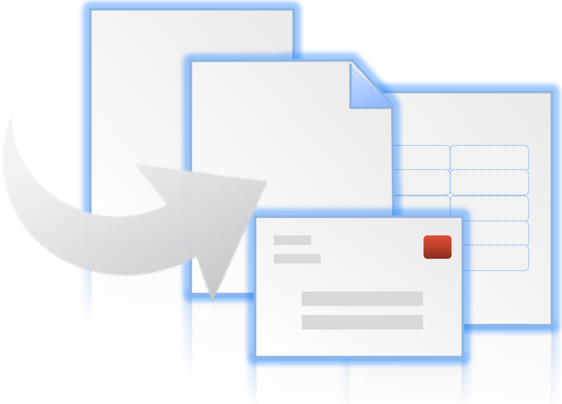
ADVANCED OPERATING SYSTEMS

UNIT 2 FILE AND DIRECTORY I/O

BY

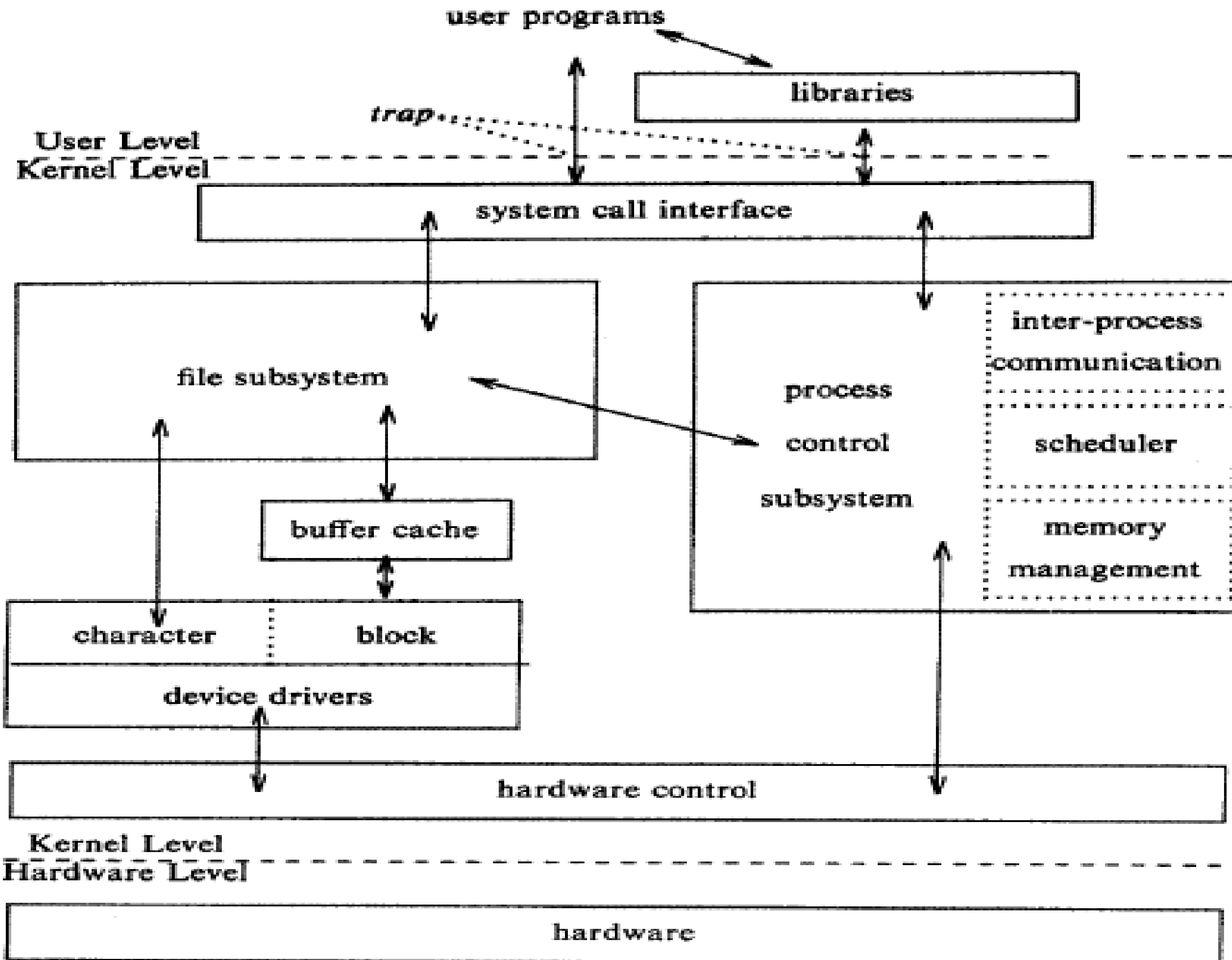
MR.PRASAD SAWANT



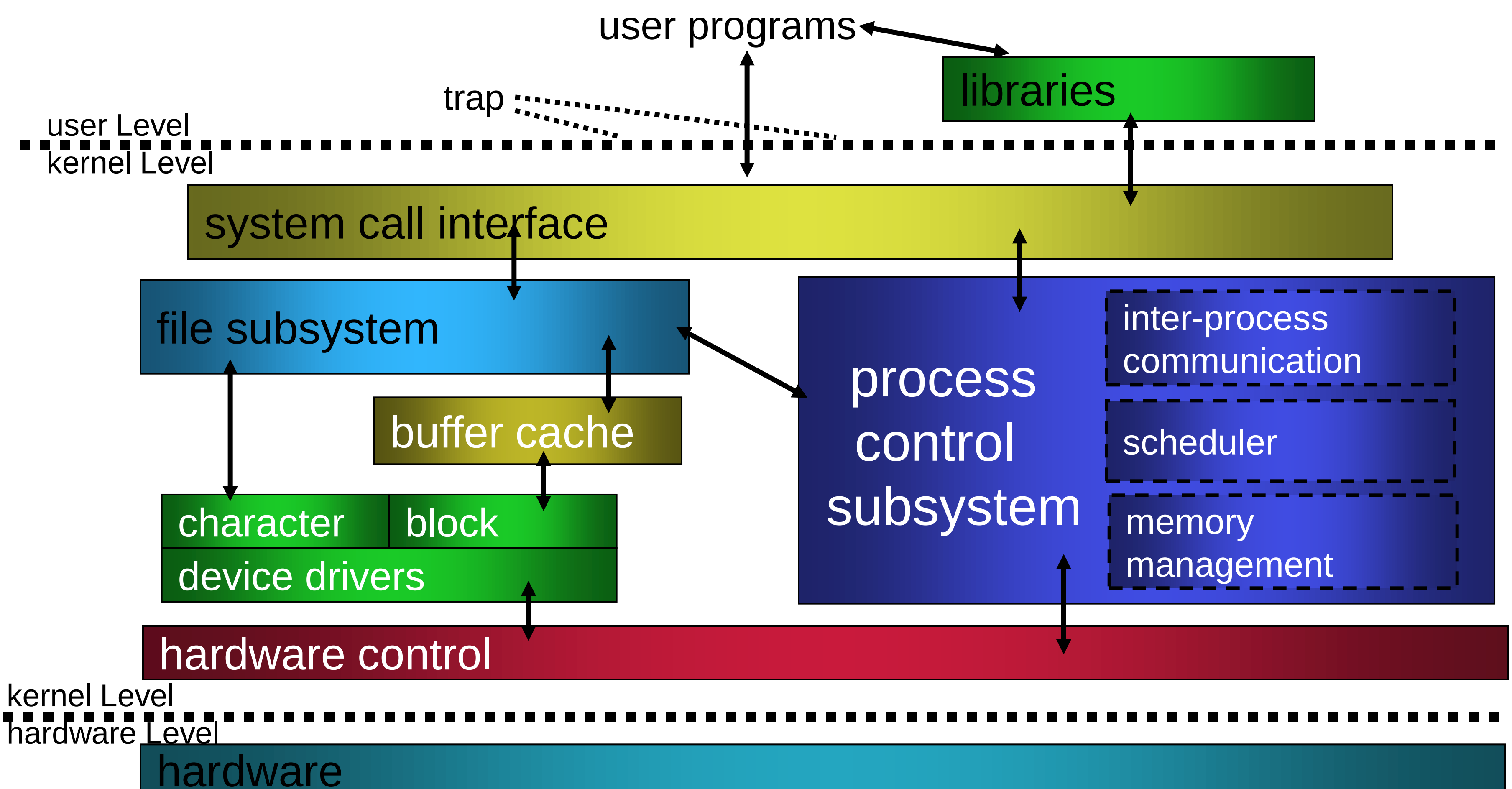
Prof.Prasad Sawant ,Assitiant Professor ,Dept. Of CS PCCCS Chichwad

OUT LINE OF SESSION

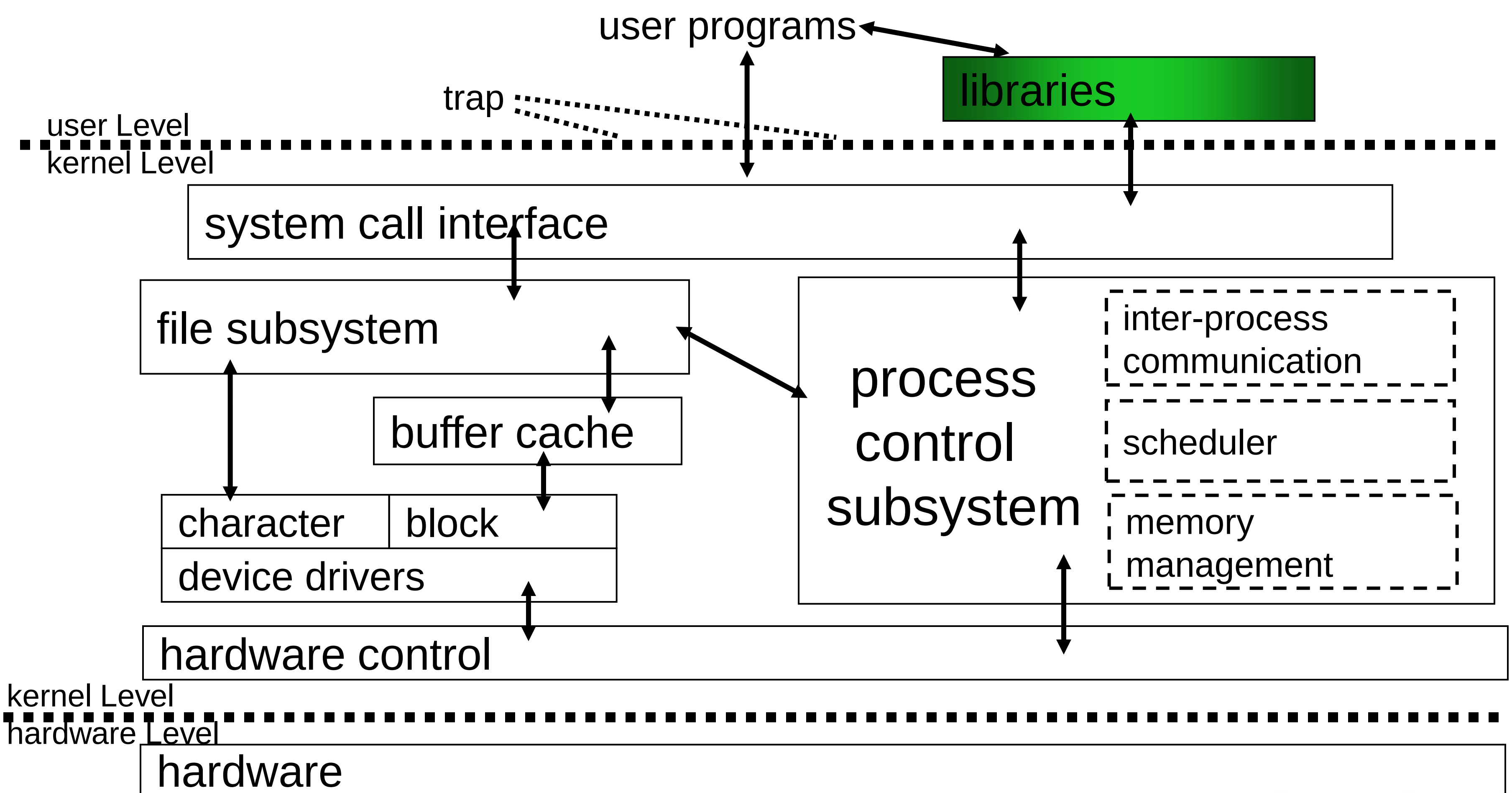
1. Buffer headers
2. structure of the buffer pool
3. scenarios for retrieval of a buffer
4. reading and writing disk blocks
5. Inodes
6. structure of regular file
7. Open
8. Read
9. Write
10. Lseek
11. Pipes
12. close
13. dup


user programs
libraries
trap
User Level
Kernel Level
system call interface
file subsystem
inter-process
communication
process
control
subsystem
scheduler
memory
management
buffer cache
character
block
device drivers
hardware control
Kernel Level
Hardware Level
hardware

ARCHITECTURE OF THE UNIX



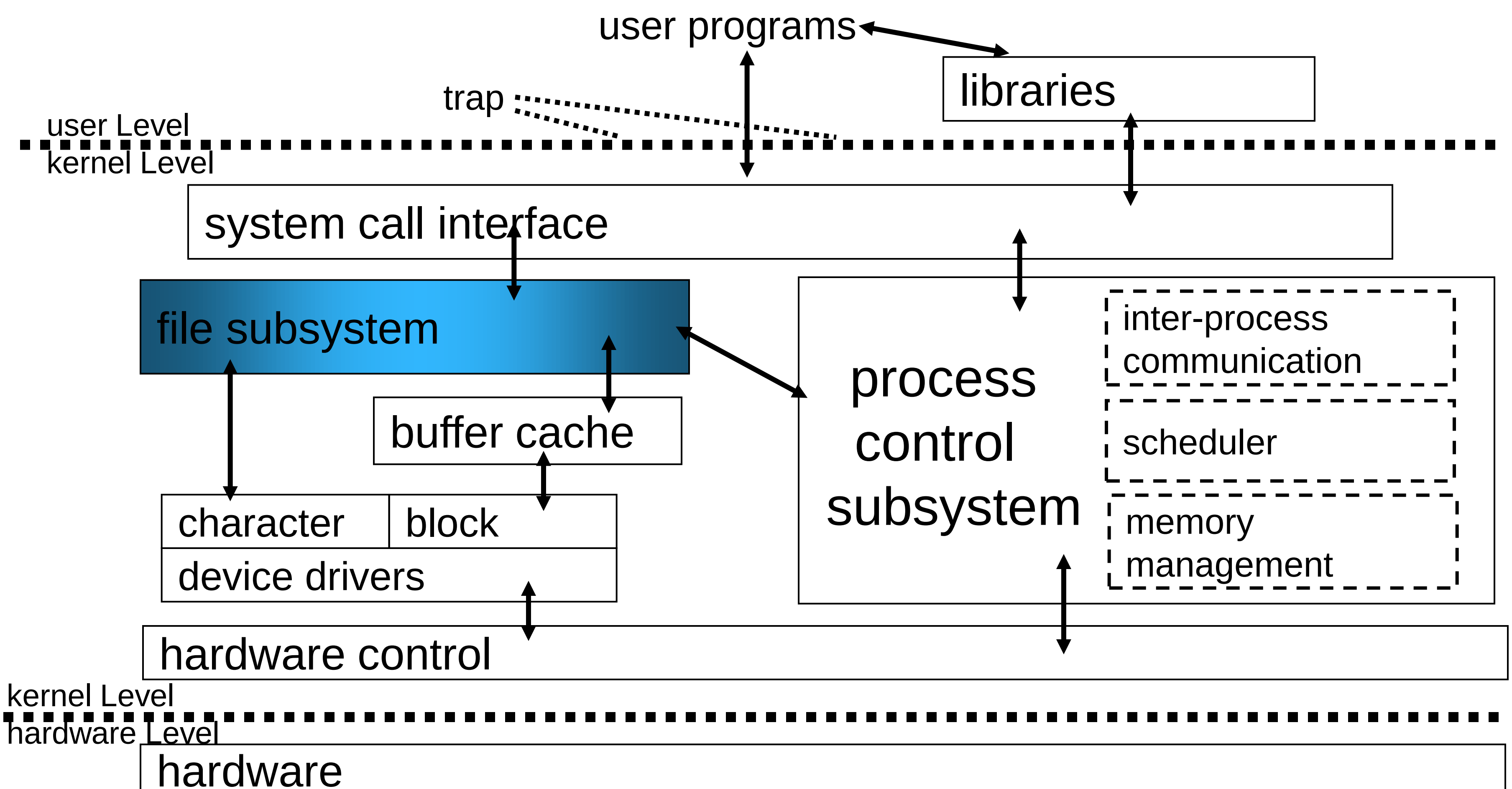
LIBRARIES (1)


user programs
libraries
trap
user Level
kernel Level
system call interface
file subsystem
buffer cache
character
block
device drivers
process
control
subsystem
inter-process
communication
scheduler
memory
management
hardware control
kernel Level
hardware Level
hardware

LIBRARIES (2)

1. Make system calls look like ordinary function call.
2. Map these function call to the primitives needed to enter the OS.

FILE SUBSYSTEM (1)

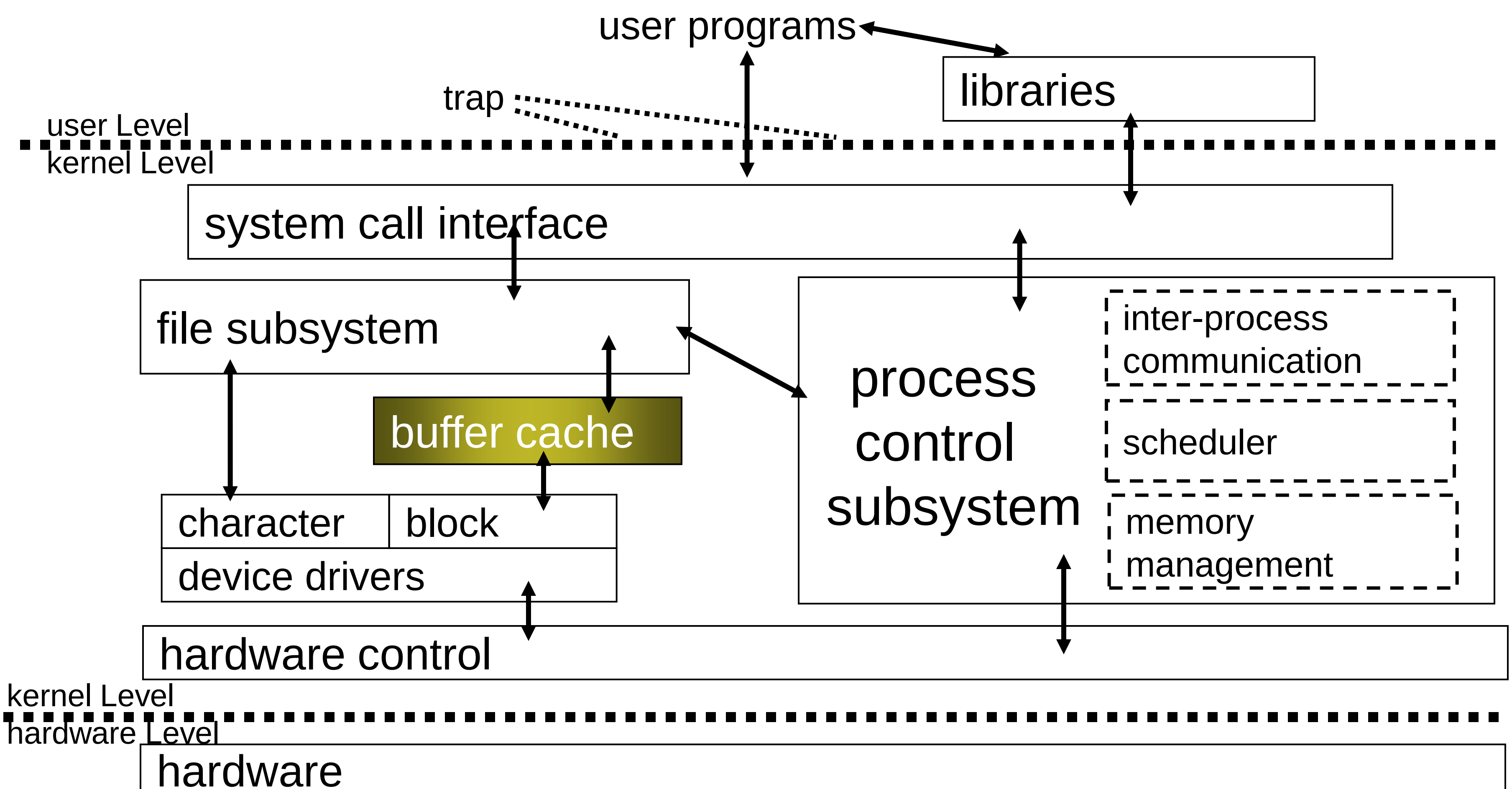


FILE SUBSYSTEM (2)

1. Managing files
2. Allocating file space
3. Administering free space
4. Controlling access to files
5. Retrieving data for users

1. Interact with set of system calls
    1. *open, close, read, write, state, chown, chmod …*

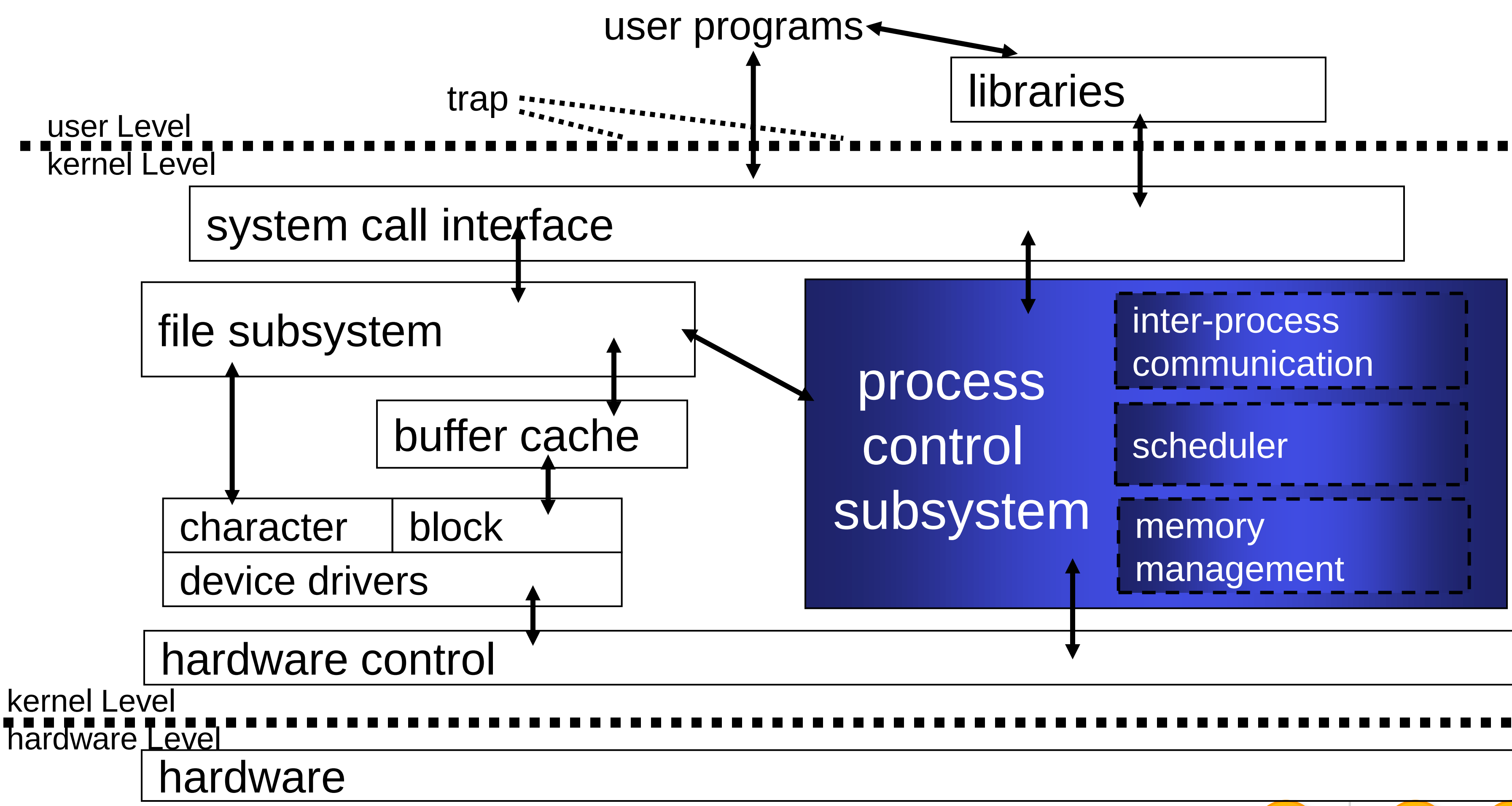
BUFFERING MECHANISM (1)



BUFFERING MECHANISM (2)

Interact with block I/O device drivers to initiate data transfer to and from kernel.

PROCESS CONTROL SUBSYSTEM (1)



PROCESS CONTROL SUBSYSTEM (2)

Responsible for process synchronization.

Interprocess communication (IPC)

Memory management

Process scheduling

Interact with set of system calls

- *fork, exec, exit, wait, brk, signal …*

PROCESS CONTROL SUBSYSTEM (3)

Memory management module

- Control the allocation of memory

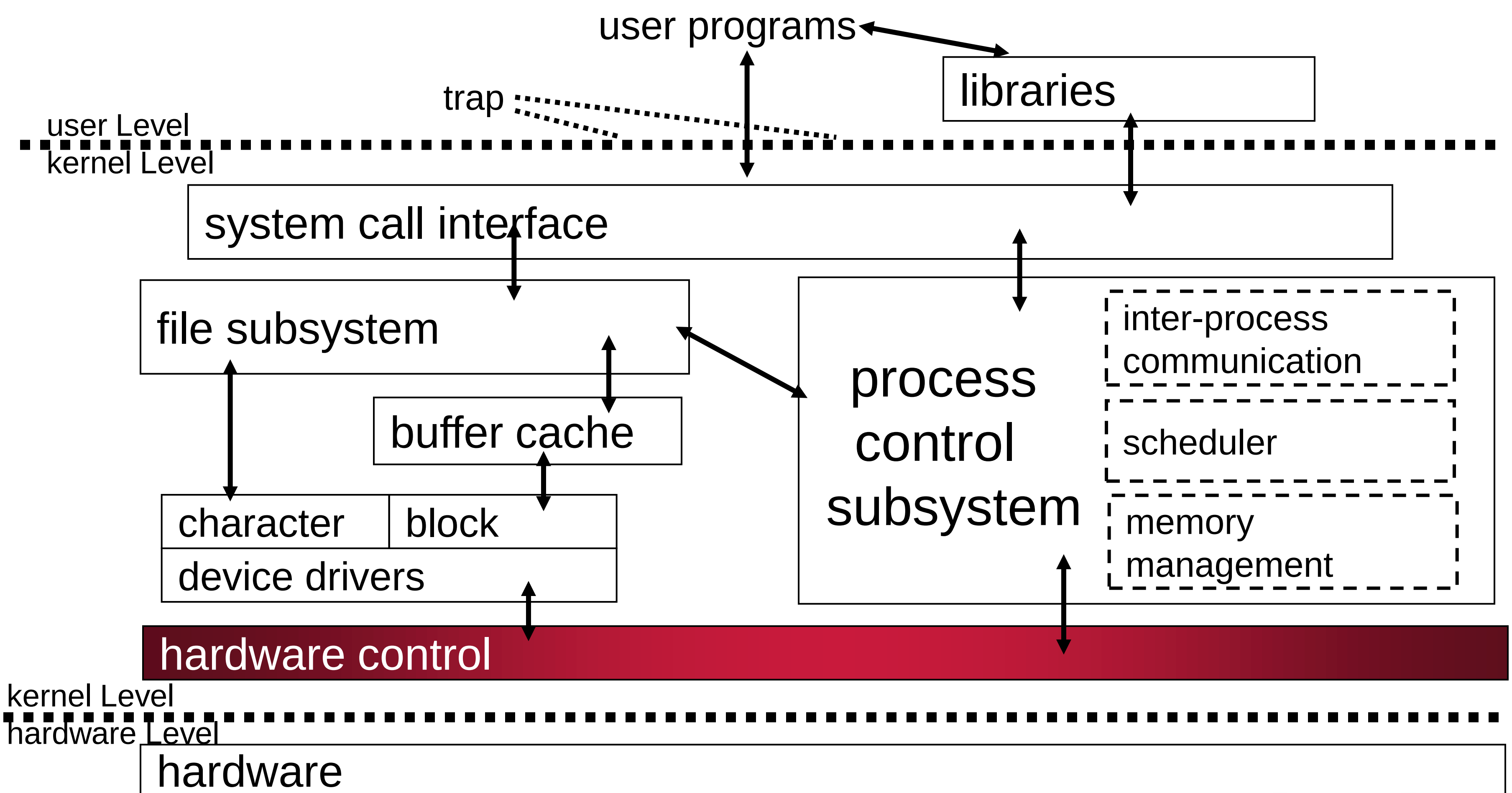
Scheduler module

- Allocate the CPU to processes

Interprocess communication

- There are several forms.

HARDWARE CONTROL (1)



HARDWARE CONTROL (2)

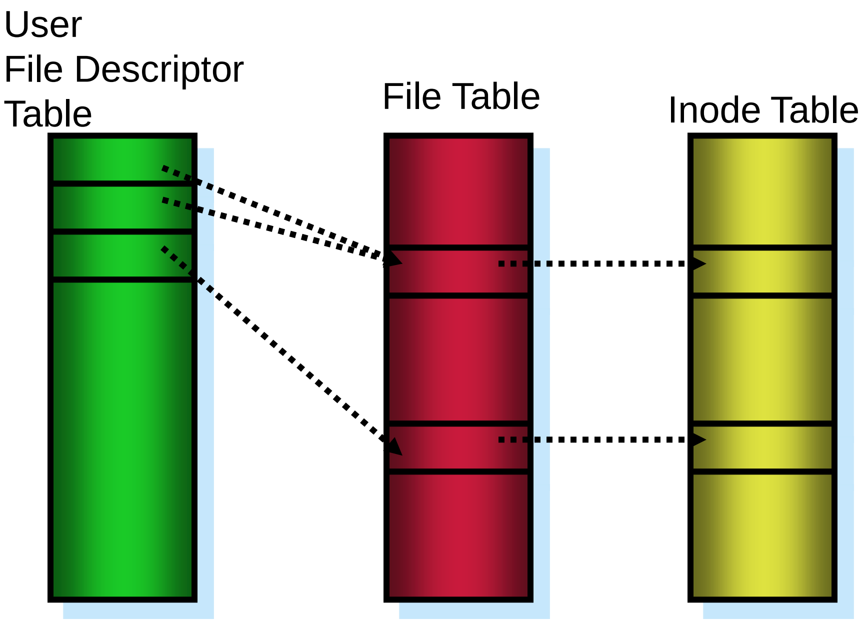
Responsible for handling interrupts and for communicating with the machine.

AN OVERVIEW OF THE FILE SUBSYSTEM

inode (index node)

- a description of the disk layout of the file data and other information

FILE ACCESS



FILE SYSTEM LAYOUT

boot block

- Be needed to boot the system

super block

- Describes the state of a file system

inode list

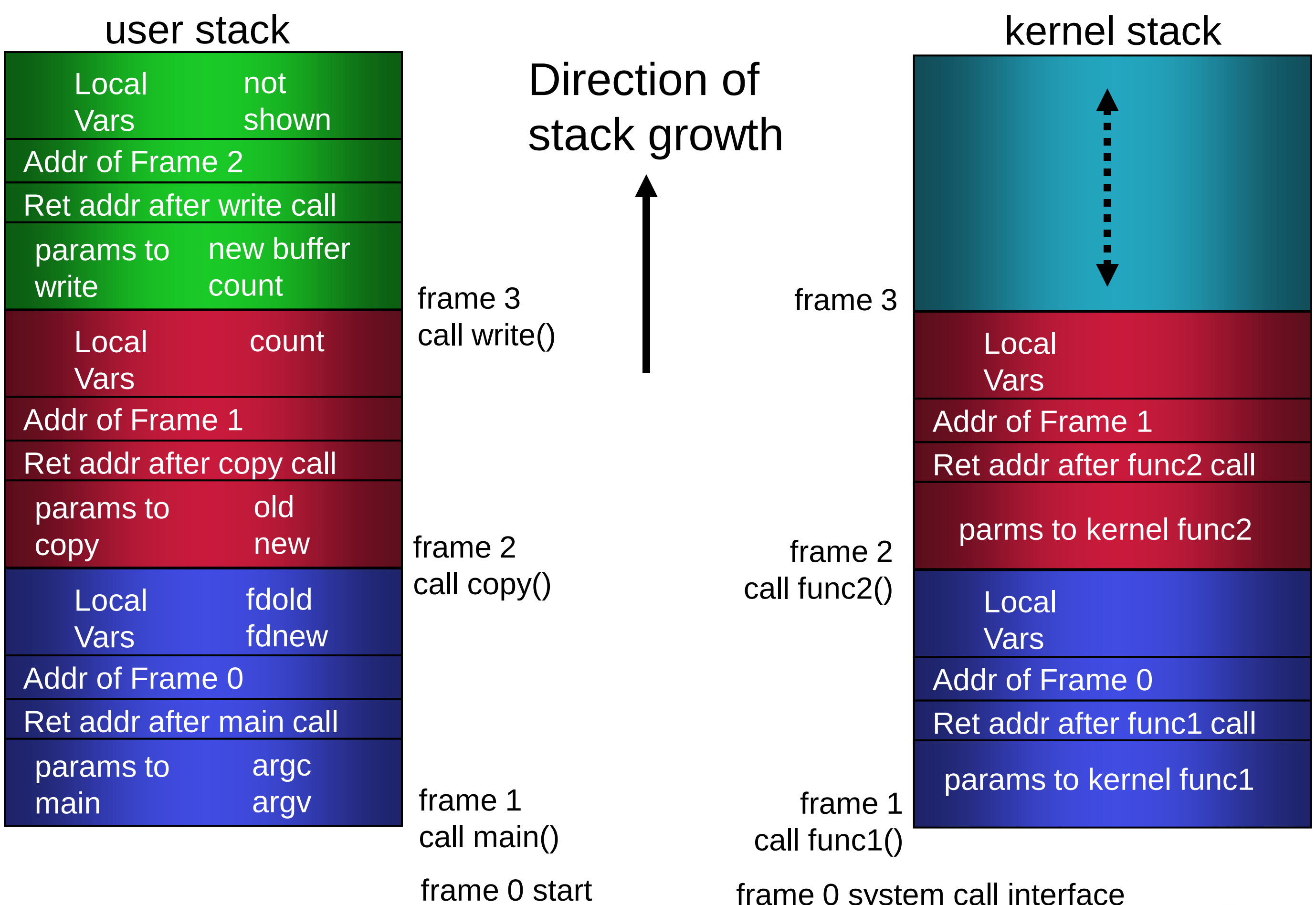
- a list of inodes

data block

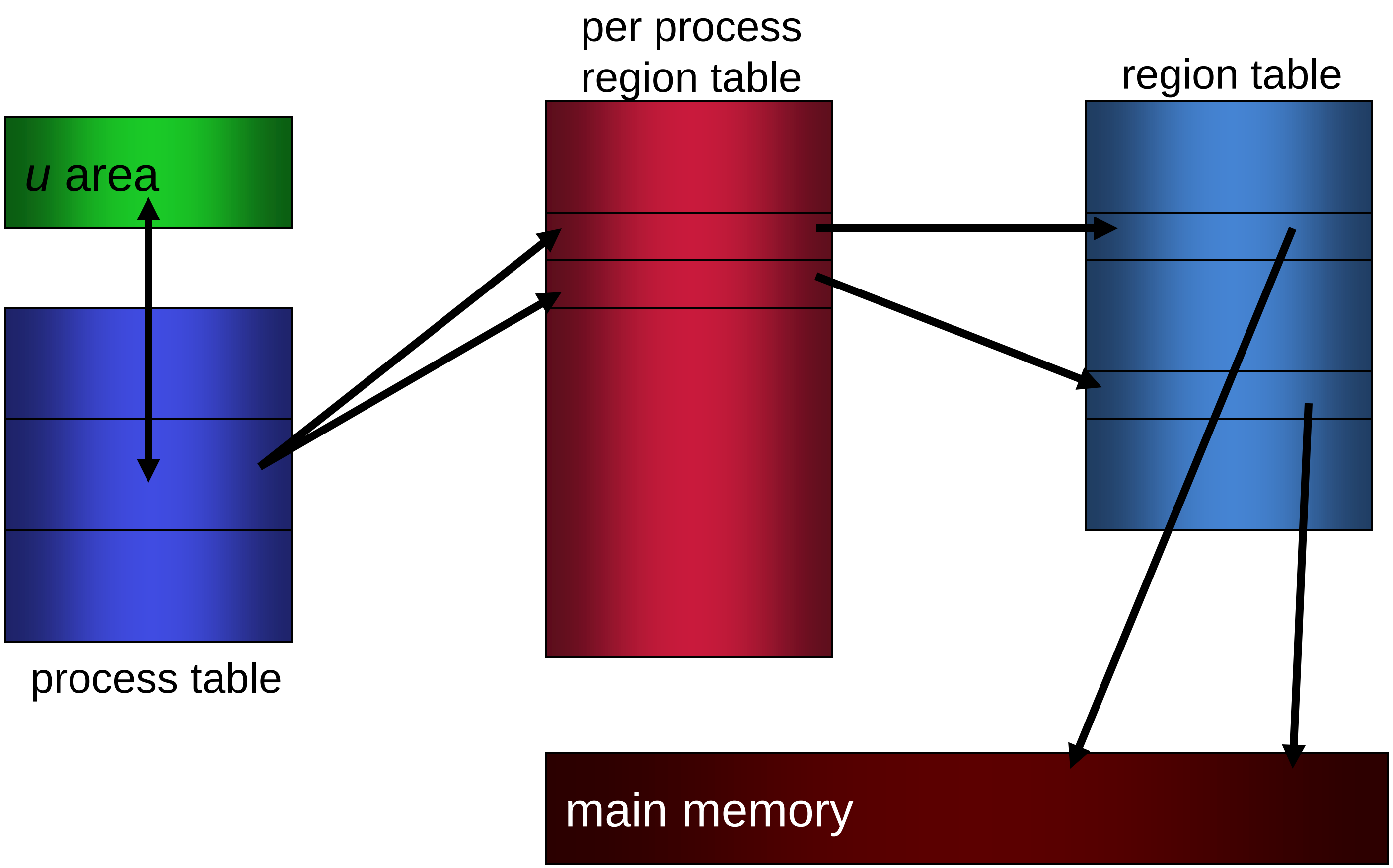
- contain file data and administrative data

```
#include  <fcntl.h>
char buffer[2048];
int version = 1;          /* Chapter 2 explains this */

main(argc, argv)
        int argc;
        char *argv[];
{
        int fdold, fdnew;

        if (argc != 3)
        {
                printf("need 2 arguments for copy program\n");
                exit(1);
        }
        fdold = open(argv[1], O_RDONLY);    /* open source file read only */
        if (fdold == -1)
        {
                printf("cannot open file %s\n", argv[1]);
                exit(1);
        }
        fdnew = creat(argv[2], 0666);    /* create target file rw for all */
        if (fdnew == -1)
        {
                printf("cannot create file %s\n", argv[2]);
                exit(1);
        }
        copy(fdold, fdnew);
        exit(0);
}

copy(old, new)
        int old, new;
{
        int count;

        while ((count = read(old, buffer, sizeof(buffer))) > 0)
                write(new, buffer, count);
}
```

USER AND KERNEL STACK FOR COPY PROGRAM



DATA STRUCTURES FOR PROCESSES



PROCESS TABLE

State, ownership, event descriptor set

u pointer (address)

U AREA

- Pointer to the process table slot
- System call parameters
- File descriptor
- Internal I/O information
- Current directory and current root
- Process and file size limits

REGION TABLE

Text / Data

Shared / Private

PROCESS STATES

User mode

- currently executing

Kernel mode

- currently executing

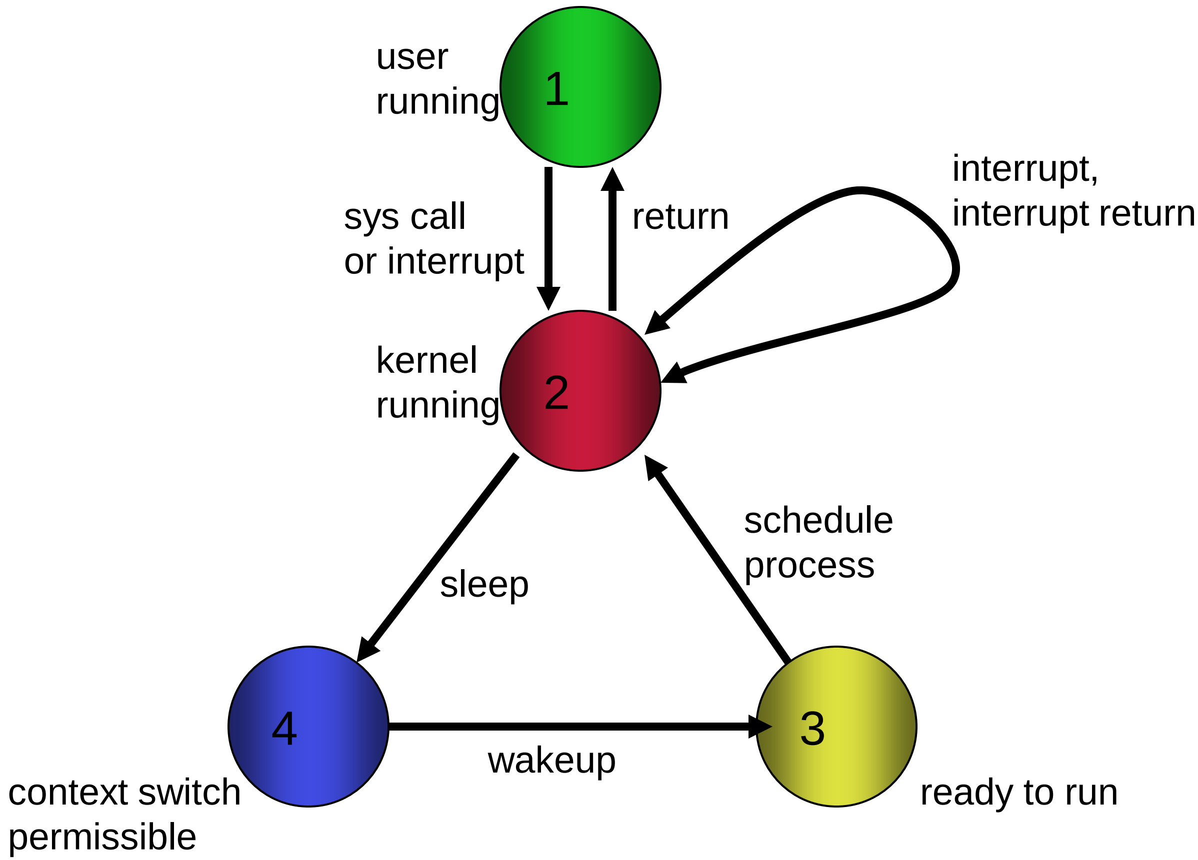
Ready to run

- soon as the scheduler chooses it.

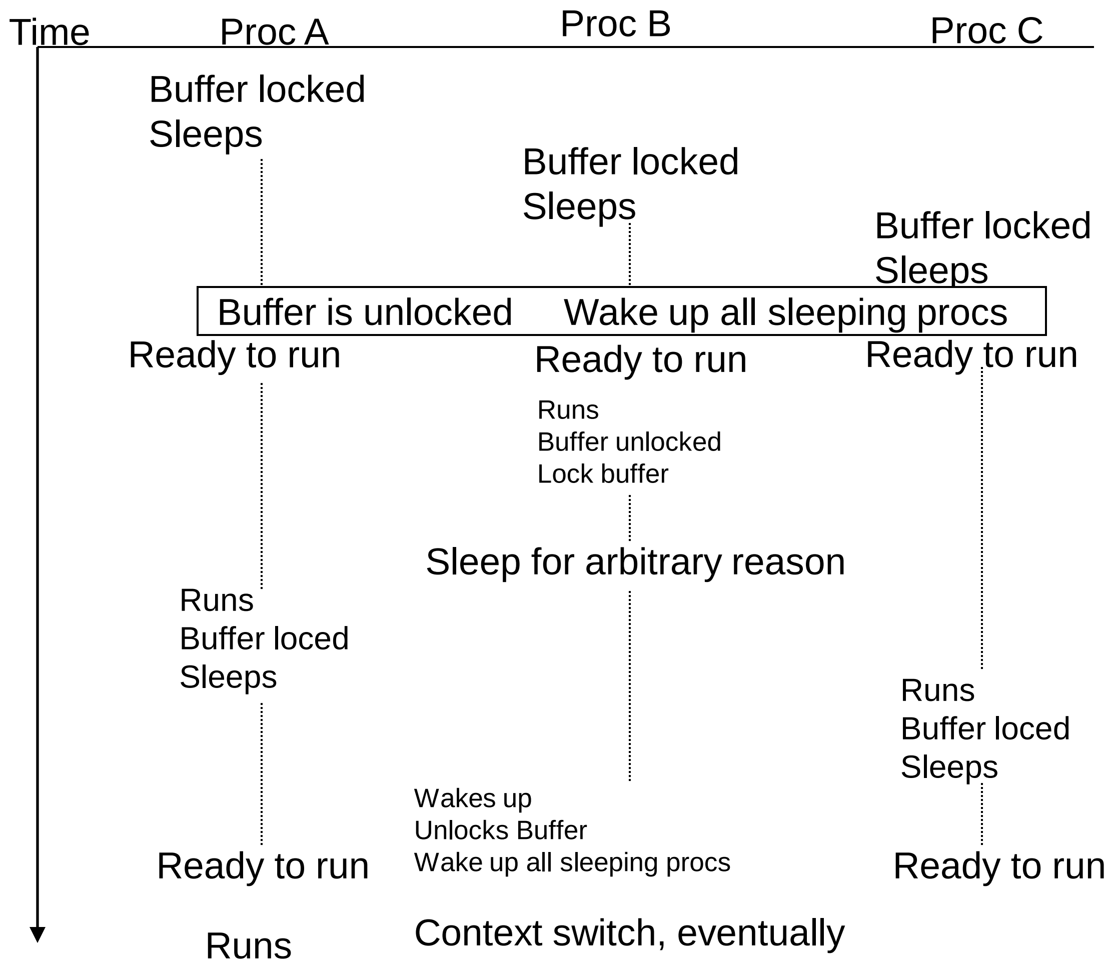
Sleeping

- no longer continue executing
- eg) waiting for I/O to complete.

PROCESS TRANSITION



MULTIPLE PROCESES SLEEPING ON A LOCK



Prof.Prasad Sawant ,Assitiant Professor ,Dept. Of CS PCCCS Chichwad

Session Contents

- Buffer Headers
- Structure of the Buffer Pool
- Scenarios for Retrieval of a Buffer
- Reading and Writing Disk Blocks
- Advantages & Disadvantages of the Buffer Cache

THE BUFFER CACHE

Kernel could read & write directly,but …

- System response time & throughput be poor

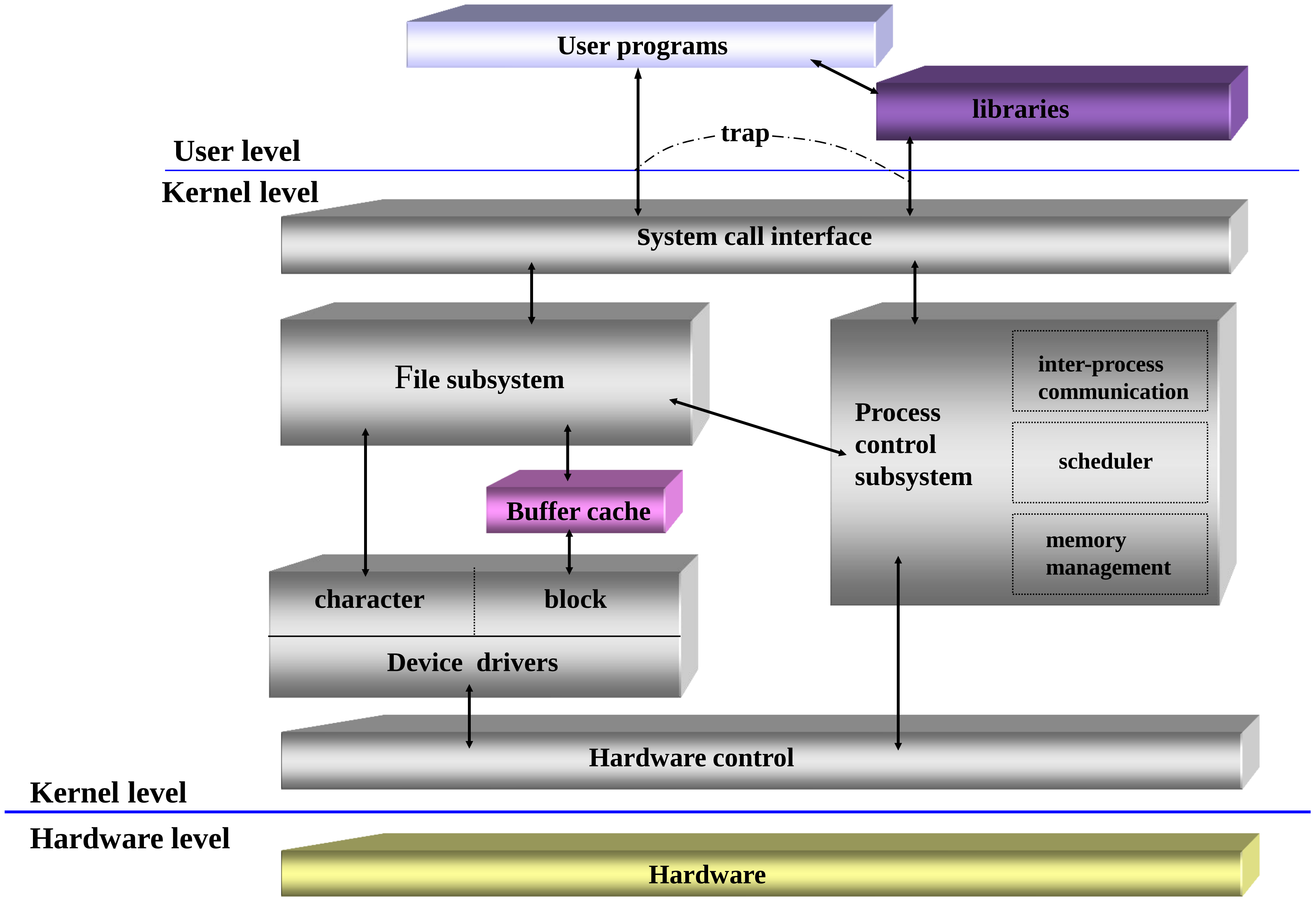
Kernel minimize the frequency of disk access

- By keeping a pool of internal data buffers

Transmit data between application programs and the file system via the buffer cache.

Transmit auxiliary data between higher-level kernel algorithms and the file system.

- super block – free space available on the file system
- inode – the layout of a file


User programs
libraries
trap
User level
Kernel level
System call interface
File subsystem
inter-process
communication
Process
control
subsystem
scheduler
Buffer cache
memory
management
character
block
Device drivers
Hardware control
Kernel level
Hardware level
Hardware

BUFFER HEADERS

Kernel allocates space for many buffers, during system initialization

A buffer consists of two parts

- a memory array
- buffer header

Data in logical disk block = Data in buffer

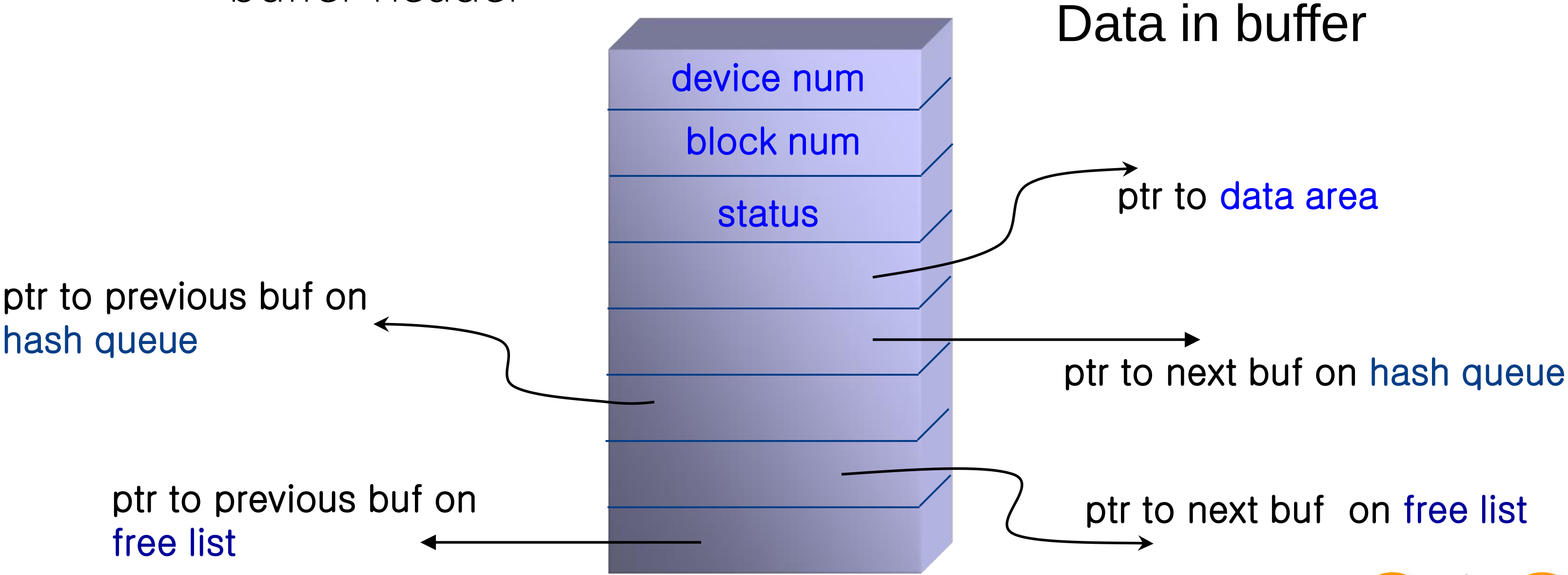


Figure 3.1 Buffer Header

device number

- logical file system number

block number

- block number of the data on disk
- Identify the buffer uniquely

Status is a combination condition

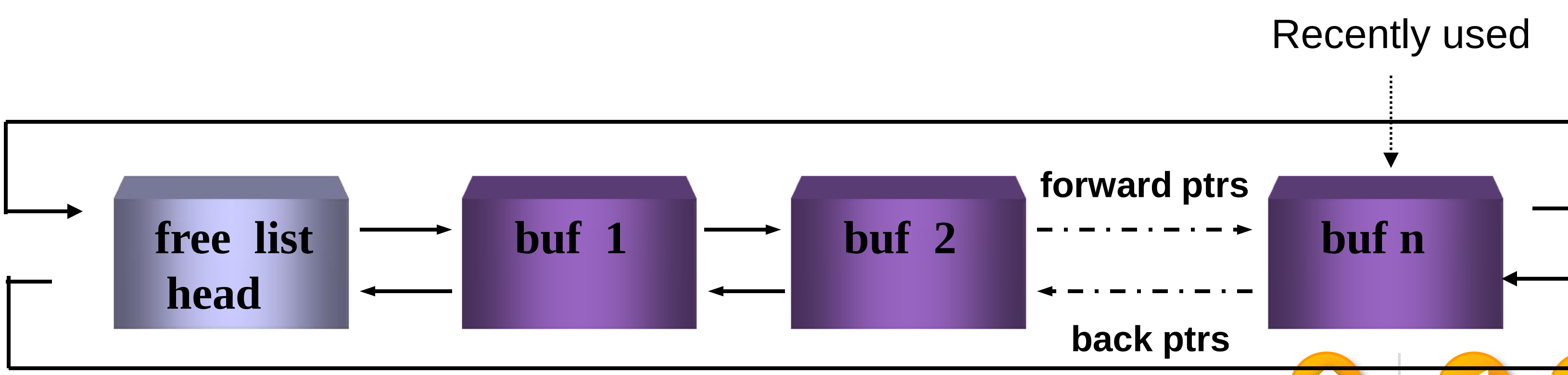
- The buffer is currently locked.
- The buffer contains valid data.
- “delayed-write” as condition
- The kernel is currently reading or writing the contents of buffer to disk.
- A process is currently waiting for the buffer to become free.

STRUCTURE OF THE BUFFER POOL

Kernel cache data in buffer pool according to a ***LRU***

A *free list* of buffer

- LRU order
- doubly linked circular list
- Kernel take a buffer from the head of the free list.
- When returning a buffer, attaches the buffer to the tail.



STRUCTURE OF THE BUFFER POOL

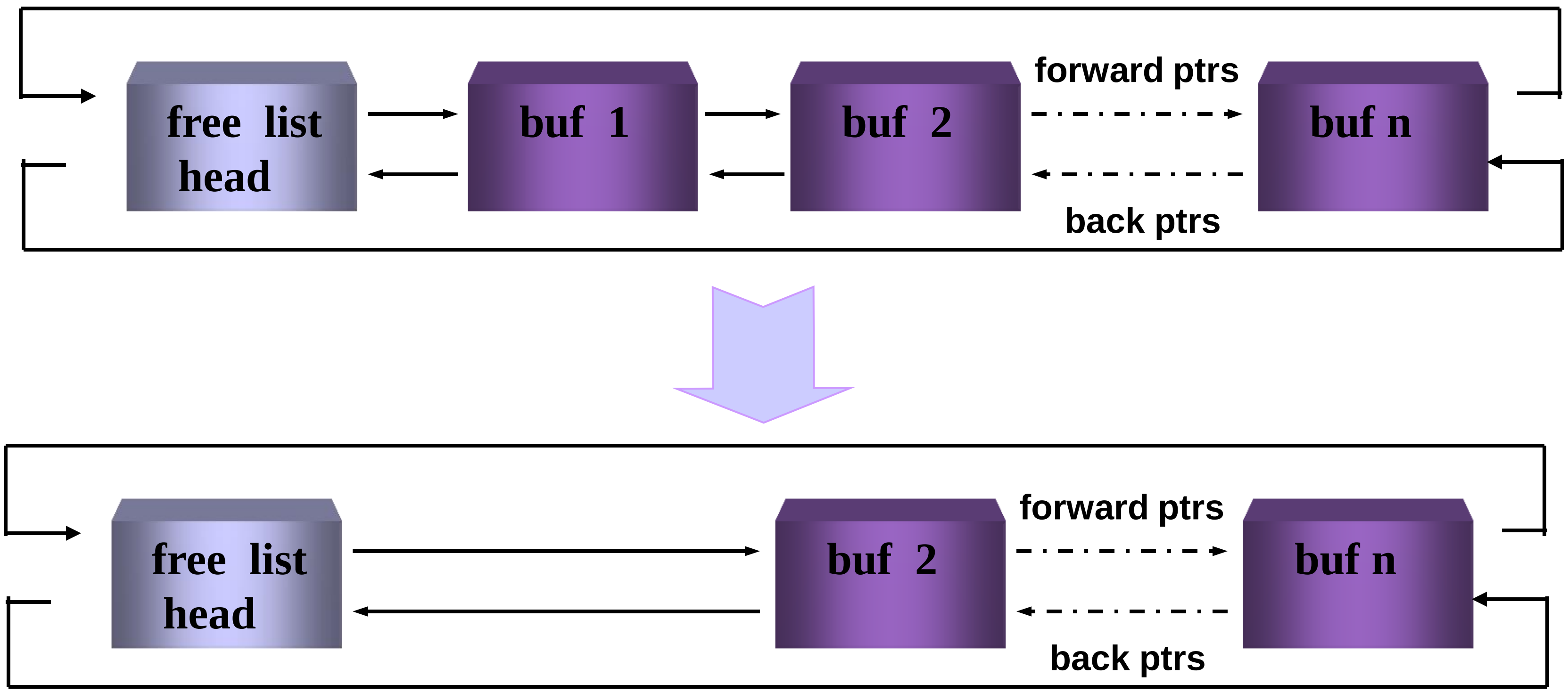


Figure 3.2. Free list of Buffers

STRUCTURE OF THE BUFFER POOL

When the kernel accesses a disk block

- Organize buffer into separate queue
  - ***hashed*** as a function of the device and block number
- Every disk block exists only on hash queue and only once on the queue

Buffer is always on a hash queue, but is may or may not be on the free list

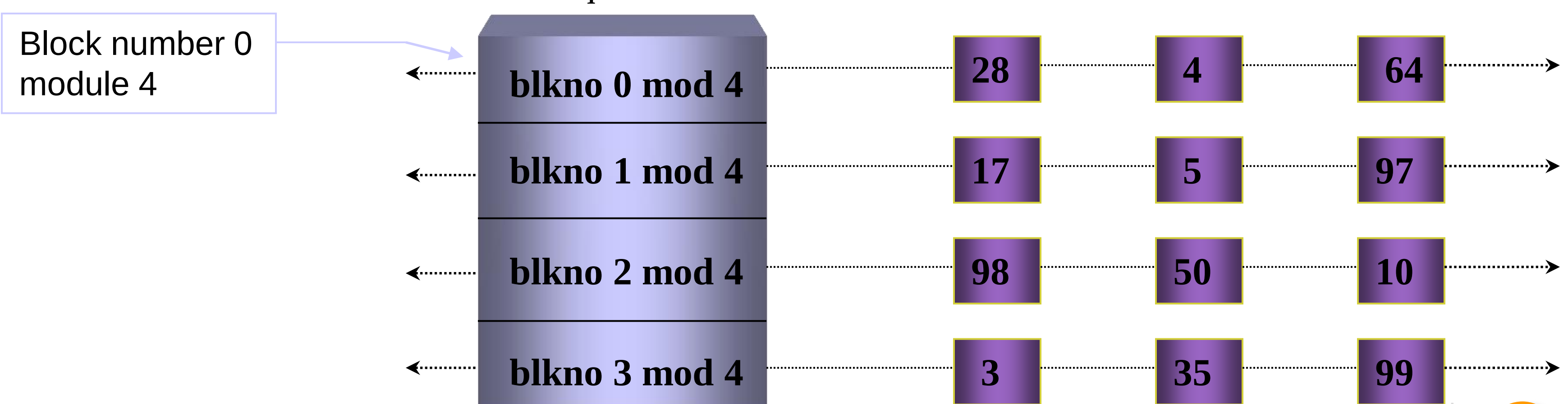


Figure 3.3 Buffers on the Hash Queues

SCENARIOS FOR RETRIEVAL OF A BUFFER

- Algorithm determine logical device # and block #
- The algorithms for reading and writing disk blocks use the algorithm *getblk*
  - Kernel finds the block on its hash queue
    - buffer is free.
    - buffer is currently busy.
  - Kernel cannot find the block on the hash queue
    - kernel allocates a buffer from the free list.
    - In attempting to allocate a buffer from the free list, finds a buffer on the free list that has been marked “delayed write”.
    - free list of buffers is empty.

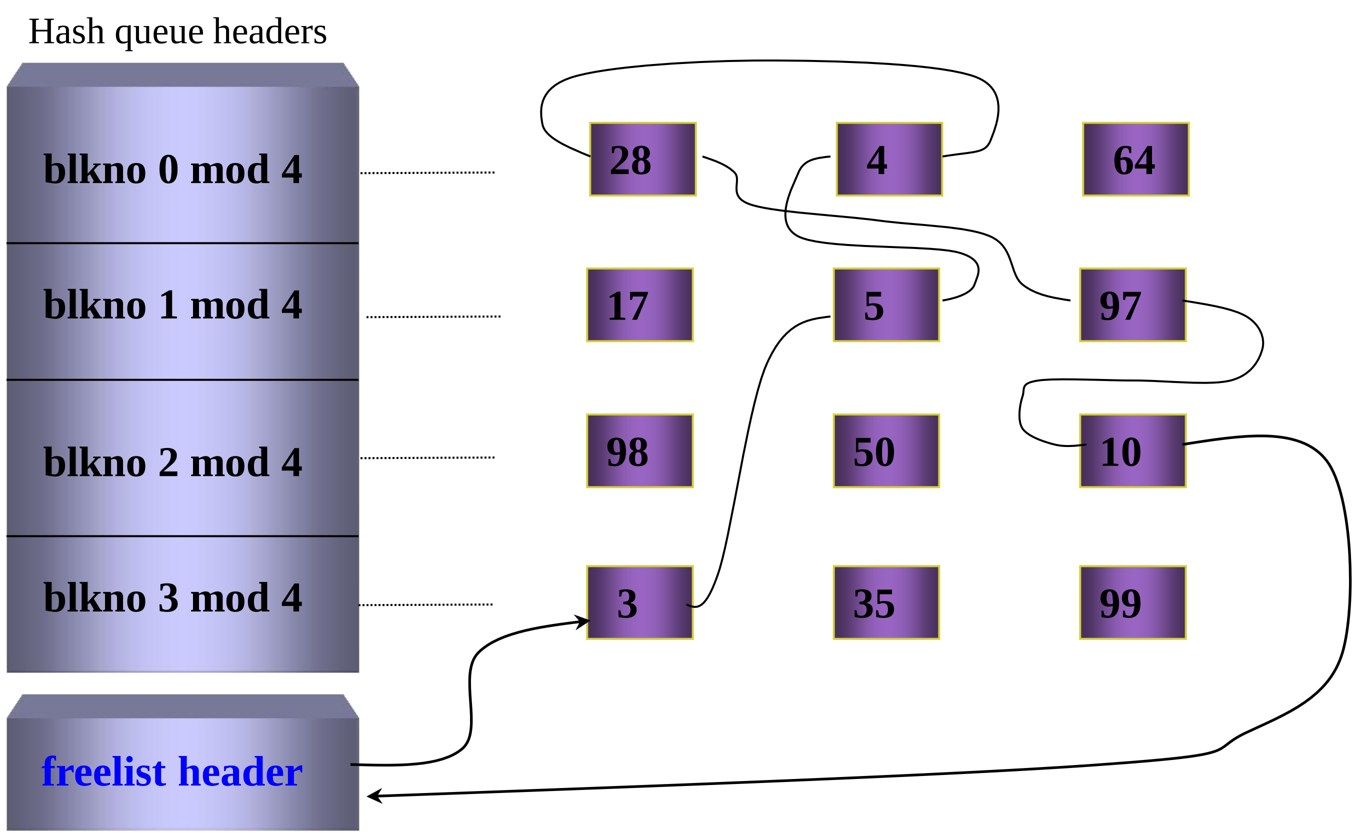
```
Algorithm getblk
Input: file system number
       block number
Output: locked buffer that can now be used for block
{
  while(buffer not found)
  {
    if(block in hash queue)
    {
      if(buffer busy)        /* scenario 5 */
      {
         sleep(event buffer becomes free);
         continue;           /* back to while loop */
      }
      make buffer busy;      /* scenario 1 */
      remove buffer from free list;
      return buffer;
    }
    else           /* block not on hash queue */
    {
      if(there are no buffers 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;
    }
  }
}
```

LINUX

```
struct  buffer_head * getblk(kdev_t dev, int block, int size)
{
            struct buffer_head * bh;
            int isize;
repeat:     bh = get_hash_table(dev, block, size);
                        if (bh) {
                        if (!buffer_dirty(bh)) {
                                    bh->b_flushtime = 0;
                        }
                        return bh;
            }
            isize = BUFSIZE_INDEX(size);

get_free:   bh = free_list[isize];
            if (!bh)
                        goto refill;
            remove_from_free_list(bh);

            init_buffer(bh, dev, block, end_buffer_io_sync, NULL);
            bh->b_state=0;
            insert_into_queues(bh);
            return bh;

refill:     refill_freelist(size);
            if (!find_buffer(dev,block,size))
                        goto get_free;
            goto repeat;
}
```

SCENARIOS FOR RETRIEVAL OF A BUFFER

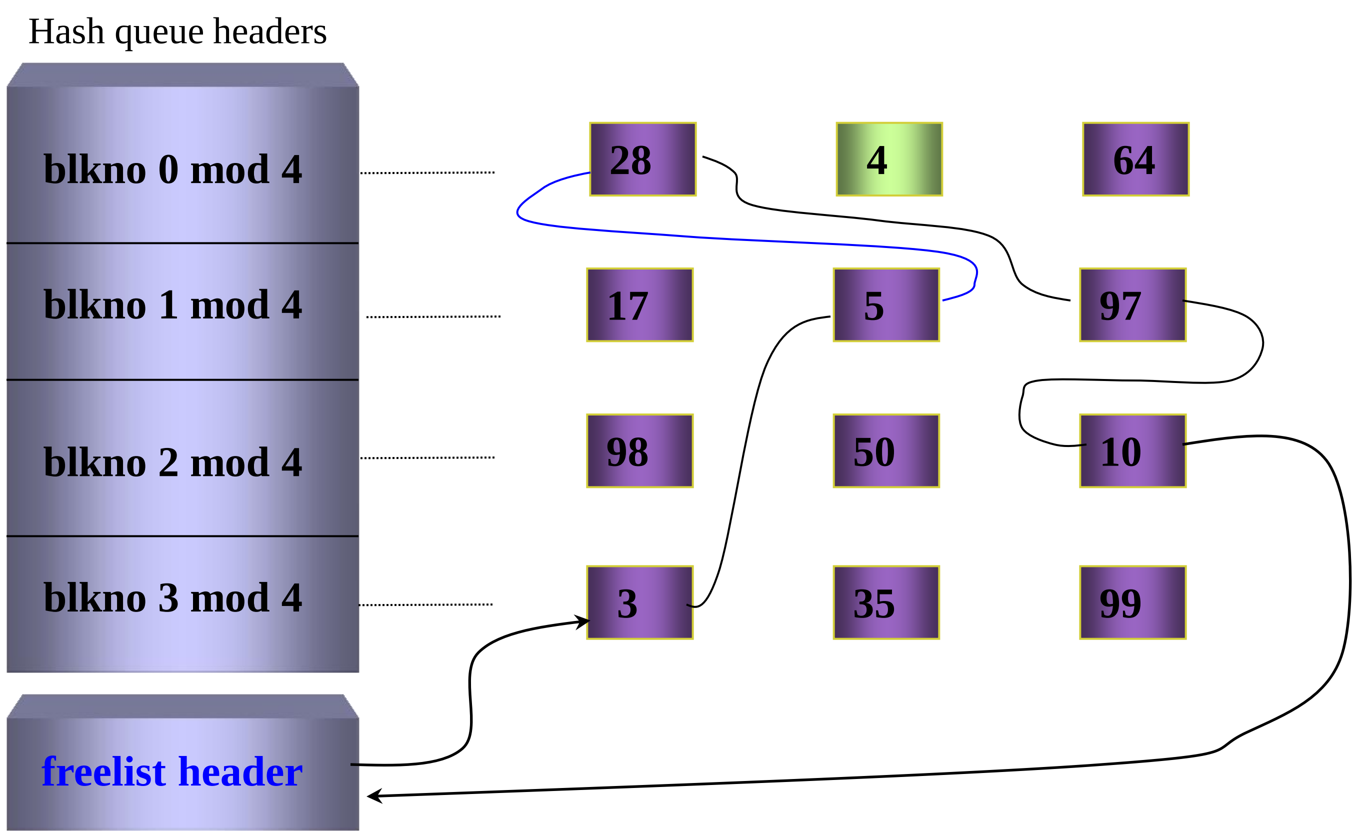
FIRST SCENARIO IN FINDING A BUFFER: BUFFER ON HASH QUEUE (A)



(a) Search for Block 4 on First Hash Queue

SCENARIOS FOR RETRIEVAL OF A BUFFER

FIRST SCENARIO IN FINDING A BUFFER: BUFFER ON HASH QUEUE (B)



(a) Remove Block 4 from Free list

SCENARIOS FOR RETRIEVAL OF A BUFFER

ALGORITHM FOR RELEASING A BUFFER

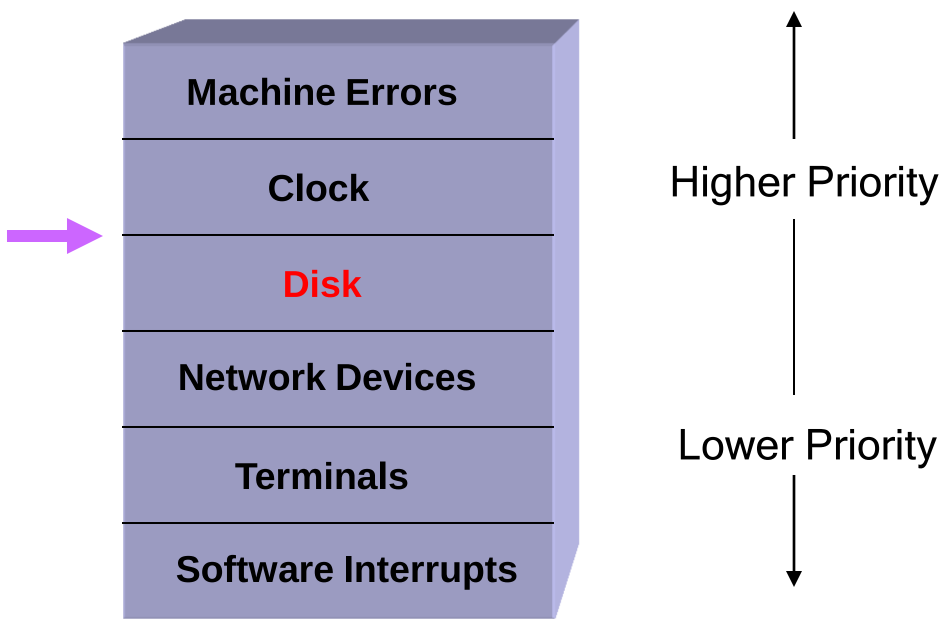
```
Algorithm  brelse
Input: locked  buffer
{
        wakeup all process: event, waiting for any buffer to become free;
        wakeup all process: event, waiting for this buffer to become free;
        raise processor execution level to block interrupts;
        if (buffer contents valid and buffer not old)
                enqueue buffer at end of free list
        else
                enqueue buffer at beginning of free list
        lower processor execution level to allow interrupts;
        unlock(buffer);
}
```

SCENARIOS FOR RETRIEVAL OF A BUFFER

ALGORITHM FOR RELEASING A BUFFER

When manipulating linked lists, block the disk interrupt

- Because handling the interrupt could corrupt the pointers

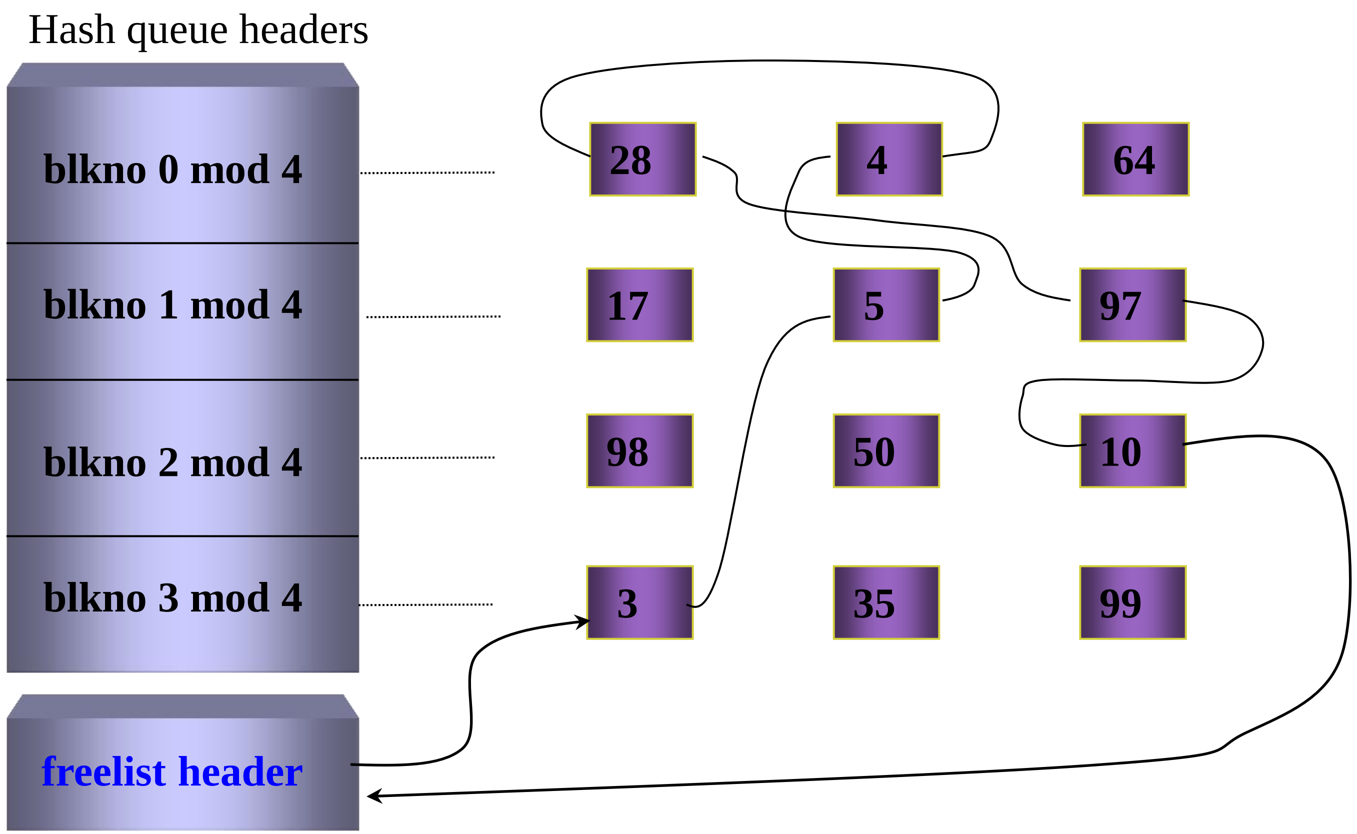


Typical Interrupt Levels

```
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);
            continue;            /* back to while loop */
         }
         make buffer busy;     /* scenario 1 */
         remove buffer from free list;
         return buffer;
      }
      else           /* block not on hash queue */
      {
         if(there are no buffers 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;
      }
   }
}
```

SCENARIOS FOR RETRIEVAL OF A BUFFER

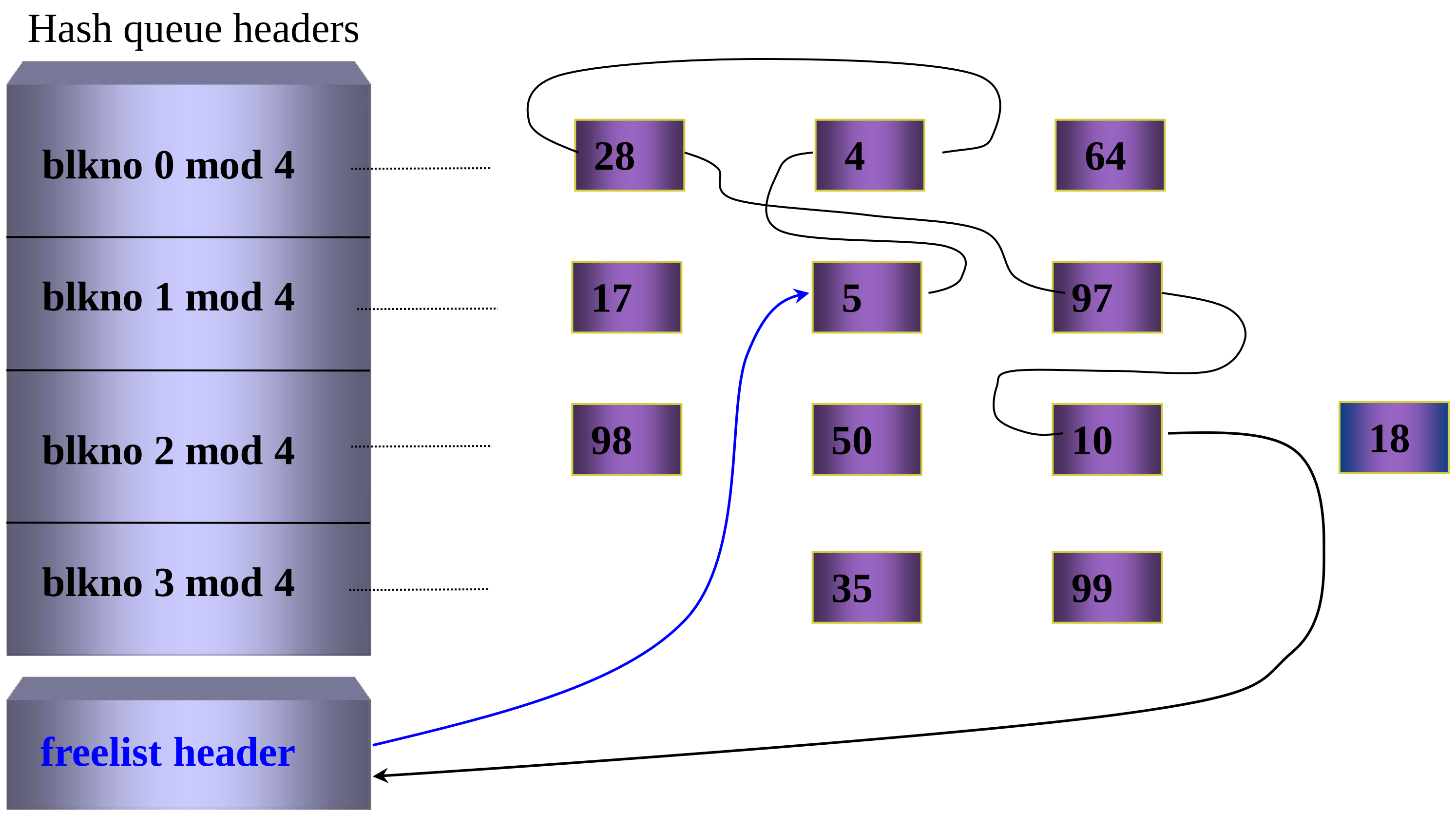
SECOND SCENARIO FOR BUFFER ALLOCATION (A)



(a) Search for Block 18 – Not in Cache

SCENARIOS FOR RETRIEVAL OF A BUFFER

SECOND SCENARIO FOR BUFFER ALLOCATION (B)

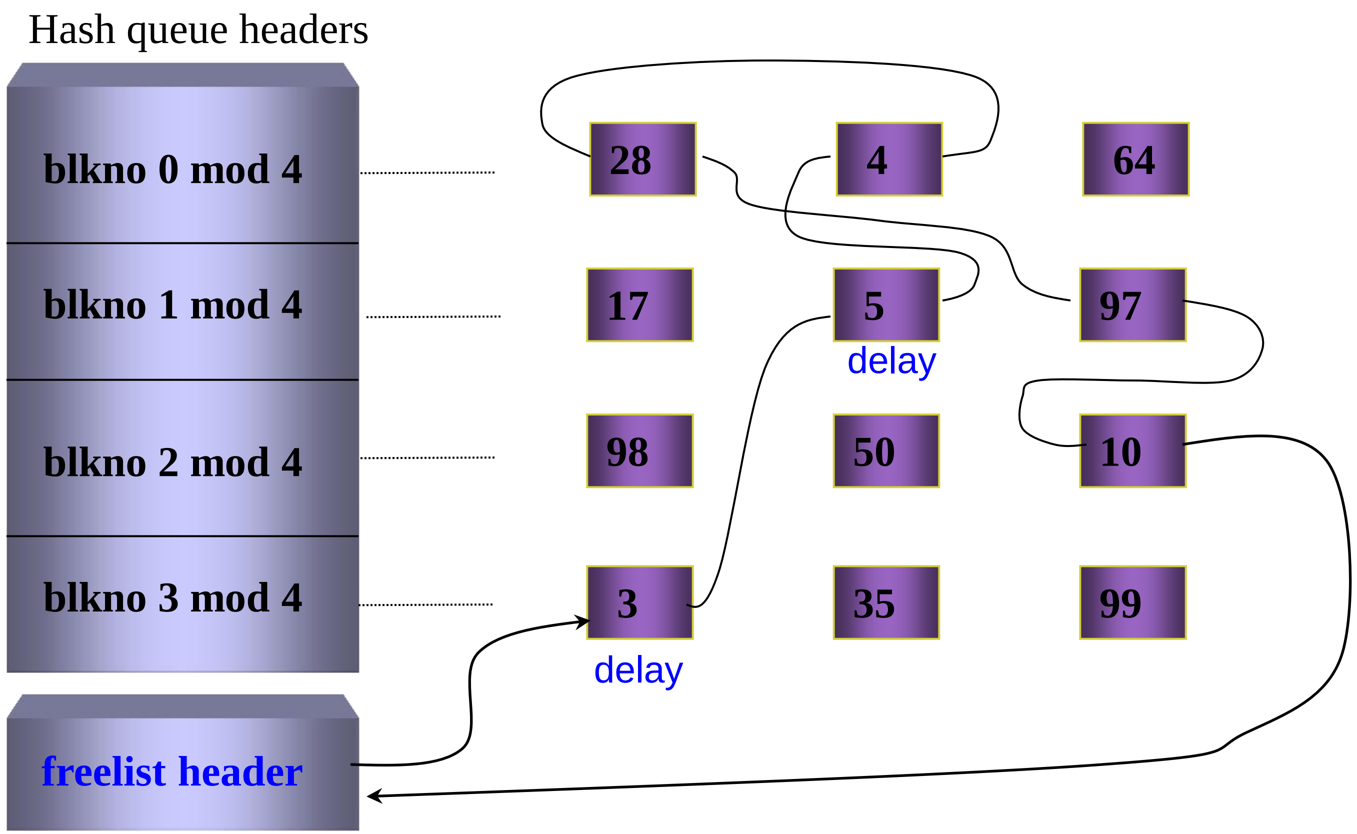


(b) Remove First Block from Free list, Assign to 18

```
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);
        continue;            /* back to while loop */
      }
      make buffer busy;      /* scenario 1 */
      remove buffer from free list;
      return buffer;
    }
    else        /* block not on hash queue */
    {
      if(there are no buffers 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;
    }
  }
}
```

SCENARIOS FOR RETRIEVAL OF A BUFFER

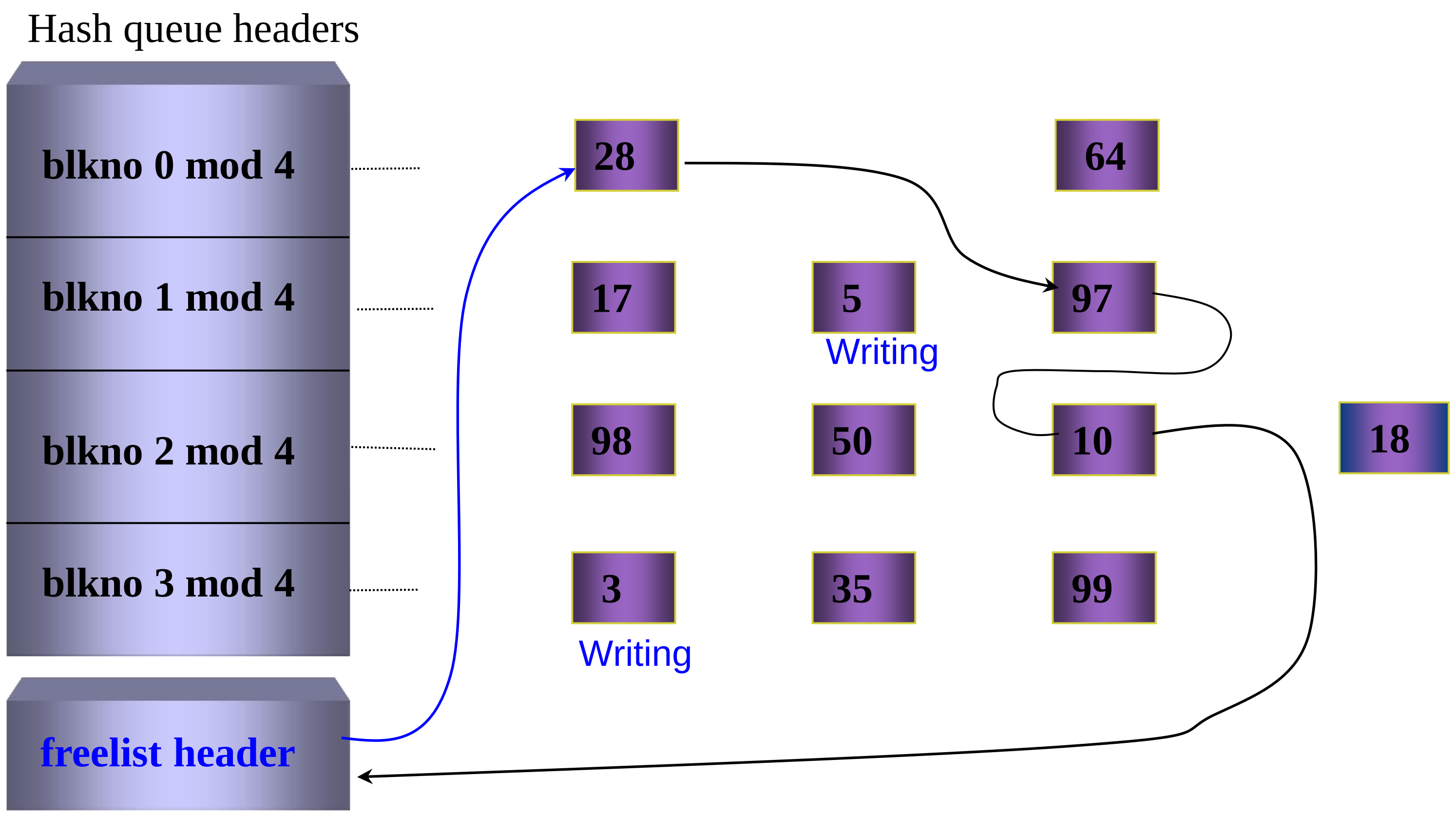
THIRD SCENARIO FOR BUFFER ALLOCATION (A)



(a) Search for Block 18, Delayed Write Blocks on Free List

SCENARIOS FOR RETRIEVAL OF A BUFFER

THIRD SCENARIO FOR BUFFER ALLOCATION (B)

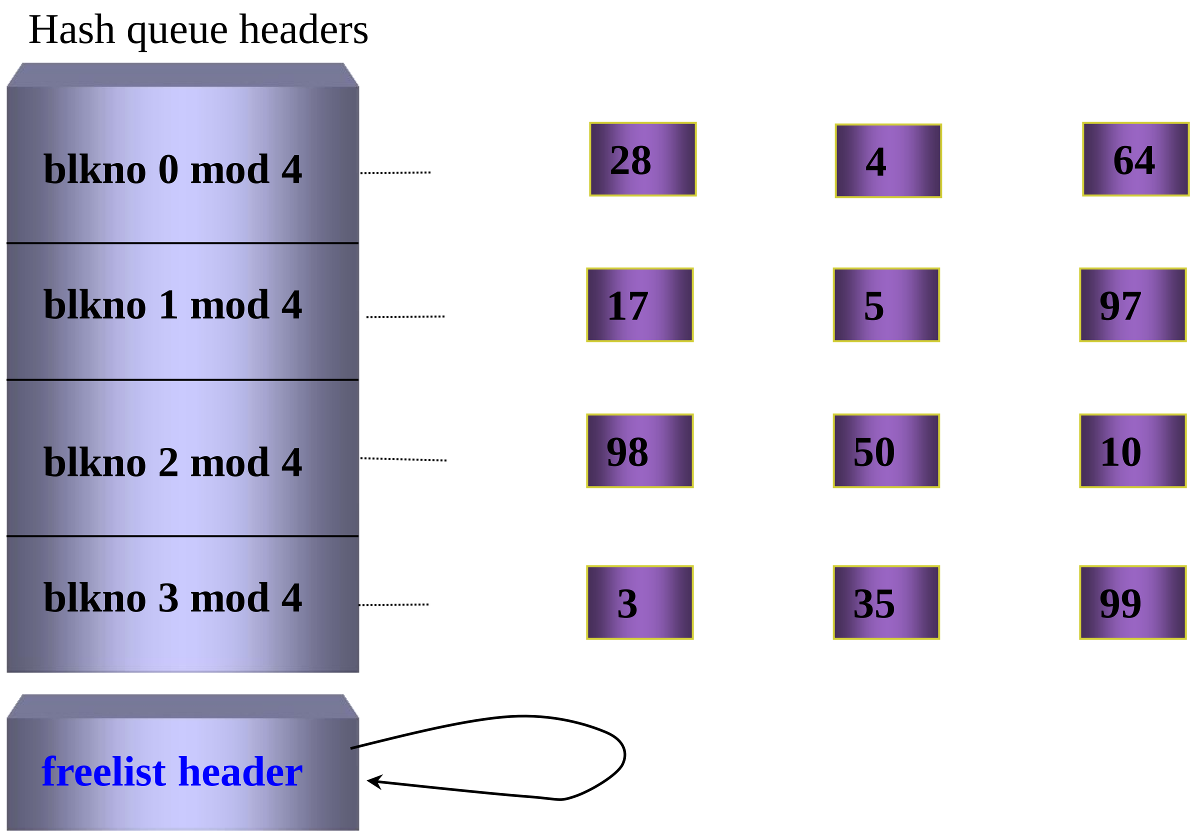


(b) Writing Blocks 3, 5, Reassign 4 to 18

```
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);
        continue;            /* back to while loop */
      }
      make buffer busy;      /* scenario 1 */
      remove buffer from free list;
      return buffer;
    }
    else          /* block not on hash queue */
    {
      if( there are no buffers 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;
    }
  }
}
```

SCENARIOS FOR RETRIEVAL OF A BUFFER

FOURTH SCENARIO FOR ALLOCATING BUFFER



Search for Block 18, Empty Free list

SCENARIOS FOR RETRIEVAL OF A BUFFER

RACE FOR FREE BUFFER

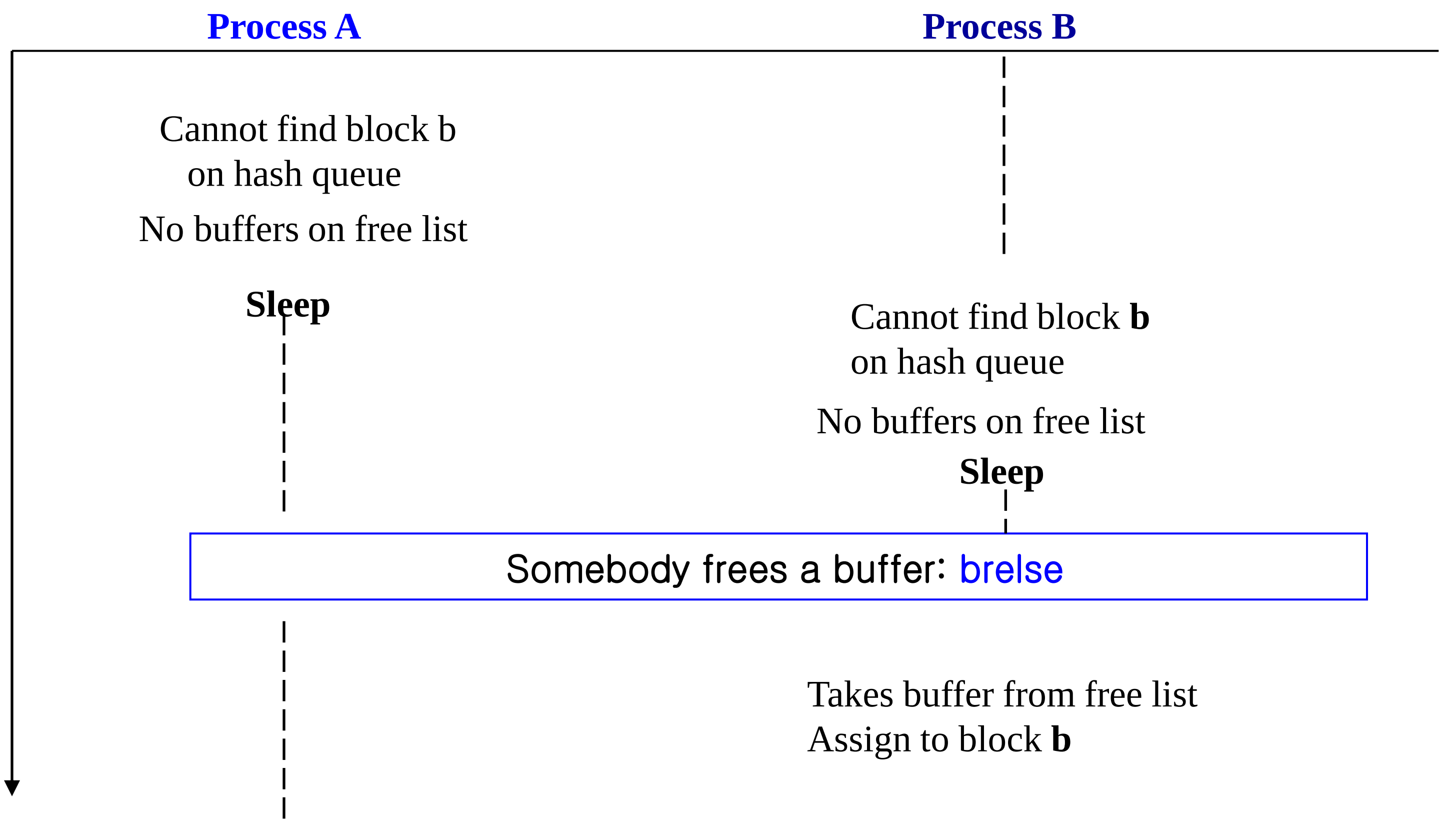


Figure 3.10. Race for Free Buffer

Algorithm **getblk**

Input: file system number

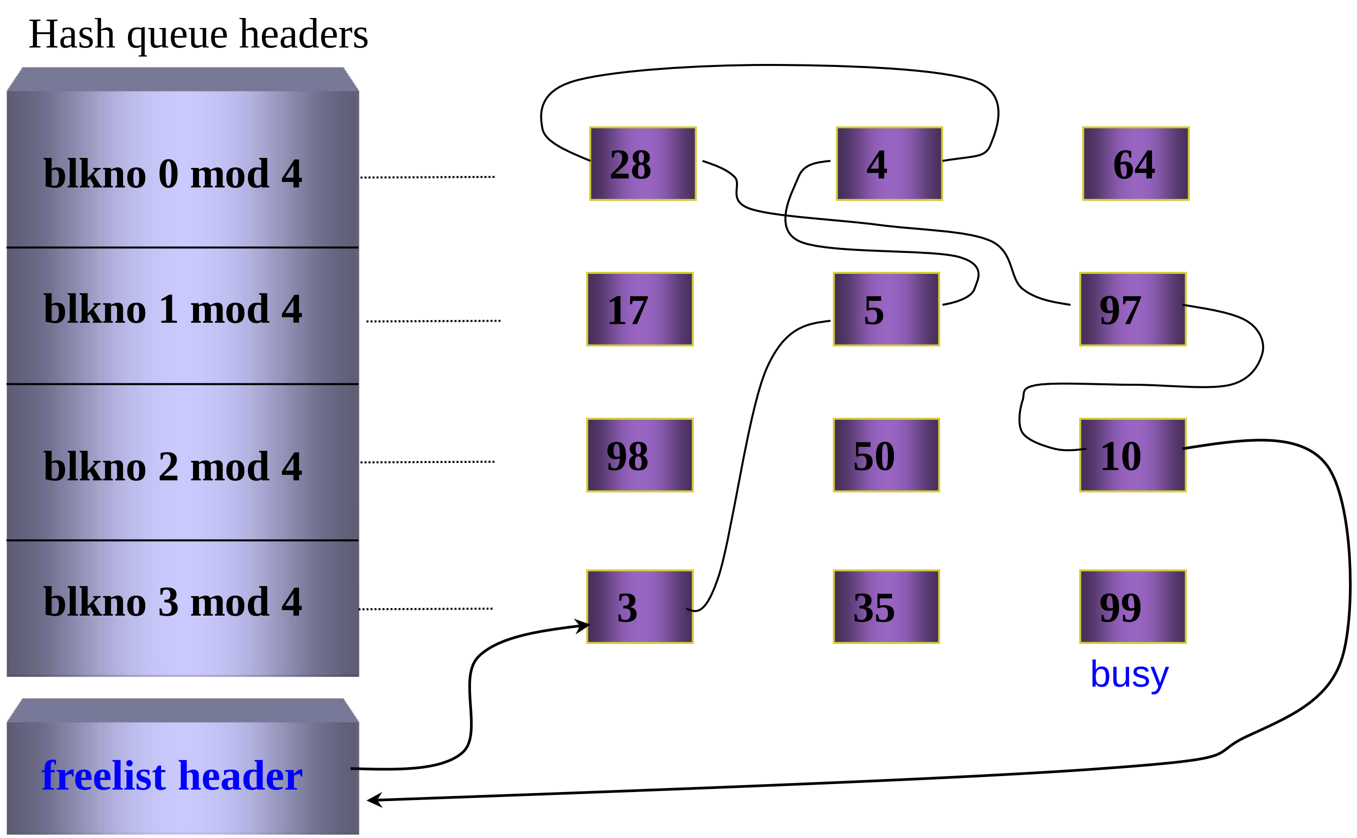
block number

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);
        continue;              /* back to while loop */
      }
      make buffer busy;        /* scenario 1 */
      remove buffer from free list;
      return buffer;
    }
    else            /* block not on hash queue */
    {
      if(there are no buffers 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;
    }
  }
}
```

SCENARIOS FOR RETRIEVAL OF A BUFFER

FIFTH SCENARIO FOR BUFFER ALLOCATION



Search for Block 99, Block busy

SCENARIOS FOR RETRIEVAL OF A BUFFER

RACE FOR A LOCKED BUFFER

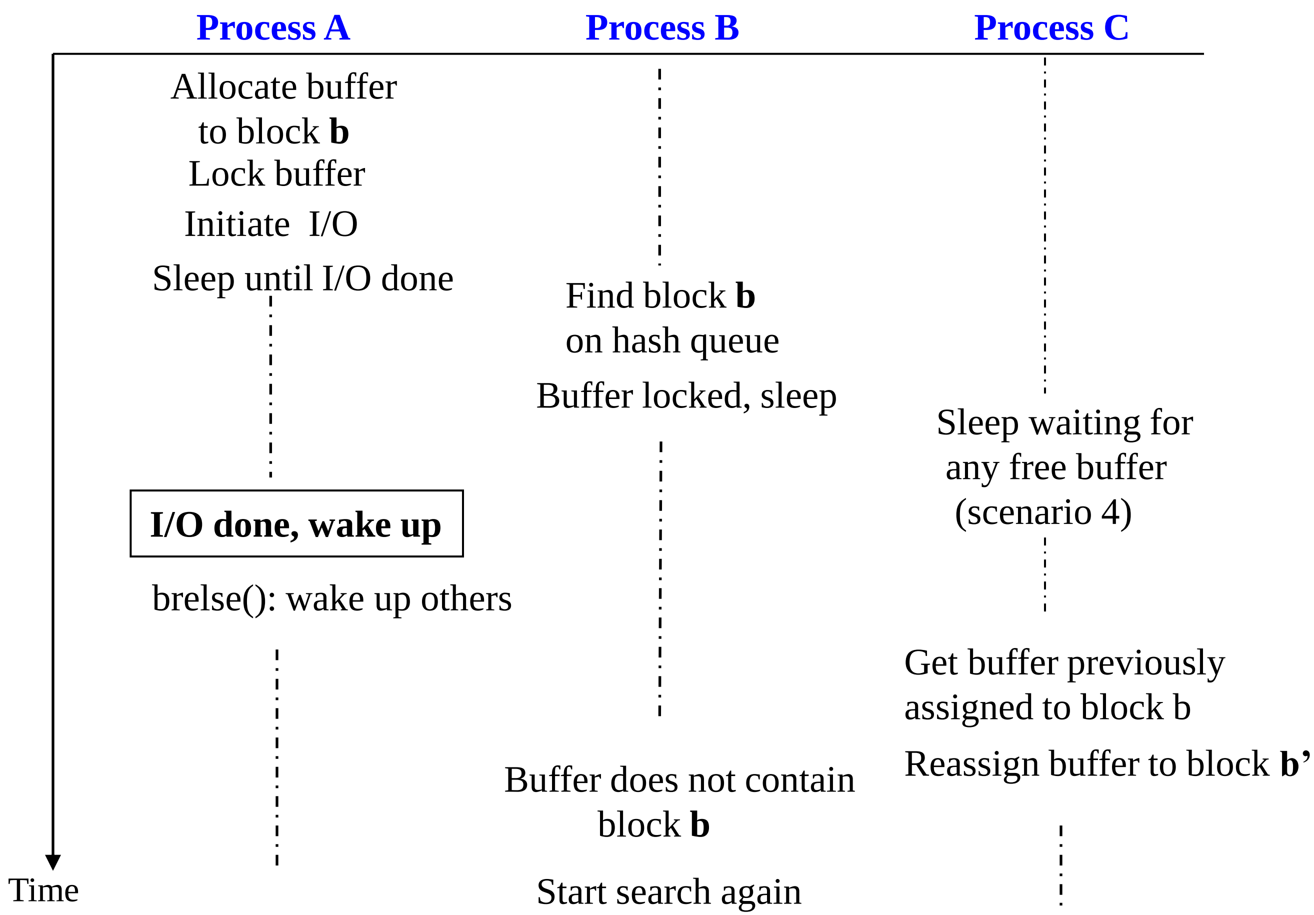


Figure 3.12 Race for a Locked Buffer

READING AND WRITING DISK BLOCKS

To read a disk block

- A process uses algorithm *getblk* to search for a disk block.
- In the cache
    - The kernel can return a disk block without physically reading the block from the disk.
- Not in the cache
    - The kernel calls the disk driver to “schedule” a read request.
    - The kernel goes to sleep awaiting the event the I/O completes.
    - After I/O, the disk controller interrupts the processor.
    - The disk interrupt handler awakens the sleeping process.

READING AND WRITING DISK BLOCKS

ALGORITHM FOR READING A DISK BLOCK

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

READING AND WRITING DISK BLOCKS

To read block ahead

- The kernel checks if the first block is in the cache or not.
- If the block in not in the cache, it invokes the disk driver to read the block.
- If the second block is not in the buffer cache, the kernel instructs the disk driver to read it asynchronously.
- The process goes to sleep awaiting the event that the I/O is complete on the first block.
- When awakening, the process returns the buffer for the first block.
- When the I/O for the second block does complete, the disk controller interrupts the system.
- Release buffer.

READING AND WRITING DISK BLOCKS

ALGORITHM FOR BLOCK READ AHEAD

```
Algorithm breada        /* block read and read ahead */
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 (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( brelse)
        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;
}
```

READING AND WRITING DISK BLOCKS

To write a disk block

- Kernel informs the disk driver that it has a buffer whose contents should be output.
- Disk driver schedules the block for I/O.
- If the write is synchronous, the calling process goes the sleep awaiting I/O completion and releases the buffer when it awakens.
- If the write is asynchronous, the kernel starts the disk write,but not wait for write to complete.
- The kernel will release buffer when I/O completes

A delayed write *vs*. an asynchronous write

READING AND WRITING DISK BLOCKS

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(algorithm brelse);
        }
        else if (buffer marked for delayed write)
                mark buffer to put at head of free list;
}
```