CSE 350: Advanced Data Structures and Indexes (Spring 2026)

Lecture 19: I/O Buffers

4/7/2026

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

How does EM Index access pages?

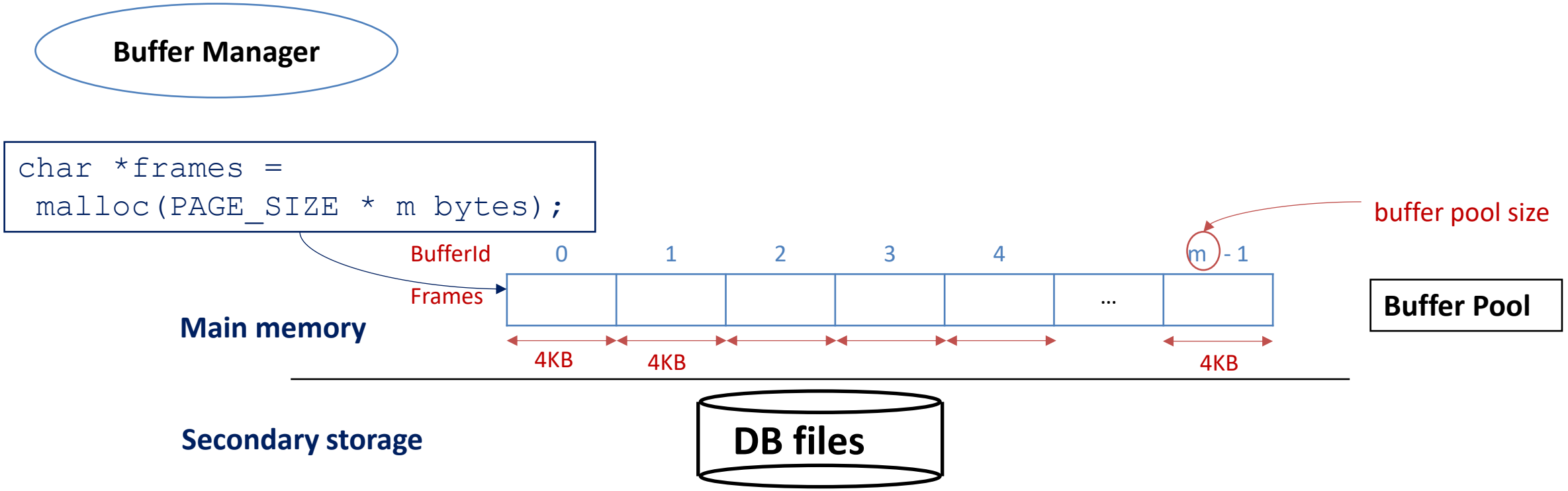
- Pages are stored in disk file
  - suppose we want to search for a key in a B+-tree
    - Need to read O(log N) pages
  - page must be loaded into memory before any computation happens

- What if we want to repeatedly search for some random keys in a B+-tree?
  - Option 1: always read page on demand <- very slow
  - Option 2: load the entire index into memory <- 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

- 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)



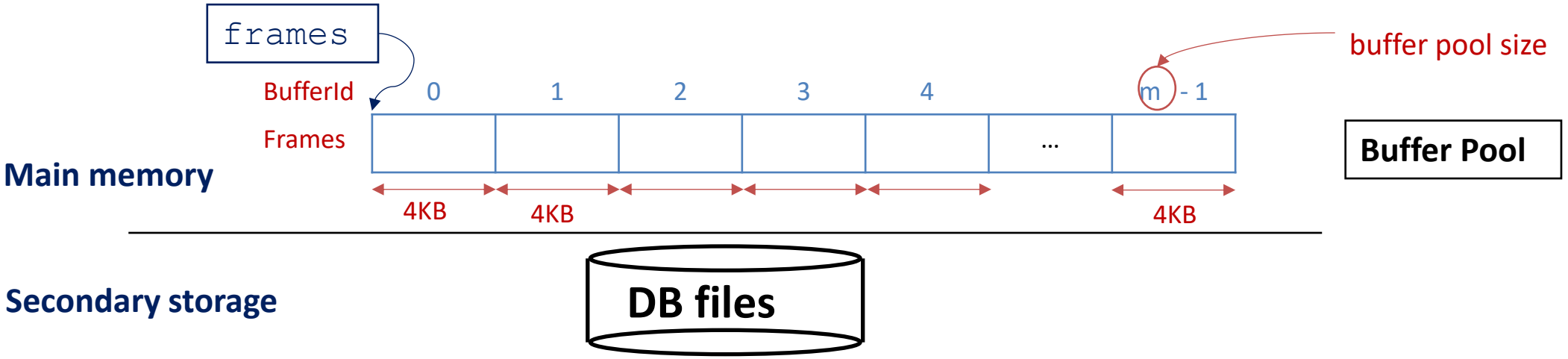
Handling a page request (buffer miss)

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

Upper level components

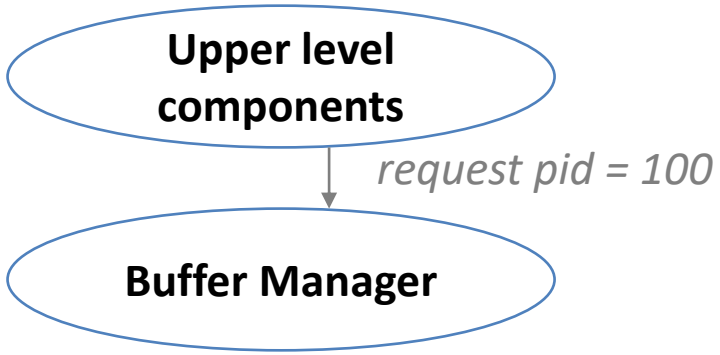
Buffer Manager

```
1 HandlePageRequest(pid):
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```

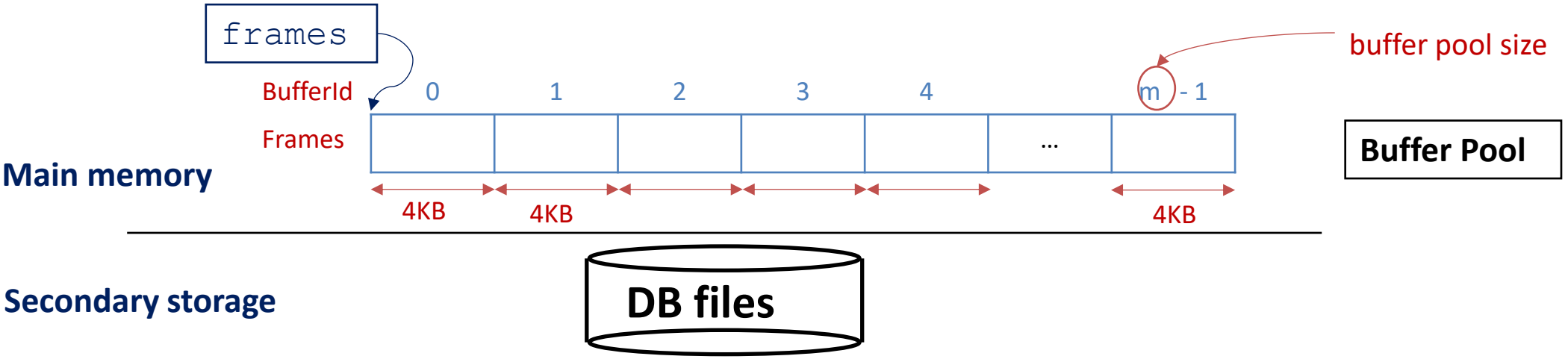


Handling a page request (buffer miss)

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

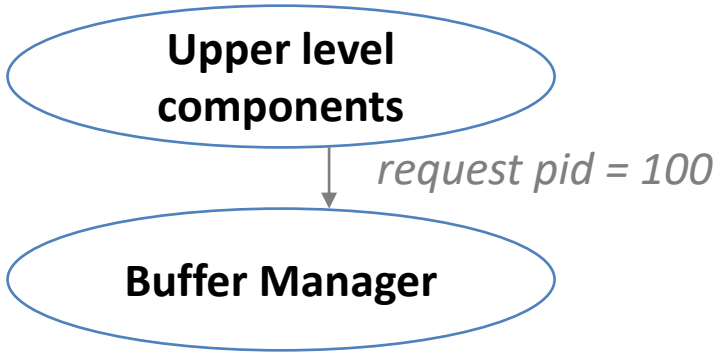



```
1 HandlePageRequest(pid):  // pid = 100
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```

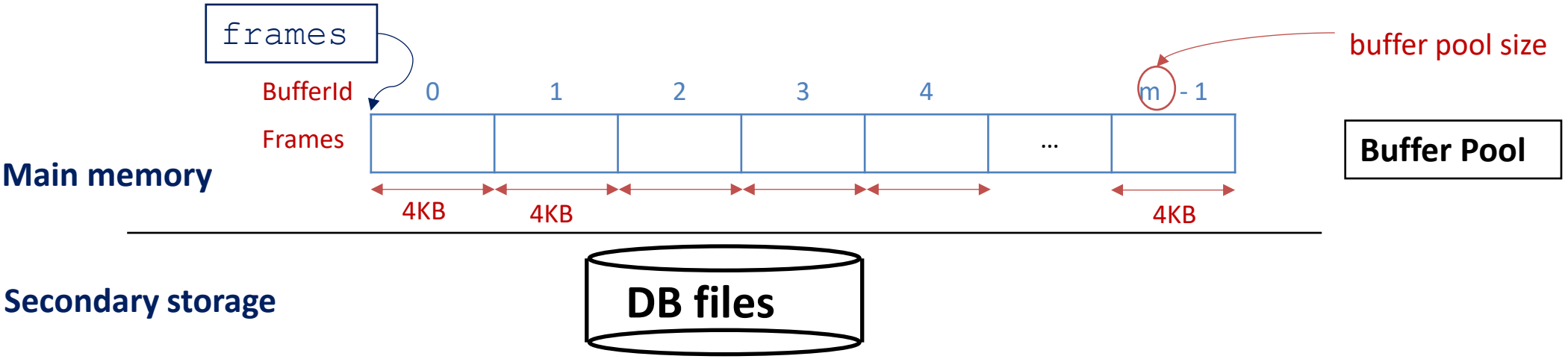


Handling a page request (buffer miss)

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

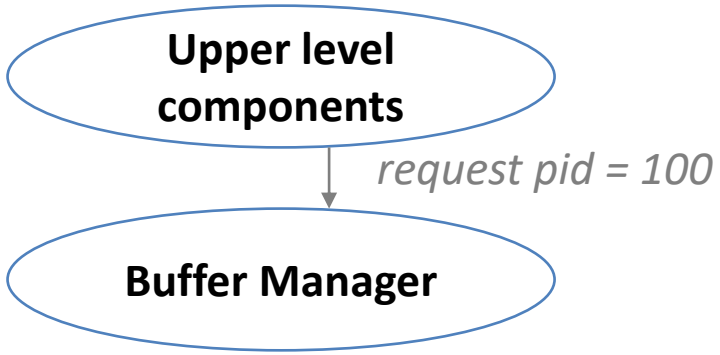



```
1 HandlePageRequest(pid):  // pid = 100
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```

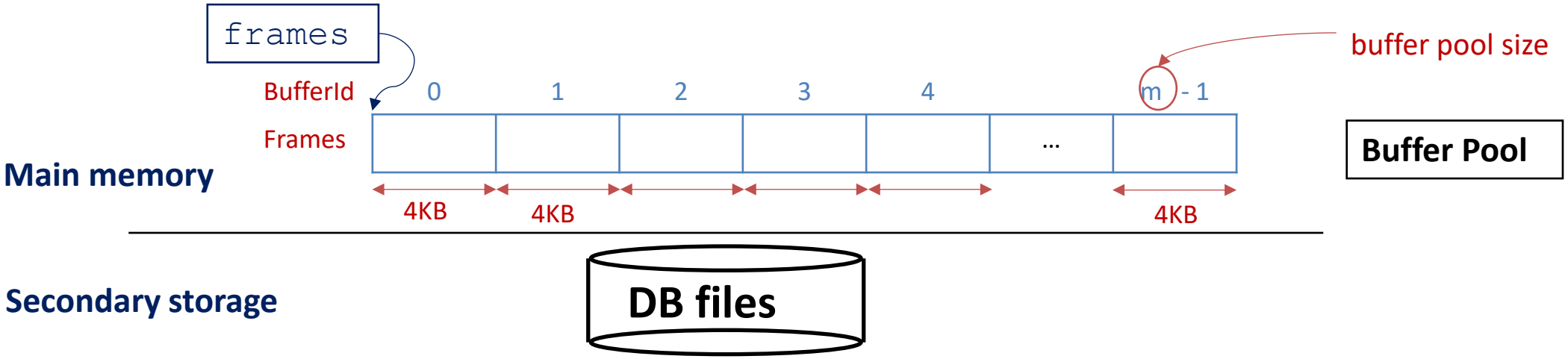


Handling a page request (buffer miss)

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

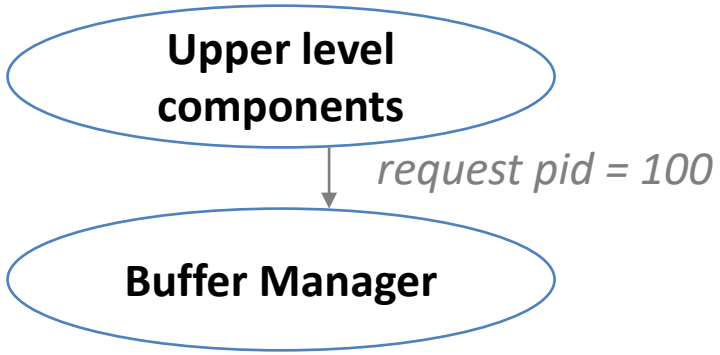



```
1 HandlePageRequest(pid):  // pid = 100
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i     // i = 0
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```

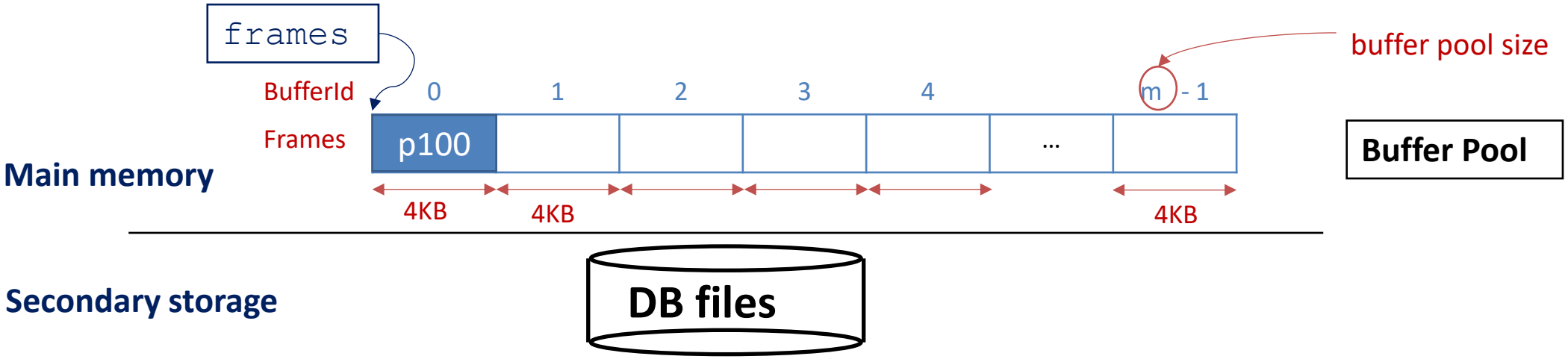


Handling a page request (buffer miss)

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

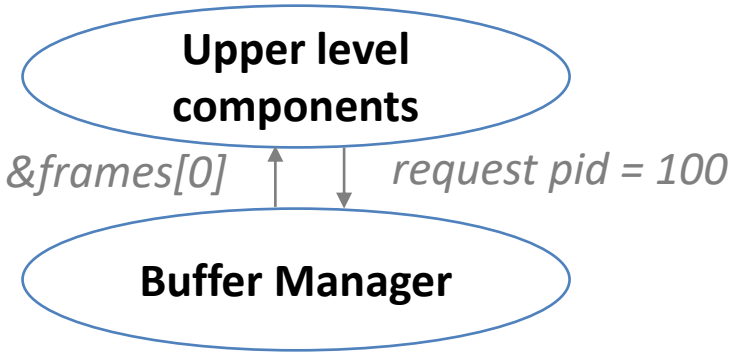



```
1 HandlePageRequest(pid):  // pid = 100
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i     // i = 0
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```

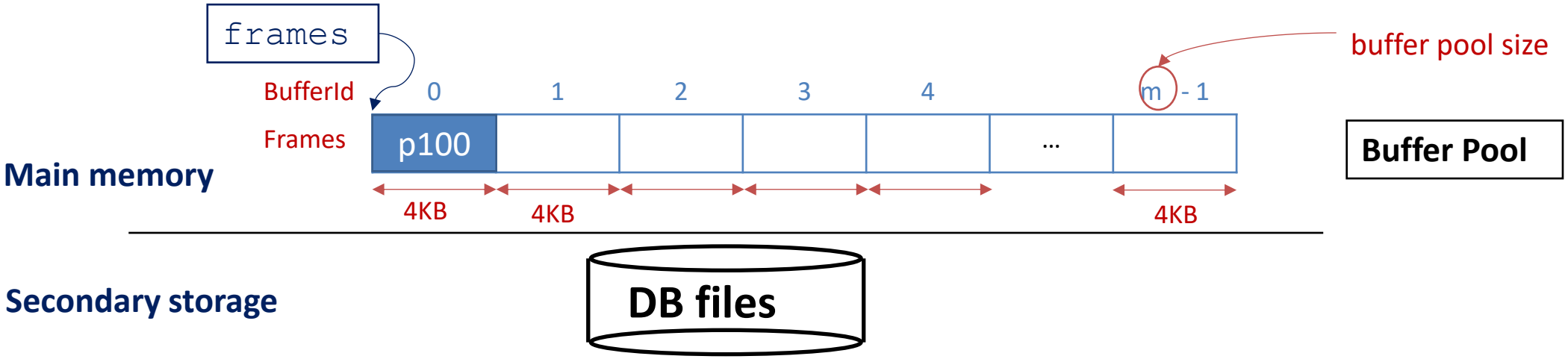


Handling a page request (buffer miss)

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




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



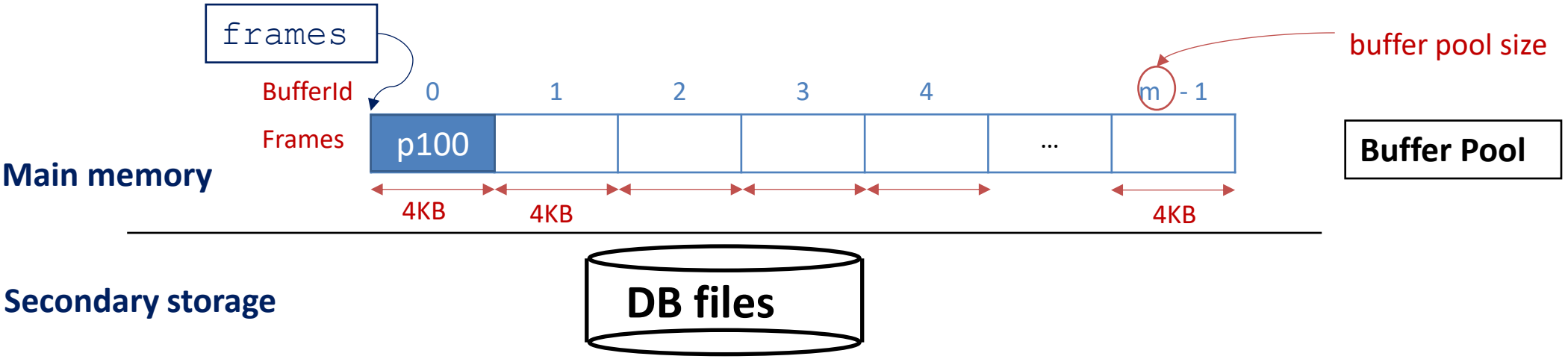
Handling a page request (buffer hit)

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

Upper level components

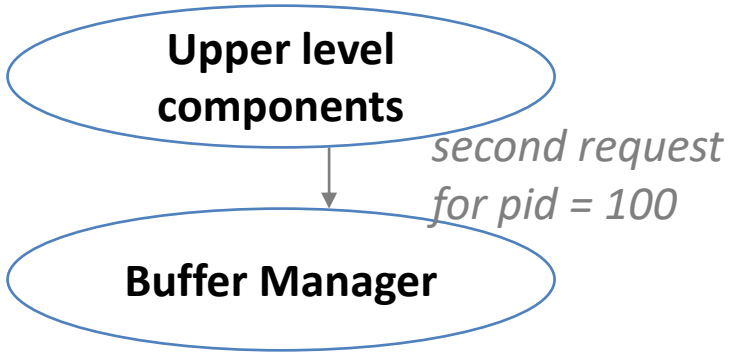
Buffer Manager

```
1 HandlePageRequest(pid):
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```

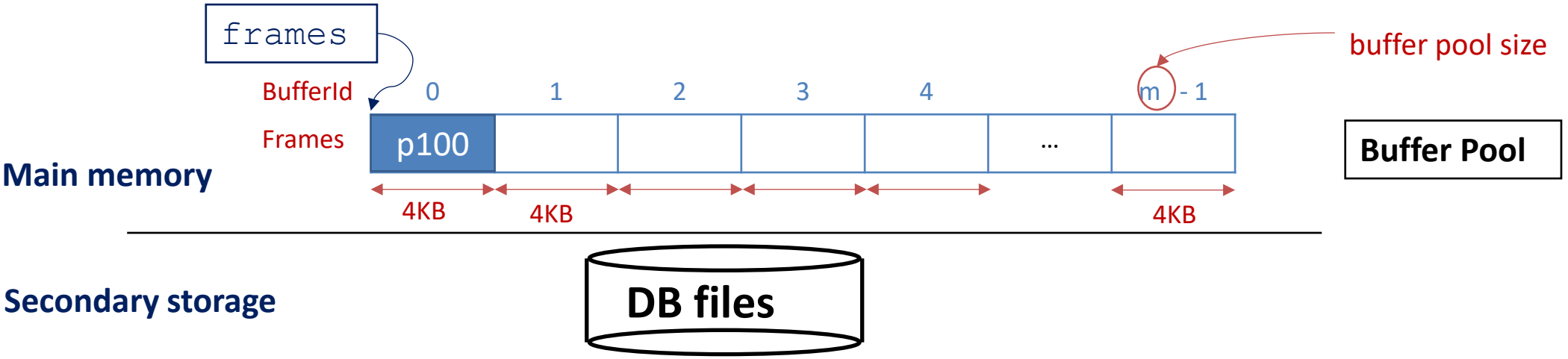


Handling a page request (buffer hit)

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



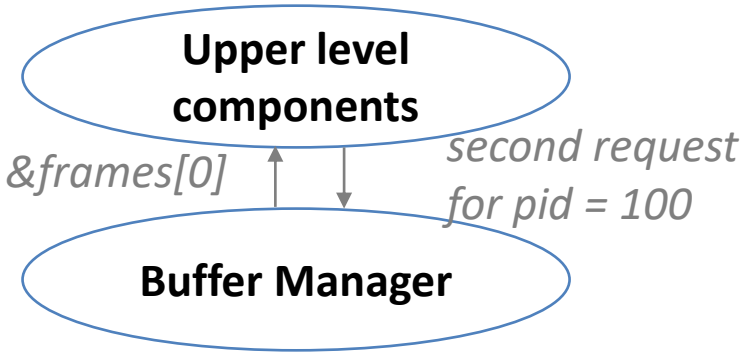

```
1 HandlePageRequest(pid):  // pid = 100
2  if pid exists in some buffer frame i:
3    return &frames[i]
4  else:
5    find a free frame i
6    ReadPage(pid, &frames[i])
7    return &frames[i]
```



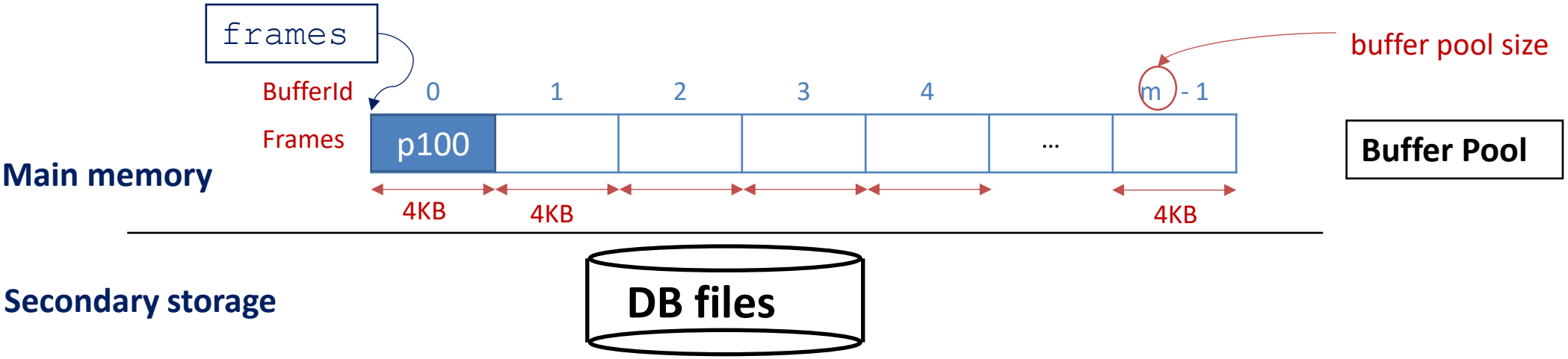
Handling a page request (buffer hit)

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

Cost: 0 I/O




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

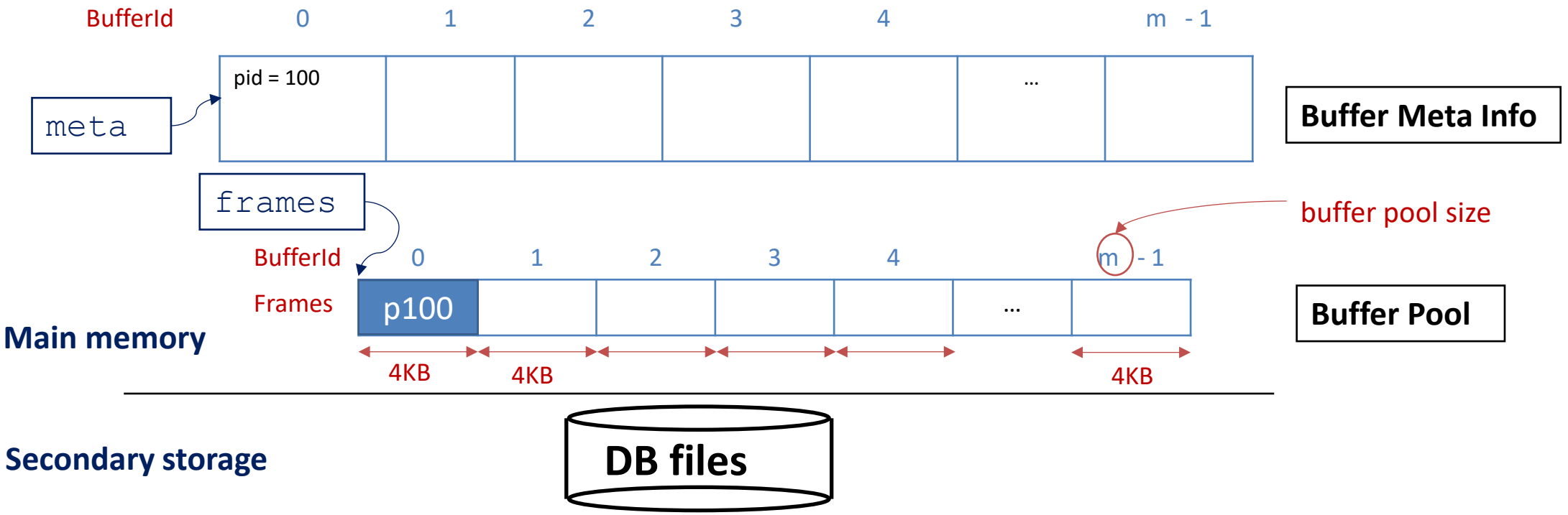


Map page numbers to buffer frames

- 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**.



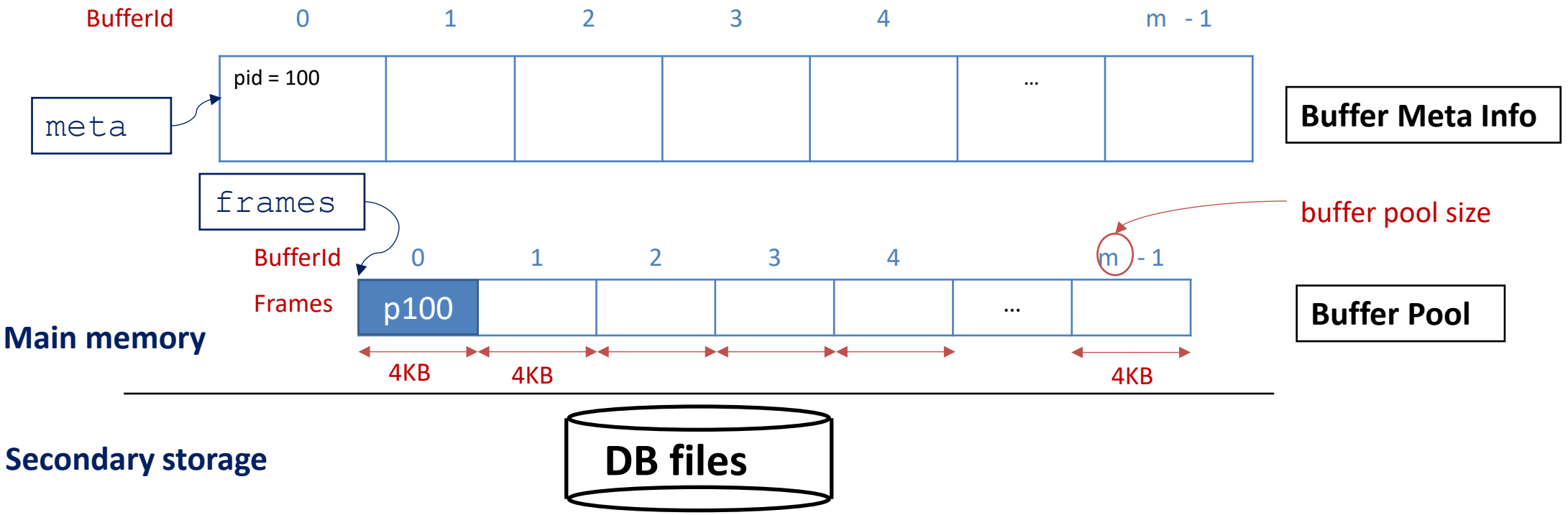
Map page numbers to buffer frames

- 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

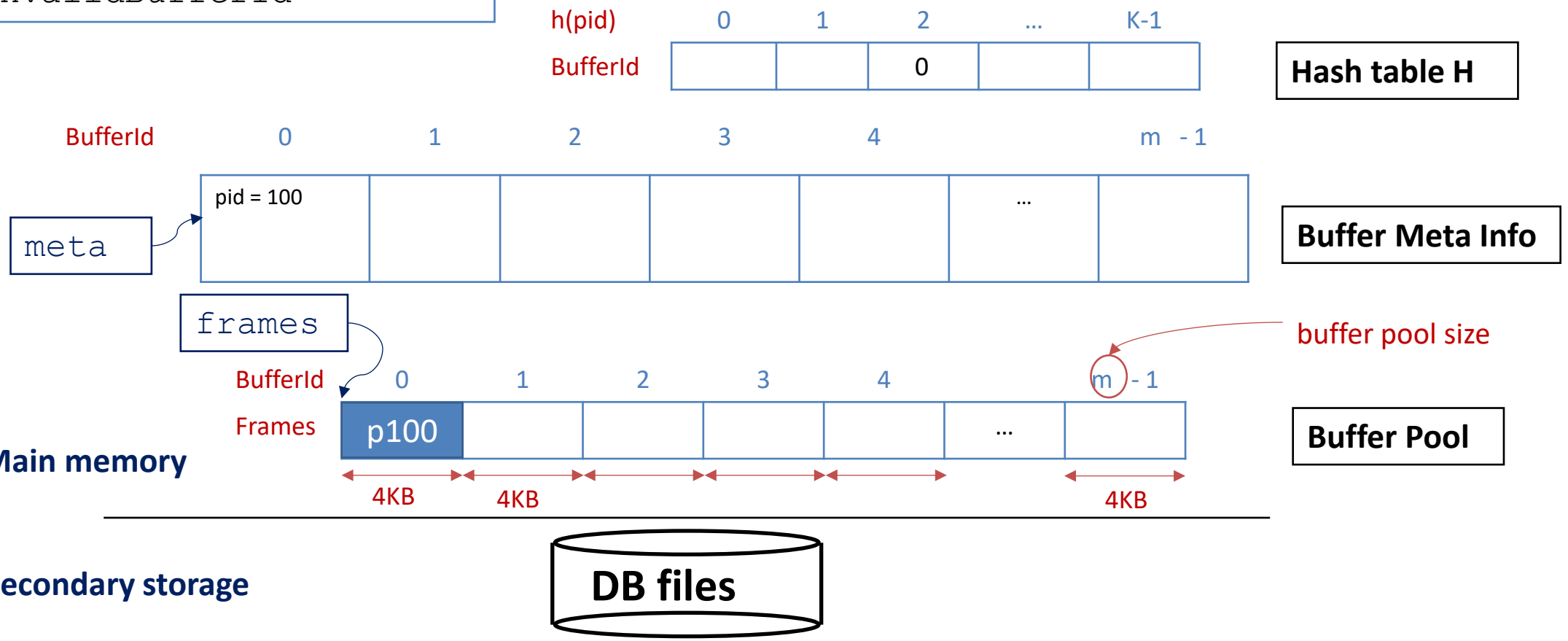
- How to implement line 2?

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

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

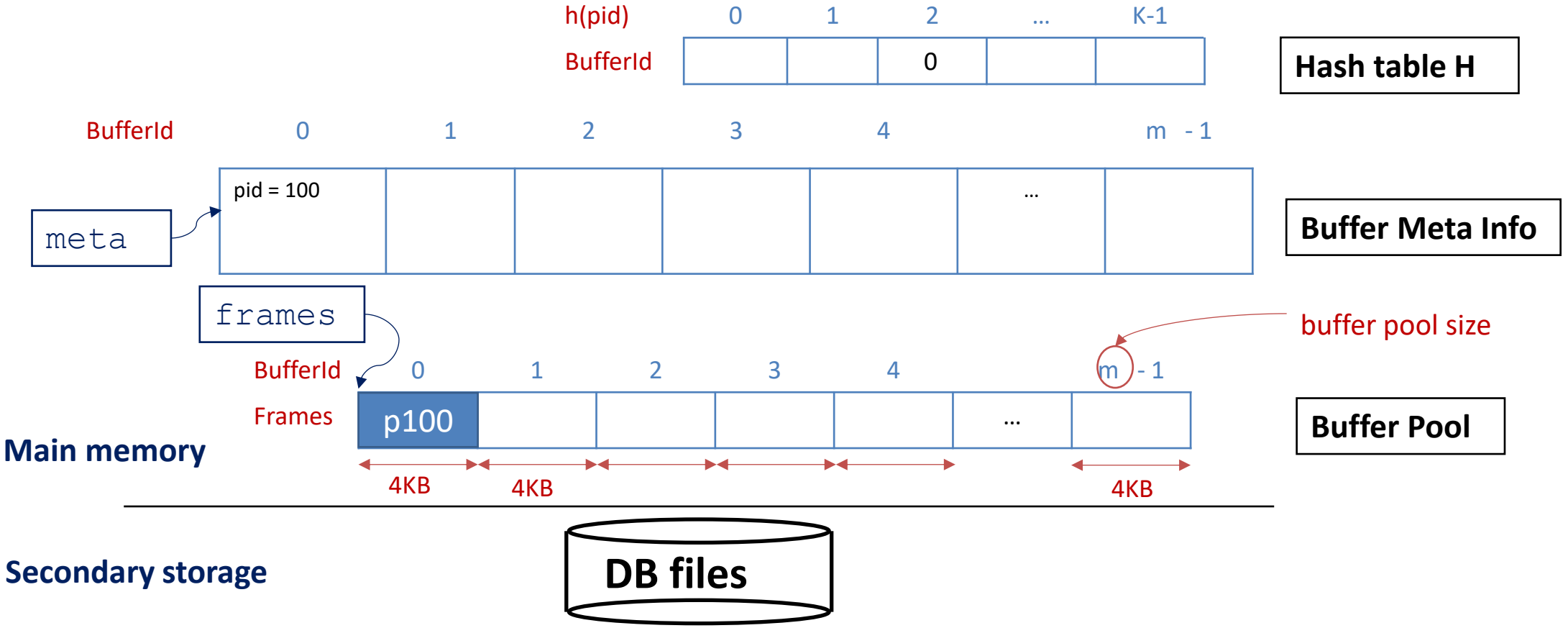
O(1) time in expectation

suppose h(100) == 2



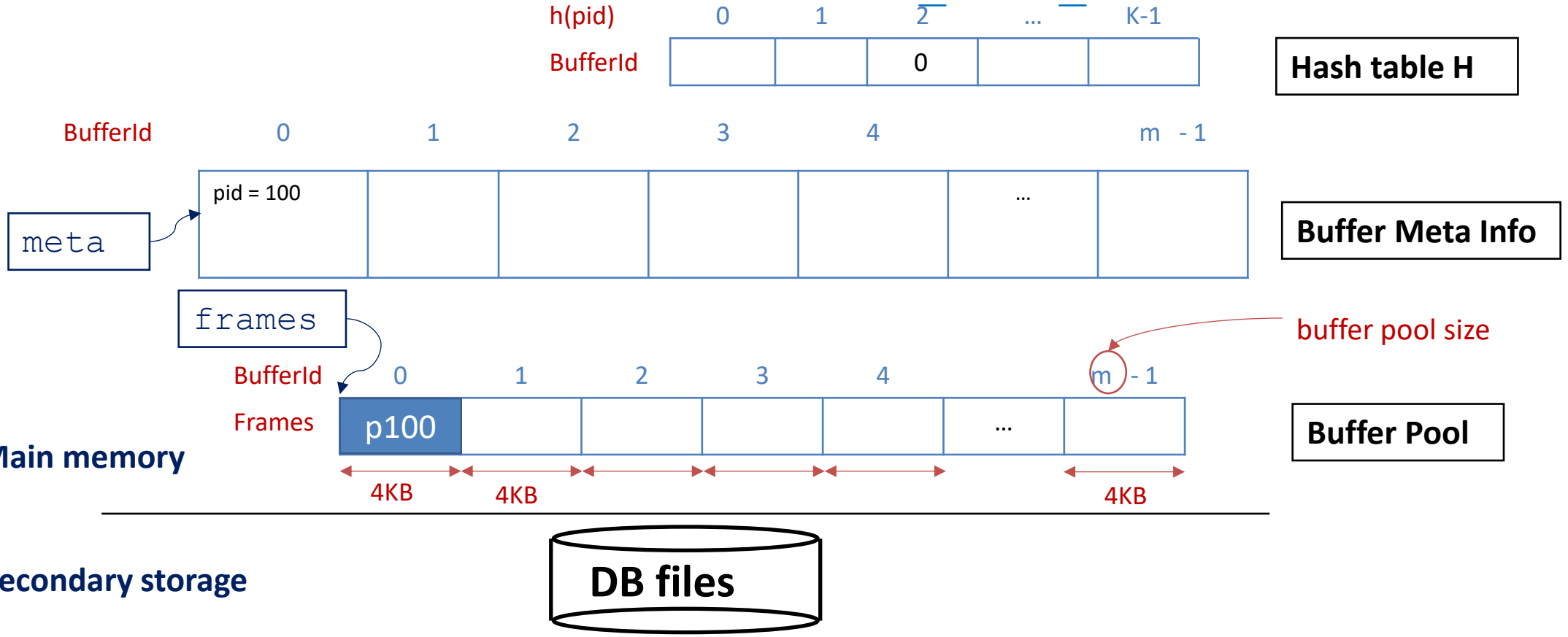
Map page numbers to buffer frames

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



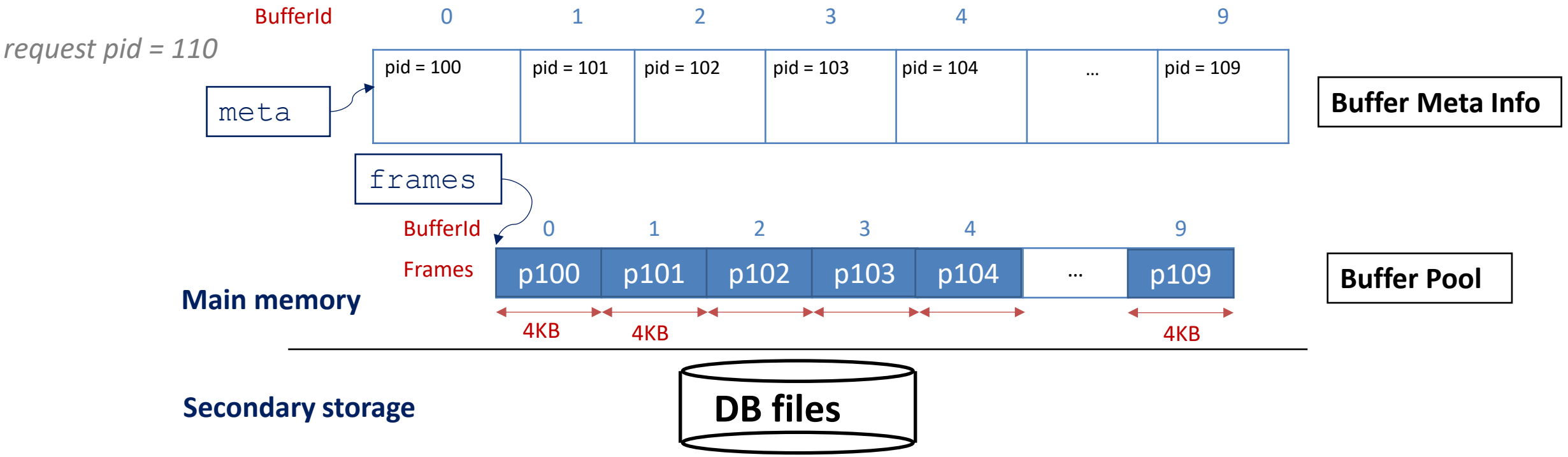
Map page numbers to buffer frames

- 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()`



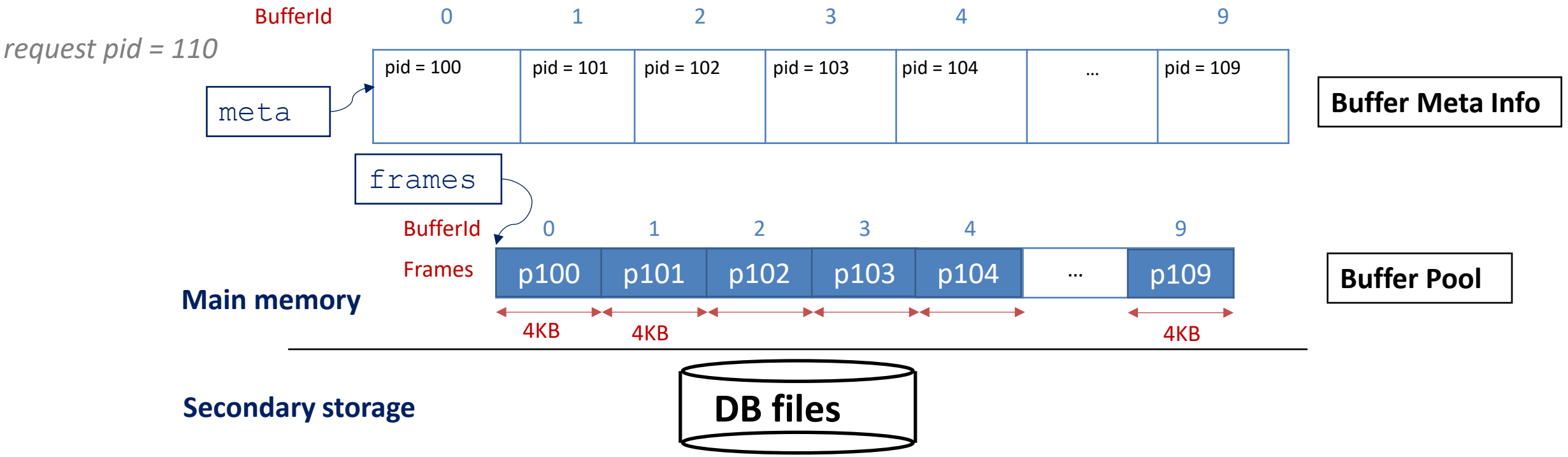
Buffer eviction

- 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



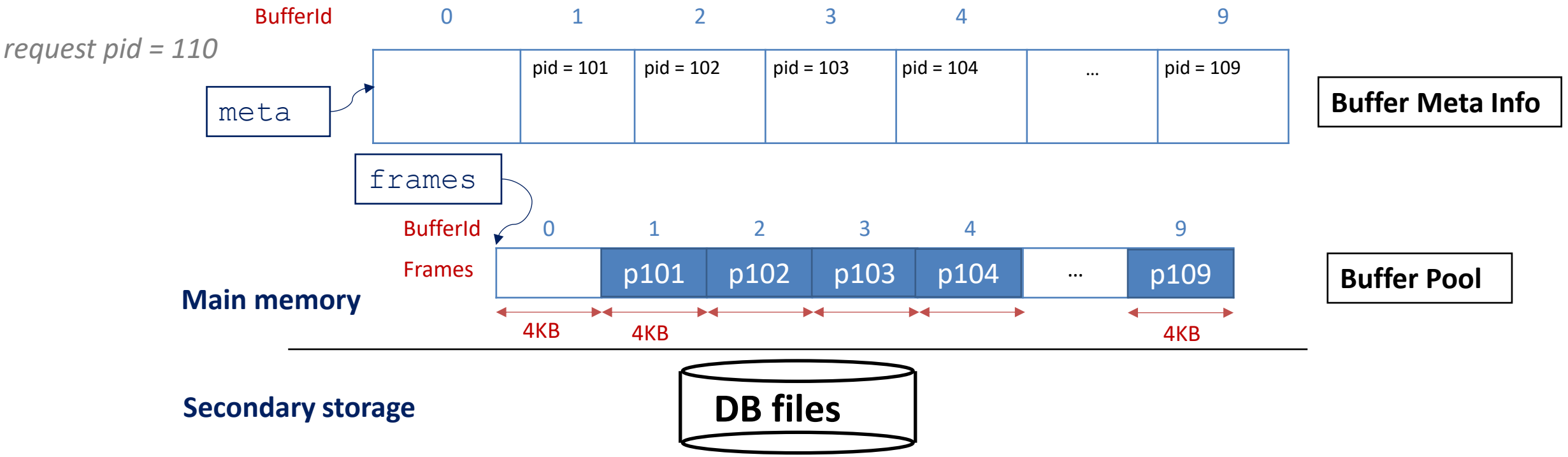
Buffer eviction

- 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)



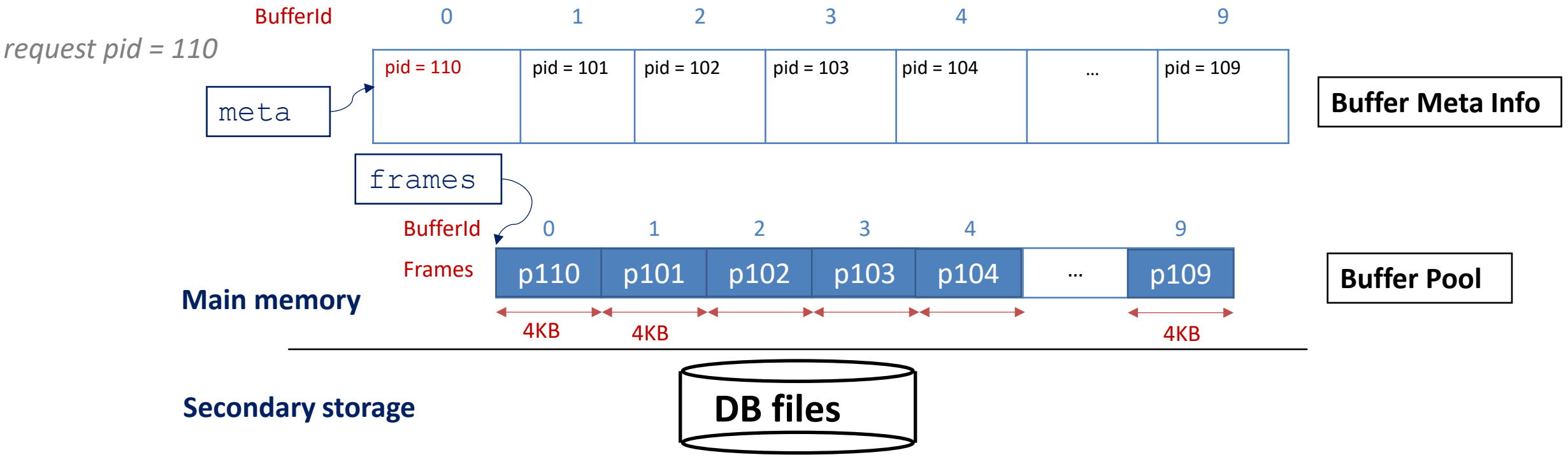
Buffer eviction

- 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)



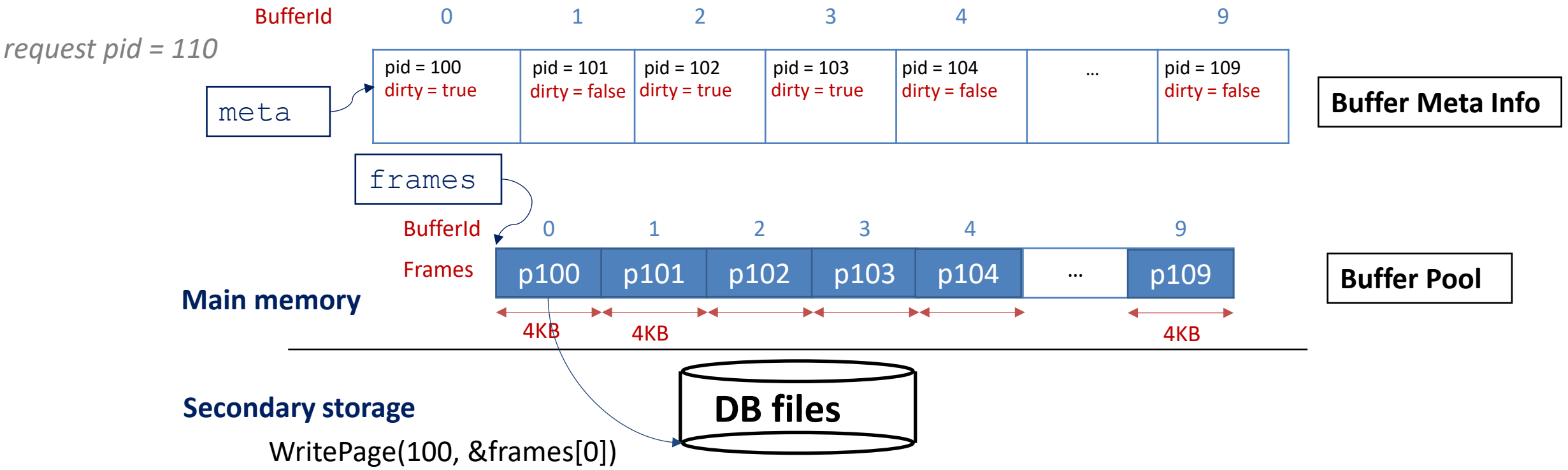
Buffer eviction

- 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)



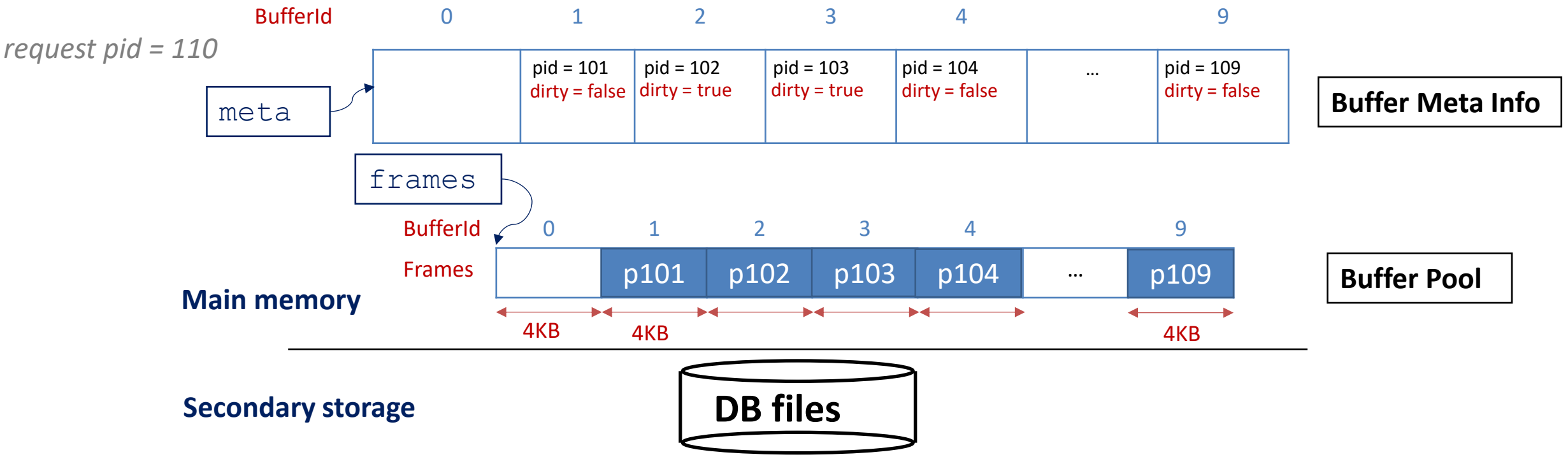
Page requested for writes

- 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



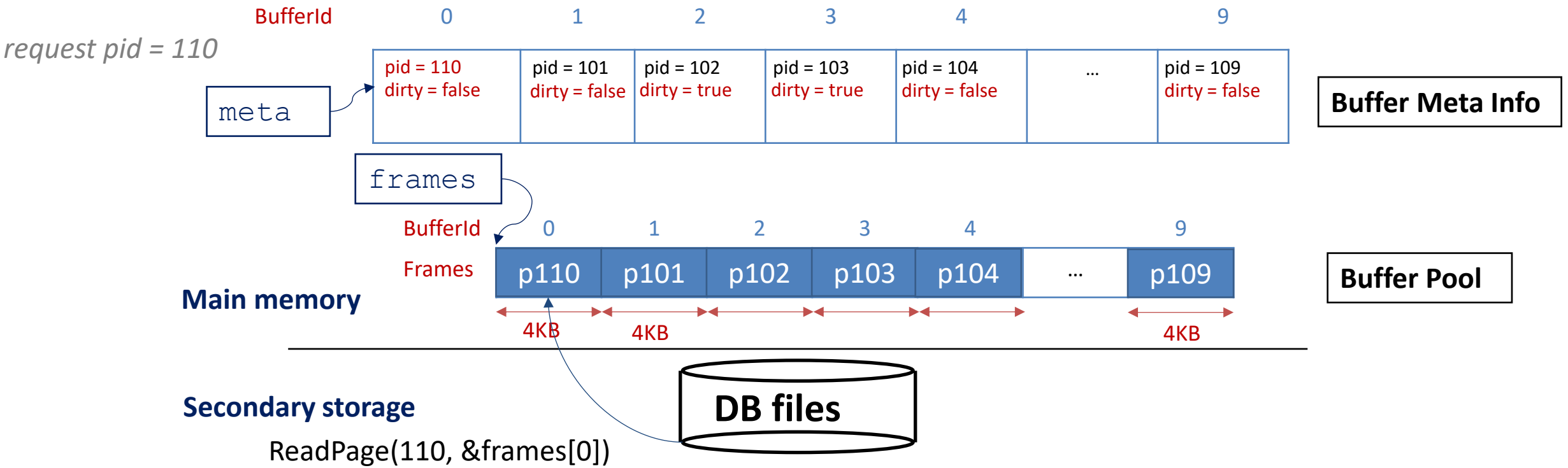
Page requested for writes

- 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

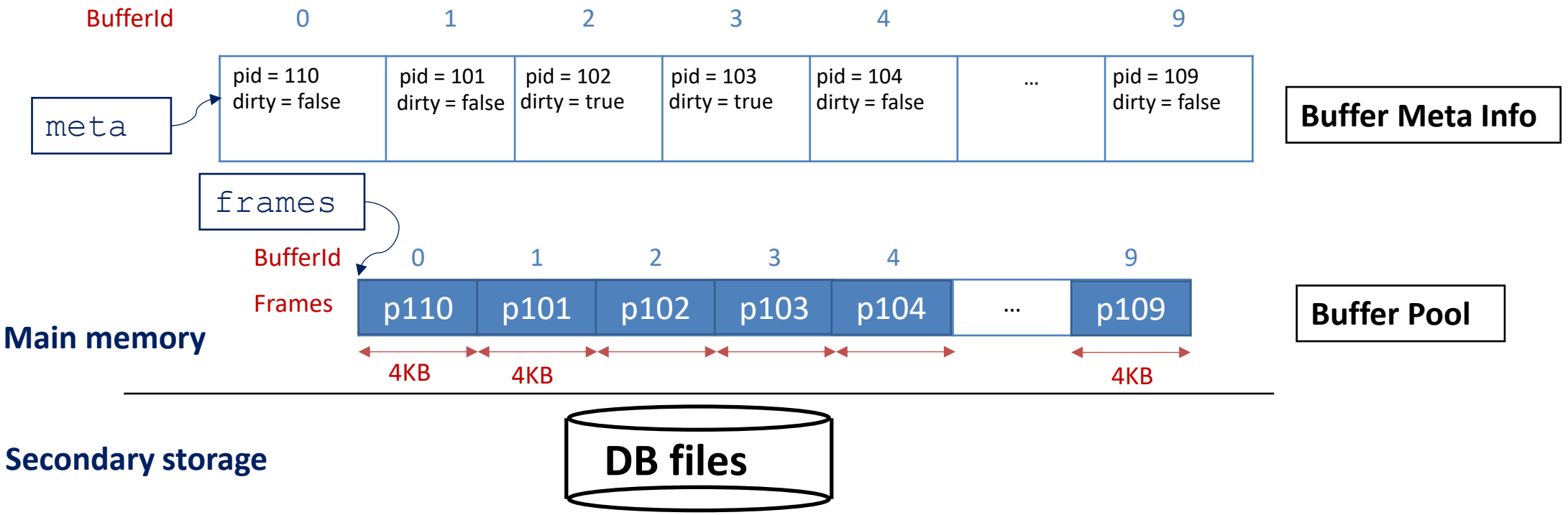
- Potential problem with page eviction?
  - What if the evicted page is modified? (e.g., `UPDATE A SET x = x + 10 WHERE …`)
    - We must write modified page back before eviction



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]

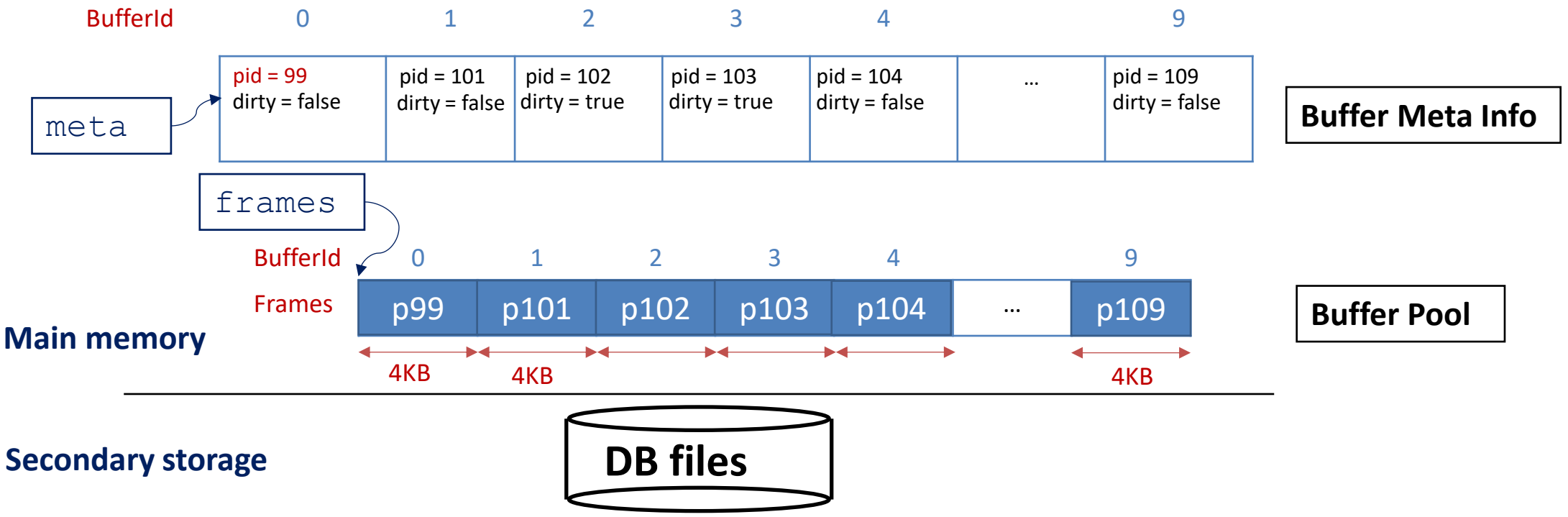


Buffer pins

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

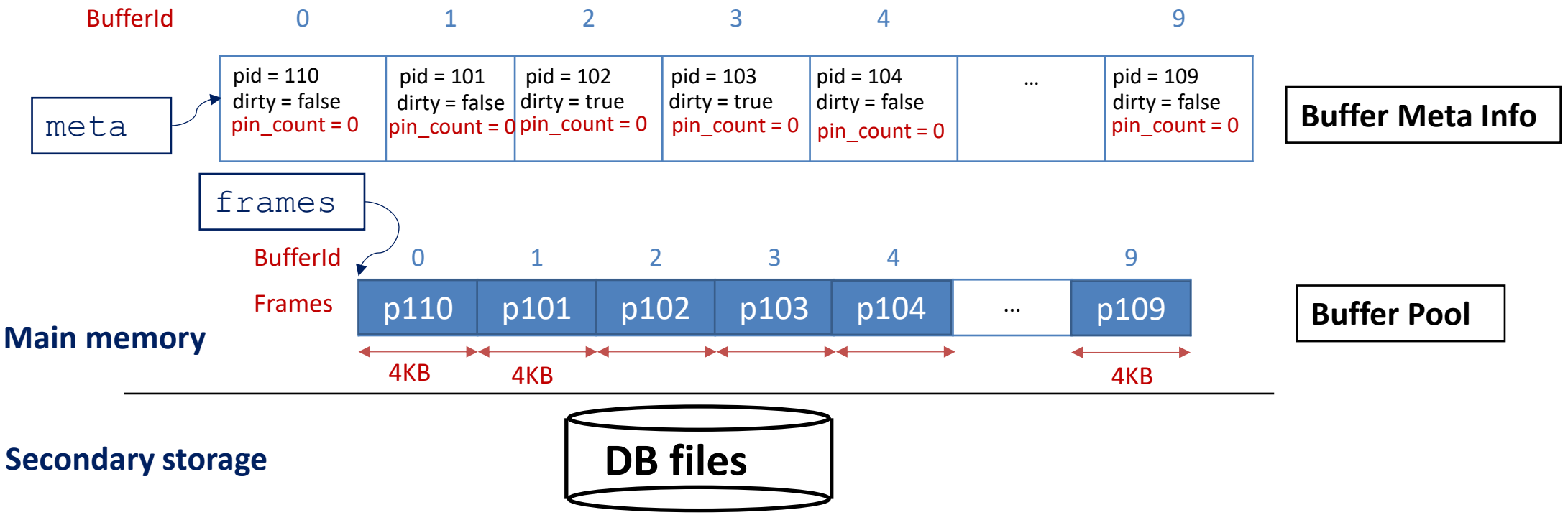
T1: char * f1 = BufMgr.HandlePageRequest(110) // &frames[0] *f1 now contains a wrong page for T1*

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



Buffer pins

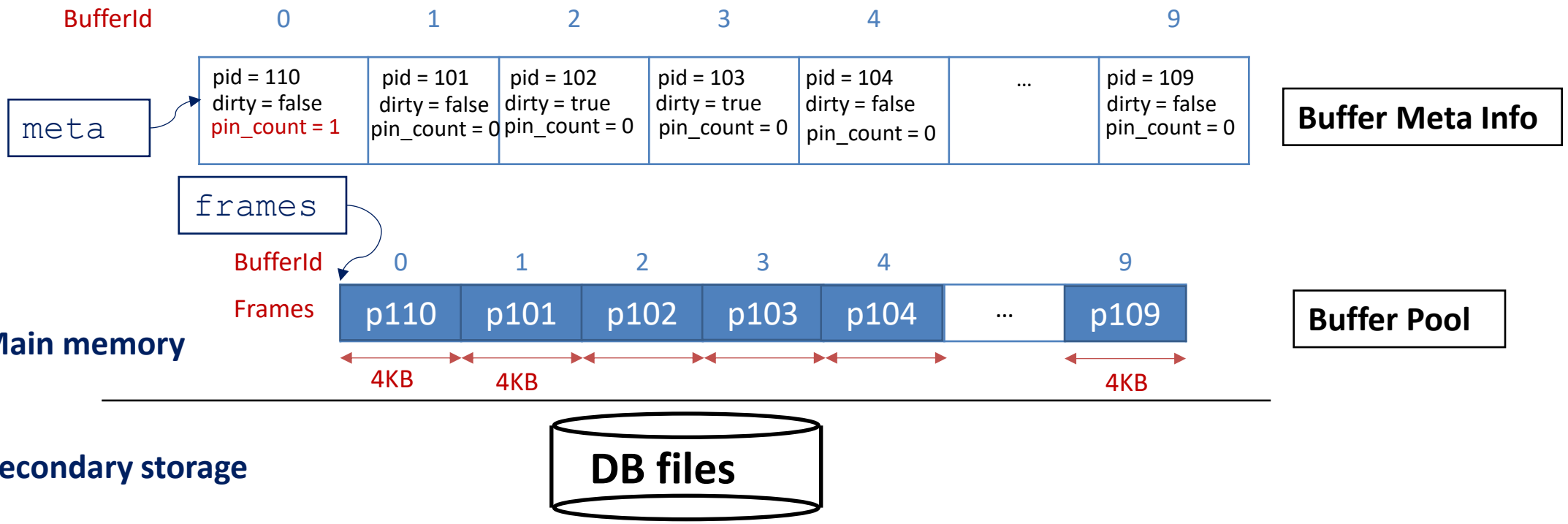
- Solution: introducing a buffer pin count per buffer frame
  - Upon page request, pin count++
  - Upon page release, pin count--
  - Never evict a page with pin count > 0



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

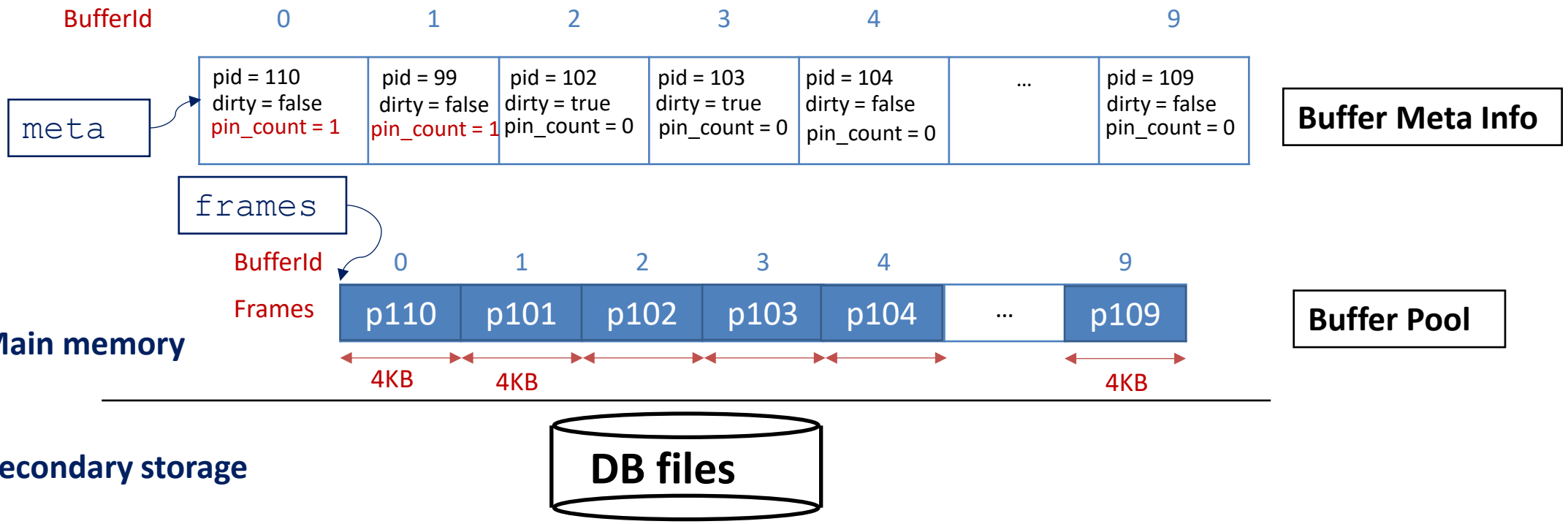
T1: BufferId b1 = BufMgr.PinPage(110, &f1) // b1 = 0



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

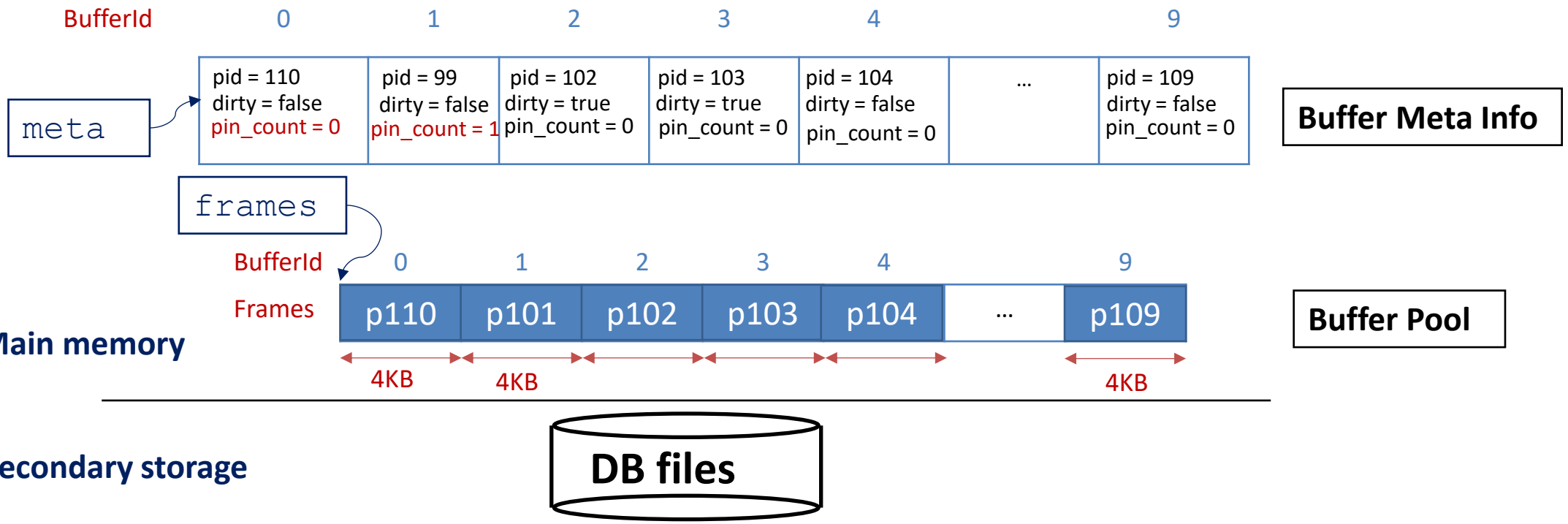
```
T1: BufferId b1 = BufMgr.PinPage(110, &f1)      // b1 = 0
T2: BufferId b2 = BufMgr.PinPage(99, &f2)       // b2 = 1
```



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)
```



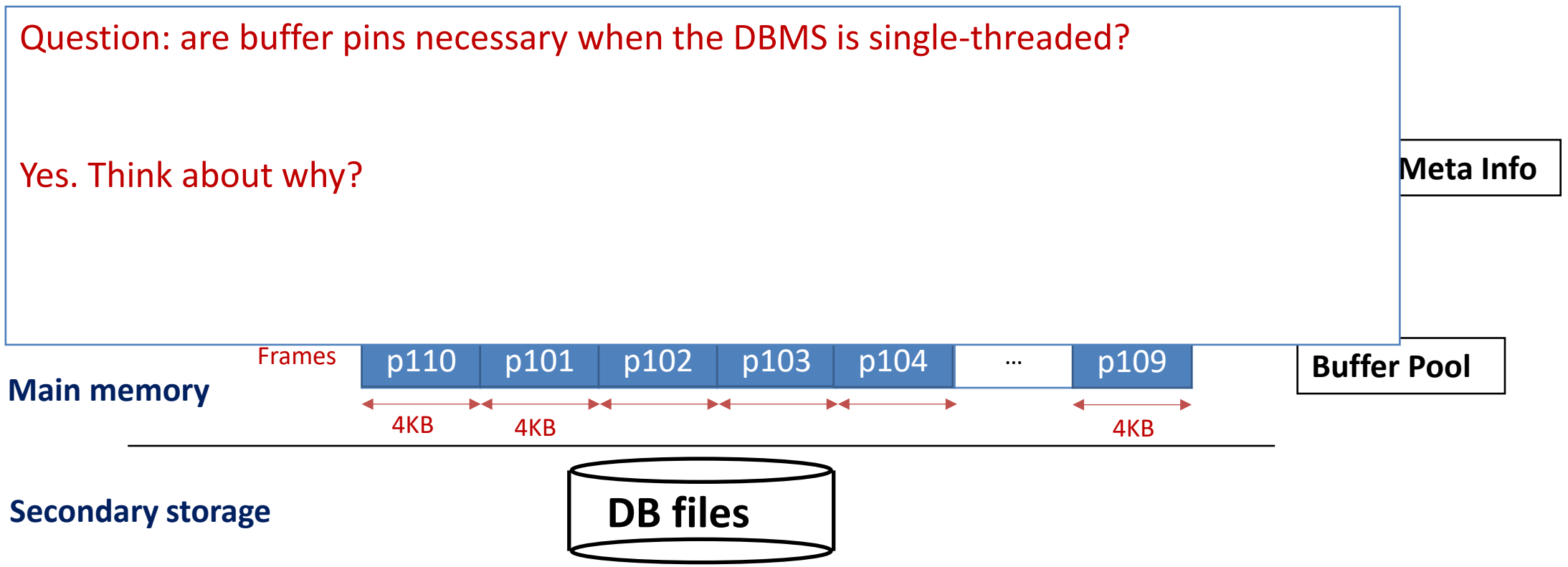
Buffer pins

- 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)
```

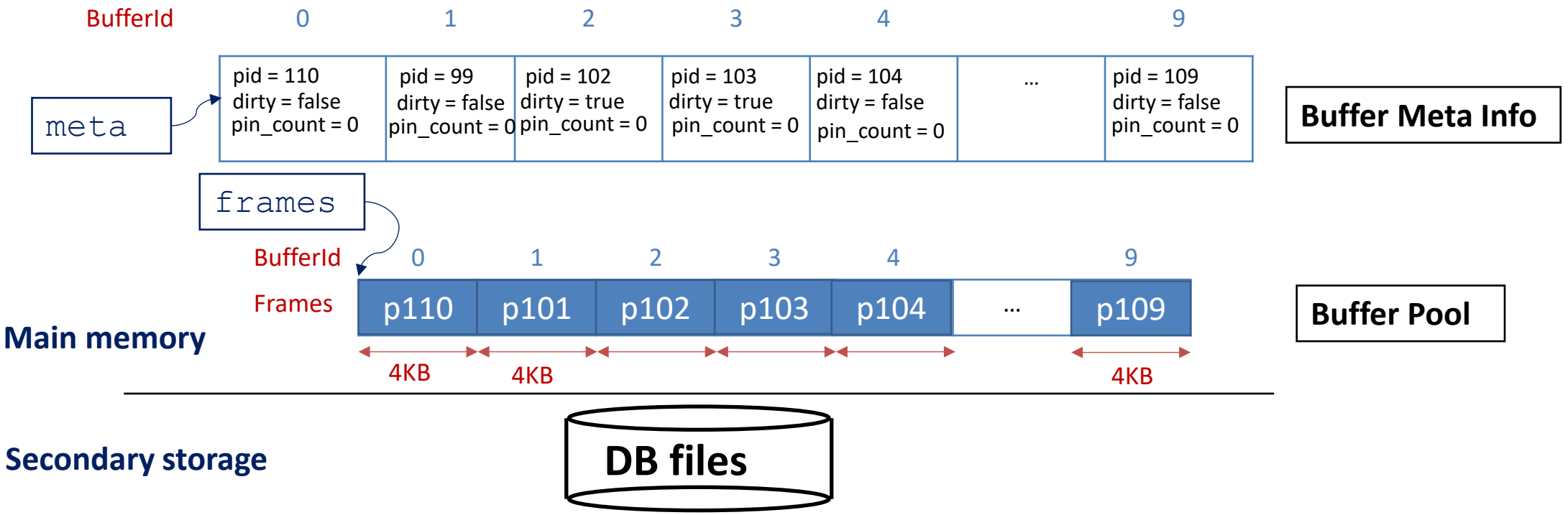
Question: are buffer pins necessary when the DBMS is single-threaded?

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!)



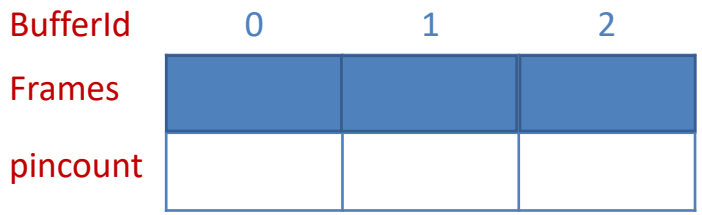
Eviction policy

- 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)



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

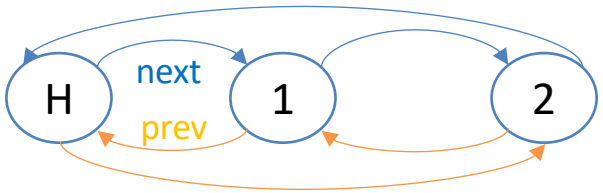
LRU list:

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)

LRU list:



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

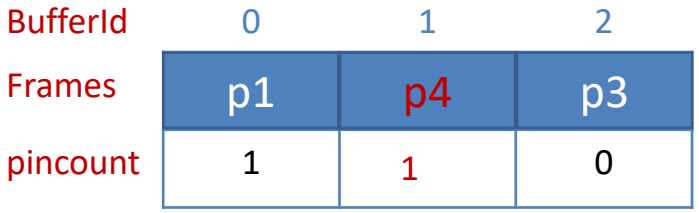
How to implement in practice?

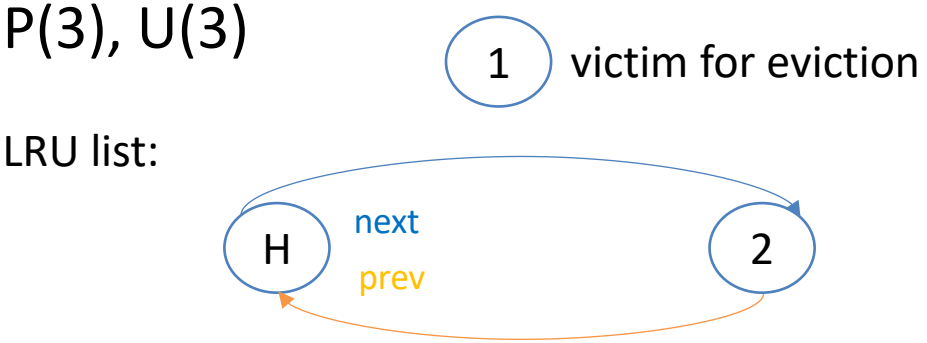
Exercise: how to remove a node in the middle of LRU list when there's a buffer hit?

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)





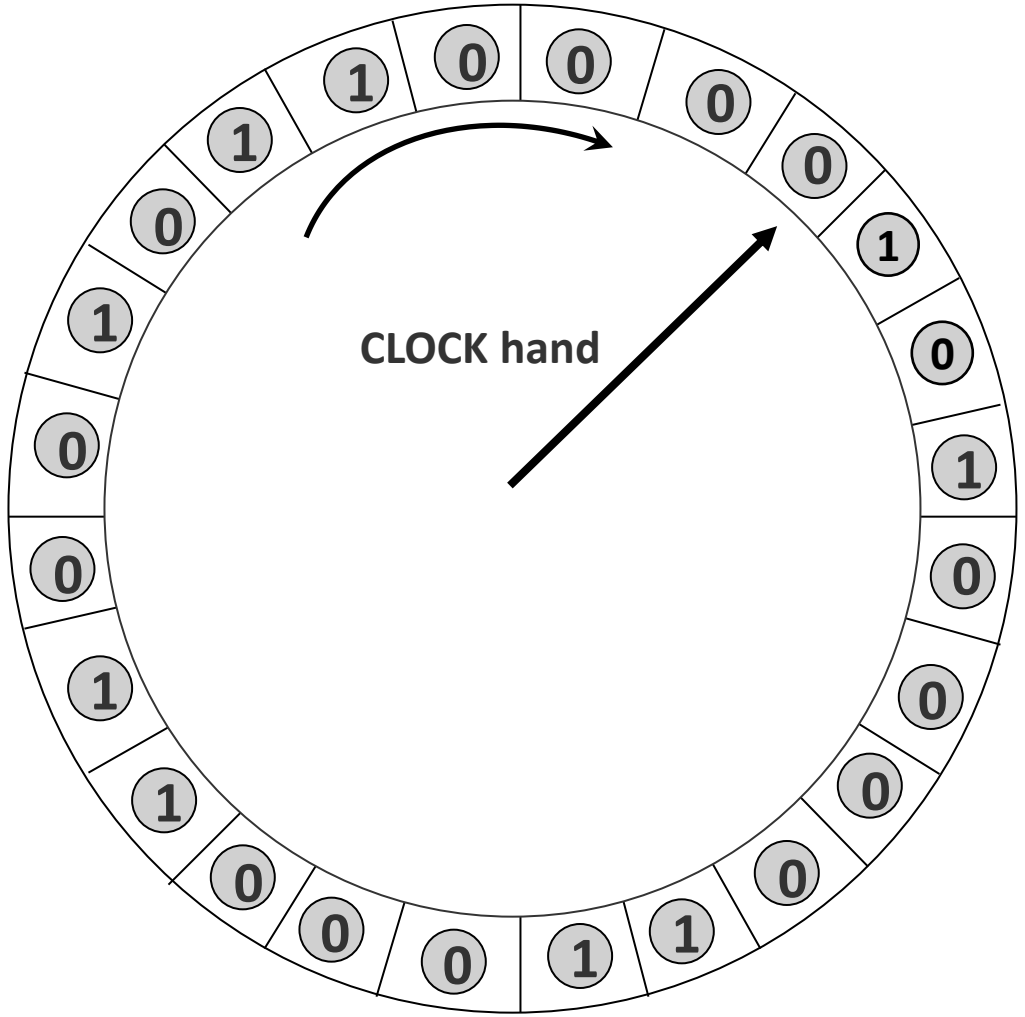
Least Recently Used (LRU) policy

- 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

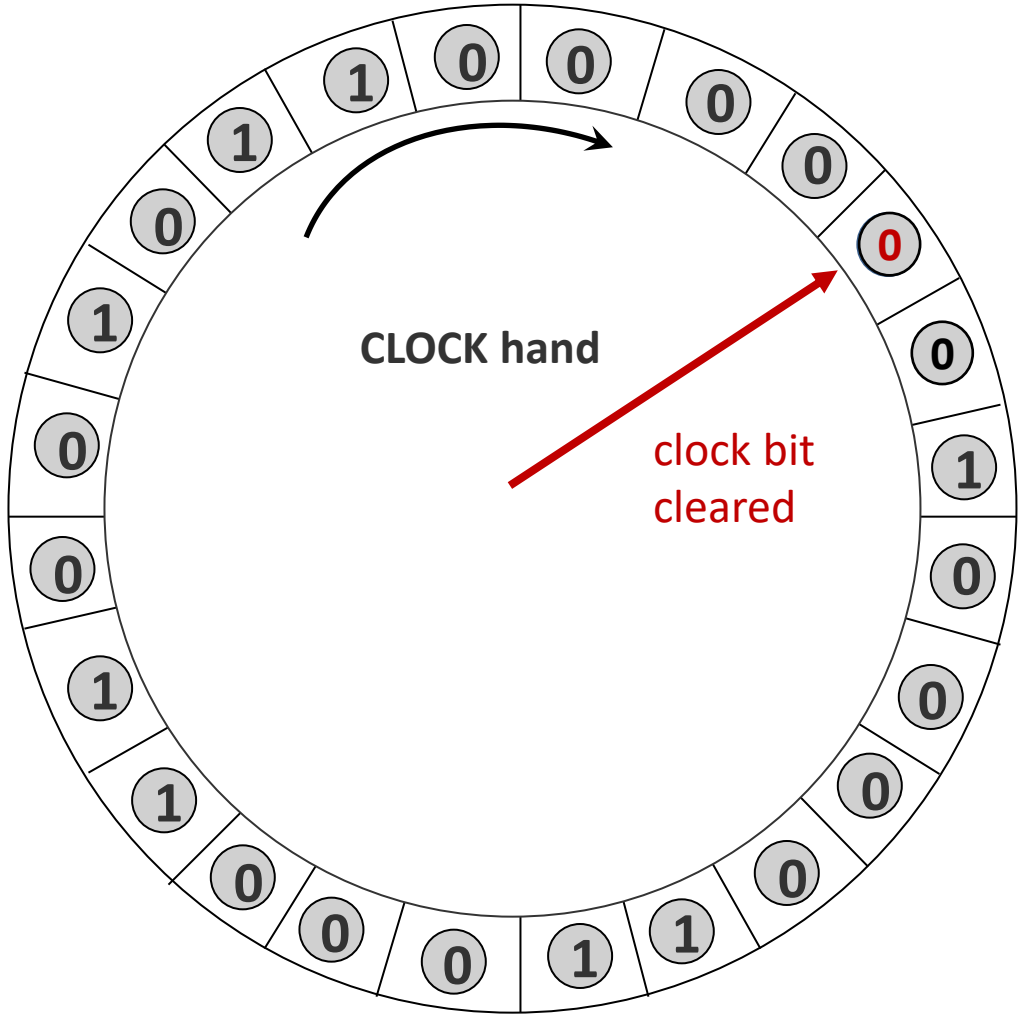
Clock policy

- 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



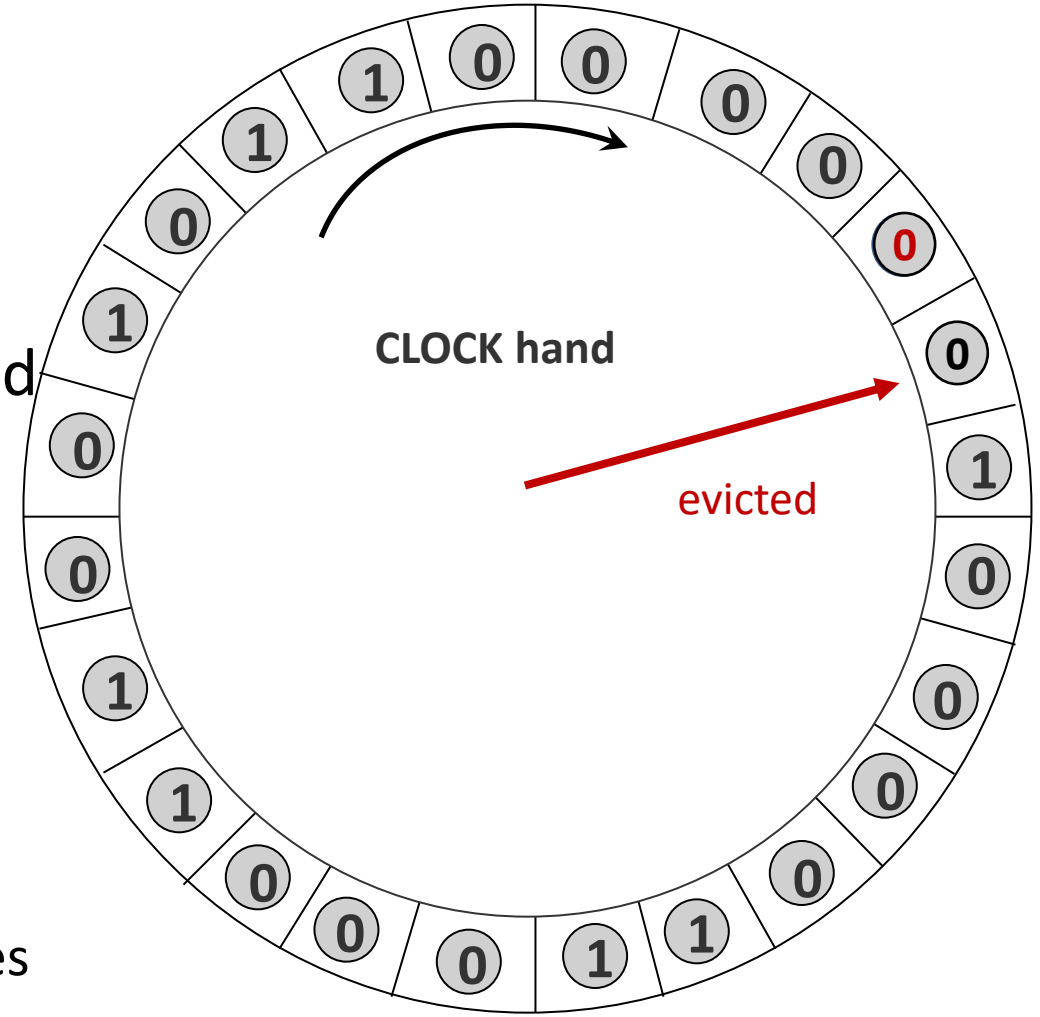
Clock policy

- 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



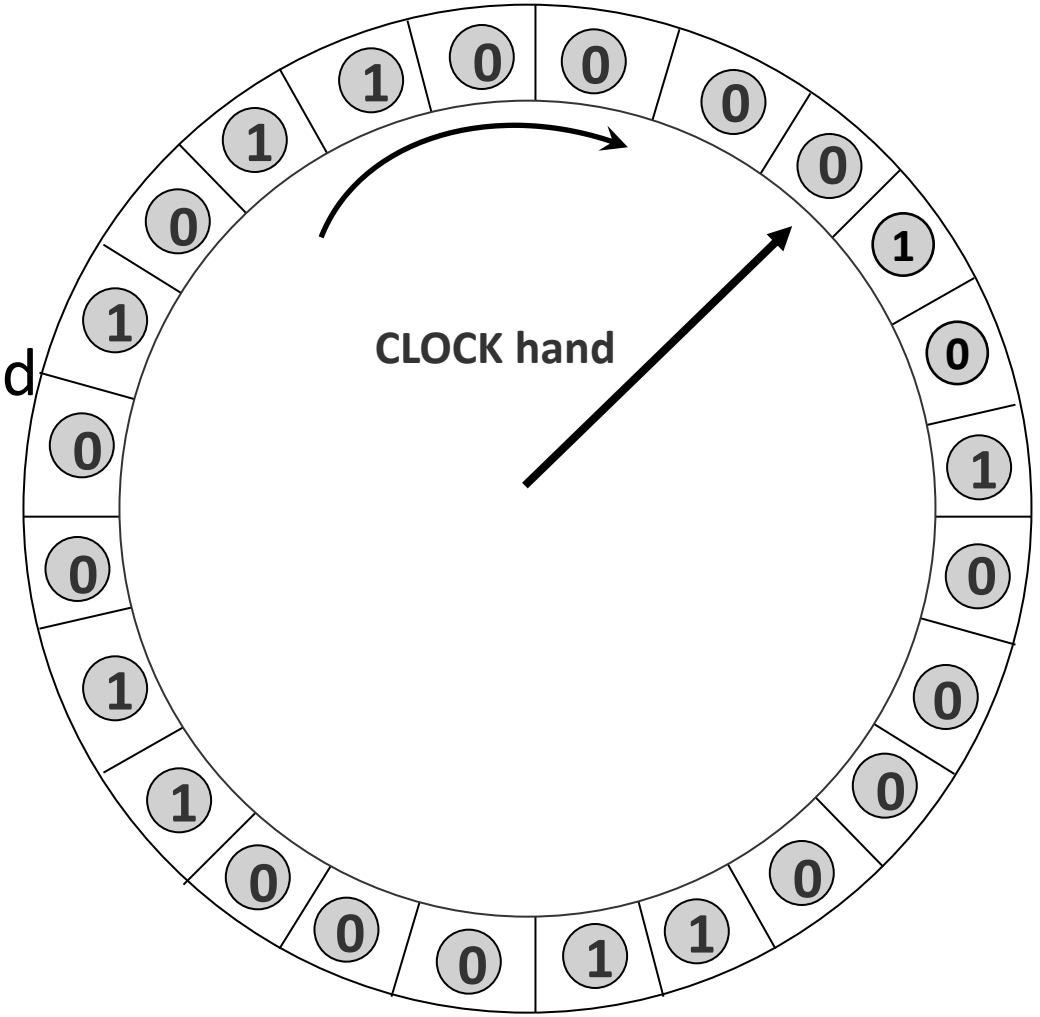
Clock policy

- 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 may apply
      - E.g., cannot flush dirty (uncommitted data) without undo logging