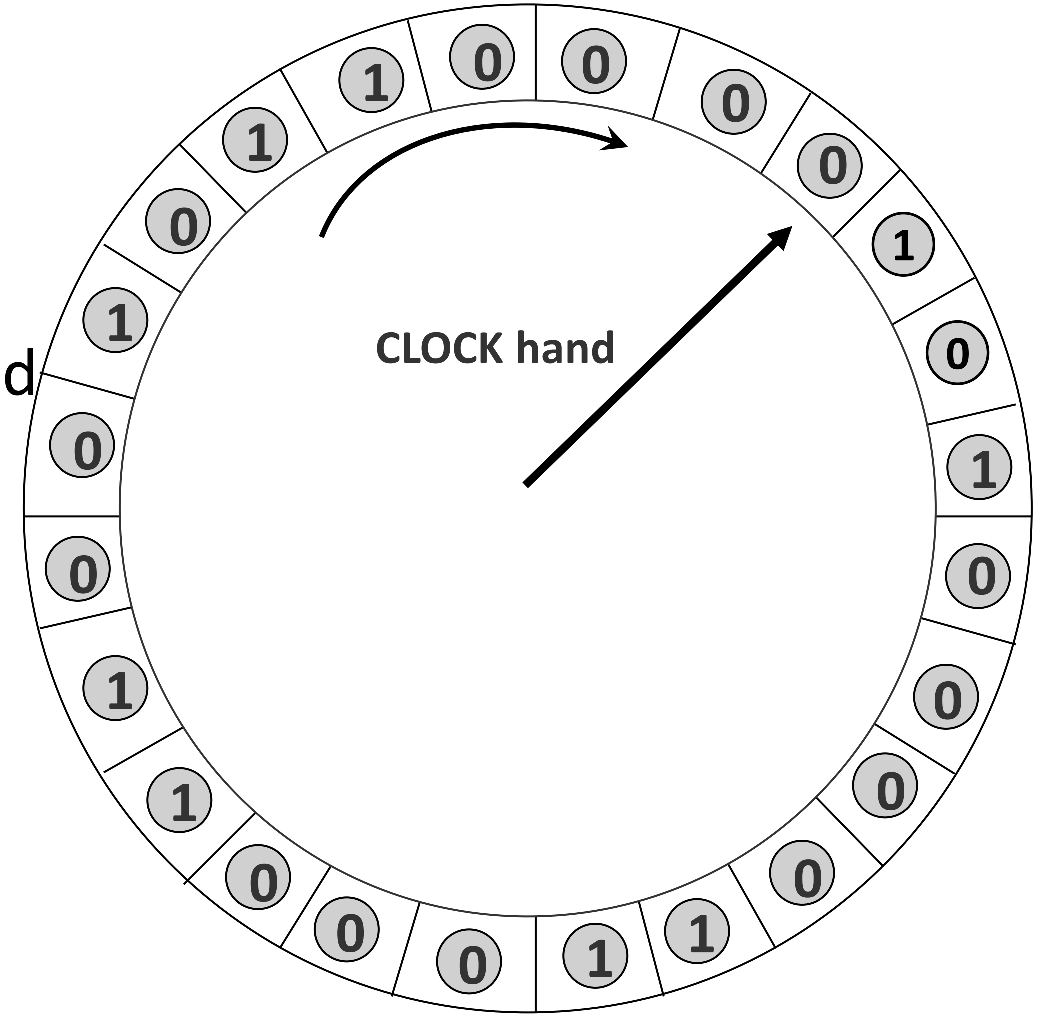
Clock policy

- Approximate LRU
- Each buffer frame has a clock bit
  - Set upon page pinned or unpinned (why?)
- When we need an eviction, move the clock hand
  - Ignore any page that is still pinned
  - Otherwise
    - If bit is set, clear it
    - If bit is clear, evict it
    - i.e., second chance
- Why this might be faster and easier to implement than LRU?
  - Hint: put the clock bit into the buffer meta structures
    - scan buffer meta structures instead



Clock policy

- Approximate LRU
- Each buffer frame has a clock bit
  - Set upon page pinned or unpinned (why?)
- When we need an eviction, move the clock hand
  - Ignore any page that is still pinned
  - Otherwise
    - If bit is set, clear it
    - If bit is clear, evict it
    - i.e., second chance
- Alternative: third/fourth/… chance
  - allowing clock counters up to 2/3/…



Buffer flush

- When are dirty pages written back to disk?
  - When evicted
  - During shutdown

  - Forced flush: flushing certain dirty pages to disk
    - when data need to be persisted for data consistency
    - only unpinned page may be flushed
    - other constraints apply (discussed later this semester)

DBMS vs. OS File System

OS does disk space & buffer management as well: why not let OS manage these tasks?

- Some limitations, e.g., files can't span disks.
- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a page to disk & order writes (important for implementing CC, concurrency control, & recovery)
  - adjust *eviction policy,* and prefetch pages based on access patterns in typical DB operations.