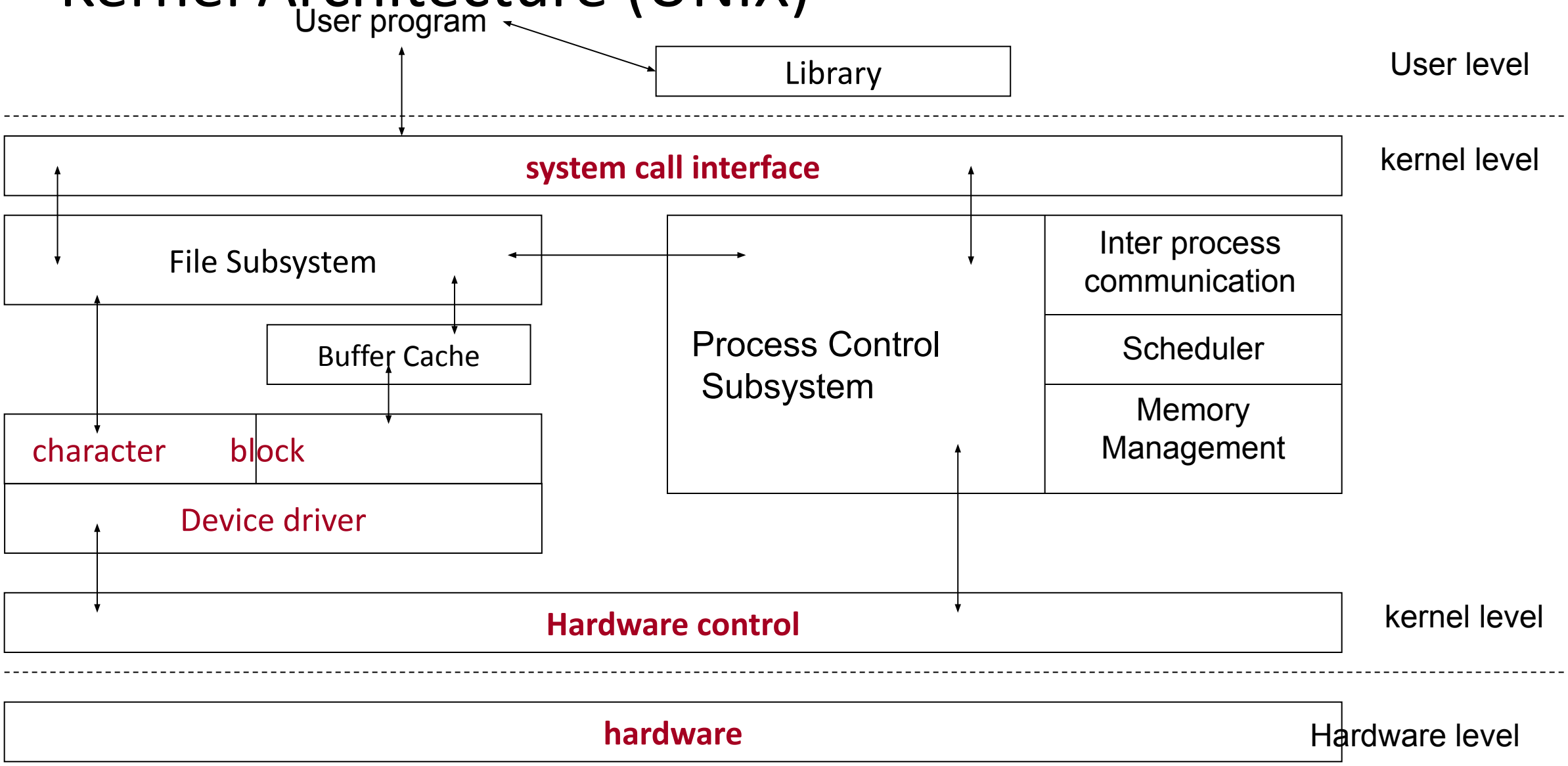
BUFFER CACHE

Kernel Architecture (UNIX)



Contents

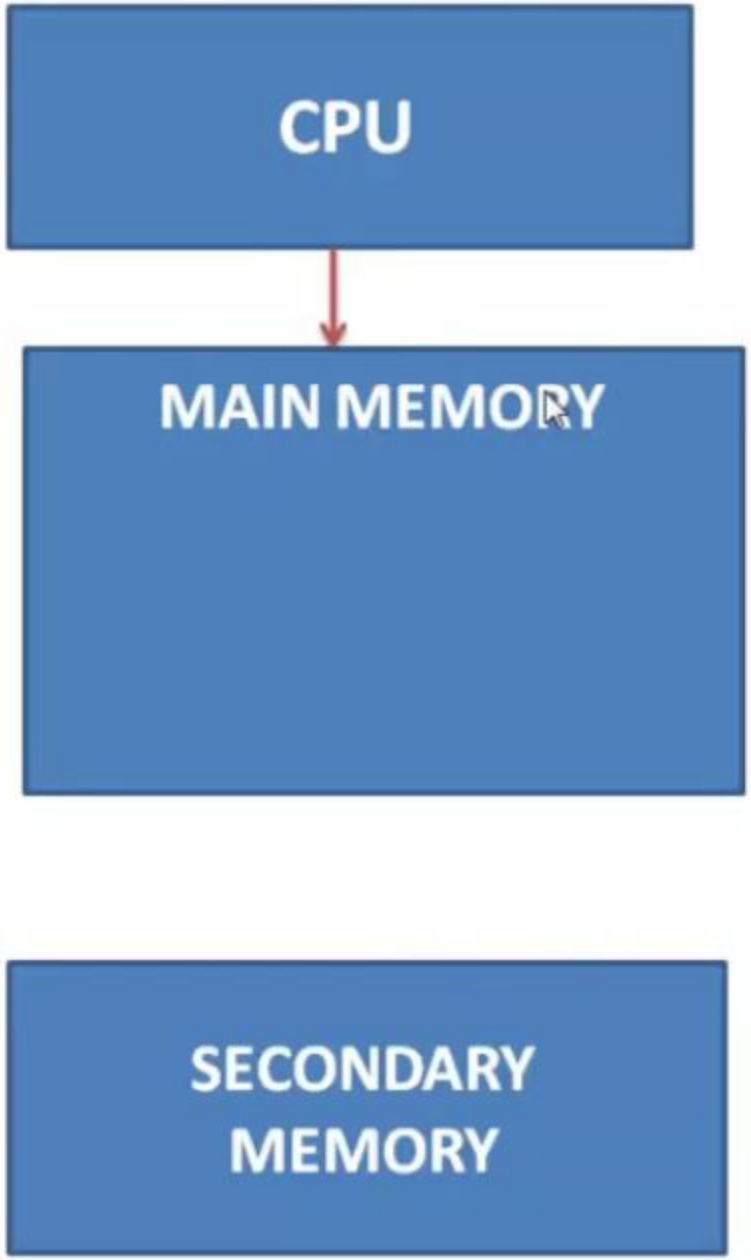
Buffer Cache

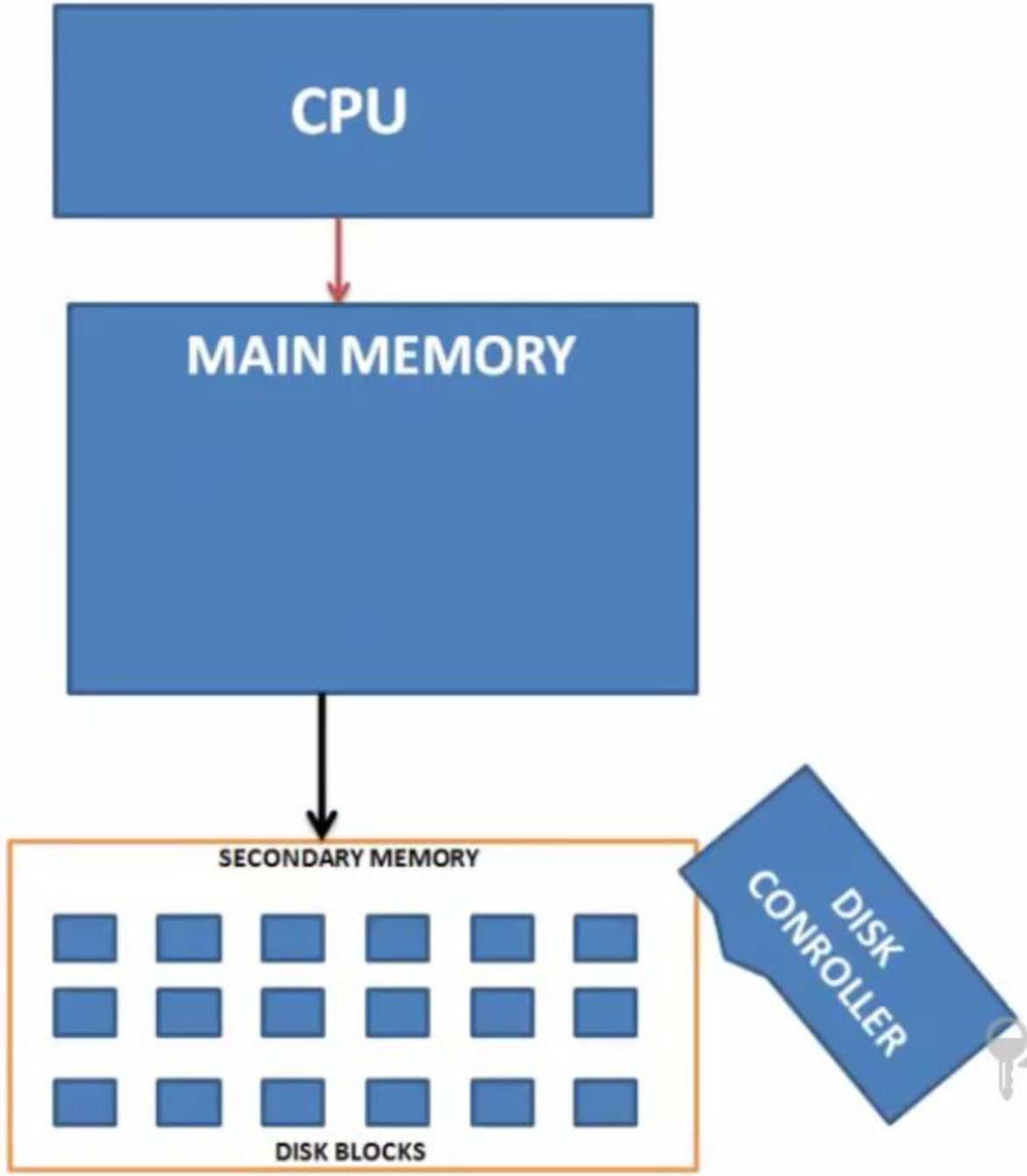
- The kernel could read and write directly to and from the disk for all the file system accesses but system response time and throughput would be poor because of the slow disk transfer rate.
- The kernel therefore attempts to minimize the frequency of disk access by keeping a pool of internal data buffers, called the buffer cache.

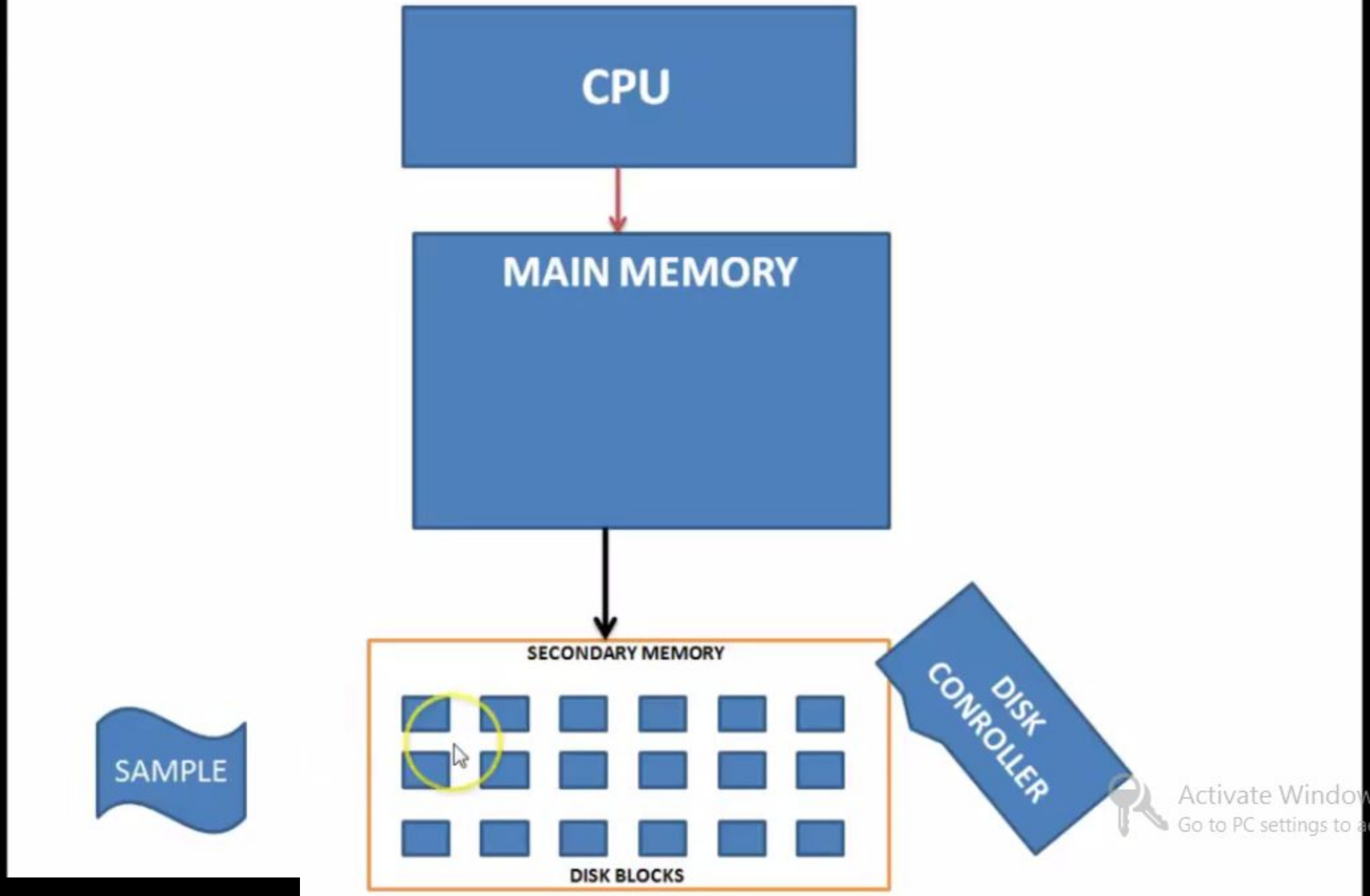
Need for Buffer Cache

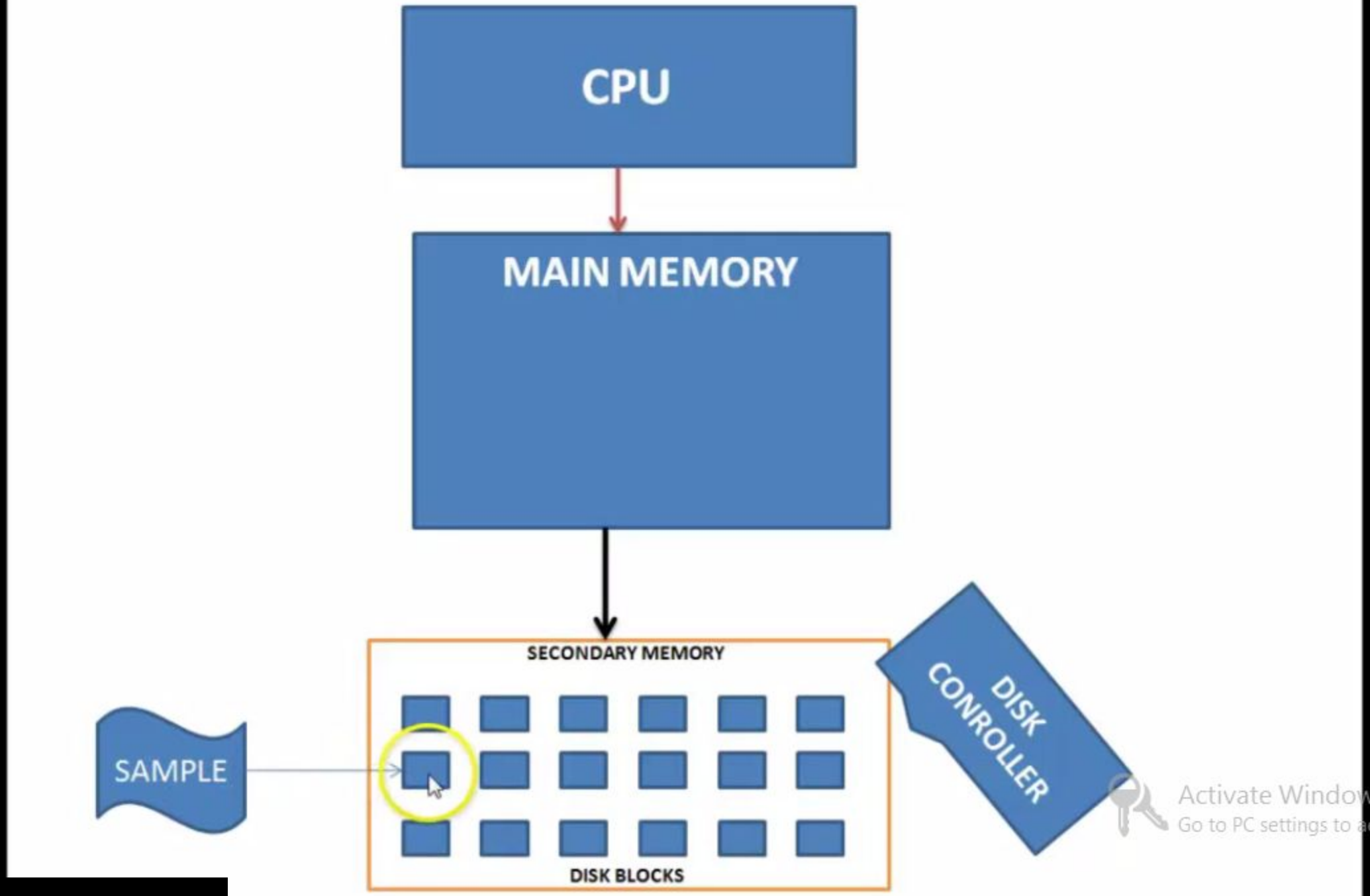
- Reading information from the disk
  - Very slow
- Reading same part of the disk several times within short periods
  - Reduces response time and throughput of the system
- Solution - **Buffer Cache**
  - Reading the information from disk - once
  - Keeping it in part of main memory - for subsequent access
  - Speeds up all except the first read
- Part of main memory used for this purpose - **Buffer Cache**

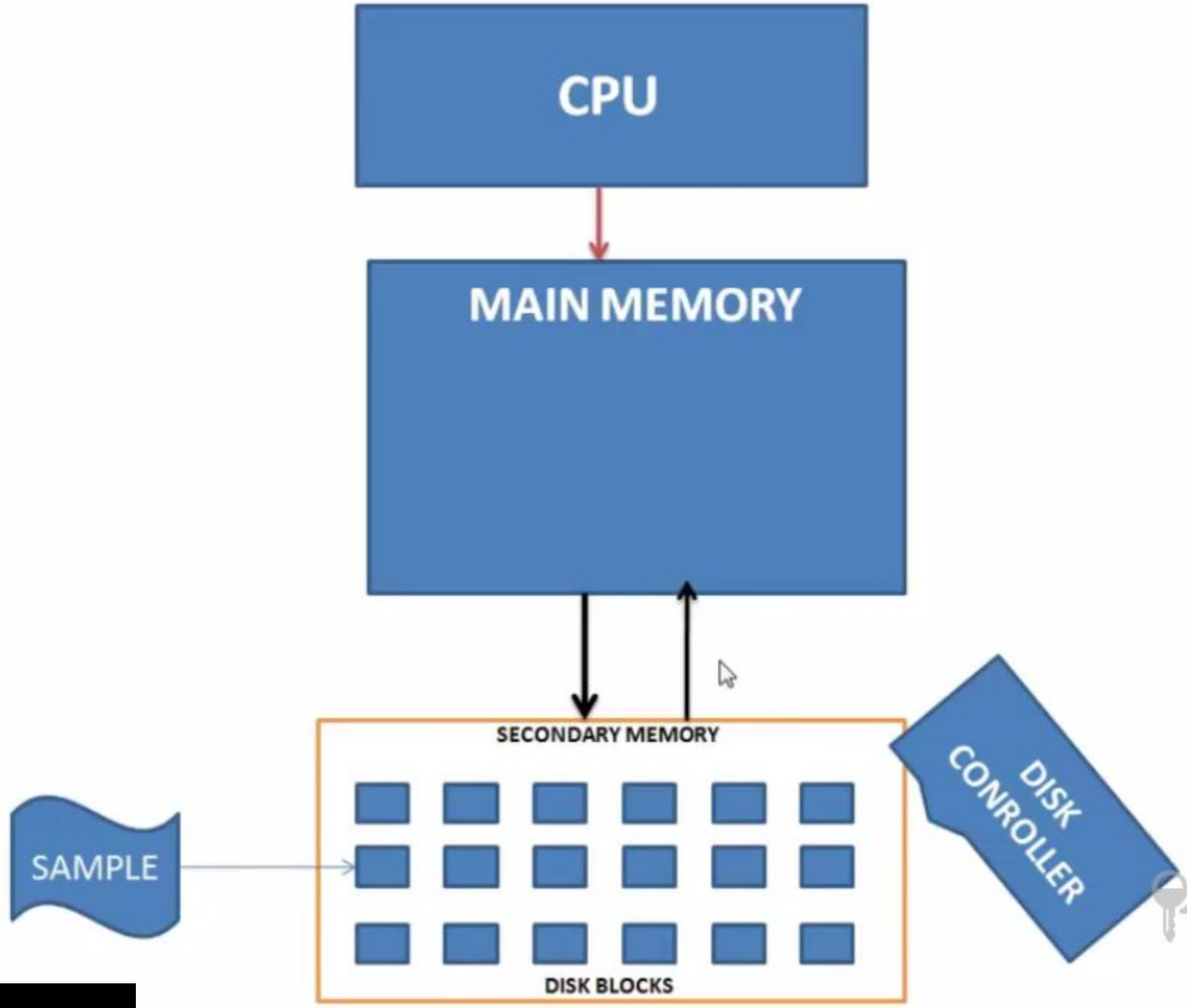
Need for Buffers

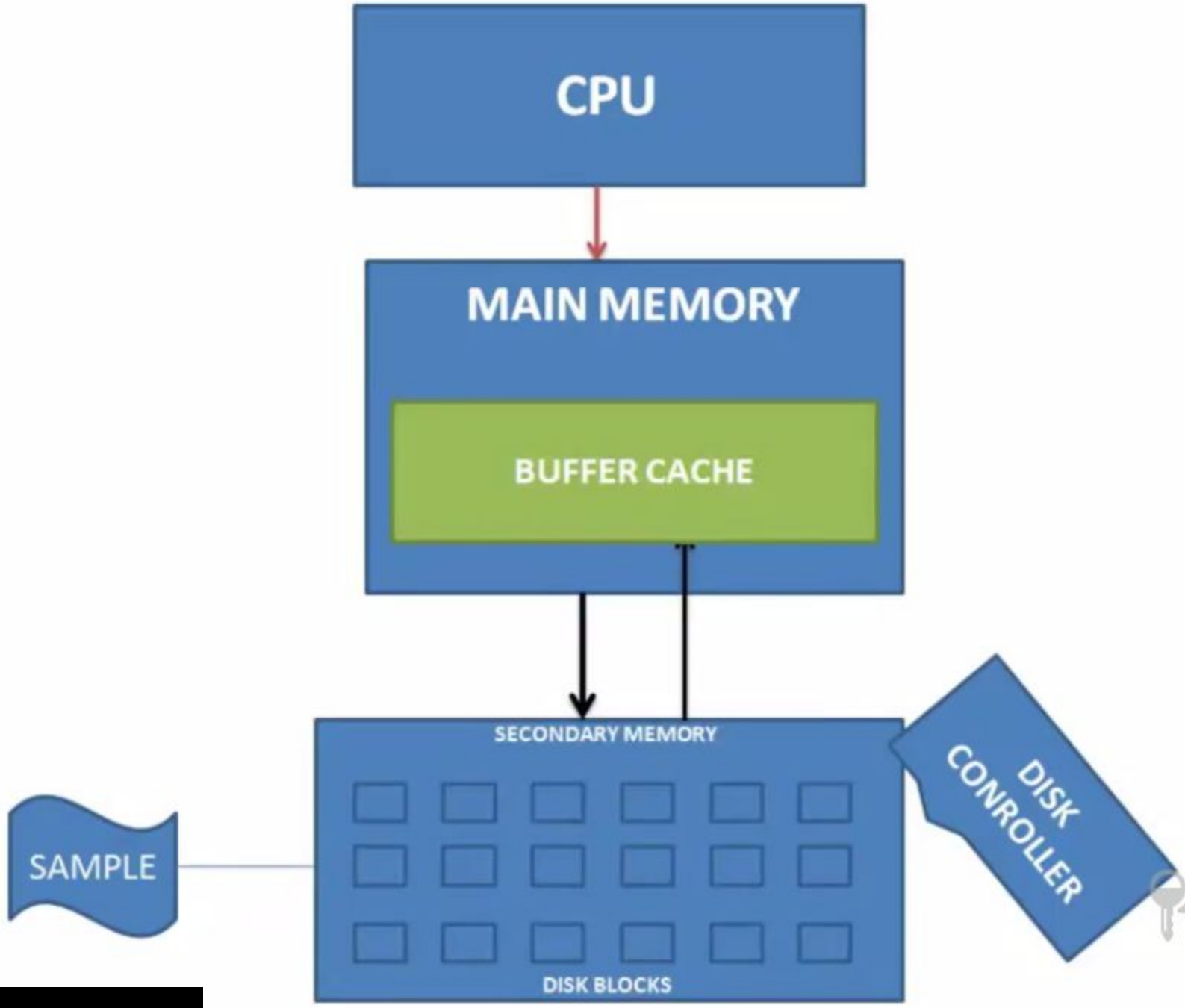












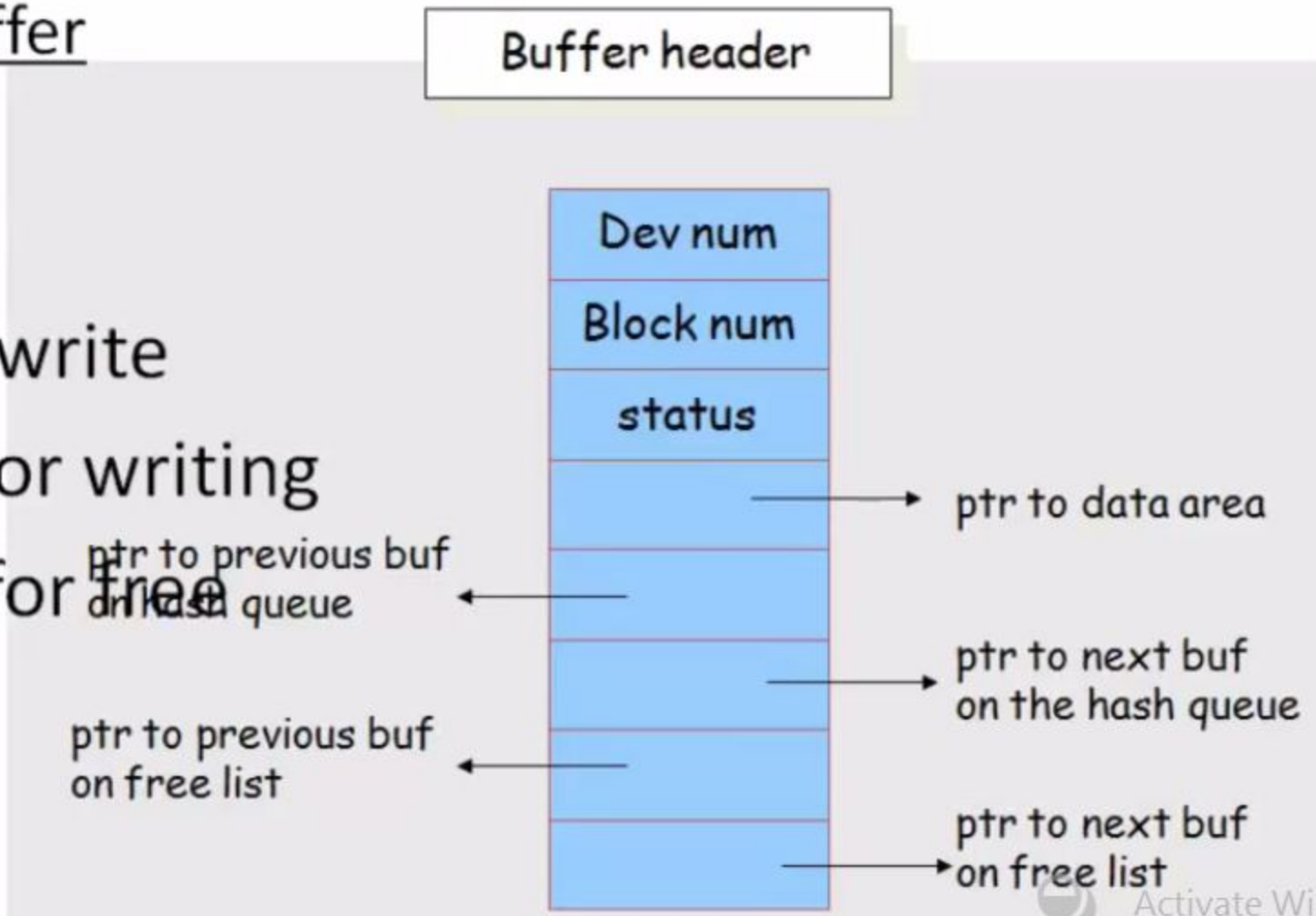
Buffer Headers

- During system initialization, the kernel allocates space for a number of buffers, configurable according to memory size and system performance constraints.
- A buffer consists of two parts:
  - A memory array that contains data from the disk.
  - A buffer header that identifies the buffer.

Buffer Headers

Status of buffer

- Locked
- Valid
- Delayed write
- Reading or writing
- Waiting for free



Buffer Headers

- **device num**
  - **logical file system number**
- **block num**
  - **block number of the data on disk**
- **status**
  - **The buffer is currently locked.**
  - **The buffer contains valid data.**
  - **delayed-write**
  - **The kernel is currently reading or writing the contents of the disk.**
  - **A process is currently waiting for the buffer to become free.**
- **kernel identifies the buffer content by examining device num and block num.**

Buffer Headers

```
struct buffer_head{
        /*First cache line: */
        struct buffer_head *b_next;          /*Hash queue list*/
        unsigned long b_blocknr;             /*block number*/
        unsigned long b_size;                /*block size*/
        kdev_t b_dev;                /*device(B_FREE = free)*/
        kdev_t b_rdev;               /*Read device*/
        unsigned long b_rsector;             /*Real Buffer location on disk*/
        struct buffer_head *b_this_page;     /*circular list of buffers in one page*/
        unsigned long b_state;               /*buffer state bitmap(see above)*/
        struct buffer_head *b_next_free;
        unsigned int  b_count;               /*users using this block*/
        char *b_data;                /*pointer to data block(1024 bytes)*/
        unsigned int b_list;                 /*List that this buffer appears*/
        unsigned long b_flushtime;           /*Time when this(dirty) buffer should be written*/
        struct wait_queue *b_wait;
        struct buffer_head **b_pprev;        /*doubly linked list of hash-queue*/
        struct buffer_head *b_prev_free;     /* double linked list of buffers*/
        struct buffer_head *b_reqnext;       /*request queue*/

        /*I/O completion*/
        void (*b_end_io)(struct buffer_head *bh,  int uptodate);
        void *b_dev_id;
};
```

Buffer Headers

/* buffer head state bits*/

- #define BH_Uptodate 0 /*1 if the buffer contains valid data*/
- #define BH_Dirty 1 /*1 if the buffer is dirty*/
- #define BH_Lock 2 /*1 if the buffer is locked*/
- #define BH_Req 3 /*0 if the buffer has been invalidated*/
- #define BH_Protected 6 /*1 if the buffer is protected*/

Buffer Headers

- The data in a buffer corresponds to the data in a logical disk block on a file system.
- The kernel identifies the buffer contents by examining identifier fields in the buffer header.

Buffer Headers

- The buffer is the in-memory copy of the disk block;
  - The contents of the disk block map into the buffer, but the mapping is temporary until the kernel decides to map another disk block into the buffer. A disk block can never map into more than one buffer at a time.
  - If two buffers were to contain the data for one disk block, the kernel would not know which buffer contained the current data and could write incorrect data back

Buffer Headers

- The buffer headers contains a device number field and a block number field that specifies the file system and block number of the data on disk and uniquely identify the buffer.

Buffer Headers

- The device number is the logical file system number, not a physical device (disk) unit number.
- The buffer header also contains a pointer to a data array for the buffer, whose size must be at least as big as the size of a disk block.
- The buffer header contains a status field that summarizes the current status of the buffer.

Buffer Headers

- The status of a buffer is a combination of following conditions:
  - The buffer is currently locked. (lock, busy or free, unlock).
  - If the buffer contains valid data then
    - The kernel must write the buffer contents to disk before reassigning the buffer; if the buffer is marked as delayed write.

Buffer Headers

- If the kernel is currently reading or writing the contents of the buffer to disk
- A process is currently waiting for the buffer to become free

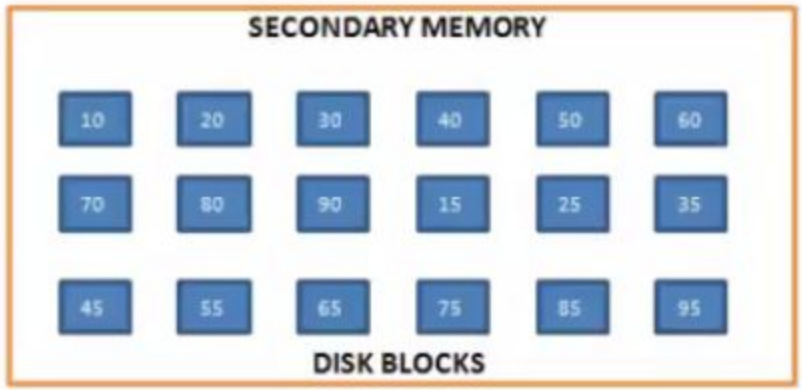
- The buffer header also contains two sets of pointers; used by the buffer allocation algorithms to maintain the overall structure of the buffer pool.

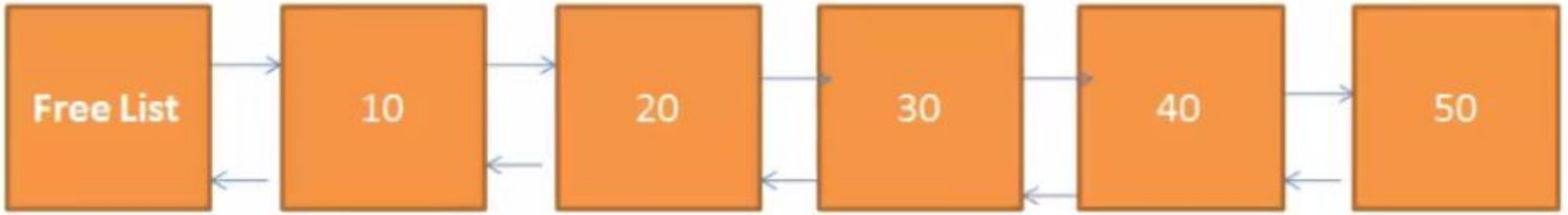
Structure of the buffer pool – **Free list**

- The kernel maintains a free list of buffers that preserves the least recently used order.
  - The free list is a doubly linked circular list of buffers with a dummy buffer header that marks its beginning and end.
- Every buffer is put on the free list when the system is booted.

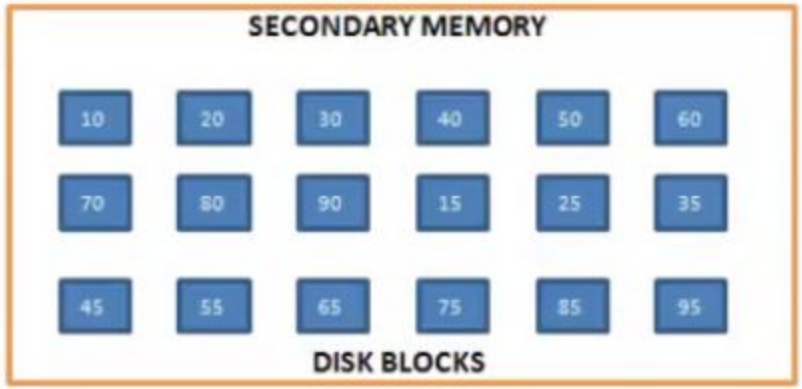
Structure of the Buffer Pool

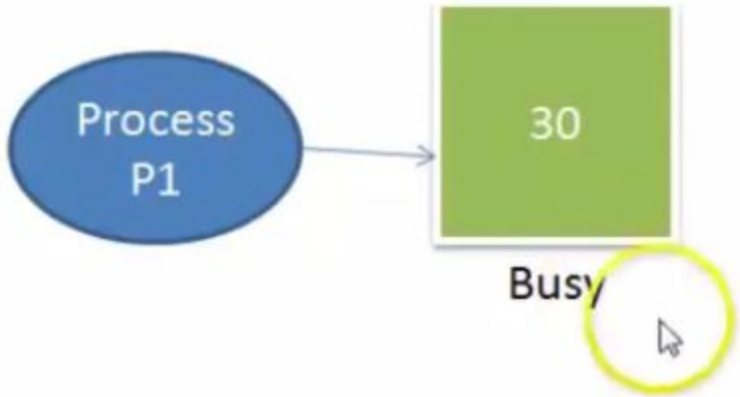
Free List

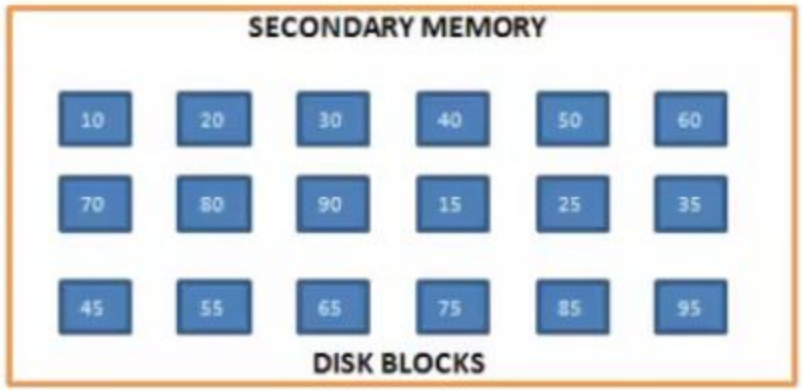




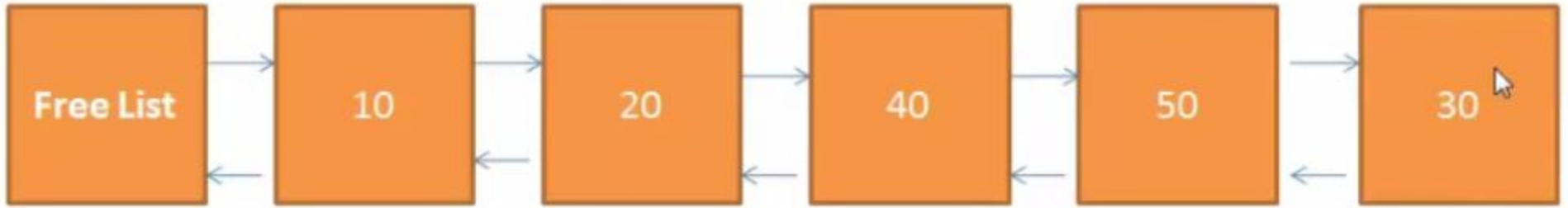
Process p1 requests for file in block number 30

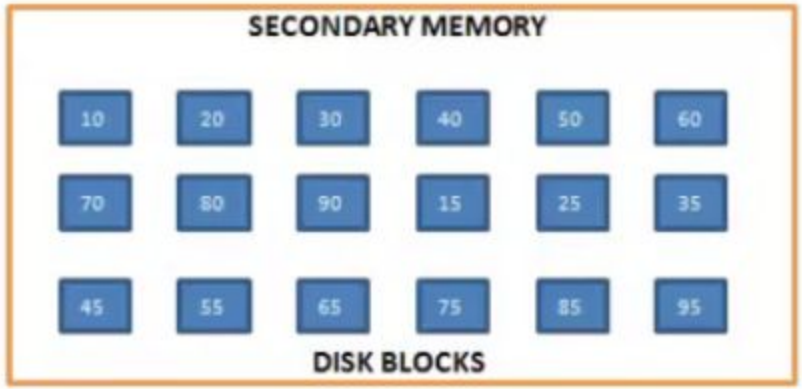


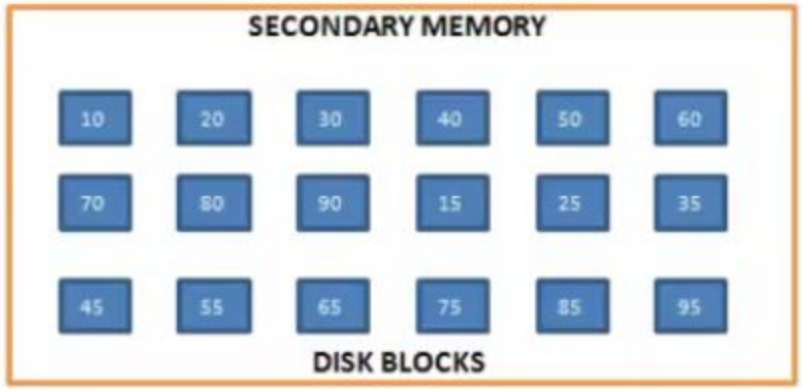


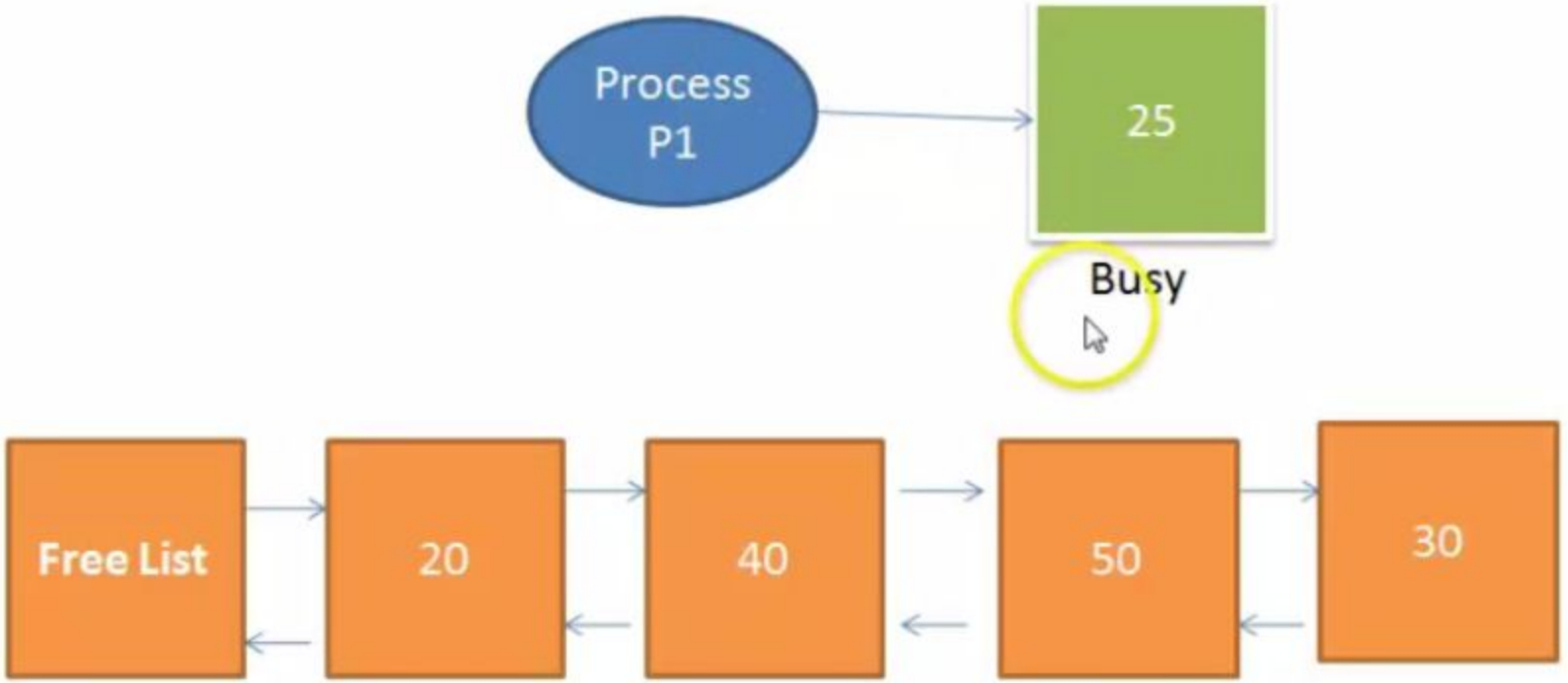


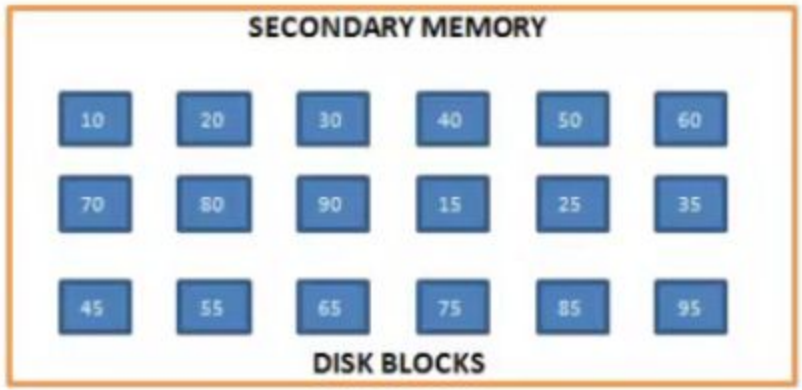
Process p1 requests for a file in block number 25











Structure of the buffer pool

- The kernel takes a buffer from the head of the free list (case 1) when it wants any free buffer,
- But it can take a buffer from the middle of the free list (case 2), if it identifies a particular block in the buffer pool.
- In both cases, it removes the buffer from the free list.

Structure of the buffer pool

- After the buffer becomes idle, when the kernel returns a buffer to the buffer pool, it usually attaches the buffer
  - to the tail of the free list,
  - occasionally to the head of the free list (for error cases),
  - but never to the middle.
- As the kernel removes buffers from the free list, a buffer with valid data moves closer and closer to head of the free list

Structure of the buffer pool – Hash Queue

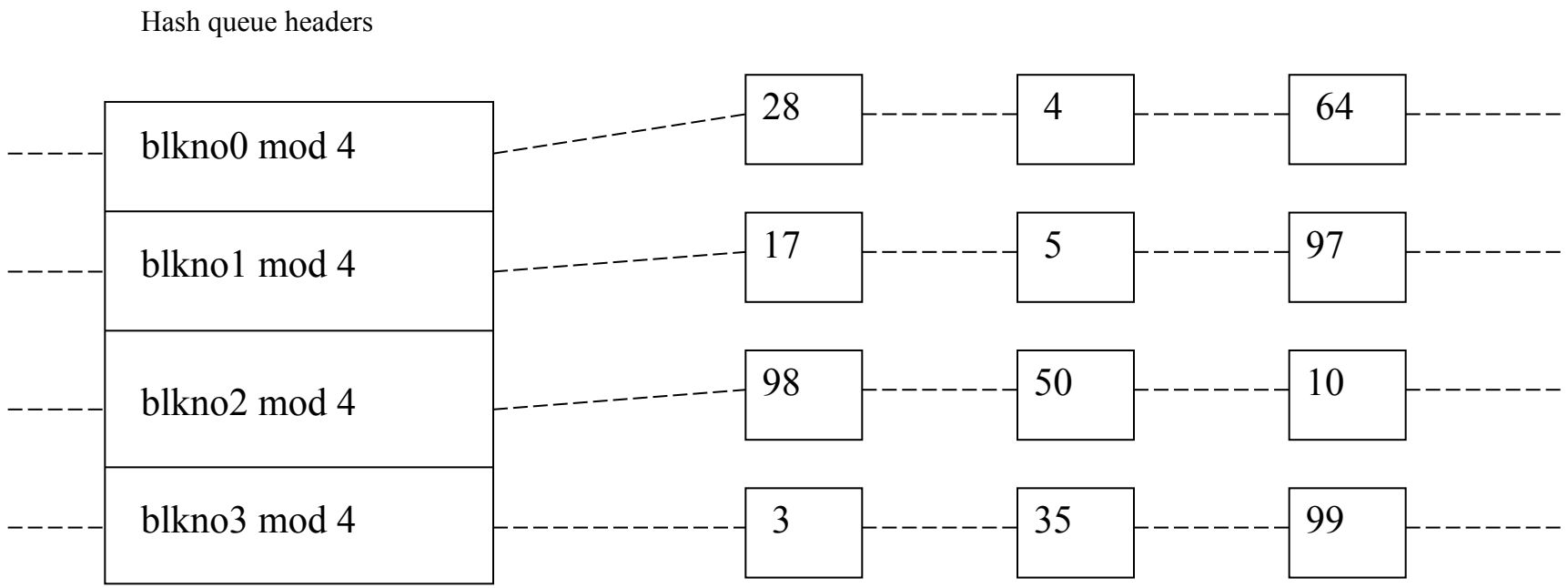
- Rather than search the entire buffer pool, kernel organizes the buffers into separate queues, hashed as a function of the device and block number.
- The kernel links the buffer on a hash queue into a circular, doubly linked list similar to structure of the free list.

Structure of the buffer pool– Hash Queue

- The number of buffers on a hash queue varies during the lifetime of the system.
- The kernel must use a hashing function that distributes the buffers uniformly across the set of hash queue.
  - Hash function must be simple so that performance does not suffer.
- System administrators configure the number of hash queues when generating the operating system.

Structures of the buffer pool– Hash Queue

- The below figure shows buffers on their hash queues.
- The headers of the hash queues are on the left side of the figure and the squares on each row are buffers on a hash queue



Buffers on the Hash Queues

Structure of the buffer pool

- Thus, squares marked 28, 4, 64 represent buffers on the hash queue for blk no 0 mod 4.
- Dotted lines between the buffers represent the forward and backward pointers for the hash queue.

Structure of the buffer pool– Hash Queue

- The fig., identifies blocks only by their blk no., and it uses a hash function dependent only on a blk no., however, implementation use the device no., too.
- Each buffer always exists on a hash queue, but there is no significance to its position on the queue.

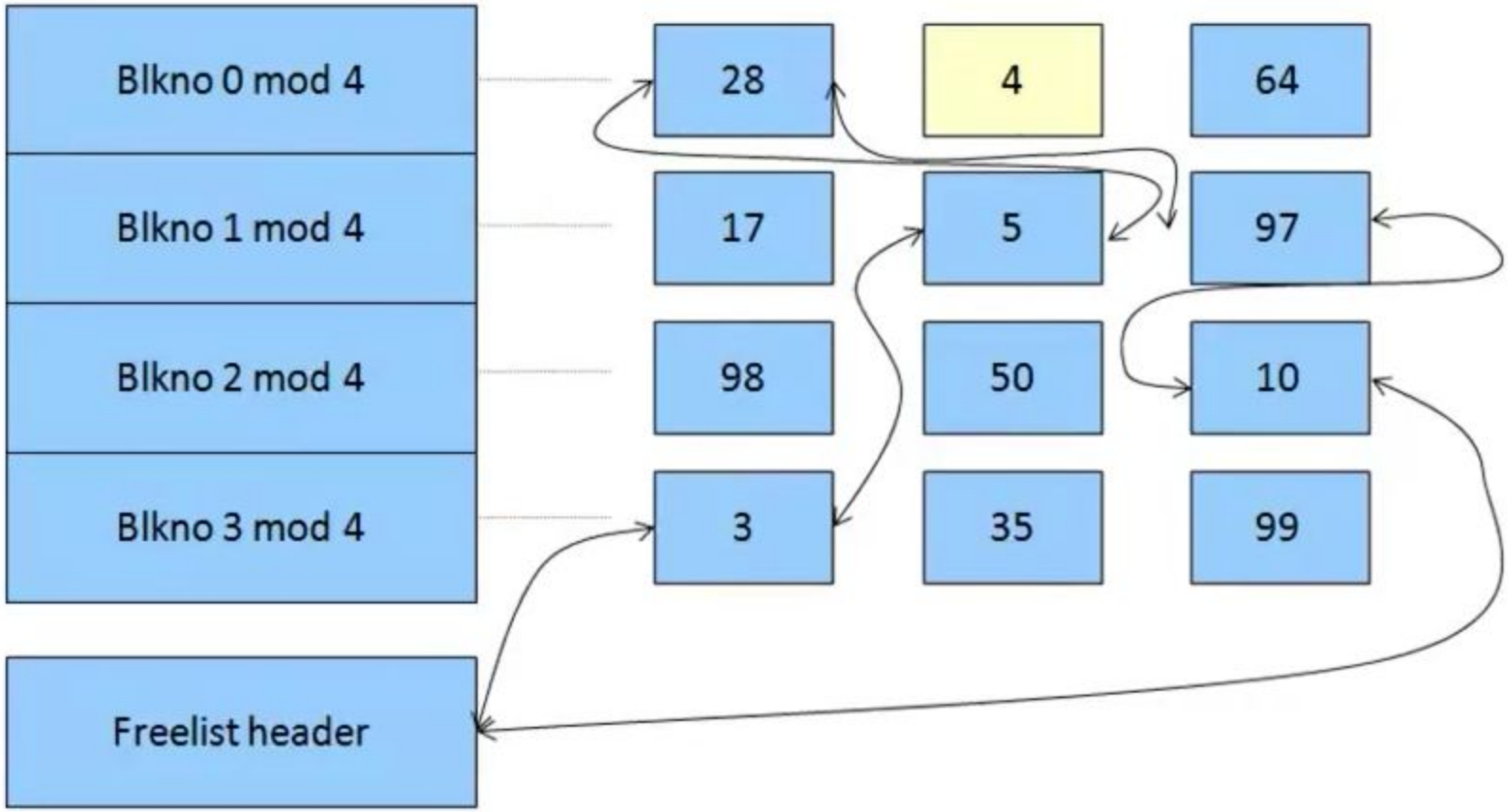
Structure of the buffer pool

- No two buffers may simultaneously contain the contents of the same disk block; therefore, every disk block in the buffer pool exists on one and only one hash queue and only once on that queue.
- However, a buffer may be on the free list as well if its status is free.
  - Because a buffer may be simultaneously on a hash queue and on the free list;

Structure of the buffer pool

- The kernel has two ways to find it;
  - It searches the hash queue if it is looking for a particular buffer, and
  - It removes a buffer from the free list if it is looking for any free buffer.
- A buffer is always on a hash queue, but it may or may not be on the free list.

Hash Queue


Blkno 0 mod 4
Blkno 1 mod 4
Blkno 2 mod 4
Blkno 3 mod 4
Freelist header
28
4
64
17
5
97
98
50
10
3
35
99

Structure of the Buffer Pool

- Free List
  - Doubly linked circular list
  - Every buffer is put on the free list when boot
  - When insert
    - Push the buffer in the head of the list (error case)
    - Push the buffer in the tail of the list (usually)
  - When take
    - Choose first element
- Hash Queue
  - Separate queues : each Doubly linked list
  - When insert
    - Hashed as a function device num, block num

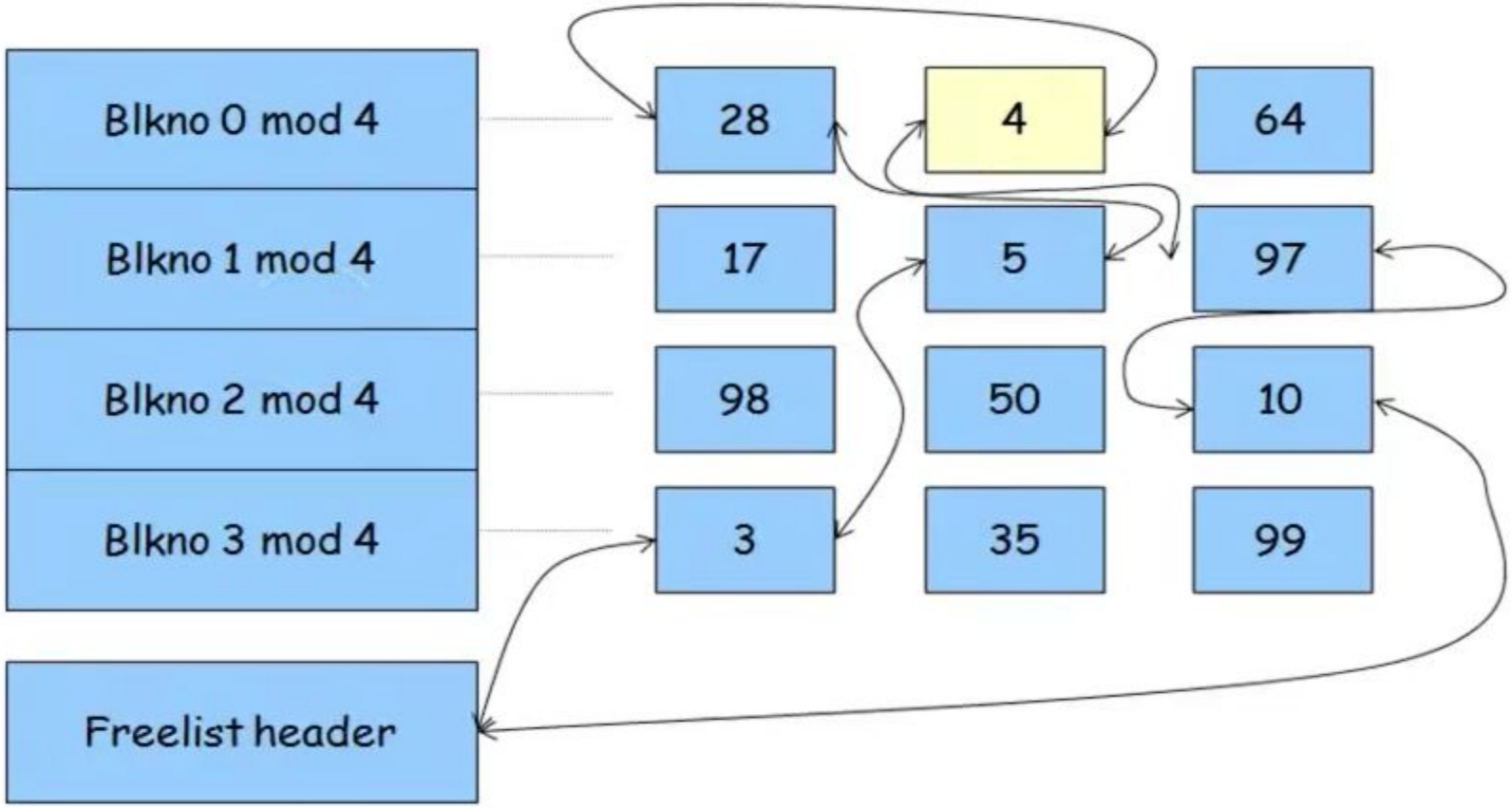
Scenarios for Retrieval of a Buffer

Scenarios for Retrieval of a Buffer

- The kernel finds the block on its hash queue, and its buffer is free.

1st Scenario

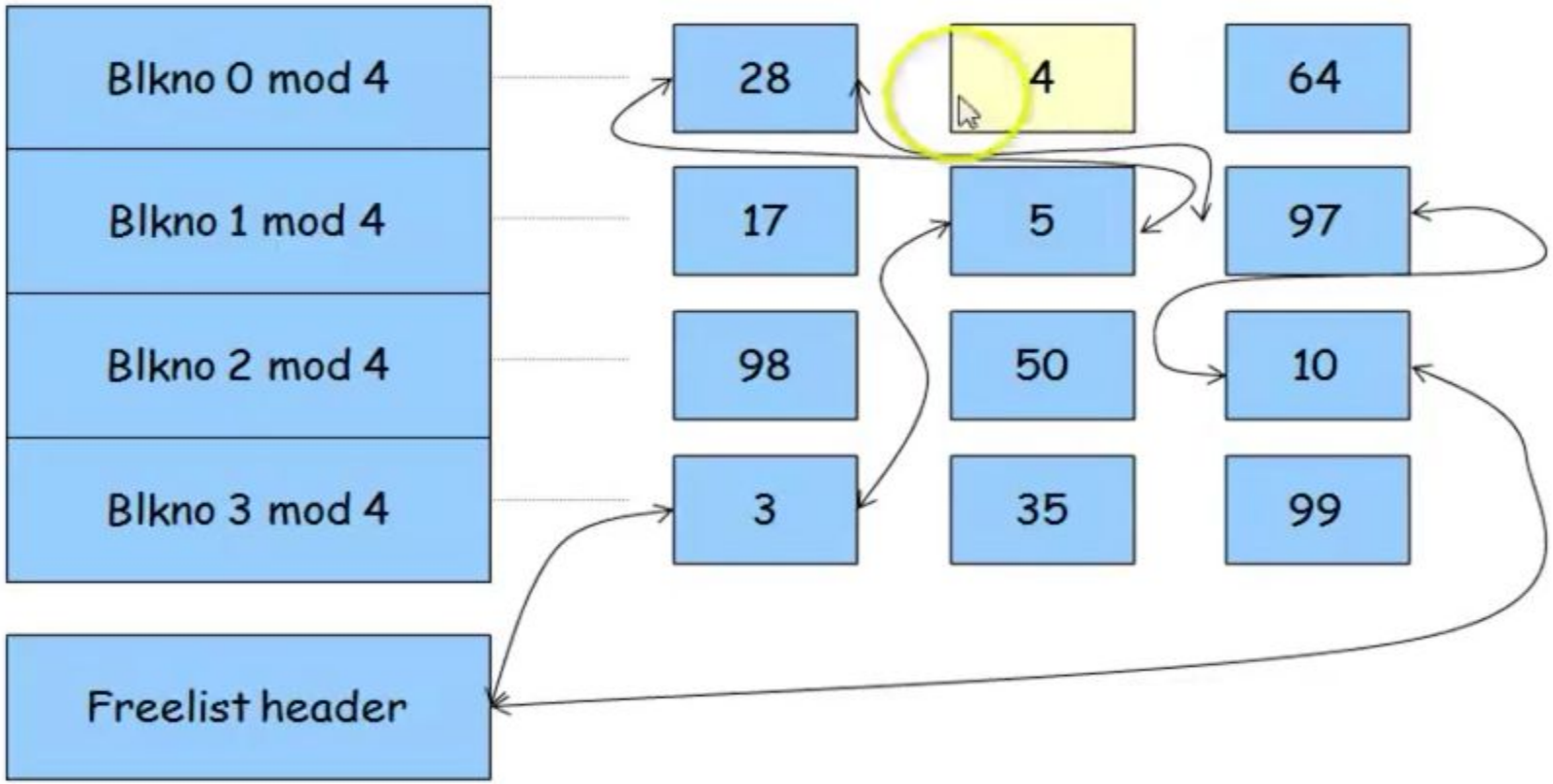
- Block is in hash queue, not busy
  - Choose that buffer



(a) Search for block 4 on first hash queue

1st Scenario

- After allocating



(b) Remove block 4 from free list

Scenario 1

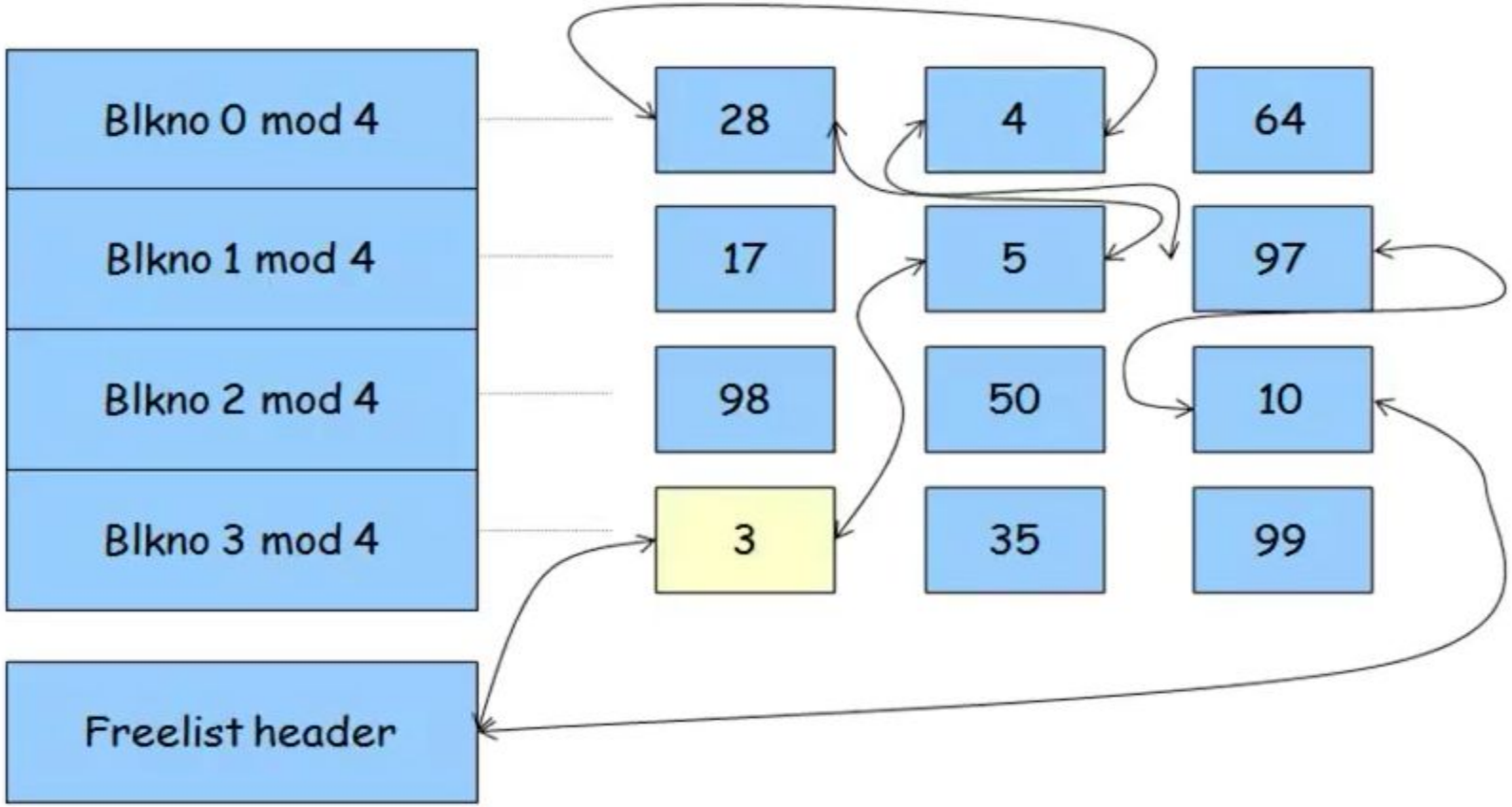
- The kernel removes the buffer(4) from the free list, leaves the buffer marked busy; no other process can access it and change its contents while it is busy, thus preserving the integrity of the data in the buffer.
- When the kernel finishes using the buffer, it releases the buffer.
- It wakes up processes that had fallen asleep because the buffer was busy and processes that had fallen asleep because no buffers remained on the free list.
- The kernel places the buffer at the end of the free list, unless an I/O error occurred or unless it specifically marked the buffer "old," as will be seen later.
- Else it places the buffer at the beginning of the free list. The buffer is now free for another process to claim it.
- The kernel raises the processor execution level to prevent disk interrupts

Scenarios for Retrieval of a Buffer

- The kernel finds the block on its hash queue, and its buffer is free.
- The kernel cannot find the block on the hash queue, so it allocates a buffer from the free list.

2nd Scenario

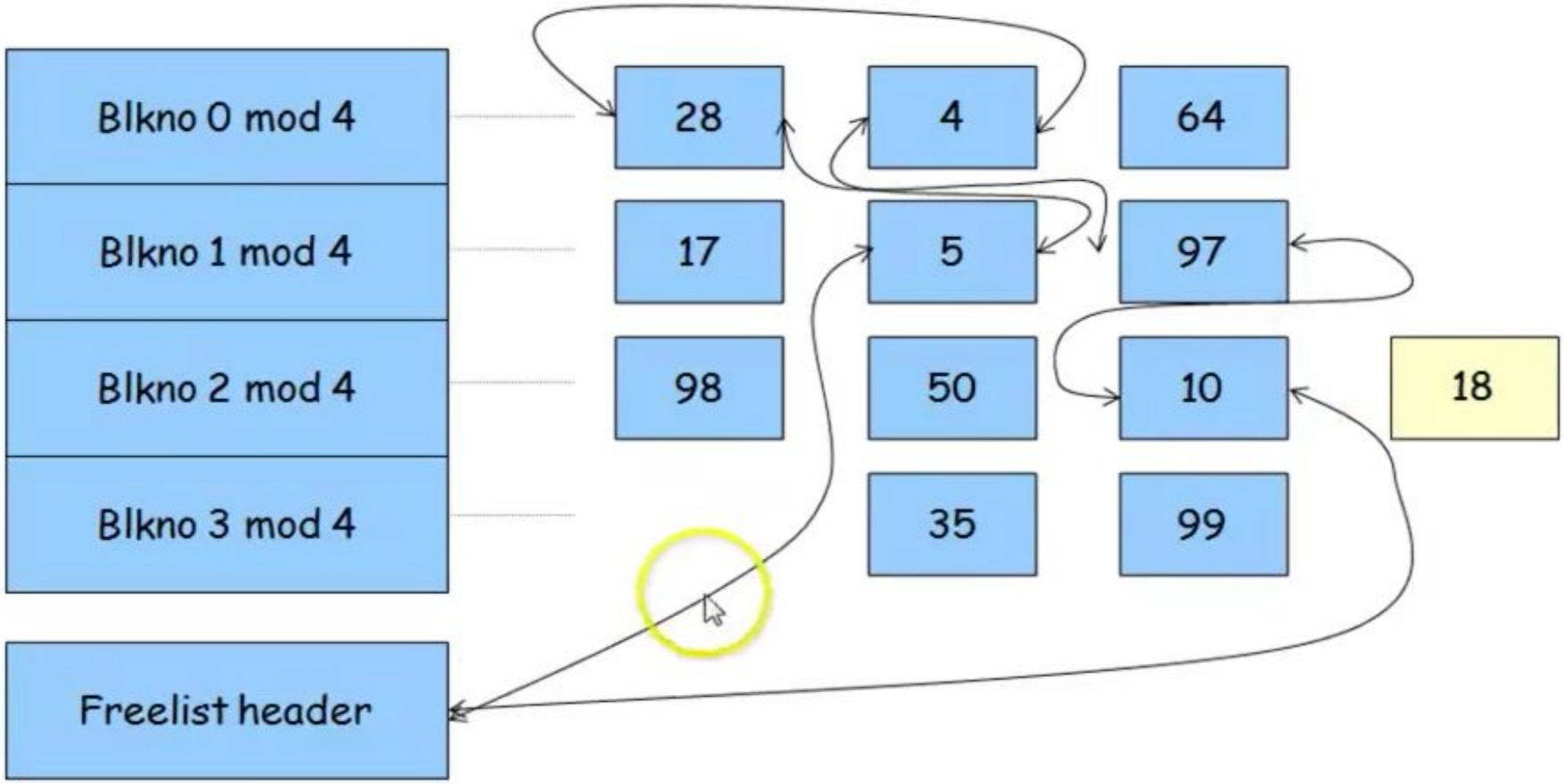
- Not in the hash queue and exist free buff.
  - Choose one buffer in front of free list



(a) Search for block 18 – Not in the cache

2nd Scenario

- After allocating



(b) Remove first block from free list, assign to 18

Scenario 2

- When kernel could not find the requested buffer, The kernel removes the first buffer from the free list instead; that buffer had been allocated to another disk block and is also on a hash queue.
- The kernel marks the buffer busy, removes it from the hash queue where it currently resides, reassigns the buffer header's device and block number to that of the disk block for which the process is searching, and places the buffer on the correct hash queue.
- Another process searching for the old disk block (3), will not find it in the pool and will have to allocate a new buffer for it from the free list.
- When the kernel finishes with the buffer, it releases it.

Scenarios for Retrieval of a Buffer

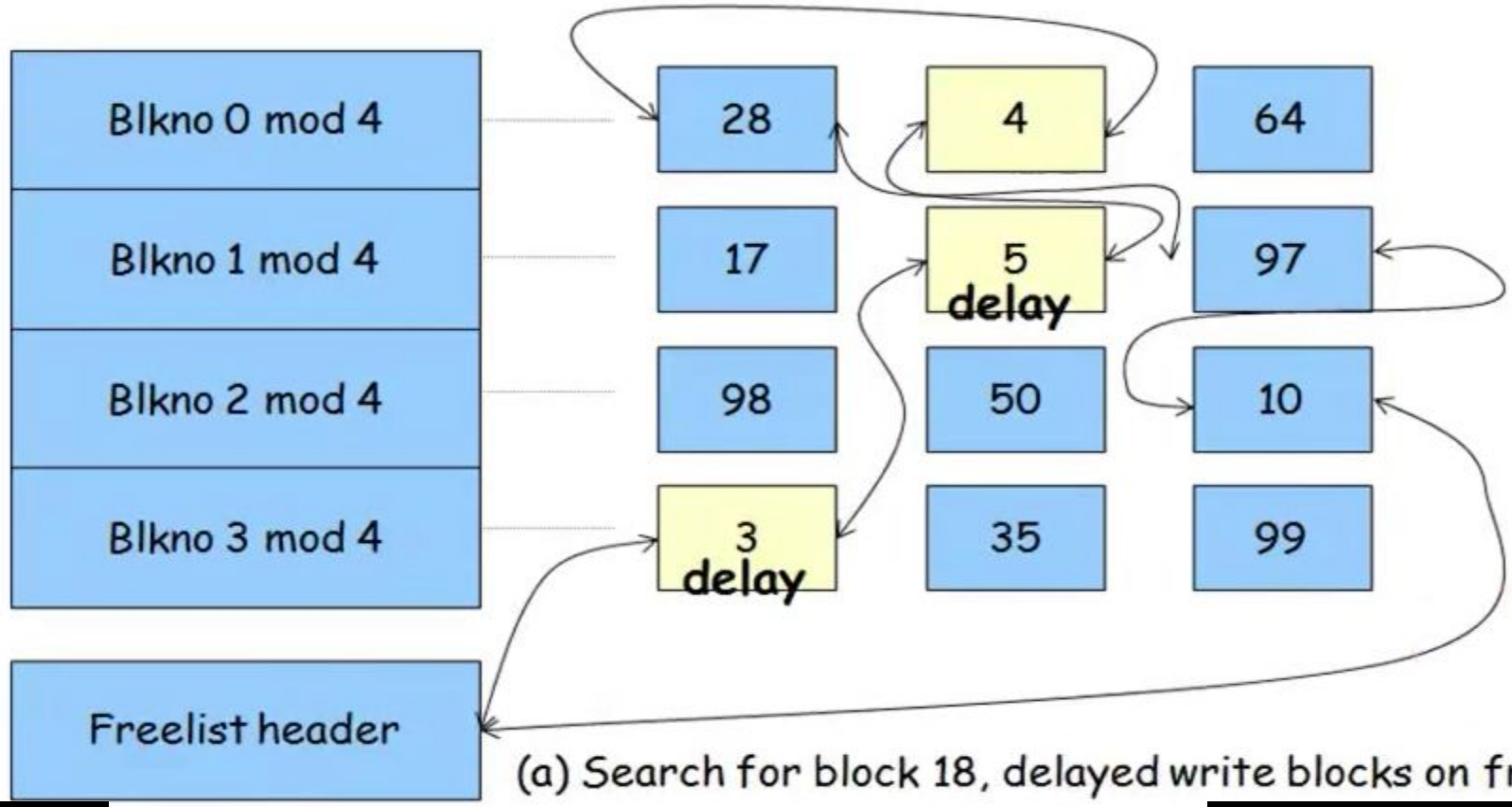
- The kernel finds the block on its hash queue, and its buffer is free.

- The kernel cannot find the block on the hash queue, so it allocates a buffer from the free list.

- The kernel cannot find the block on the hash queue and, in attempting to allocate a buffer from the free list (as in scenario 2) , finds a buffer on the free list that has been marked "delayed write." The kernel must write the "delayed write" buffer to disk and allocate another buffer.

3rd Scenario

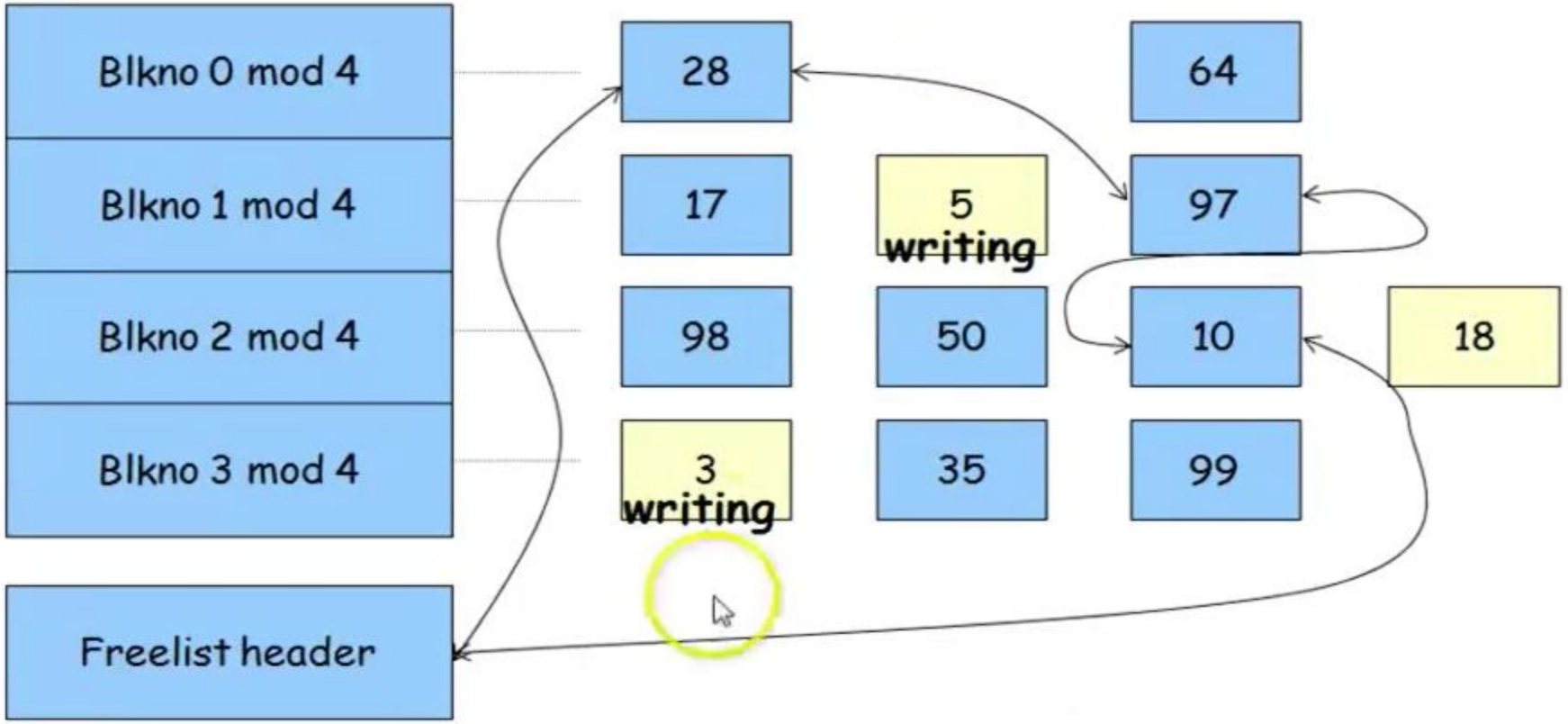
- Not in the hash queue and there exists delayed write buffer in the front of free list
  - Write delayed buffer async. and choose next



(a) Search for block 18, delayed write blocks on free list

3rd Scenario

- After allocating



(b) Writing block 3, 5, reassign 4 to 18

Scenario 3

- Requested buffer not on hash queue, so the kernel also has to allocate a buffer from the free list.
- However, it discovers that the buffer it removes from the free list has been marked for "delayed write," so it must write the contents of the buffer to disk before using the buffer.
- The kernel starts an asynchronous write to disk and removes the delayed write buffer from free list.
- And the kernel tries to allocate another(next free) buffer from the free list for the requesting process.
- When the asynchronous write completes for the delayed write buffer, the kernel releases the buffer and places it at the head of the free list

Scenarios for Retrieval of a Buffer

- The kernel finds the block on its hash queue, and its buffer is free.

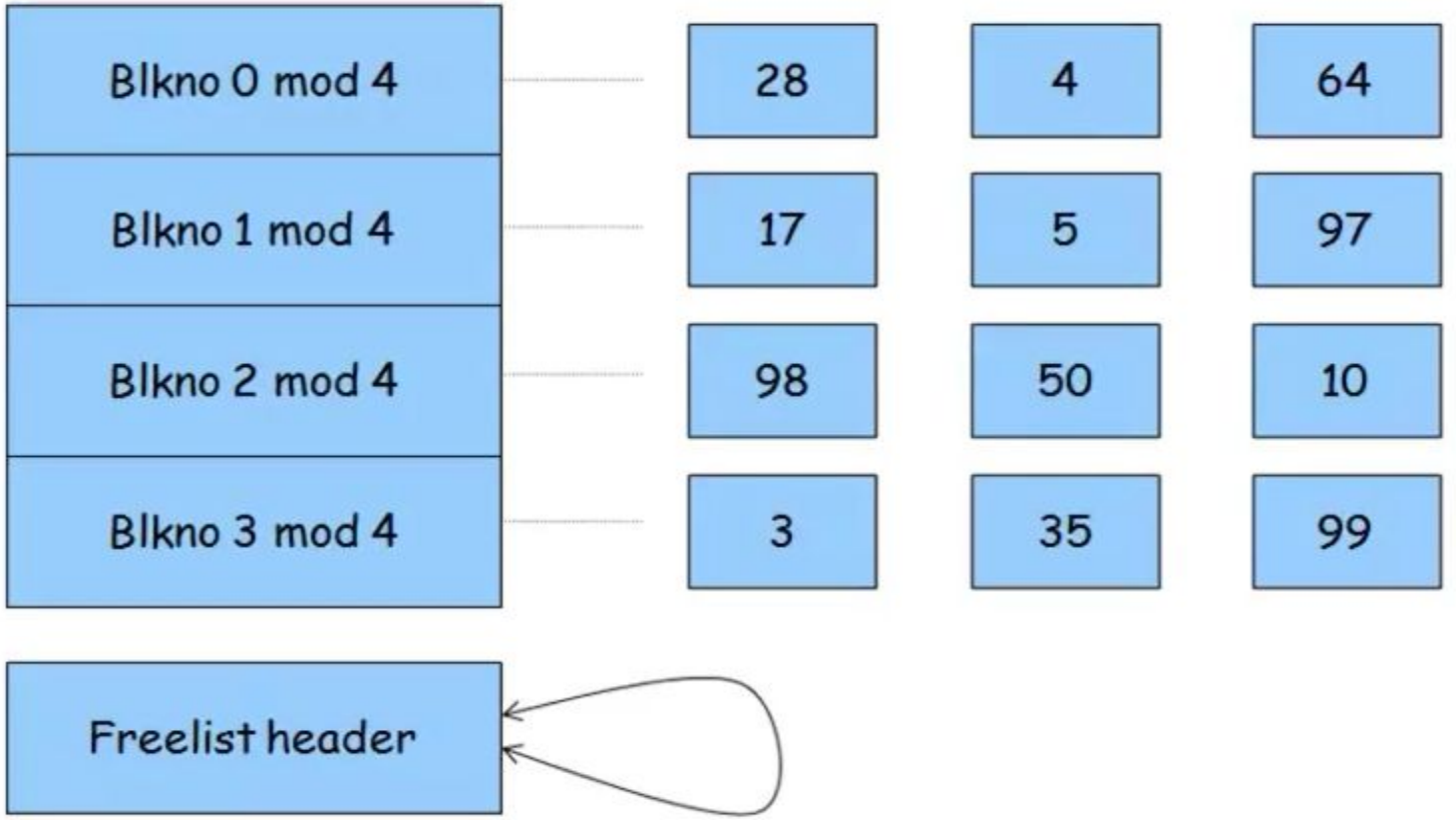
- The kernel cannot find the block on the hash queue, so it allocates a buffer from the free list.

- The kernel cannot find the block on the hash queue and, in attempting to allocate a buffer from the free list (as in scenario 2) , finds a buffer on the free list that has been marked "delayed write." The kernel must write the "delayed write" buffer to disk and allocate another buffer.

- The kernel cannot find the block on the hash queue, and the · free list of buffers is empty.

4th Scenario

- Not in the hash queue and no free buffer
  - Wait until any buffer free and re-do



(a) Search for block 18, empty free list

Scenario 4

- Here the requested buffer 18 it not on the hash queue, so the kernel attempts to allocate a new buffer from the free list, as in the second scenario.
- However, no buffers are available on the free list, so process A goes to sleep until another process frees a buffer.
- When the kernel schedules process A, it must search the hash queue again for the block.
- It cannot allocate a buffer immediately from the free list, because it is possible that several processes were waiting for a free buffer and that one of them allocated a newly freed buffer.

4th Scenario

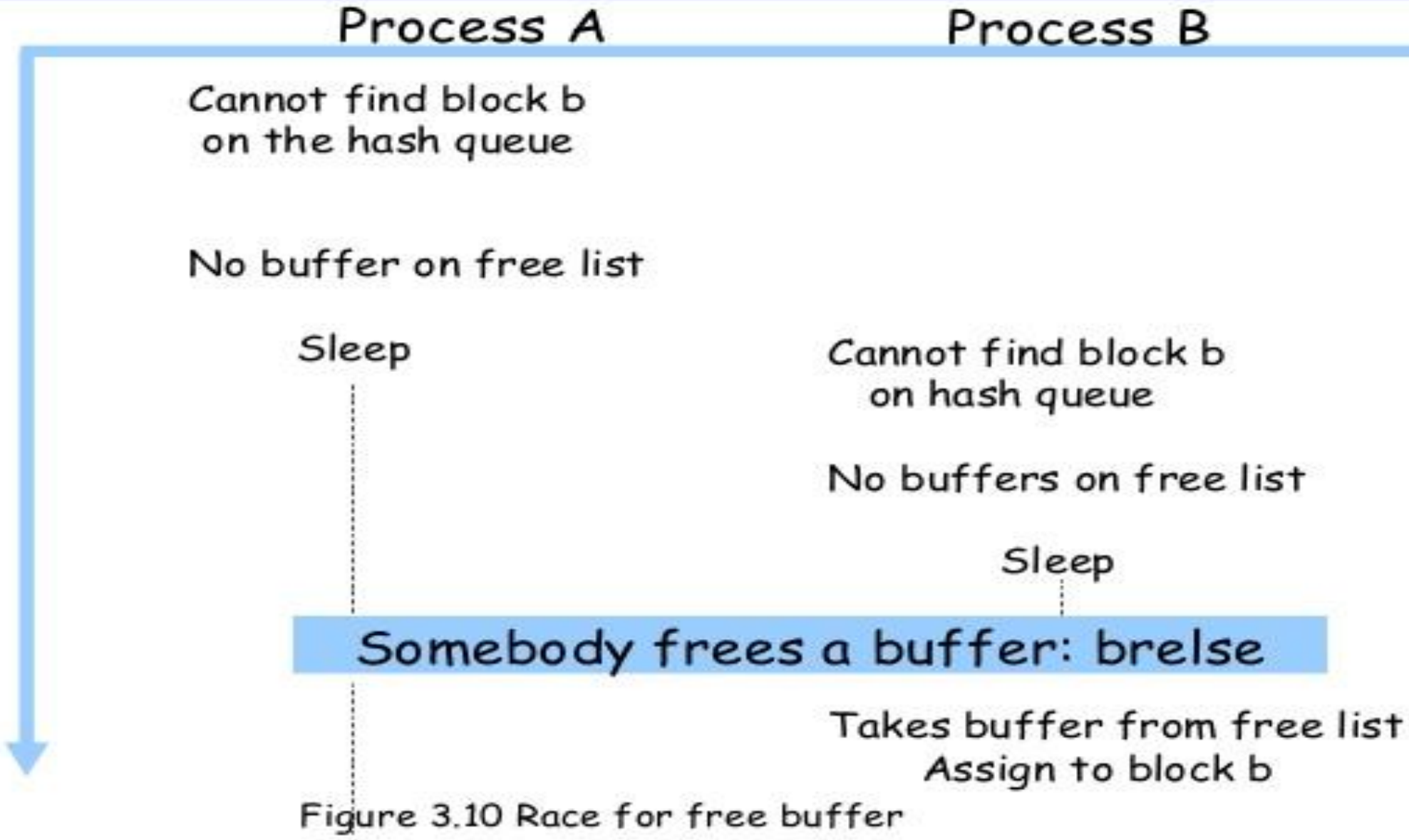


Figure 3.10 Race for free buffer

Scenarios for Retrieval of a Buffer

- The kernel finds the block on its hash queue, and its buffer is free.

- The kernel cannot find the block on the hash queue, so it allocates a buffer from the free list.

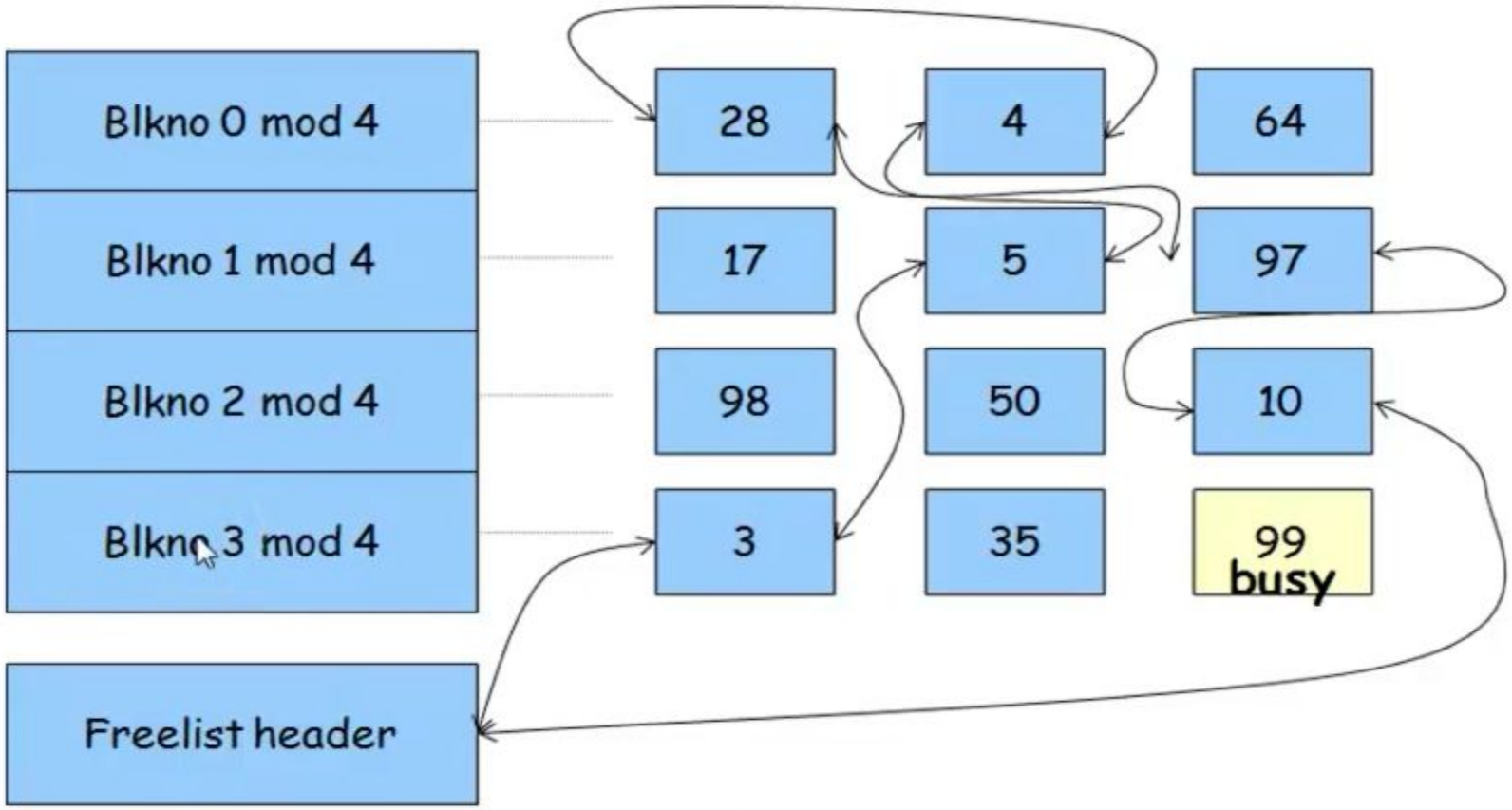
- The kernel cannot find the block on the hash queue and, in attempting to allocate a buffer from the free list (as in scenario 2) , finds a buffer on the free list that has been marked "delayed write." The kernel must write the "delayed write" buffer to disk and allocate another buffer.

- The kernel cannot find the block on the hash queue, and the · free list of buffers is empty.

- The kernel finds the block on the hash queue, but its buffer is currently busy.

5th Scenario

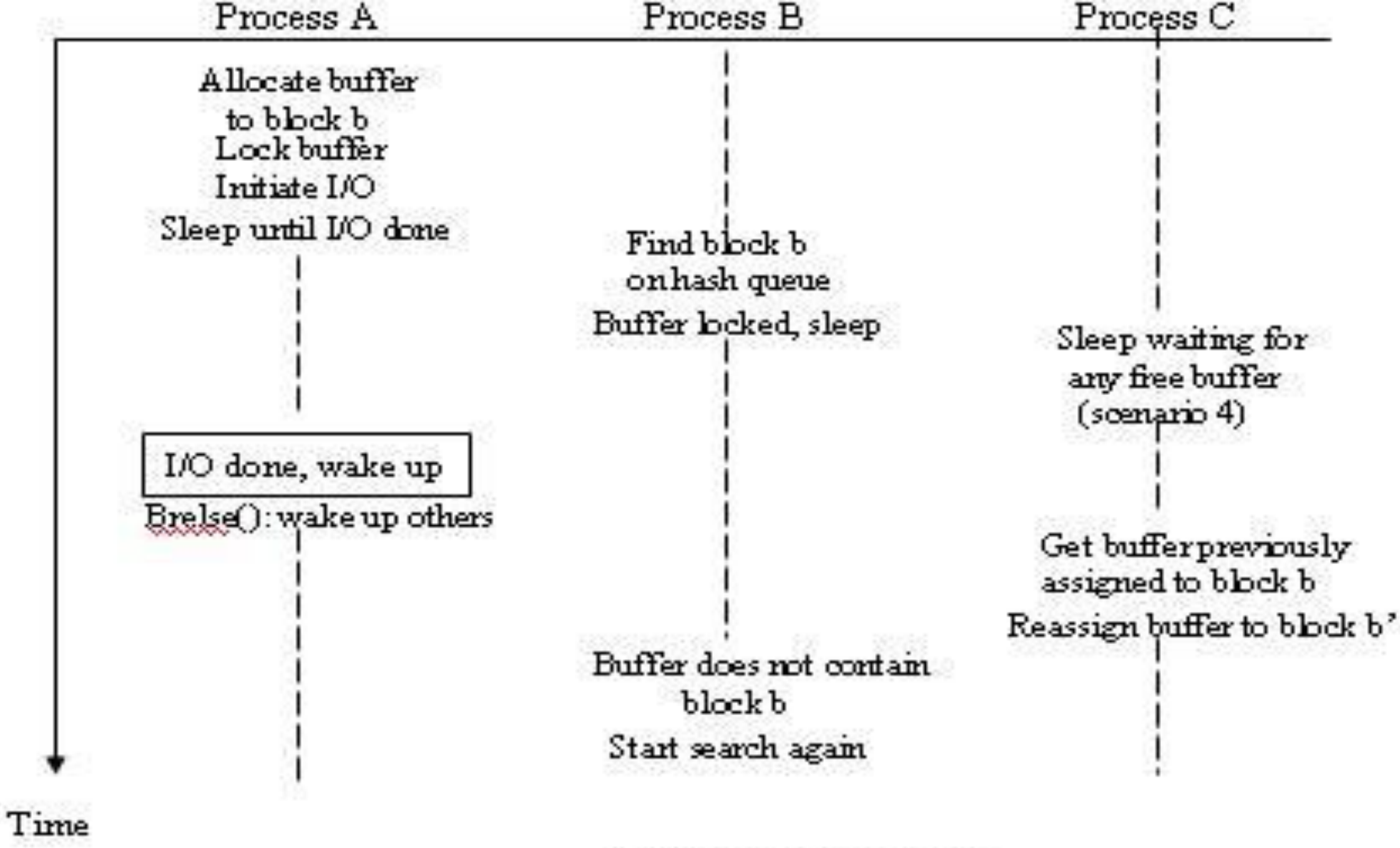
- Block is in hash queue, but busy
  - Wait until usable and re-do



(a) Search for block 99, block busy

Scenario 5

- Process B (going through scenario 5) will find the requested block, locked on the hash queue.
- Since it is illegal to use a locked buffer and it is illegal to allocate a second buffer for a disk block, process B marks the buffer "in demand" and then sleeps and waits for process A to release the buffer.
- Process A will eventually release the buffer and notice that the buffer is in demand.
- It awakens all processes sleeping on the event "the buffer becomes free,“ including process B.
- When the kernel again schedules process B, process B must verify that the buffer is free.



Race for a Locked Buffer

Two Major Condition

- Block Not Exists
  - Free List Empty → Sleep
  - Free list not empty but delayed write
  - Free list not empty and not delayed write
    -→ Use the Buffer.
- Block Exists
  - Locked -> Process Must Sleep
  - Unlocked-> Use the Buffer

Algorithm for buffer allocation

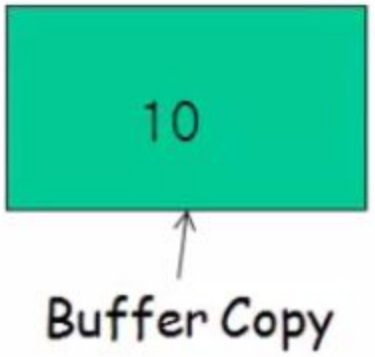
Algorithm: getblk()
Input: File System Number, Block Number
Output: Locked Buffer that can now be used for block

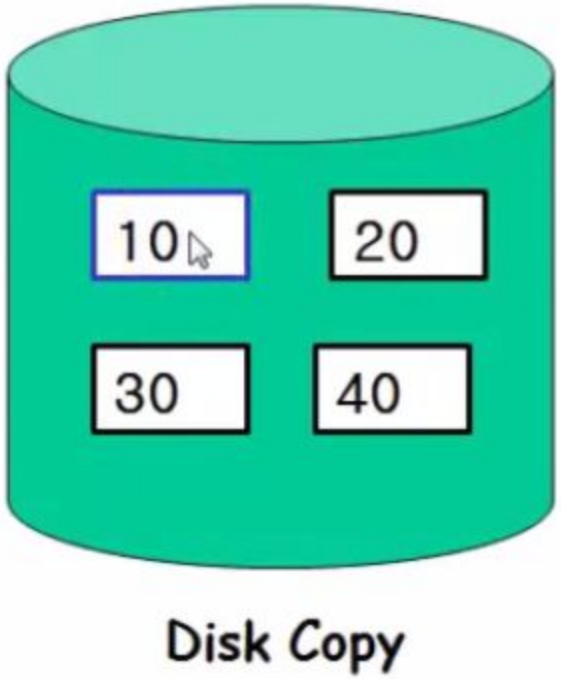
```
while(buffer not found)
{
    if(block in hash queue)
    {
        if(buffer busy)      /*scenario 5*/
        {
            sleep(event buffer becomes free);
            continues;              /*back to while*/
        }

        remove buffer from free list;
        mark buffer busy; /*scenario 1*/
        return buffer;
    }
```

Algorithm for buffer allocation

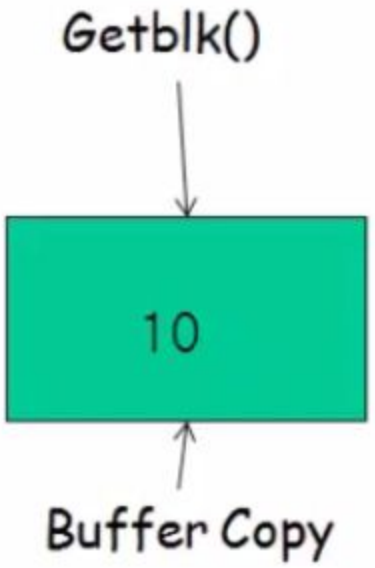
```
else
{
        if(there are no buffer on free list)                    /*scenario 4 */
        {
                sleep(event any buffer becomes free);
                continue;               /*back to while loop*/
        }
        remove buffer from free list;
        if(buffer marked for delayed write){  /*scenario 3 */
                asynchronous write buffer to disk;
                continue;               /*back to while loop*/
        }
        /* scenario 2 – found a free buffer */
        remove buffer from old hash queue;
        put buffer onto new hash queue;
        return buffer;
}
```

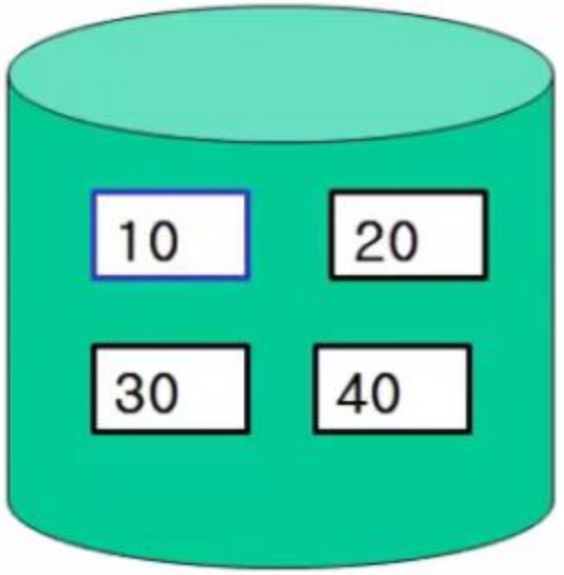
Block Read





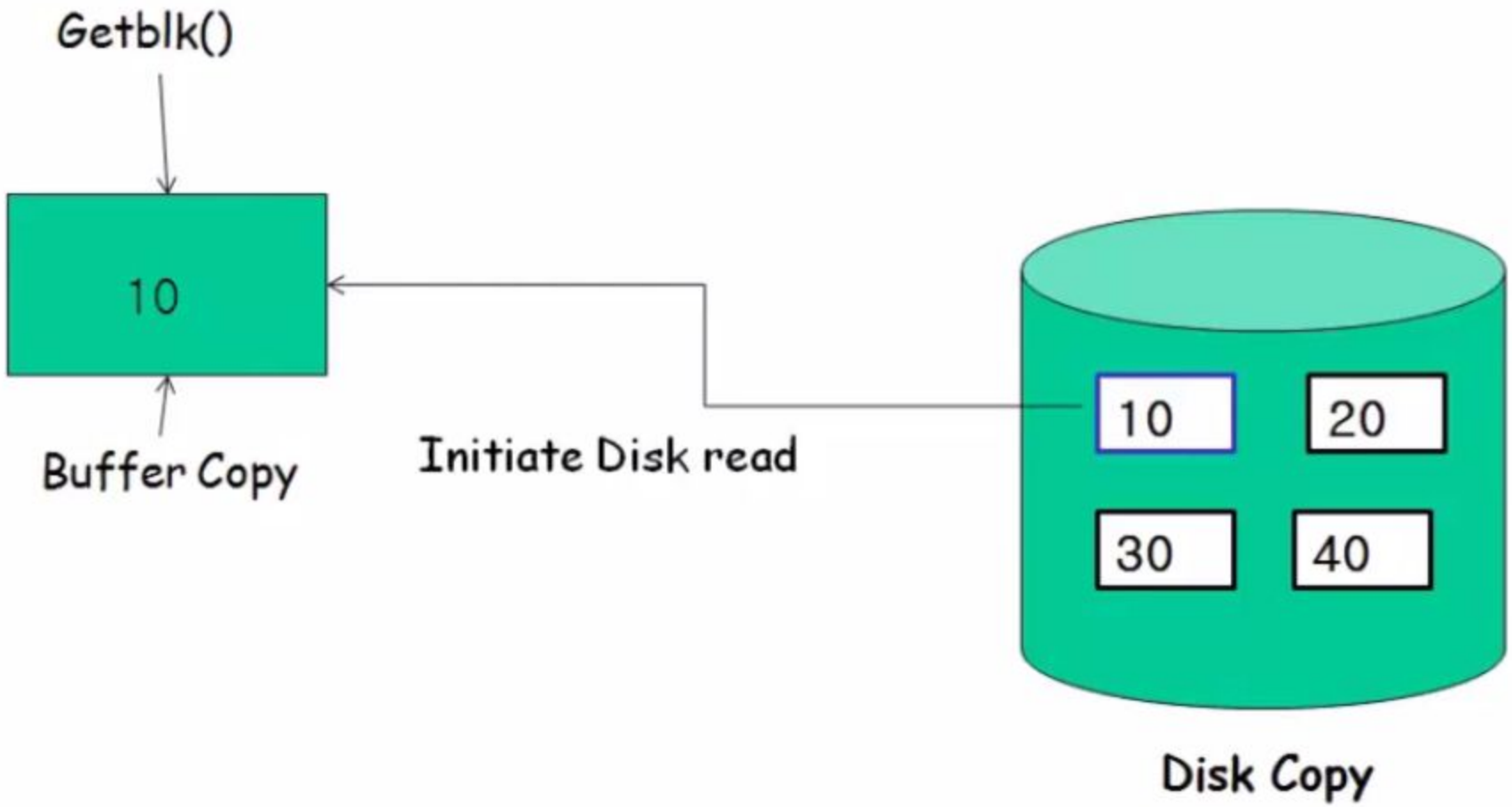
Block Read





Disk Copy

Block Read

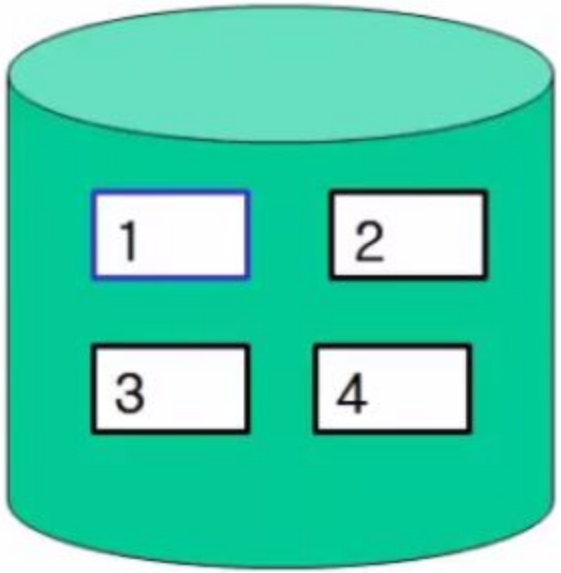


Reading Disk Blocks

algorithm

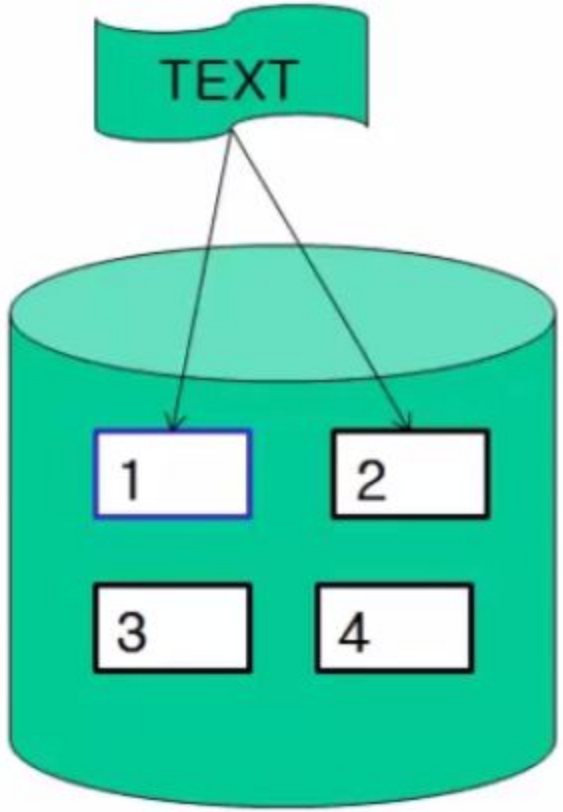
```
Algorithm bread
Input: file system block number
Output: buffer containing data
{
   get buffer for block(algorithm  getblk);
   if(buffer data valid)
      return buffer;
else
   initiate disk read;
   sleep(event disk read complete);
   return(buffer);
}
```

Block Read Ahead



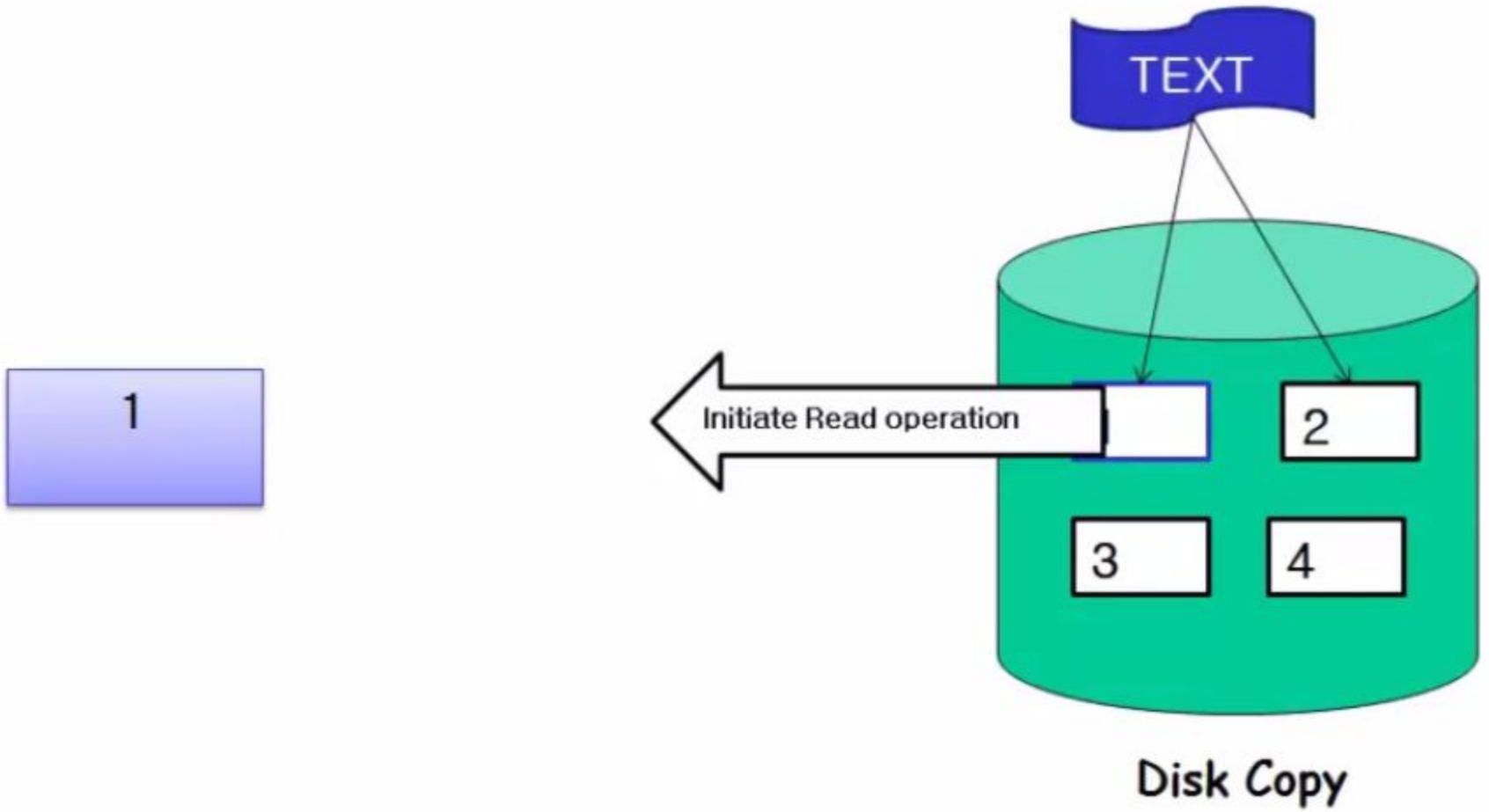
Disk Copy

Block Read Ahead

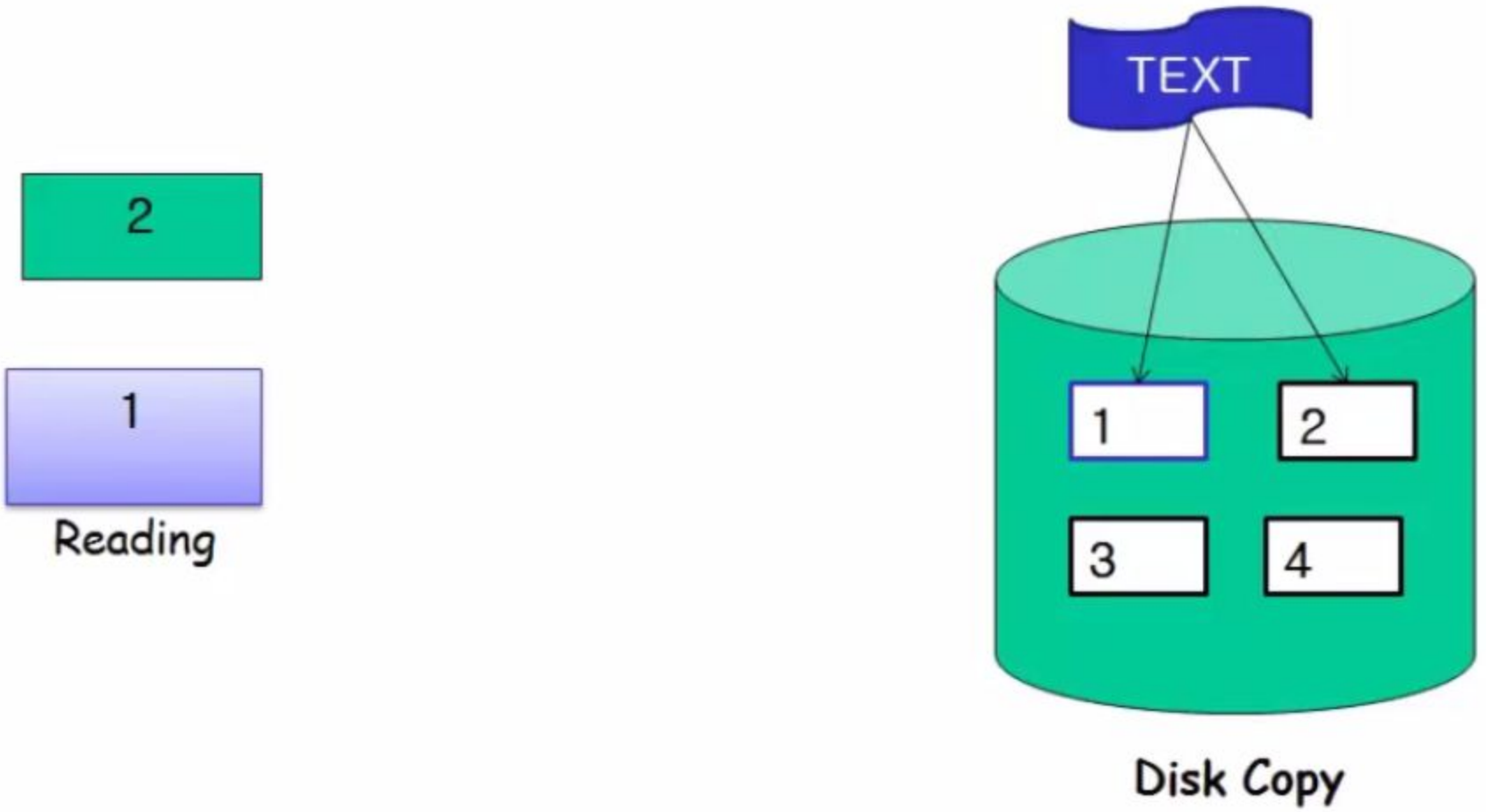


Disk Copy

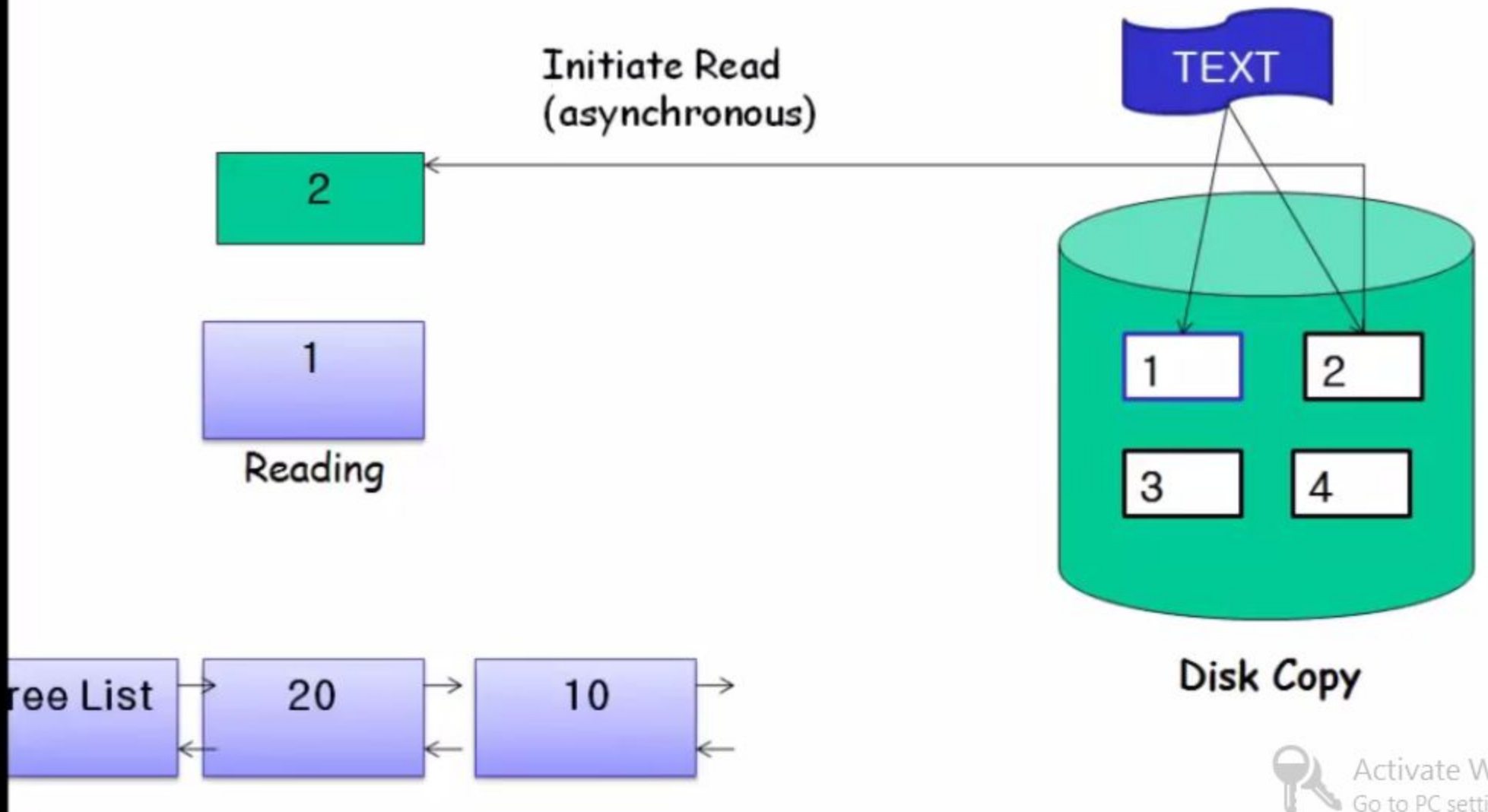
Block Read Ahead



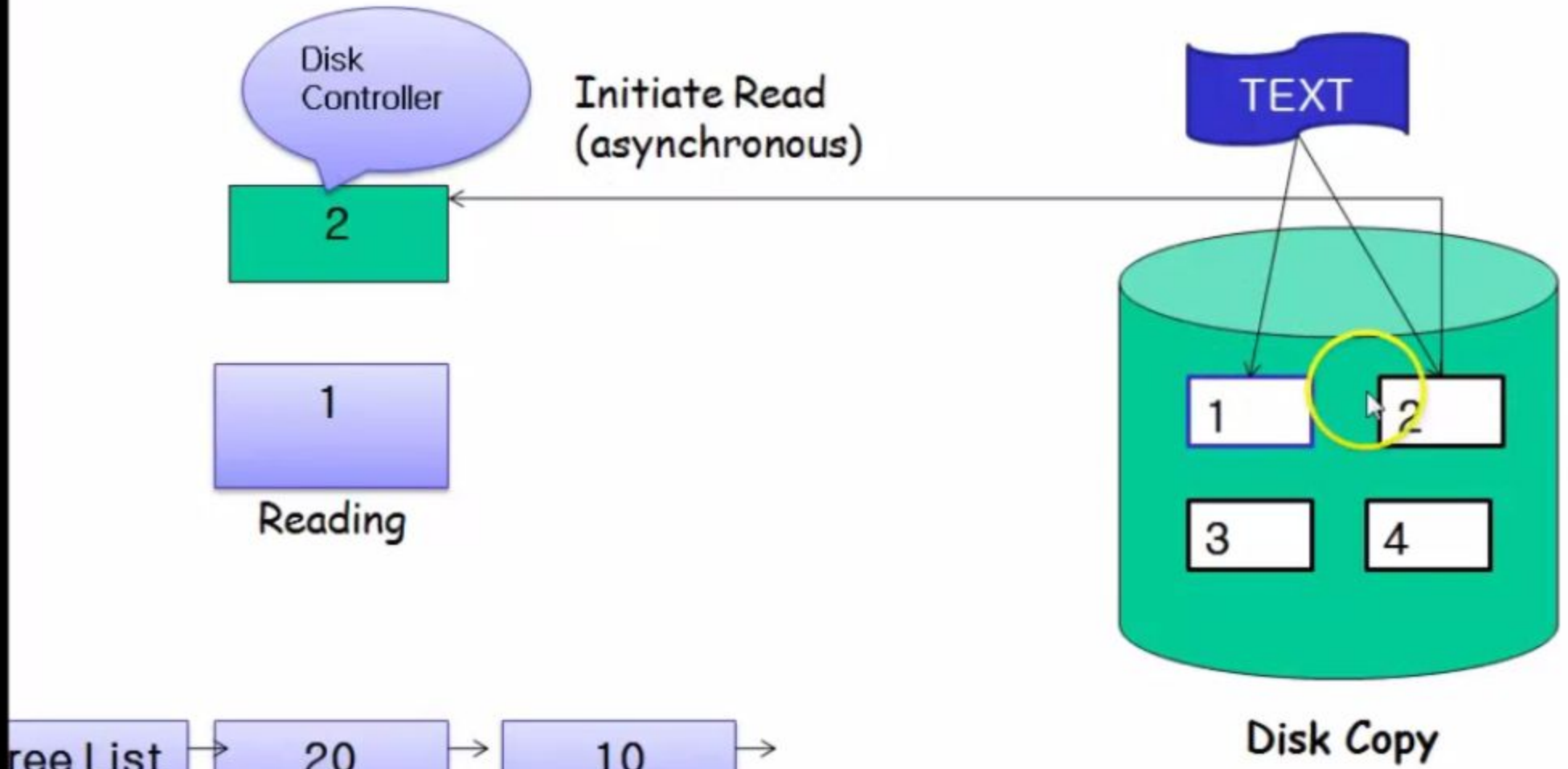
Block Read Ahead



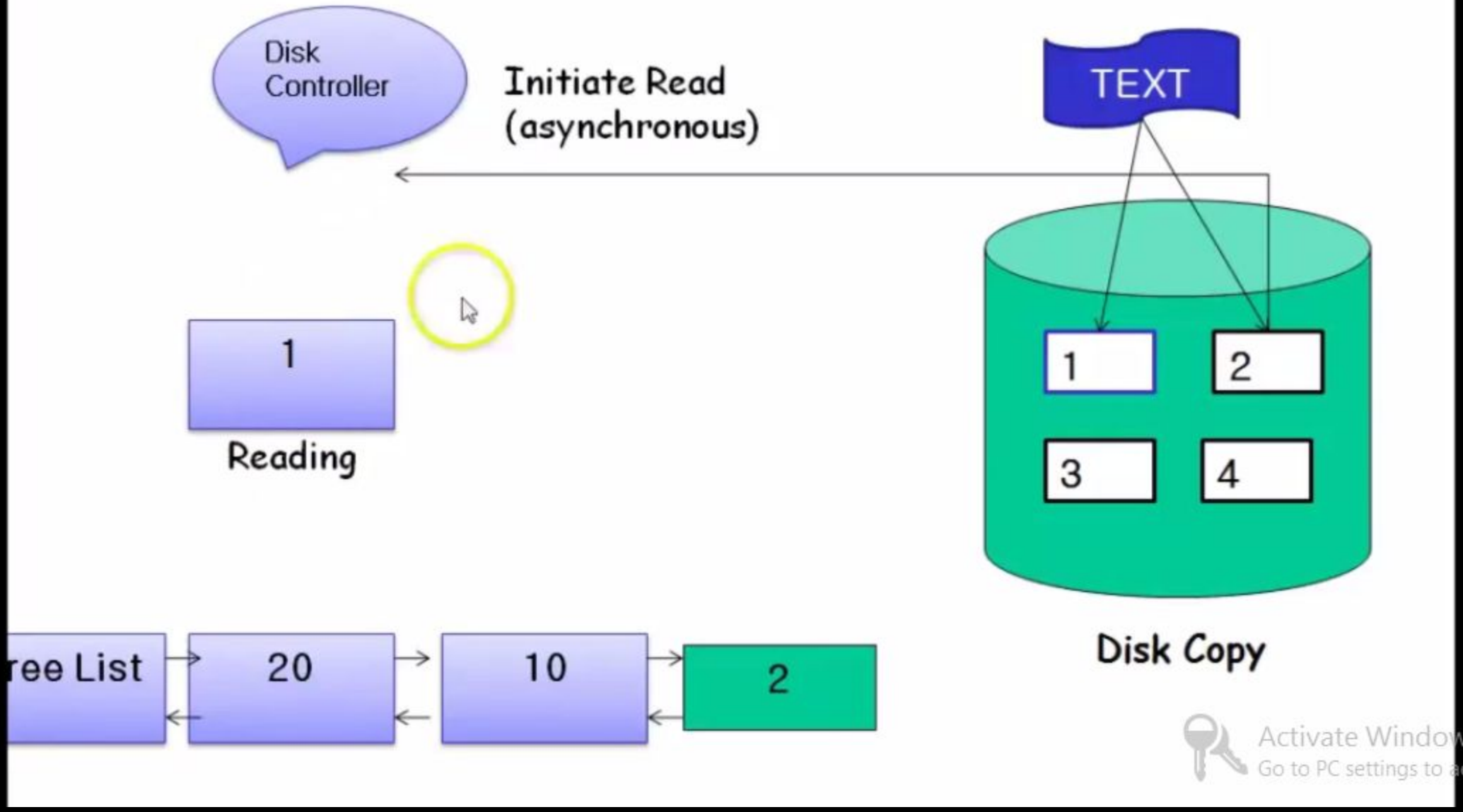
Block read Ahead



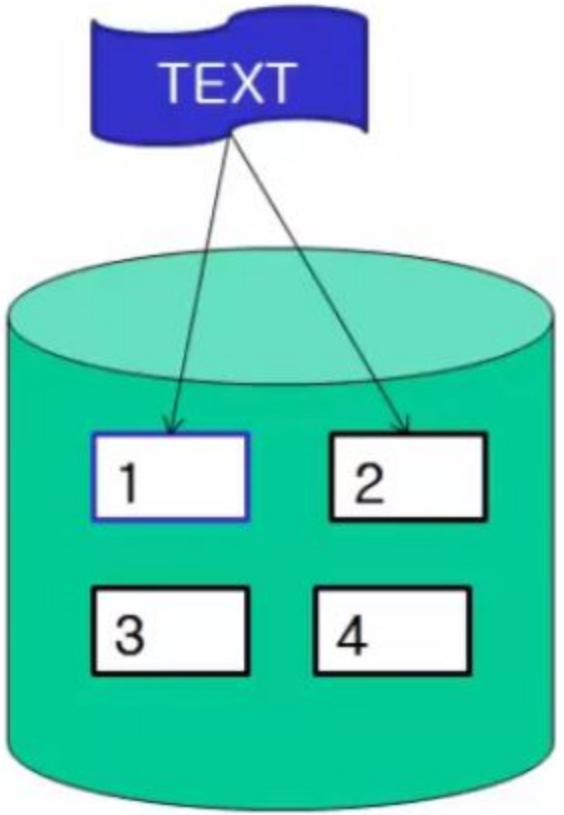
Block read Ahead



Block read Ahead

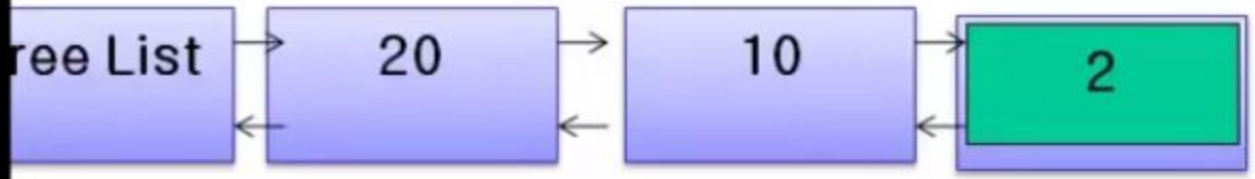


Block read Ahead



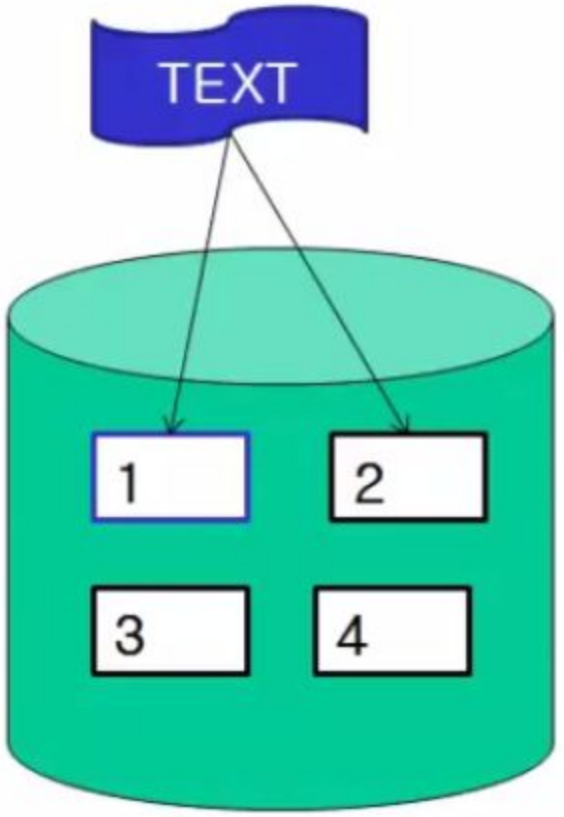
Disk Copy

Reading

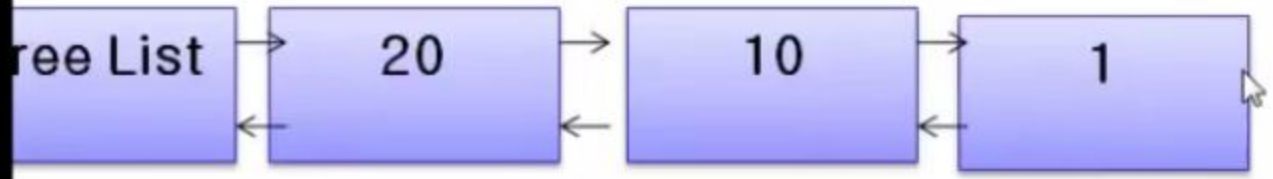


Block read Ahead

Reading



Disk Copy



Reading Disk Blocks

- Read Ahead
  - Improving performance
    - Read additional block before request
  - Use breada()

Algorithm

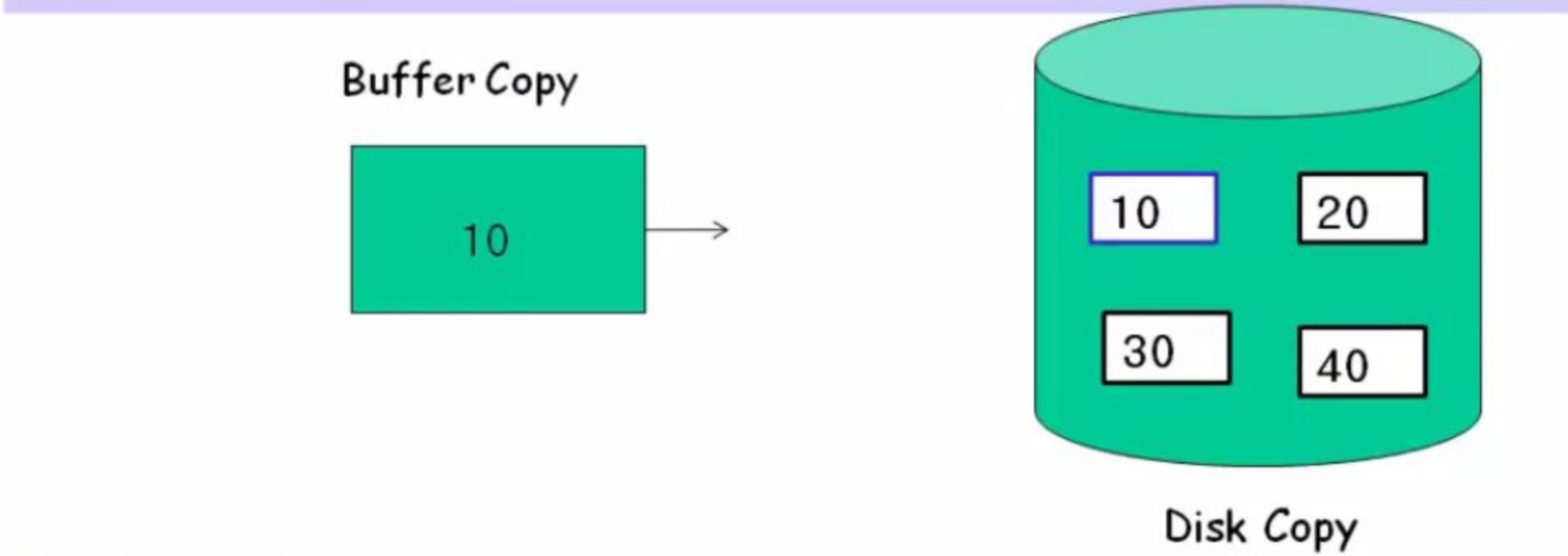
```
Algorithm breada
Input: (1) file system block number for immediate read
       (2) file system block number for asynchronous read
Output: buffer containing data for immediate read
{
  if (first block not in cache){
    get buffer for first block(algorithm getblk);
    if(buffer data not valid)
        initiate disk read;
  }
```

Reading disk Block

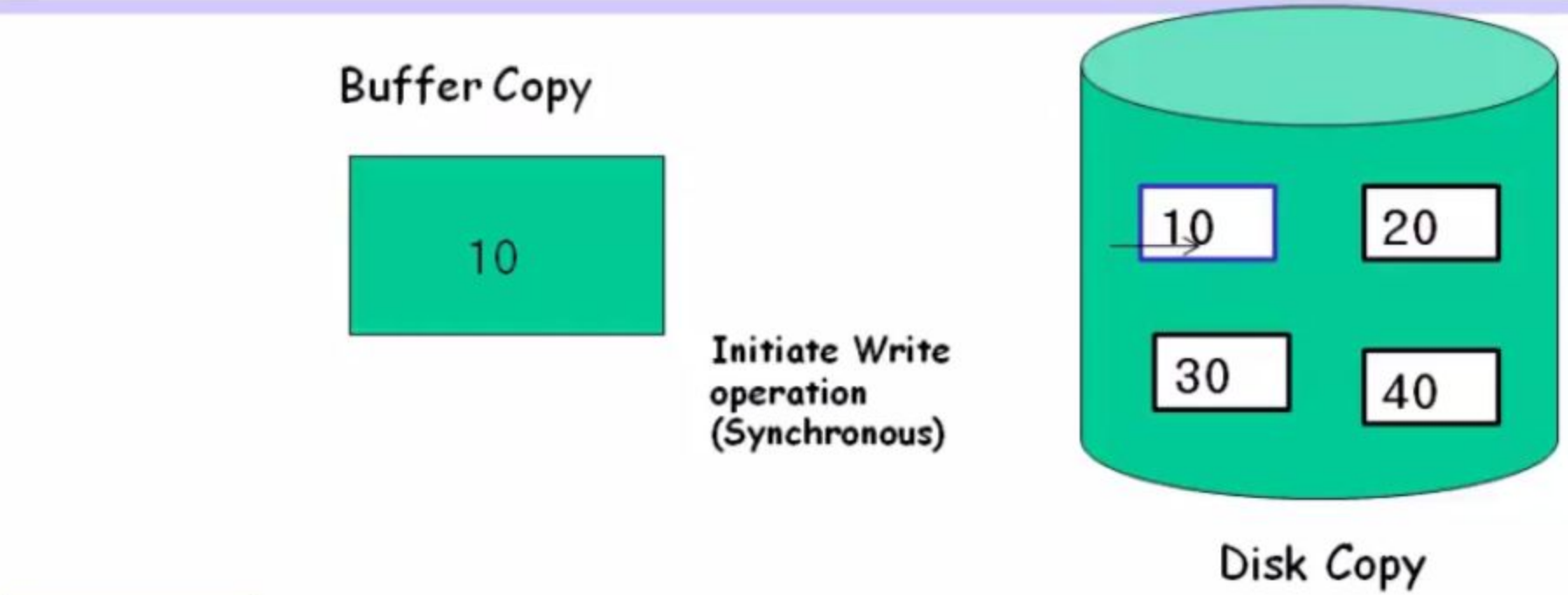
Algorithm-cont

```
if second block not in cache){
      get buffer for second block(algorithm getblk);
      if(buffer data valid)
          release buffer(algorithm brelse);
      else
          initiate disk read;
    }
    if(first block was originally in cache)
    {
        read first block(algorithm bread)
        return buffer;
     }
     sleep(event first buffer contains valid data);
     return buffer;
}
```

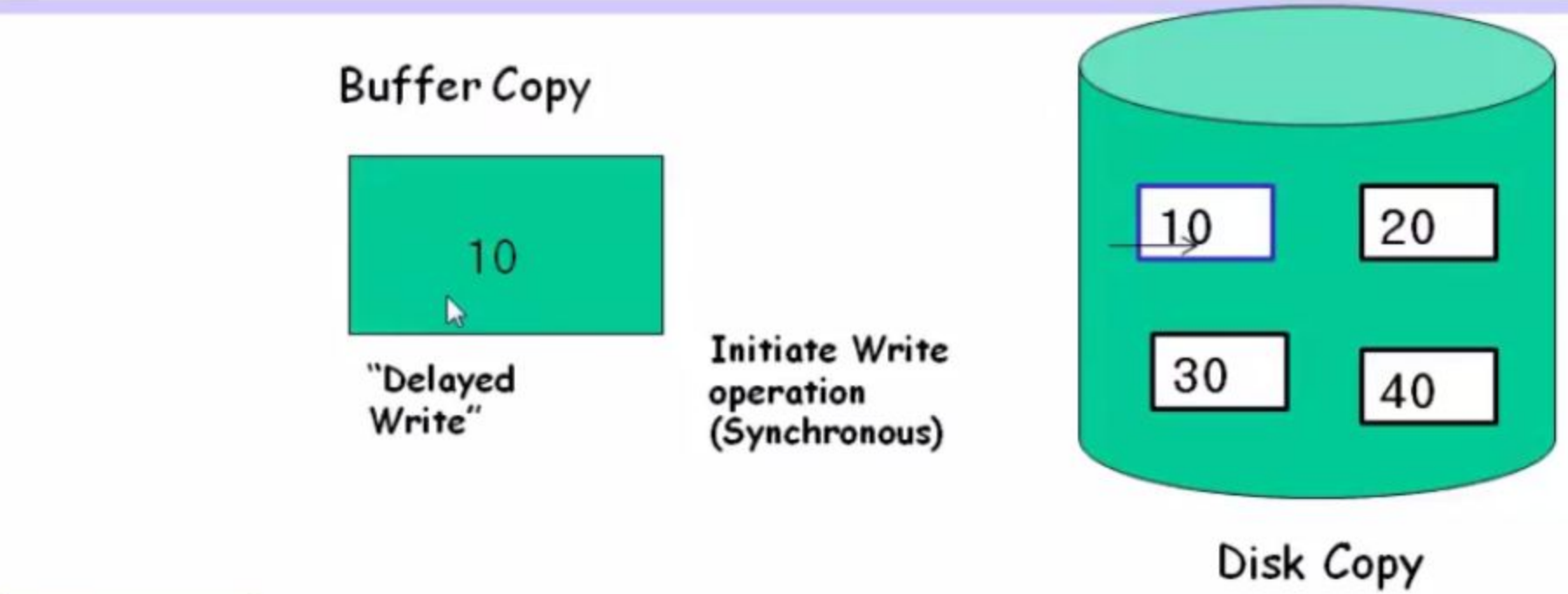
Block Write



Block Write



Block Write

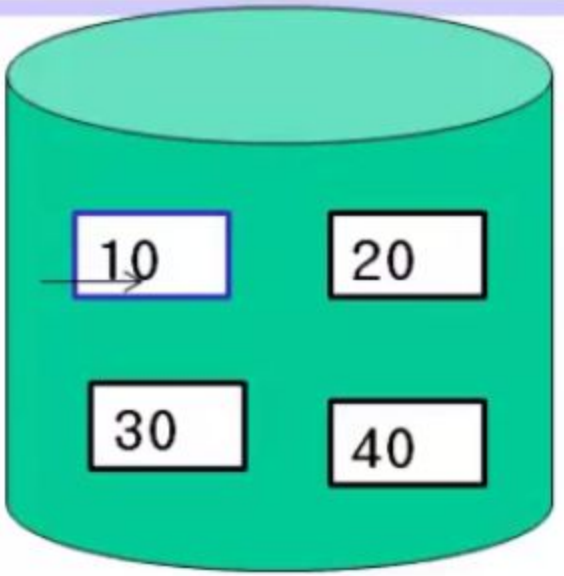


Block Write

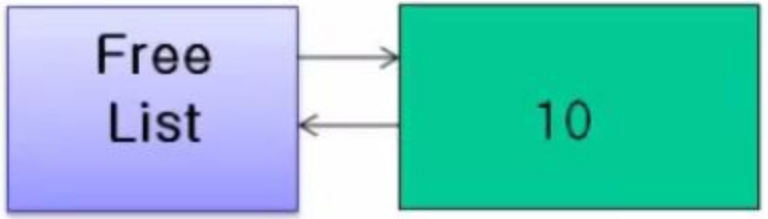
Buffer Copy

"Delayed Write"

Initiate Write operation (Synchronous)



Disk Copy



Writing Disk Block

algorithm

```
Algorithm bwrite
Input: buffer
Output: none
{
   Initiate disk write;
   if (I/O synchronous){
      sleep(event I/O complete);
      release buffer(algorithm brelse);
    }
    else if (buffer marked for delayed write)
       mark buffer to put at head of free list;
}
```

Release Disk Block

algorithm

```
Algorithm brelse
Input: locked buffer
Output: none
{
  wakeup all process; event,
        waiting for any buffer to become free;
  wakeup all process; event,
        waiting for this buffer to become free;
```

Raise processor execution level to block interrupts

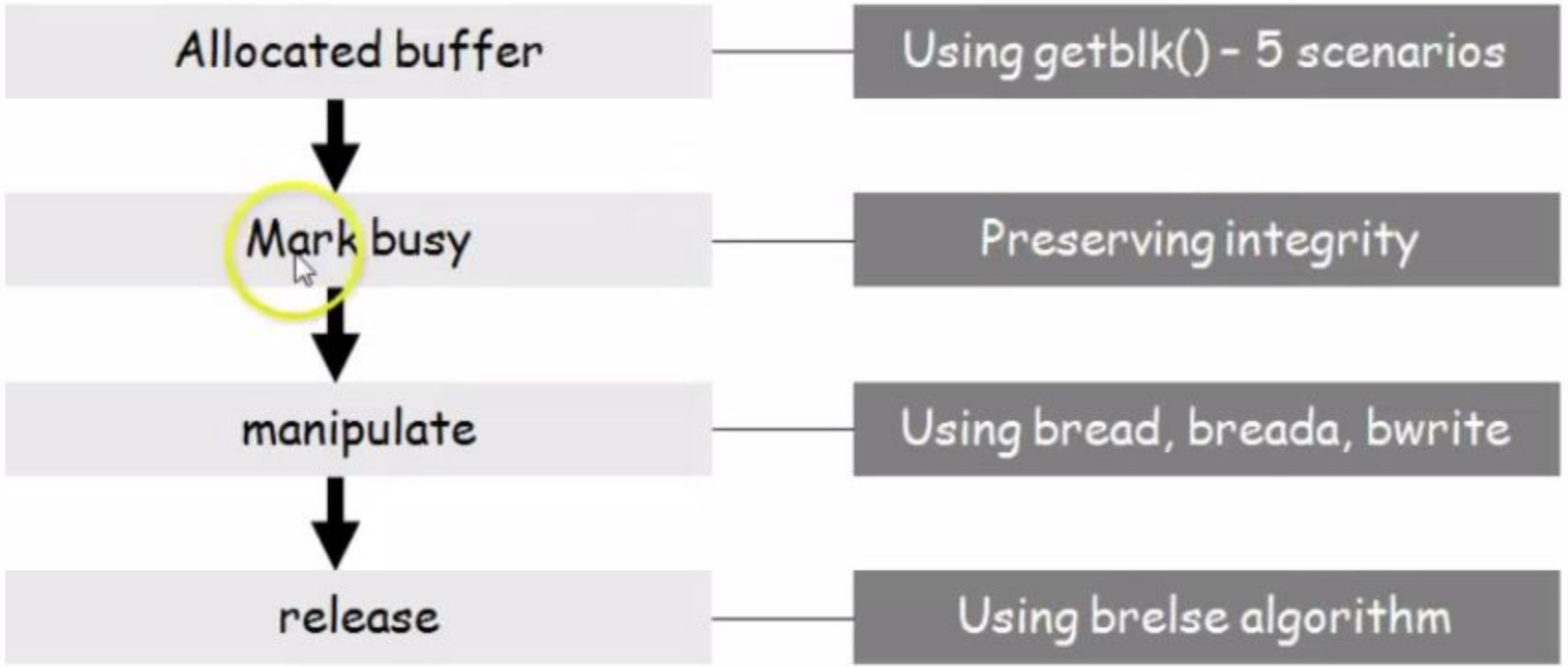
```
  if( buffer contents valid and buffer not old)
      enqueue buffer at end of free list;
  else
      enqueue buffer at beginning of free list
unlock(buffer);
}
```

Lower processor execution level to allow interrupts

Advantages and Disadvantages

- Advantages
  - Allows uniform disk access
  - Reduce the amount of disk traffic
    - less disk access
  - Insure file system integrity
    - one disk block is in only one buffer
- Disadvantages
  - Can be vulnerable to crashes
    - When delayed write
  - requires an extra data copy
    - When reading and writing to and from user processes

What happen to buffer until now



Thank you!