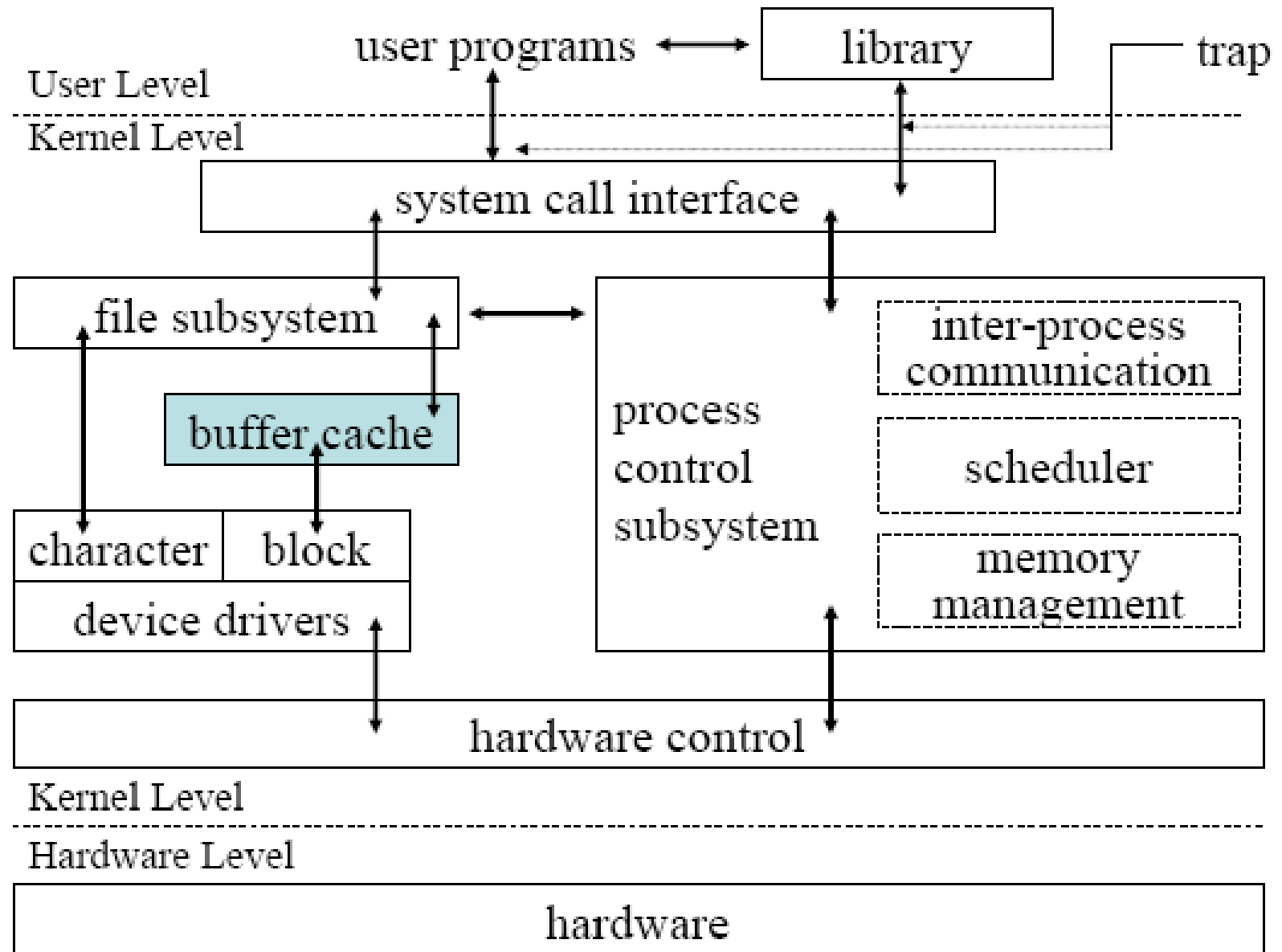


Buffer Cache

Block Diagram of the System Kernel



Introduction

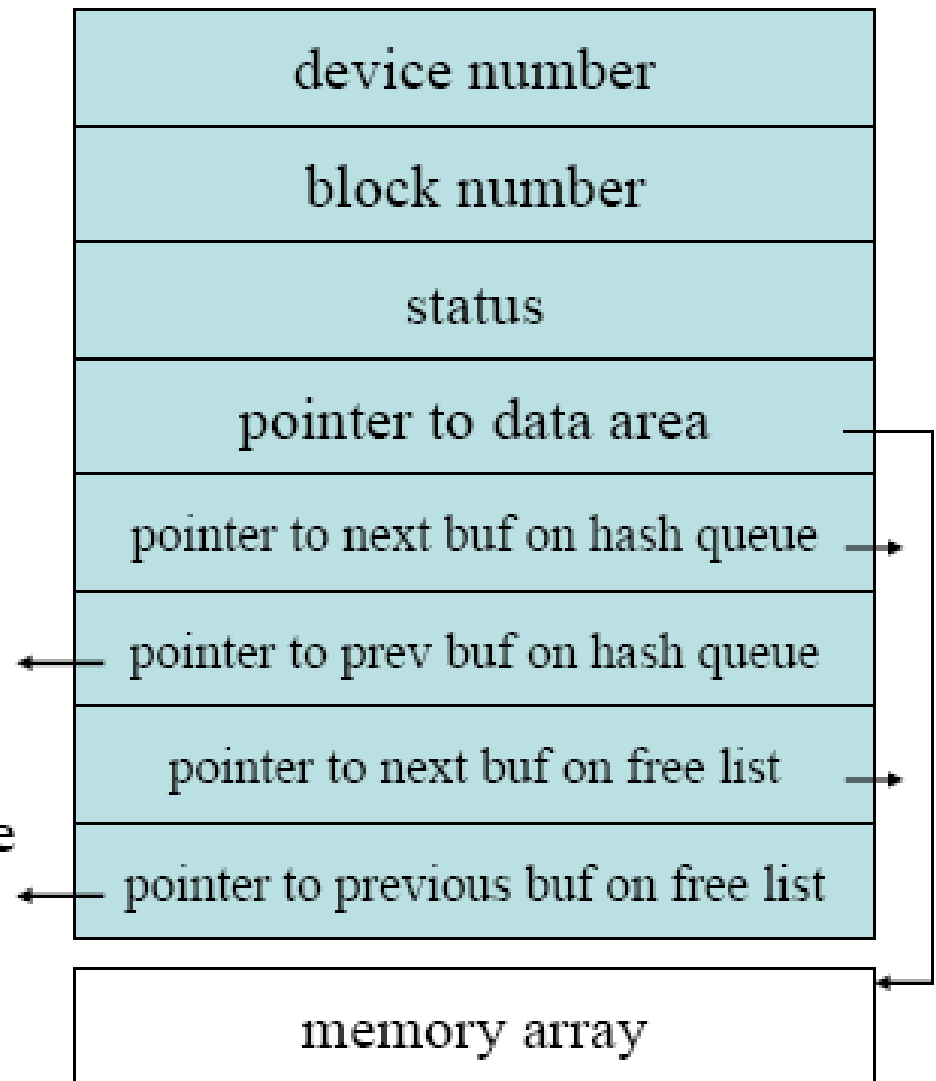
- When reading data from the disk, the kernel attempts to read from the buffer cache.
 - If the data is already in the cache, the kernel does not have to read from the disk.
 - If not, the kernel reads the data from the disk and caches it, using an algorithm that tries to save as much good data in the cache as possible.
- Data being written to disk is cached.
 - The kernel also attempts to minimize the frequency of disk write operations by determining whether the data must really be stored on disk or whether it is transient data that will soon be overwritten.

Buffer Headers

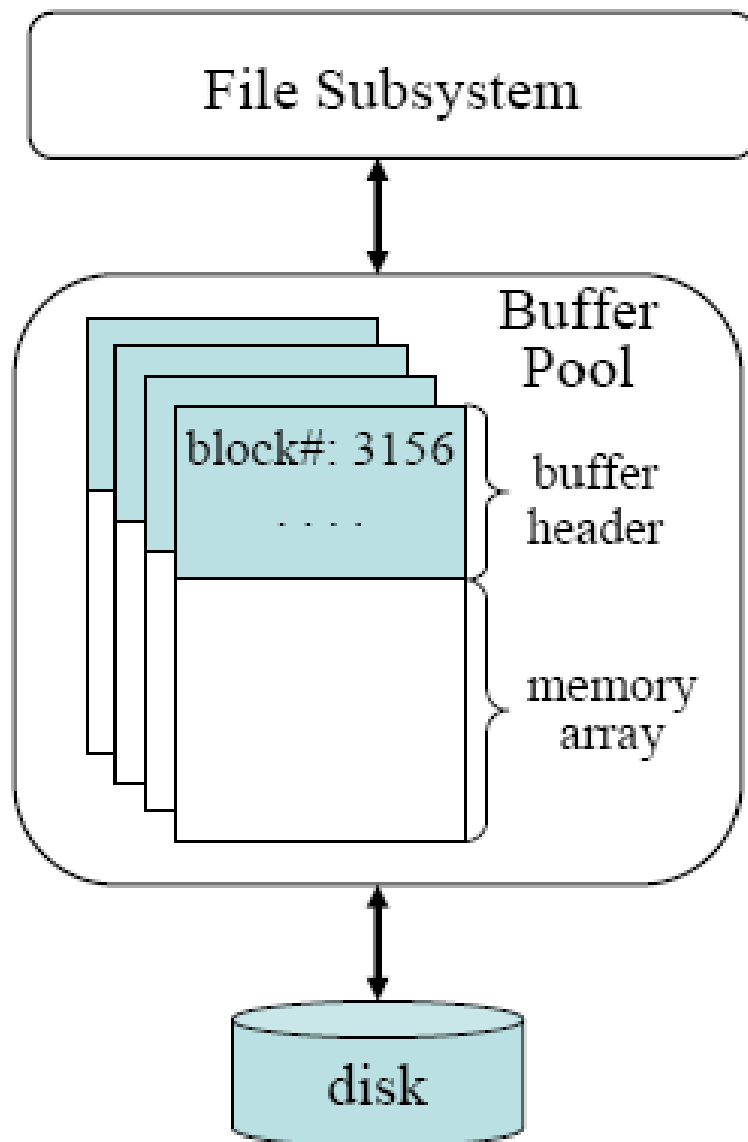
- A buffer consists of two parts: a **memory array** that contains data from the disk and a *buffer header* that identifies the buffer.
- The data in a buffer corresponds to the data in a logical disk block on a file system, and the kernel identifies the buffer contents by examining identifier fields in the buffer header.
- The contents of the disk block map into the buffer, but the mapping is temporary.
- A disk block can never map into more than one buffer at a time.

Buffer Header

- The *device number* is the logical file system number.
- The status of a buffer is combination of:
 - locked
 - contains valid data
 - delayed-write
 - kernel is reading/writing
 - a process is waiting for the buffer to become free





Buffer Pool



- The top cache block is mapped to disk block 3156.
 - This mapping is temporary.
- If the file subsystem tries to read disk block 3156, the cached data is returned without disk access.
- If the file subsystem tries to read disk block 4818 (not cached), the kernel reads the data from the disk and caches it.
 - The top cache block might be selected to cache the data of disk block 4818.

Requirements to Buffer Pool

- Fast lookup
 - The requested buffer should be looked up as fast as possible.
 - How to manage buffers for fast lookup?  hash queues
 - The buffer header has two pointers to manage a hash queue
- Efficient replacement
 - When there is no free buffer and a new disk block is read, one of currently used buffers must be selected for replacement.
 - How to manage buffers for efficient replacement?
 LRU (Least Recently Used) ordering
 - The buffer header has two pointers to manage the free list in LRU order.

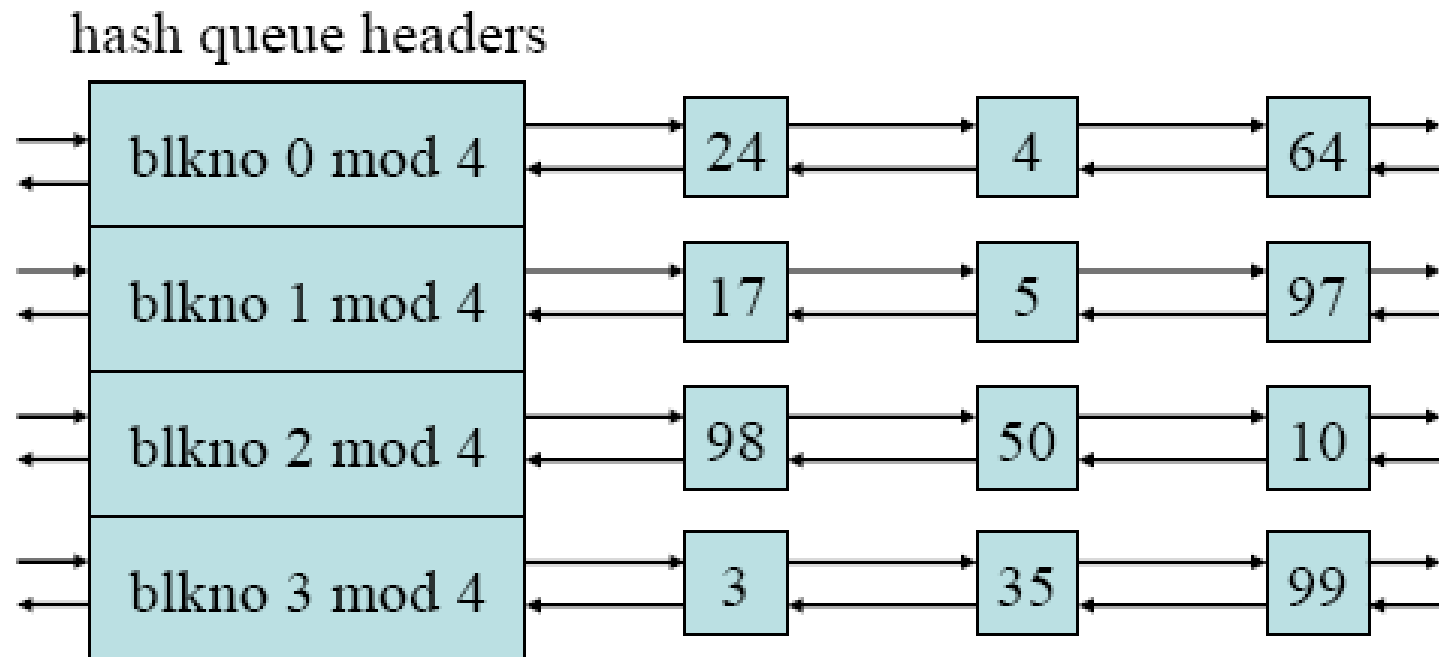
Structure of the Buffer Pool

- The kernel caches data in the buffer pool according to a *least recently used (LRU)* algorithm.
- The kernel maintains a *free list* of buffers that preserves the least recently used order.
 - The free list is a *doubly linked circular list*.
- The kernel takes a buffer from the head of the free list for any free buffer.
 - The kernel can take a buffer from the middle of the free list for a particular block.
- The kernel returns a buffer to the tail of the free list.
- Hence, the buffers closer to the head have not been used as recently as those further from the head.

Structure of the Buffer Pool

- The kernel links the buffers on a *hash queue* into a circular, doubly linked list, similar to the structure of the free list.

e.g., $f(blkno) = blkno \bmod 4$;



Structure of the Buffer Pool

- Each buffer always exists on a hash queue.
 - Every disk block in the buffer pool exists on one and only one hash queue and only once on that queue.
- A buffer may be simultaneously on a hash queue and on the free list if its status is free.
- The kernel has two ways to find a buffer:
 - It searches the hash queue if it is looking for a particular buffer.
 - It removes a buffer from the free list if it is looking for any free buffer.

Scenarios for Retrieval of a Buffer

- 5 typical scenarios of *getblk* algorithm:
 - (1) The kernel finds the block on its hash queue, and its buffer is free.
 - (2) The kernel cannot find the block on the hash queue, so it allocates a buffer from the free list.
 - (3) The kernel cannot find the block on the hash queue and, in attempting to allocate a buffer from the free list, finds a buffer marked “delayed write.”
 - (4) The kernel cannot find the block on the hash queue, and the free list is empty.
 - (5) The kernel finds the block on the hash queue, but its buffer is currently busy.

```

algorithm getblk
input: file system number
       block number
output: locked buffer
{
  while (buffer not found) {
    if (block in hash queue) {
      if (buffer busy) { /* scenario 5 */
        sleep(event buffer becomes free);
        continue;
      }
      mark buffer busy; /*scenario 1 */
      remove buffer from free list;
      return buffer;
    } else {
      if (there are no buffers on free list) {
        /* scenario 4 */
        sleep(event any buffer becomes free);
        continue;
      }
    }
  }
}

```

```

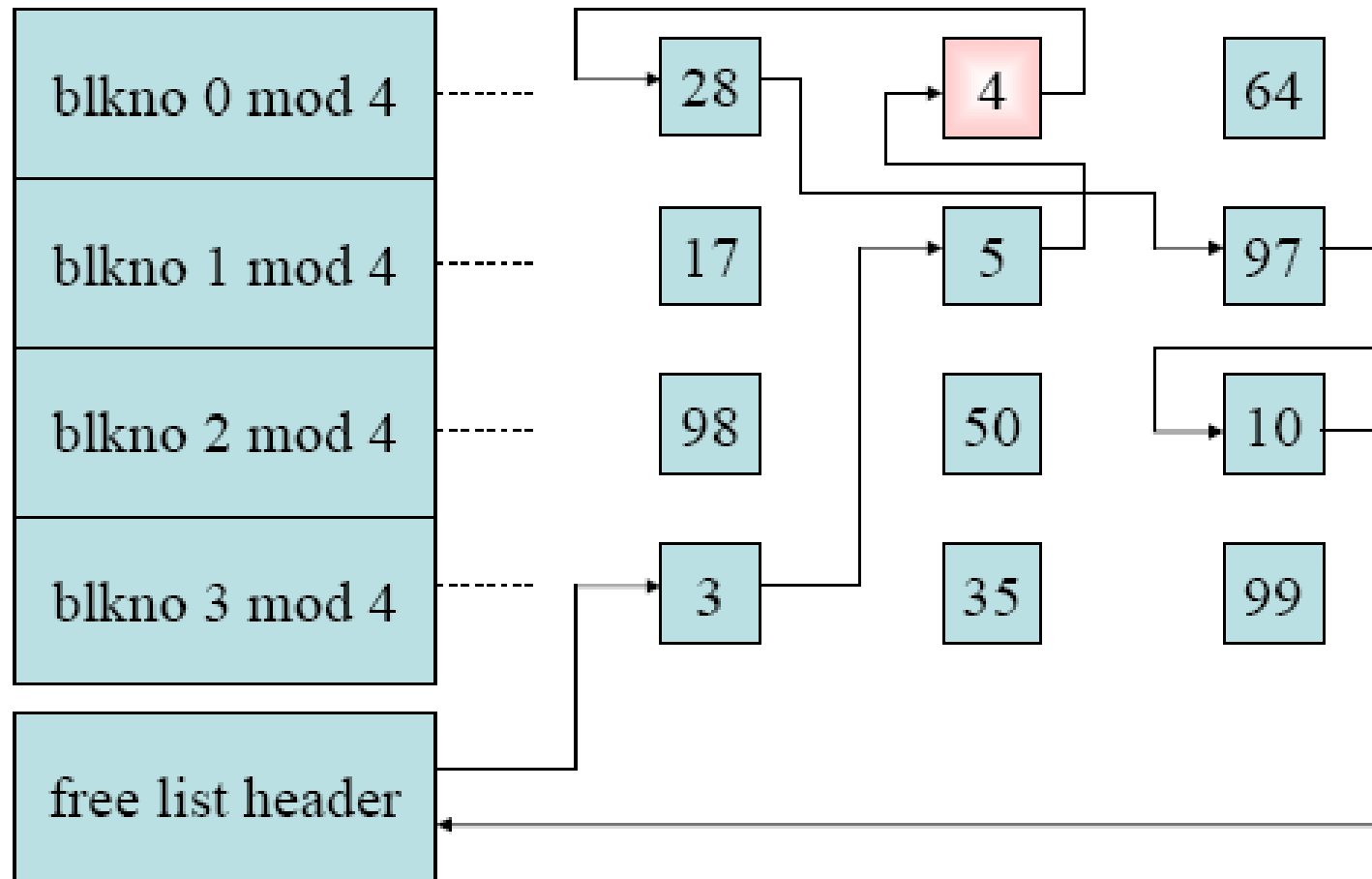
    remove buffer from free list;
    if (buffer marked for delayed
        write) { /* scenario 3 */
      asynchronous write buffer
        to disk;
      continue;
    }
    /* scenario 2 */
    remove buffer from old hash
      queue;
    put buffer onto new hash
      queue;
    return buffer;
  } /* else */
} /* while */

```

algorithm for
buffer allocation

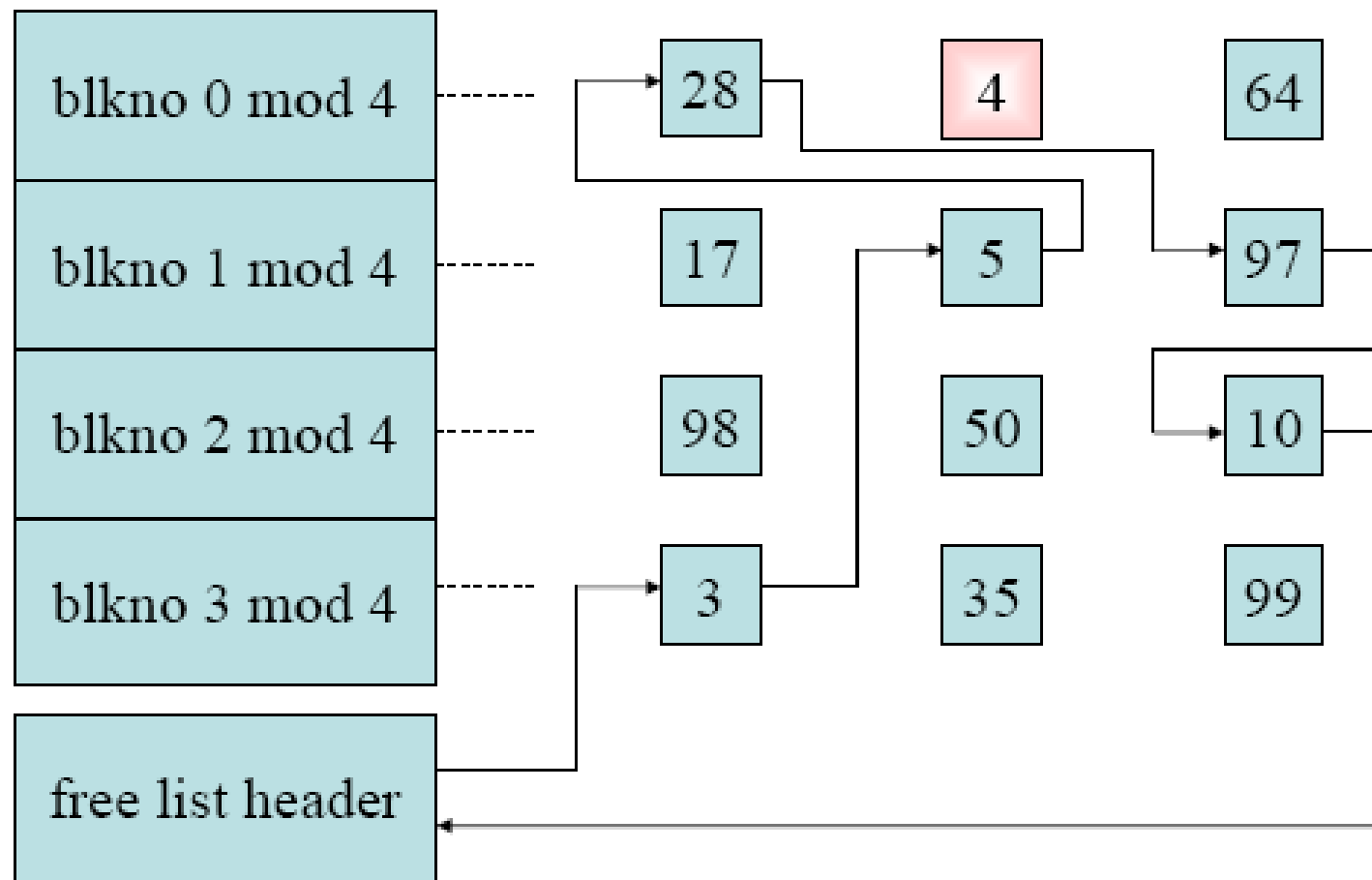
Scenario 1 (1/2)

- search for block 4 on first hash queue



Scenario 1 (2/2)

- remove block 4 from free list



After Allocating a Buffer

- The kernel leaves the buffer marked *busy*; no other process can access it.
- When the kernel finishes using the buffer, it releases the buffer according to algorithm *brelse*.
- It wakes up processes that had fallen asleep:
 - because the buffer was busy and,
 - because no buffer remained on the free list.
- The kernel places the buffer at the end of the free list.

Algorithm for Releasing a Buffer

algorithm **brelease**

input: locked buffer

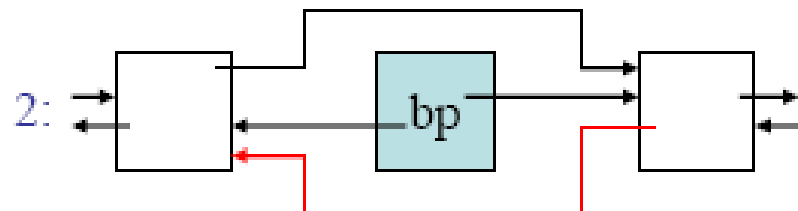
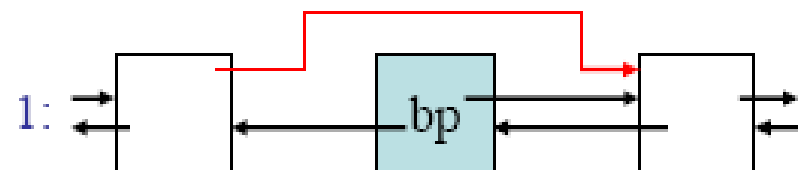
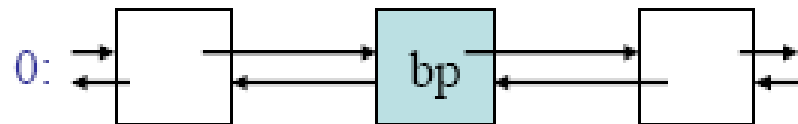
output: none

```
{  
    wakeup all procs: event, waiting for any buffer to become free;  
    wakeup all procs: 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;  
    lower processor execution level to allow interrupts;  
    unlock(buffer);  
}
```


brelse and Interrupt

- The kernel also invokes *brelse* when a handling a disk interrupt to release buffers used for asynchronous I/O.
- The kernel raises the processor execution level to prevent disk interrupts when manipulating the free list.
- Bad effects could happen if an interrupt handler invoked *brelse* while a process was executing *getblk*, so the kernel raises the processor execution level at strategic places in *getblk*, too.

1: $bp \rightarrow f_bp \rightarrow f_fp = bp \rightarrow f_fp;$
2: $bp \rightarrow f_fp \rightarrow f_bp = bp \rightarrow f_bp;$
3: $bp \rightarrow f_fp = bp \rightarrow bp = \text{NULL};$

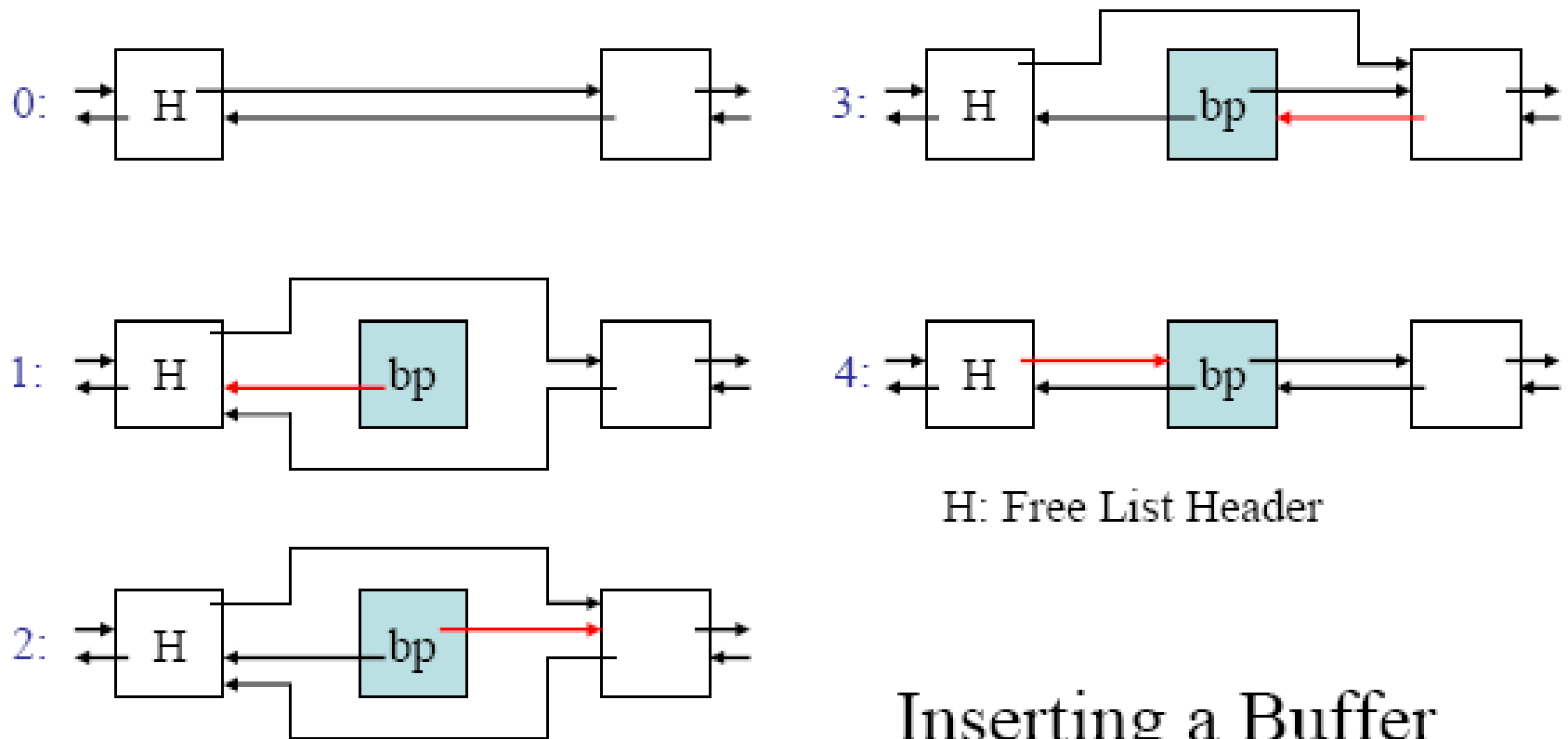


Removing a Buffer
from the Free List

```

1: bp->f_bp = H;
2: bp->f_fp = H->f_fp;
3: H->f_fp->f_bp = bp;
4: H->f_fp = bp;

```



H: Free List Header

Inserting a Buffer
at the Head of the Free List

```

1: bp->f_bp->f_fp = bp->f_fp;
6: bp->f_fp->f_bp = bp->f_bp;
7: bp->f_fp = bp->bp = NULL;

```

removing a buffer in *getblk*

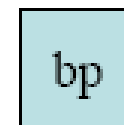
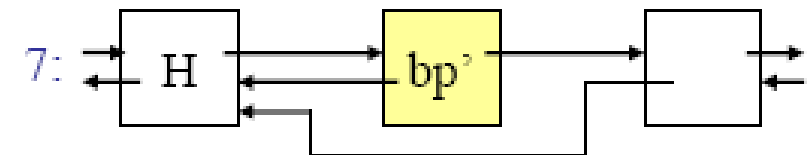
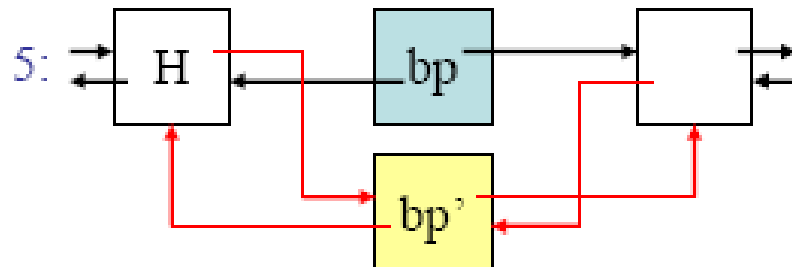
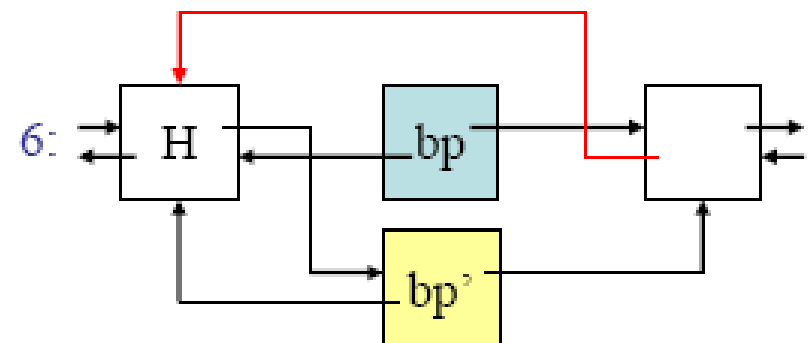
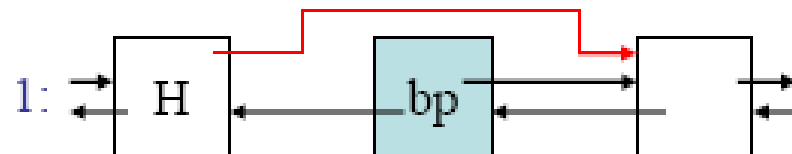
```

2: bp->f_bp = H;
3: bp->f_fp = H->f_fp;
4: H->f_fp->f_bp = bp;
5: H->f_fp = bp;

```

inserting a buffer in
interrupt handler

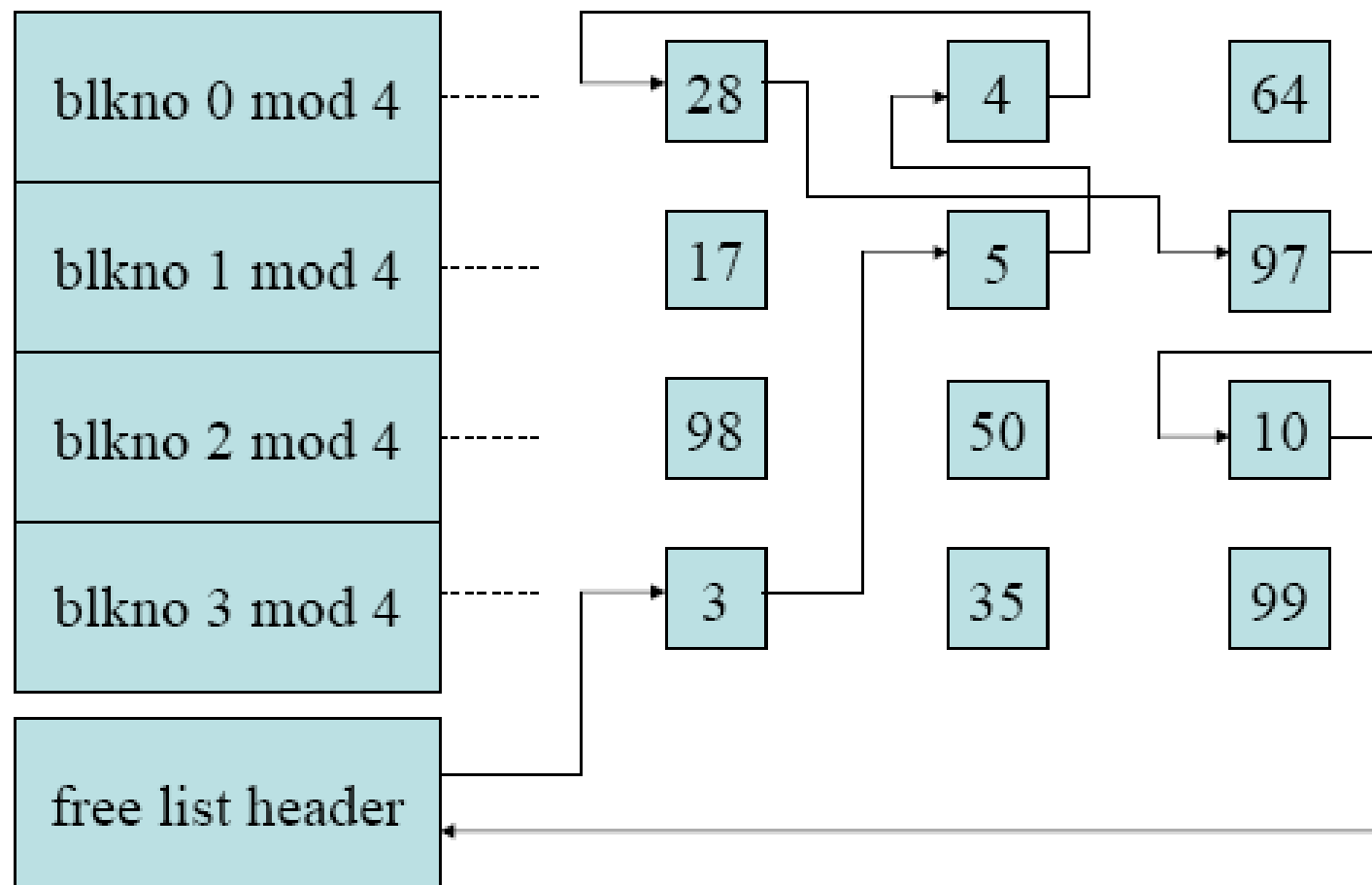
interrupt



An Example of
Bad Effect Caused by
getblk and Interrupt

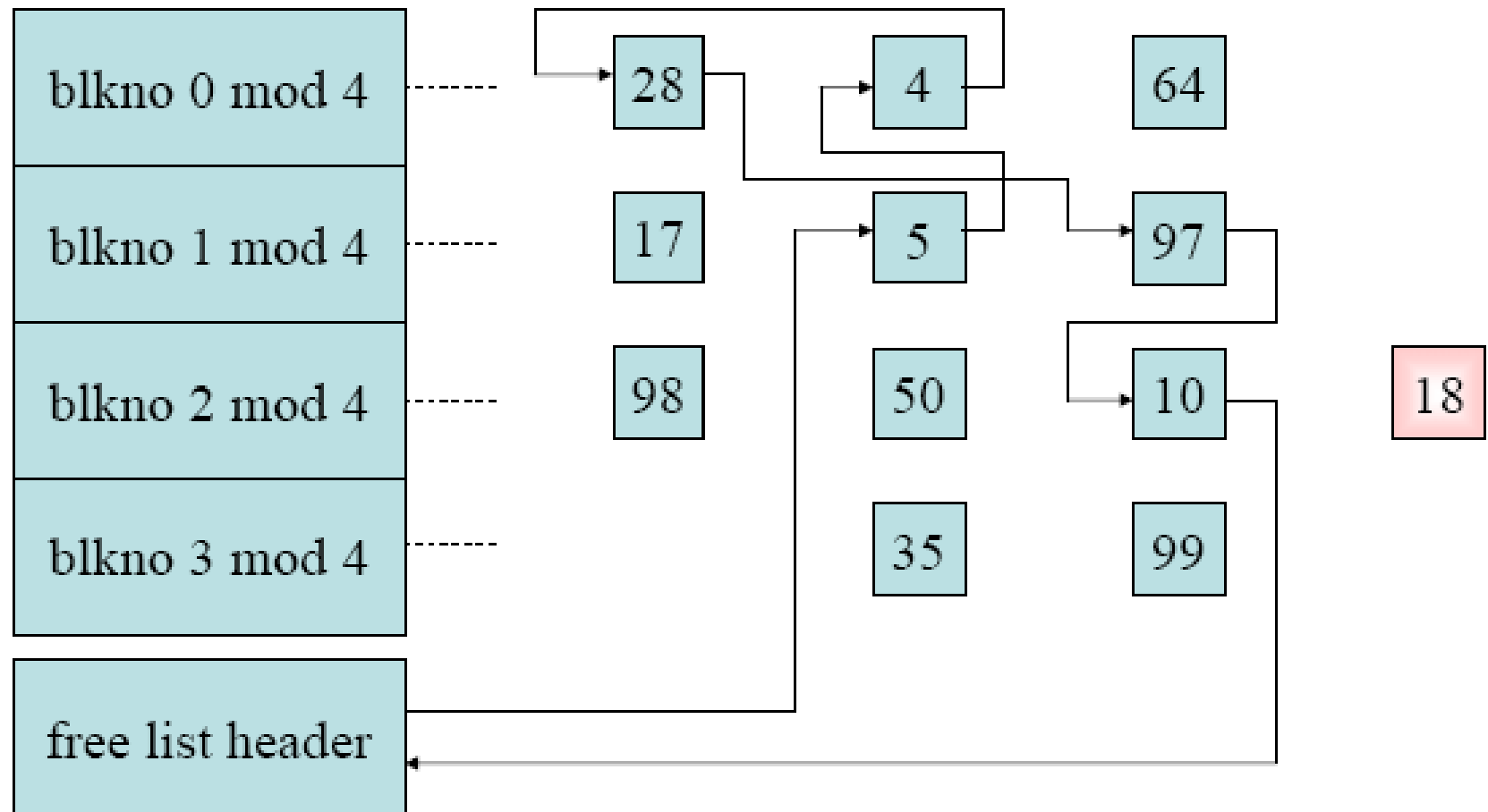
Scenario 2 (1/2)

- search for block 18 – not in cache



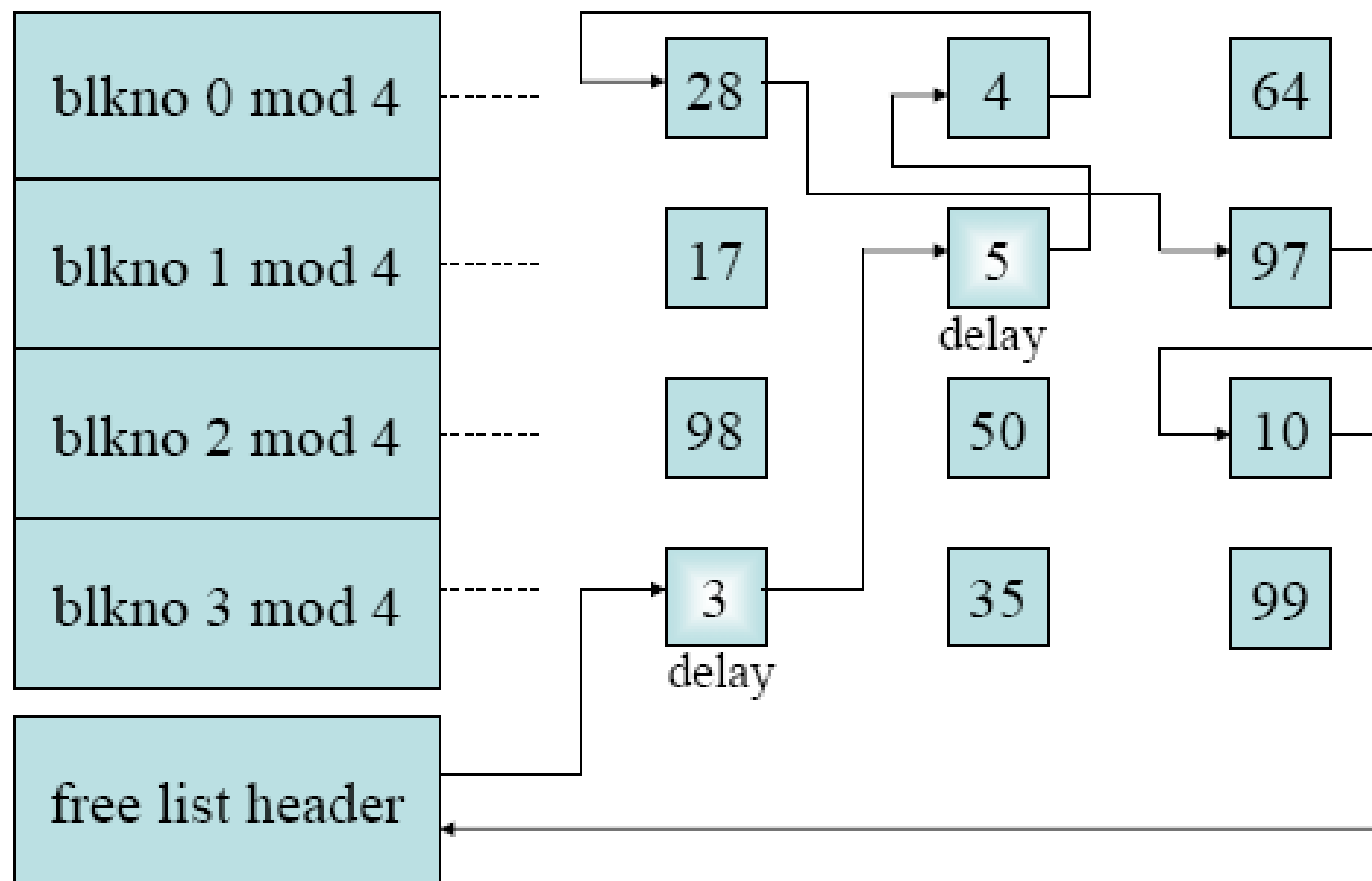
Scenario 2 (2/2)

- remove first block from free list, assign to 18.



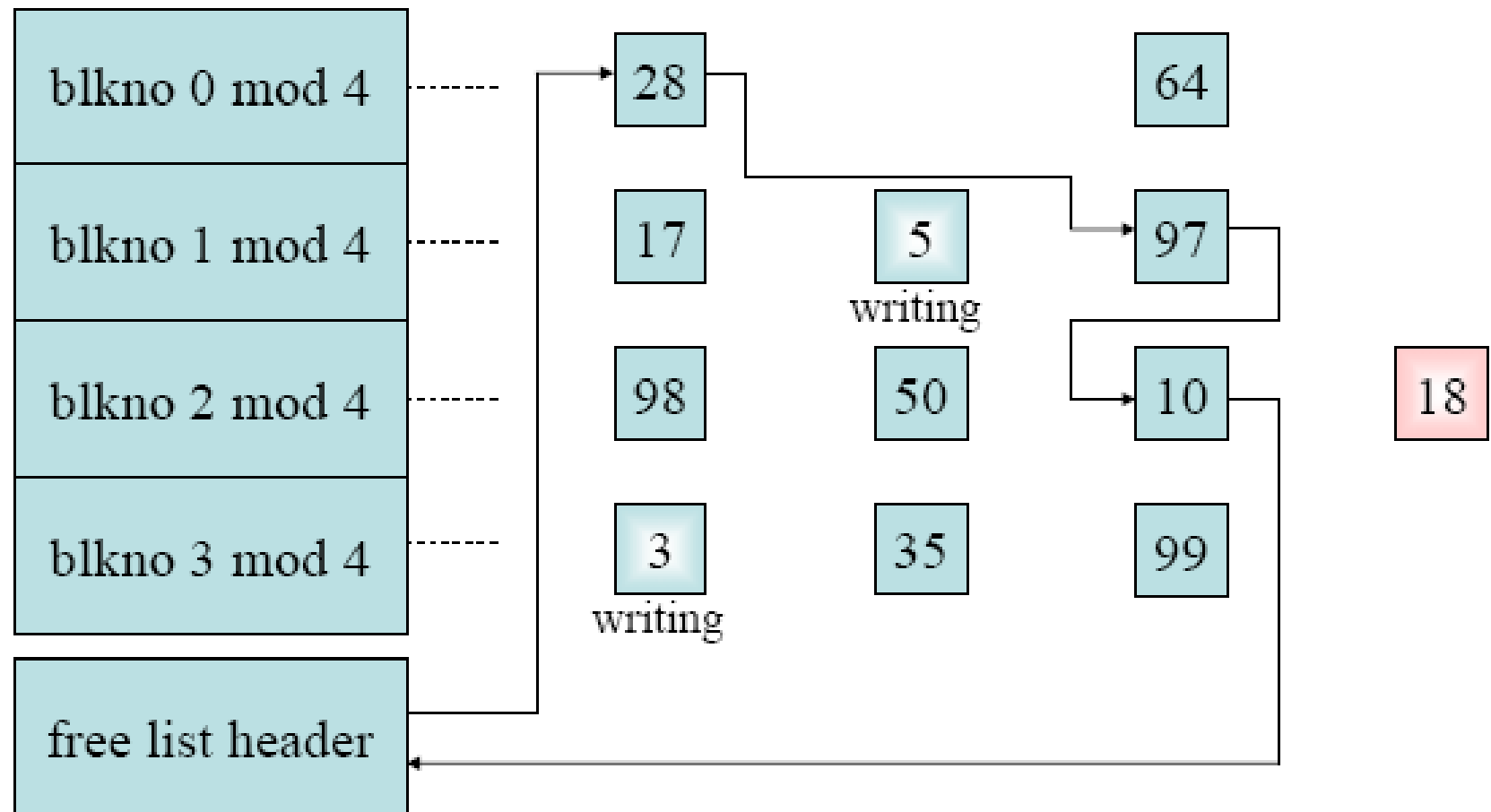
Scenario 3 (1/3)

- search for block 18 – delayed write block on free list



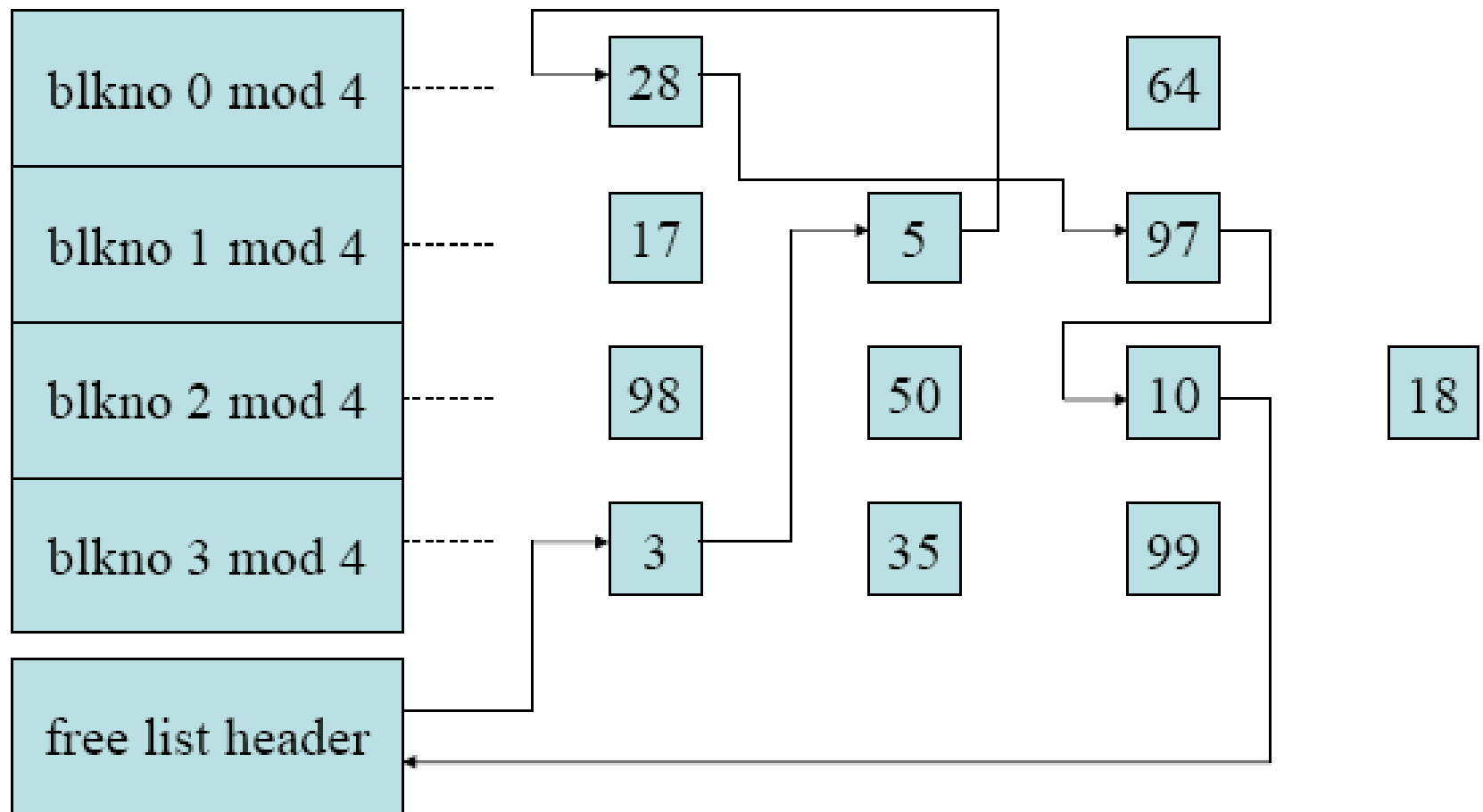
Scenario 3 (2/3)

- writing blocks 3 and 5, reassign 4 to 18



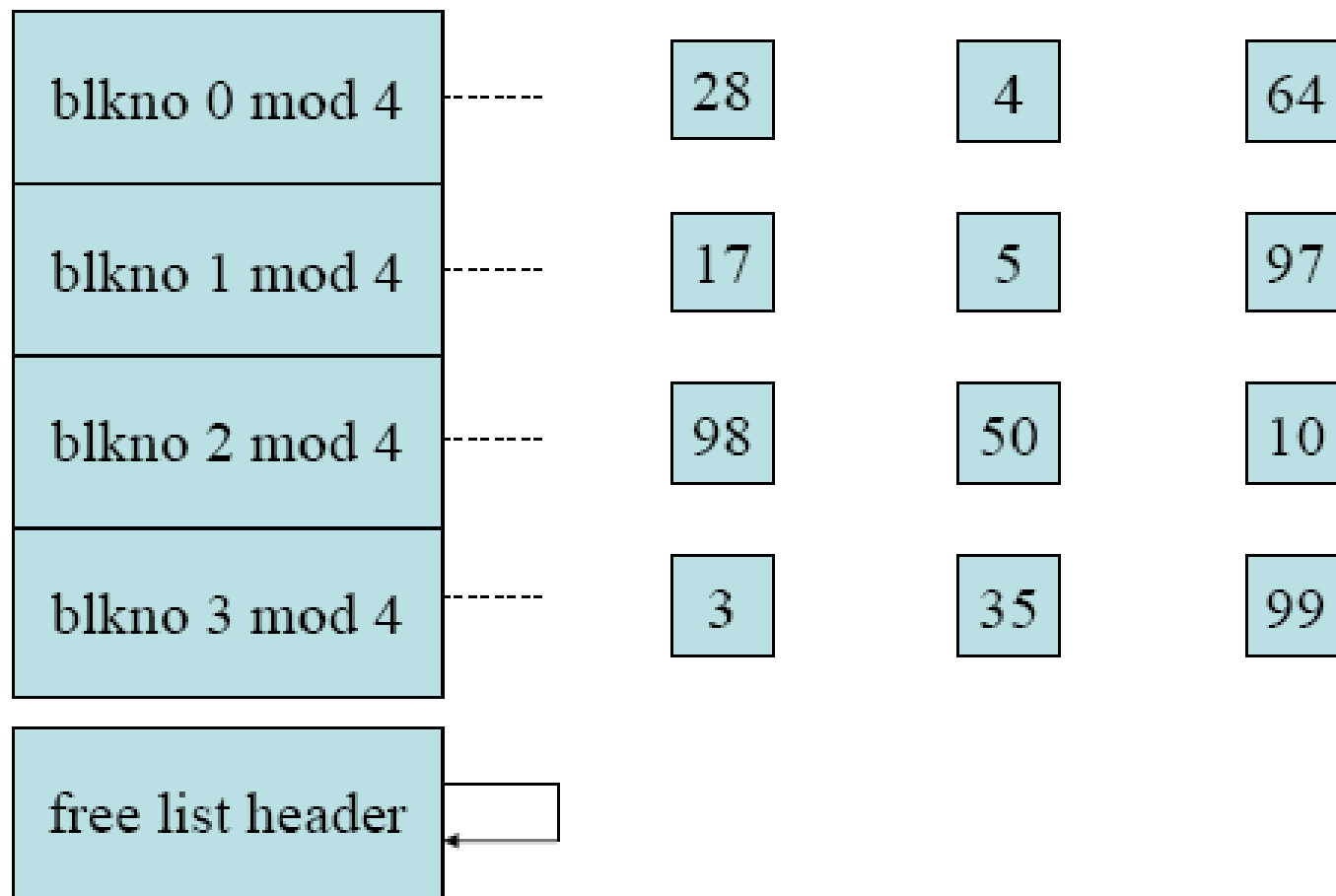
Scenario 3 (3/3)

- block 3 and 5 at the *beginning* of the free list



Scenario 4

- search for block 18, empty free list \Rightarrow asleep

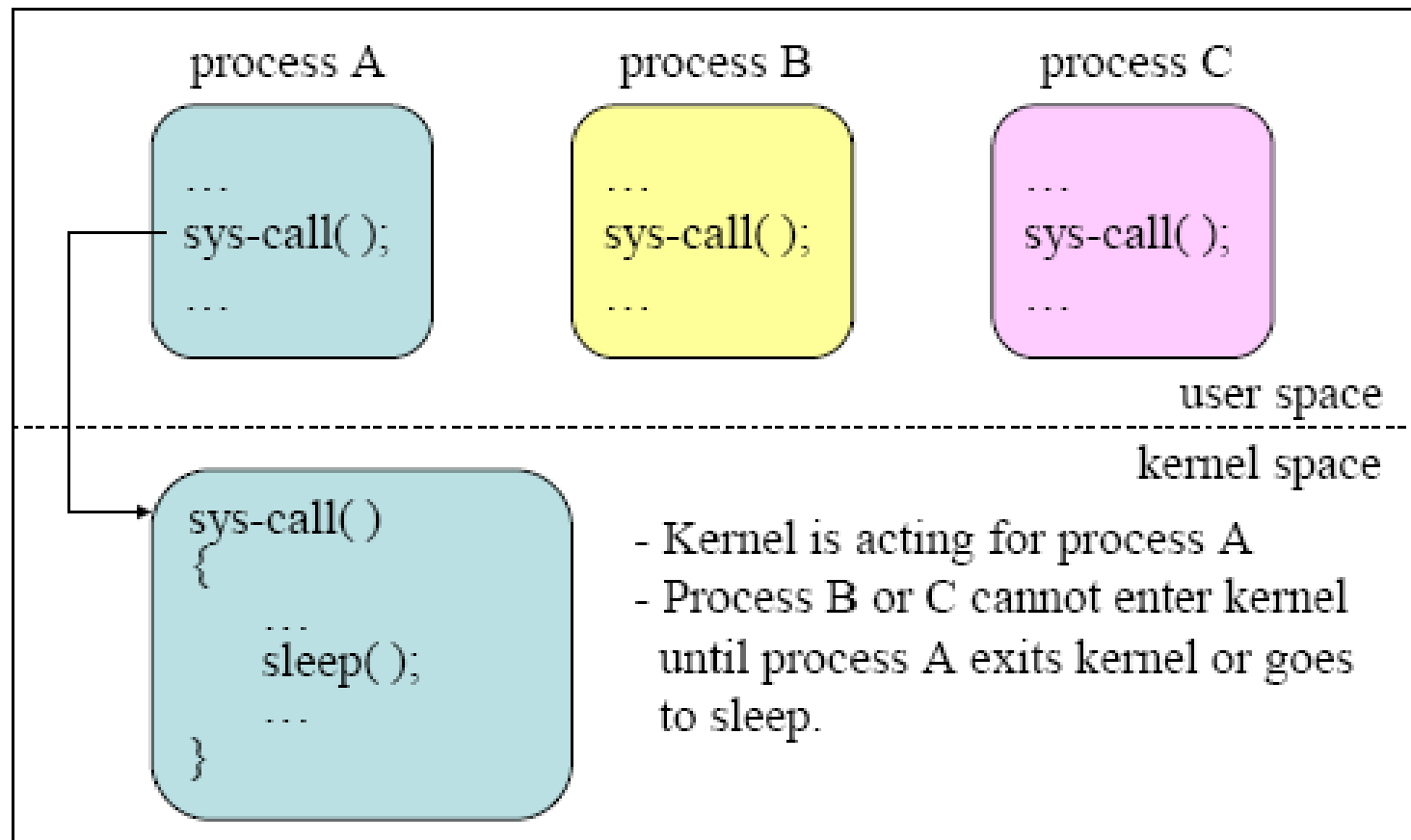


Scenario 5 (1/2)

- This scenario is complicated because it involves complex relationship between several processes.
- Suppose that:
 - The kernel, acting for *process A*, allocates a buffer but goes to sleep before freeing the buffer,
 - *Process B* finds the locked buffer, and marks the buffer “in demand” and then sleeps, and
 - *Process A* will eventually release the buffer and wake up all the process waiting for the buffer.

Supplementation

- “the kernel acting for process A” means:



Scenario 5 (2/2)

- *Process B* must verify that the buffer is free when the kernel again schedules *process B*.
 - Another *process C* may have been waiting for the same buffer, and the kernel may have scheduled *process C* to run before *process B*.
- *Process B* must also verify that the buffer contains the disk block that it originally requested.
 - *Process C* may have allocated the buffer to another block (as in scenario 2).
 - *Process B* must search for the block again.

Process A

Allocate buffer to *block b*
Lock buffer
Initiate I/O
Sleep until I/O done

I/O done, wake up
brelse(): wake up others

Process B

Find *block b* on hash queue
Buffer locked, sleep

Buffer does not contain
block b
Start search again

Process C

Sleep waiting for
any free buffer
(scenario 4)

Get buffer previously
assigned to *block b*
Reassign buffer to *block b'*

time



Properties of Buffer Allocation Algorithm

- The buffer allocation algorithm must be safe; processes must not sleep forever, and they must eventually get a buffer.
- The kernel guarantees that all processes waiting for buffers will wake up.
 - The kernel allocates buffers during the execution of system calls and frees them before returning.
 - Processes in user mode do not control the allocation of kernel buffers directly.

Properties of Buffer Allocation Algorithm

- It is also possible to imagine cases where a process is starved out of accessing a buffer.
 - e.g., If several processes sleep while waiting for a buffer to become free, the kernel does not guarantee that they get a buffer in the order that they requested.

Reading Disk Blocks

```
algorithm bread      /* block read */  
input: file system block number  
output: buffer containing data  
{  
    get buffer for block (getblk);  
    if (buffer data valid)  
        return buffer;  
    initiate disk read;  
    sleep (event disk read complete);  
    return buffer;  
}
```

algorithm **breada** /* block read and read ahead */

input: (1) block no. for immediate read

(2) block no. for asynchronous read

output: buffer containing data for
immediate read

```
{  
  if (first block not in cache) {  
    get buffer for first block; (getblk)  
    if (buffer data not valid)  
      initiate disk read;  
  }  
  if (second block not in cache) {  
    get buffer for second block; (getblk)  
    if (buffer data valid)  
      release buffer; (brelease)  
    else  
      initiate disk read;  
  }  
}
```

```
if (first block was originally  
in cache) {  
  read first block; (bread)  
  return buffer;  
}  
sleep (event first buffer  
contains valid data);  
return buffer;  
}
```

Algorithm for Block Read Ahead

Read Ahead Algorithm

- If the first block is not in the buffer, the kernel invokes the disk driver to read that block.
- If the second block is not in the buffer, the kernel instructs the disk driver to read it asynchronously.
- Then the process goes to sleep awaiting the event that the I/O is completed on the first block.
- When the process awakens, it returns the buffer for the first block, and does not care when the I/O for second block completes.
- When the I/O for the second block completes, the interrupt handler recognizes that the I/O was asynchronous and releases the buffer (algorithm *brelease*).

Algorithm for Writing a Disk Block

```
algorithm bwrite    /* block write */
input:  buffer
output: none
{
    initiate disk write;
    if (I/O synchronous) {
        sleep(event I/O complete);
        release buffer (brelse);
    } else if (buffer marked for delayed write)
        mark buffer to put at head of free list;
}
```

Writing a Disk Block

- If the write is *synchronous*, the calling process goes to sleep awaiting I/O completion and release the buffer when is awakens.
- If the write is *asynchronous*, the kernel starts the disk write but does not wait for the write to complete.
- If the kernel does a “*delayed write*,” it marks the buffer accordingly, releases the buffer using algorithm *breise*, and continues without scheduling I/O.
- The kernel writes the block to disk before another process can reallocate the buffer to another block.
(scenario 3 of *getblk*)

Asynchronous Write and Delayed Write

- When doing an **asynchronous write**, the kernel starts the disk operation immediately but does not wait for its completion.
- For a “**delayed write**,” the kernel puts off the physical write to disk as long as possible.
 - it marks the buffer “**old**” and writes the block to disk asynchronously.
 - the interrupt handler release the buffer, using *brelse*.
- The kernel must prevent interrupts in any procedure that manipulates the buffer free list because the interrupt handler can invoke *brelse*.

Advantages and Disadvantages of Buffer Cache

- Advantages:
 - Use of the buffer cache reduce the amount of disk traffic, thereby increasing overall system throughput and decreasing response time.
 - The buffer algorithms help insure file system integrity.
- Disadvantages:
 - For delayed write, the system is vulnerable to crashes that leave disk data in an incorrect state.
 - Use of the buffer cache requires an extra data copy when reading and writing to and from user processes.

End