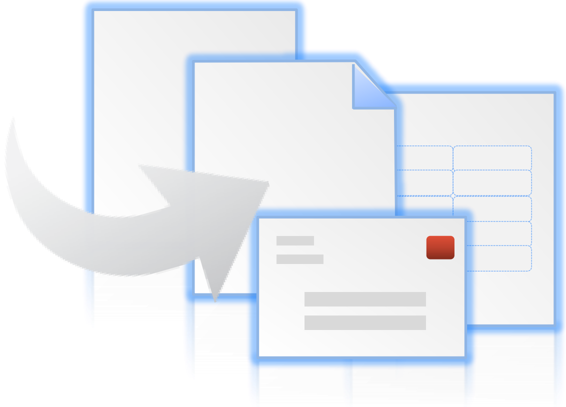
ADVANCED OPERATING SYSTEMS

UNIT 2 FILE AND DIRECTORY I/O

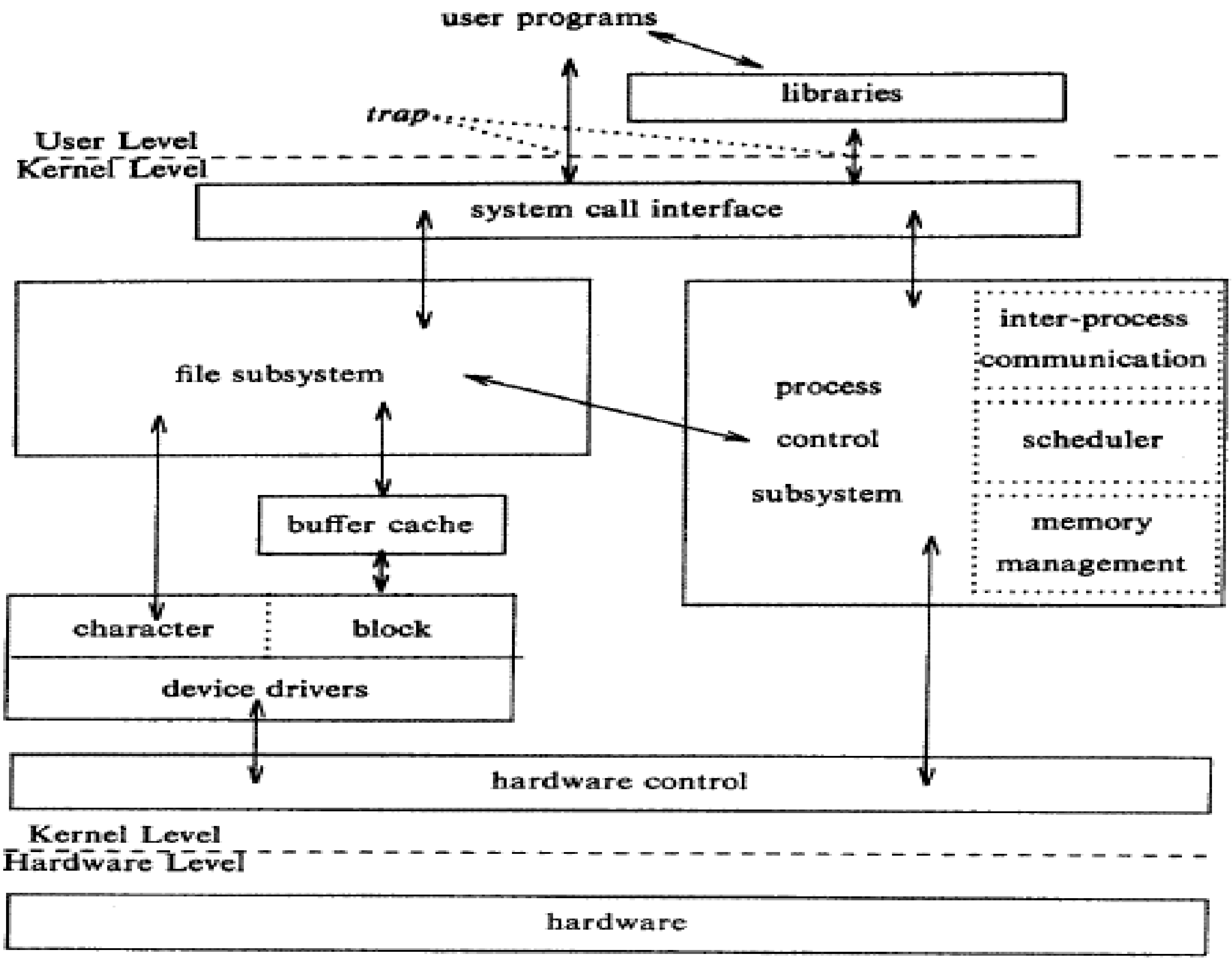
BY

MR.PRASAD SAWANT

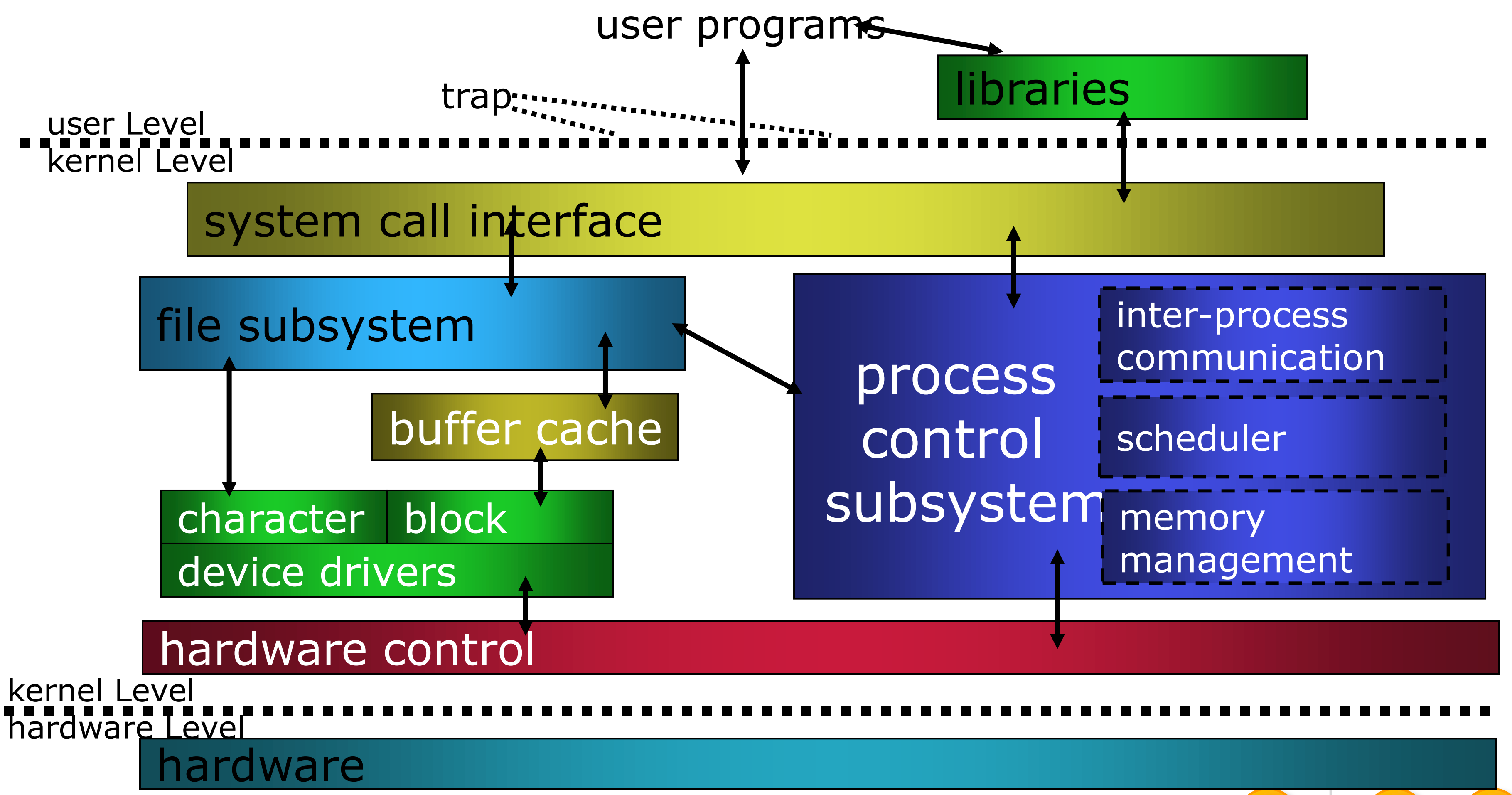


OUT LINE OF SESSION

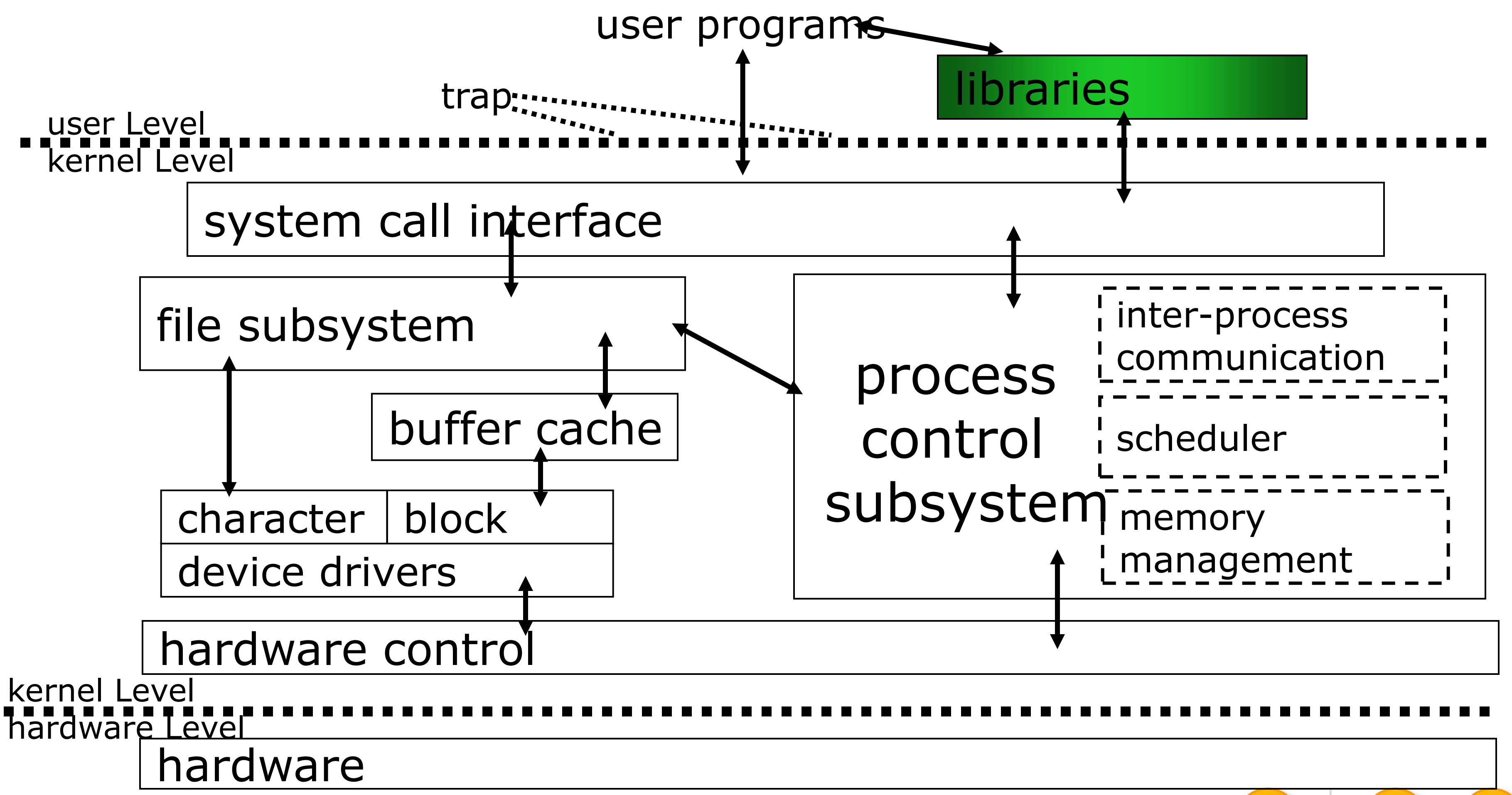
1. Buffer headers
2. structure of the buffer pool
3. scenarios for retrieval of a buffer
4. reading and writing disk blocks
5. Inodes
6. structure of regular file
7. Open
8. Read
9. Write
10. Lseek
11. Pipes
12. close
13. dup


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

ARCHITECTURE OF THE UNIX



LIBRARIES (1)

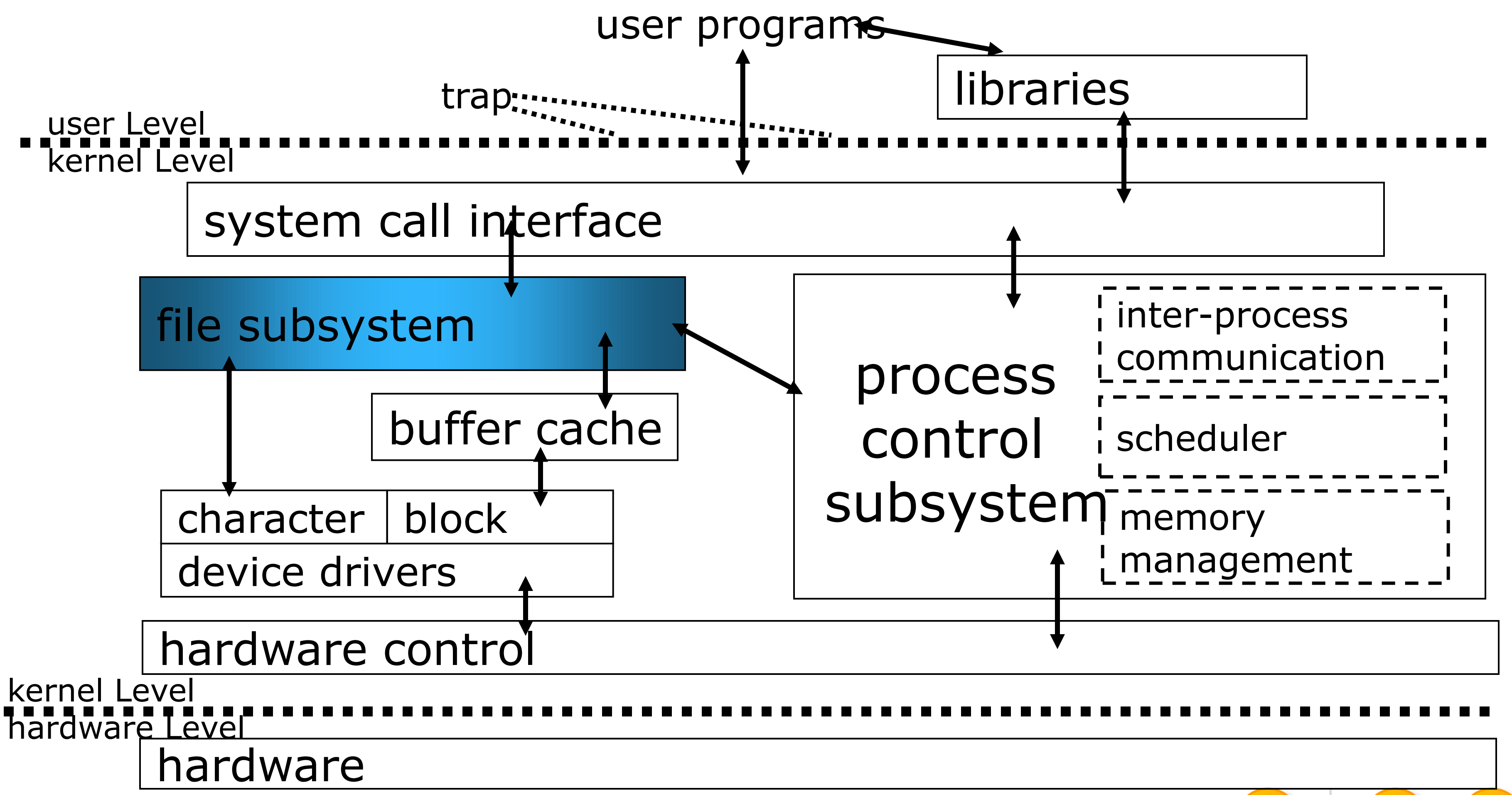


user programs
libraries
trap
user Level
kernel Level
system call interface
file subsystem
buffer cache
character
block
device drivers
process control subsystem
inter-process communication
scheduler
memory management
hardware control
kernel Level
hardware Level
hardware

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)

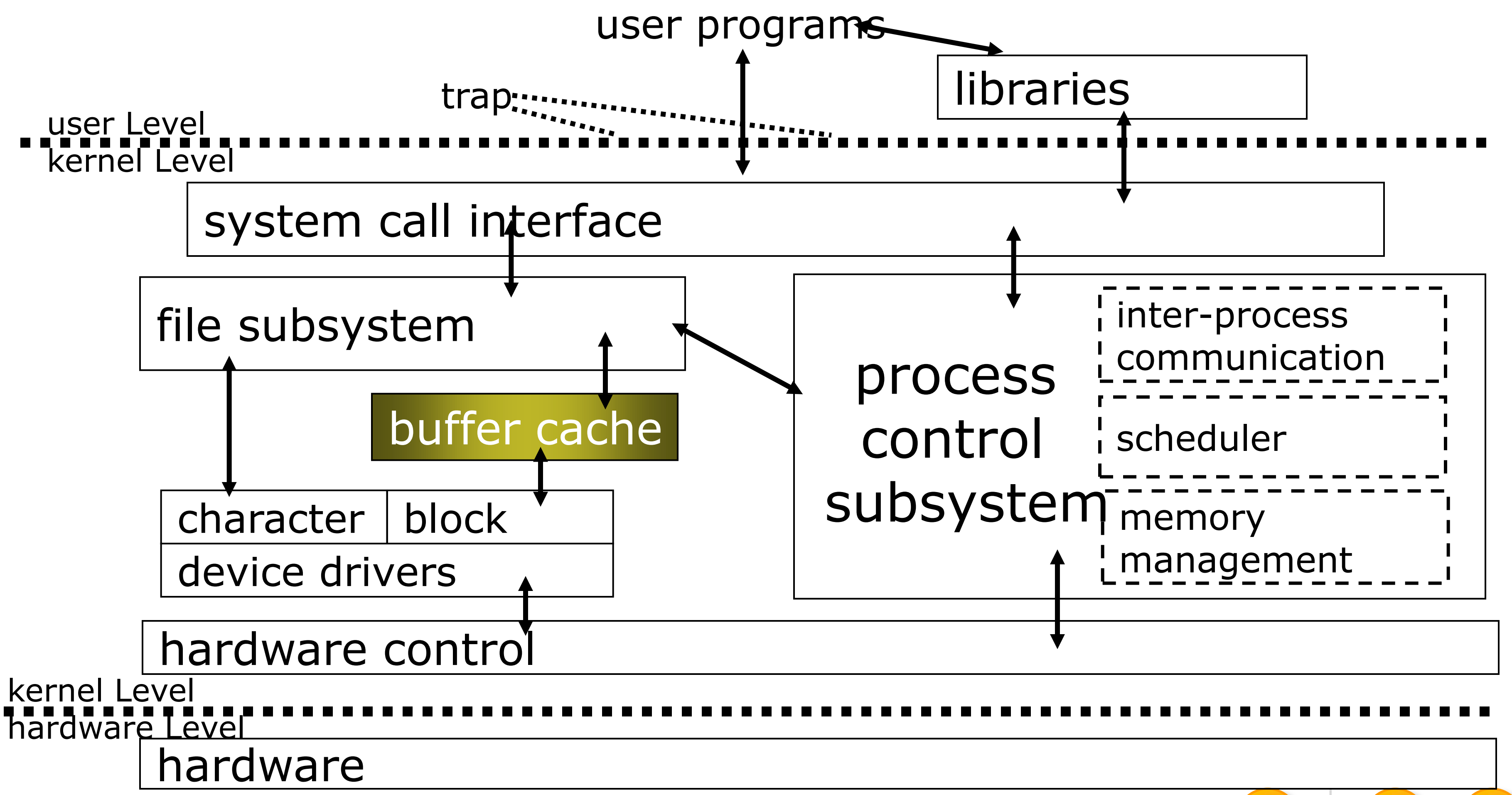


FILE SUBSYSTEM (2)

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 …*

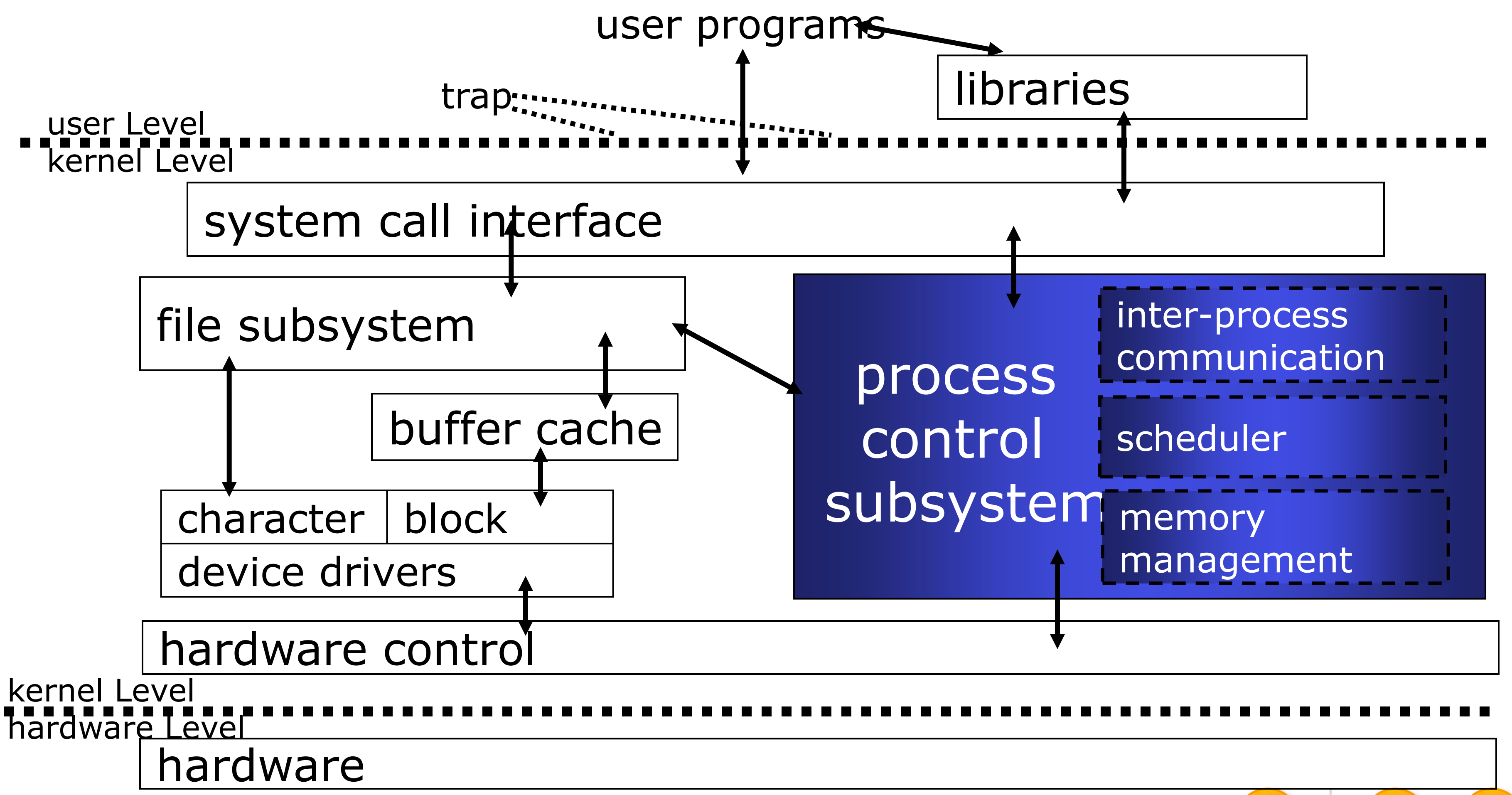
BUFFERING MECHANISM (1)



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

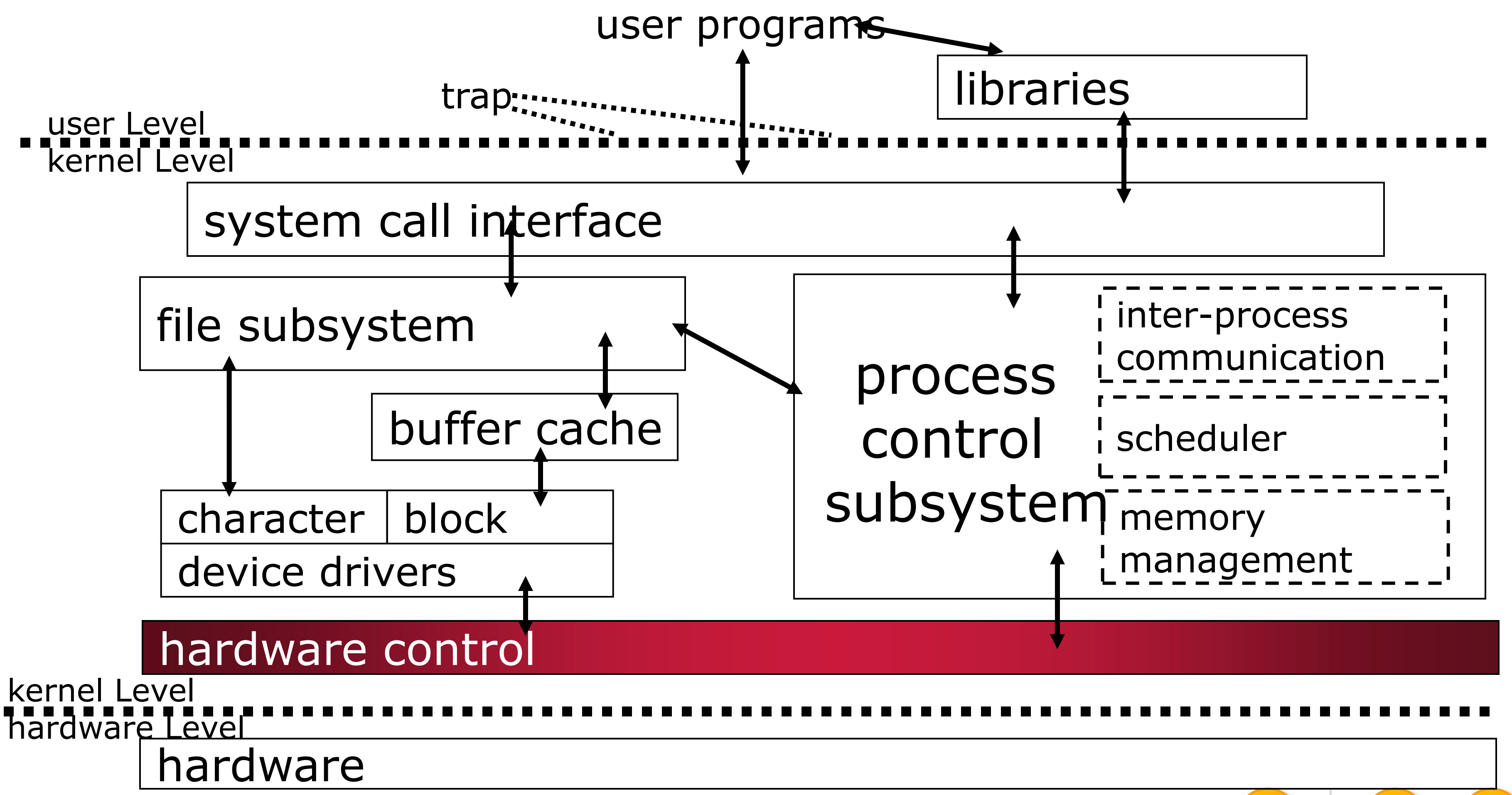
Scheduler module

- Allocate the CPU to processes

Interprocess communication

- There are several forms.

HARDWARE CONTROL (1)



HARDWARE CONTROL (2)

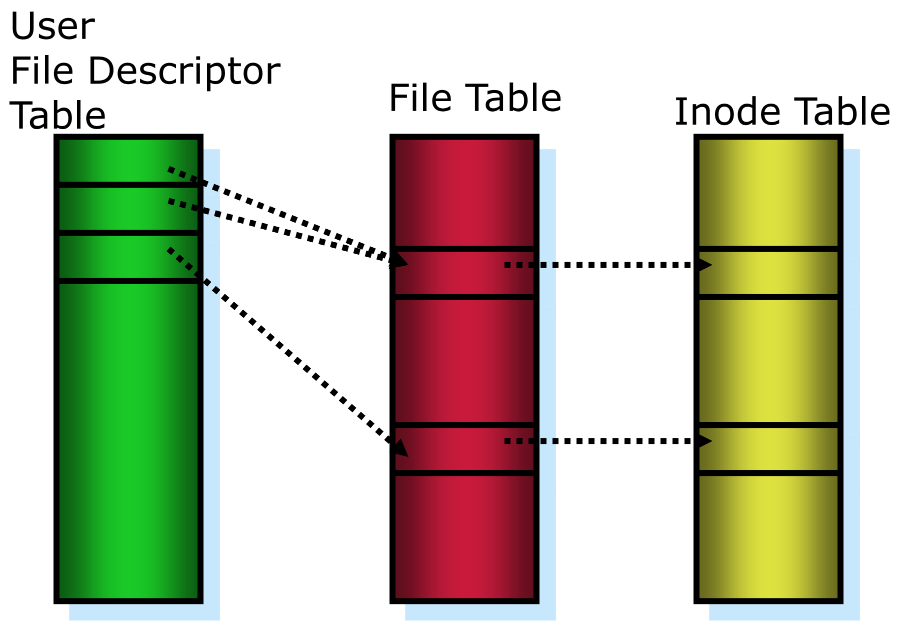
Responsible for handling interrupts and for communicating with the machine.

AN OVERVIEW OF THE FILE SUBSYSTEM

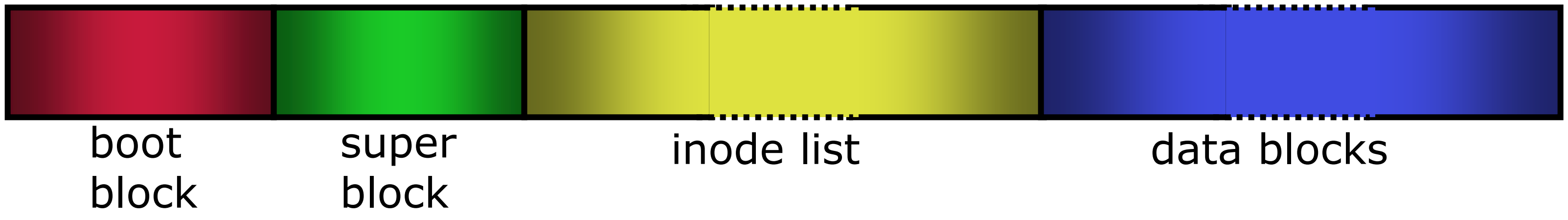
inode (index node)

- 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

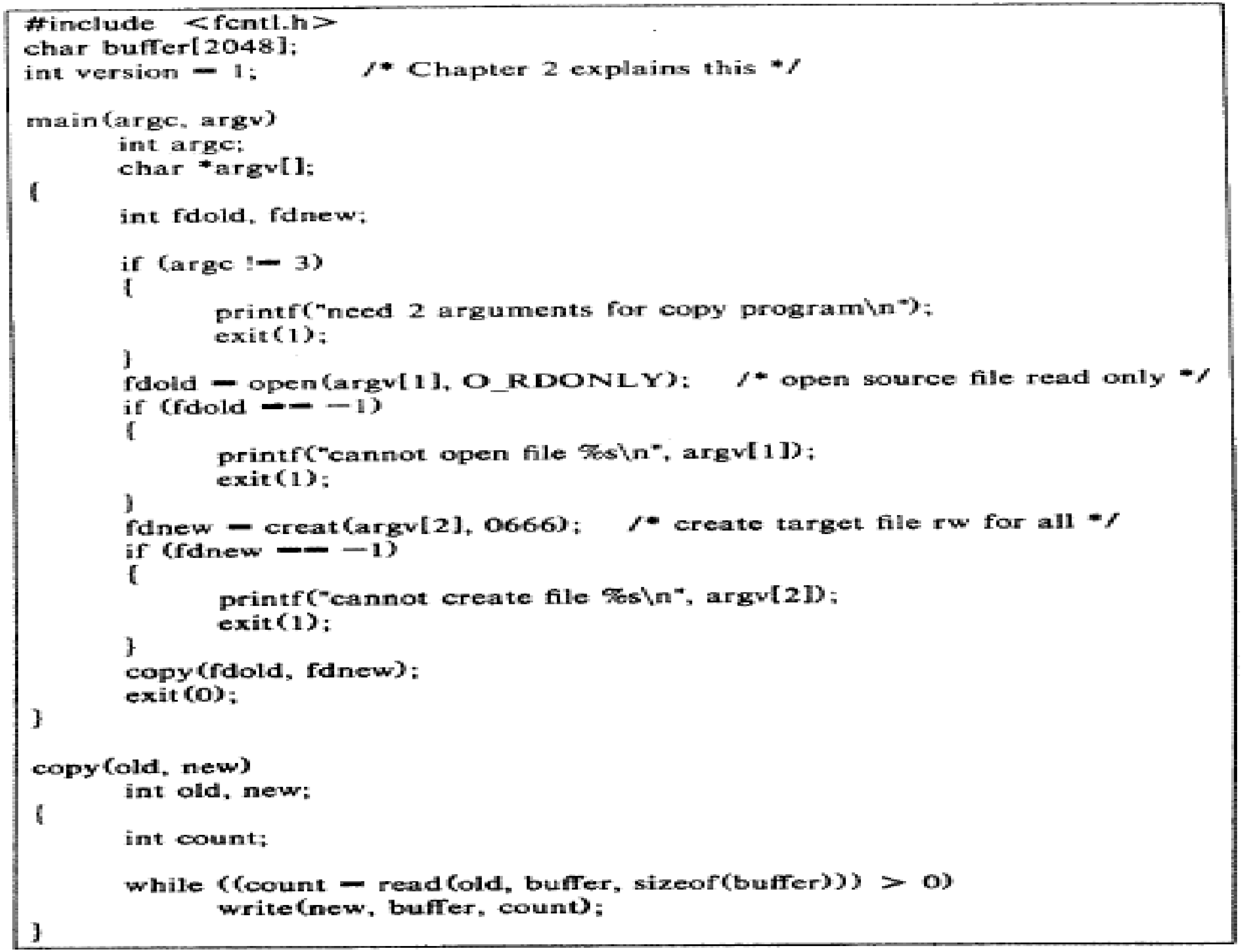
- Describes the state of a file system

inode list

- a list of inodes

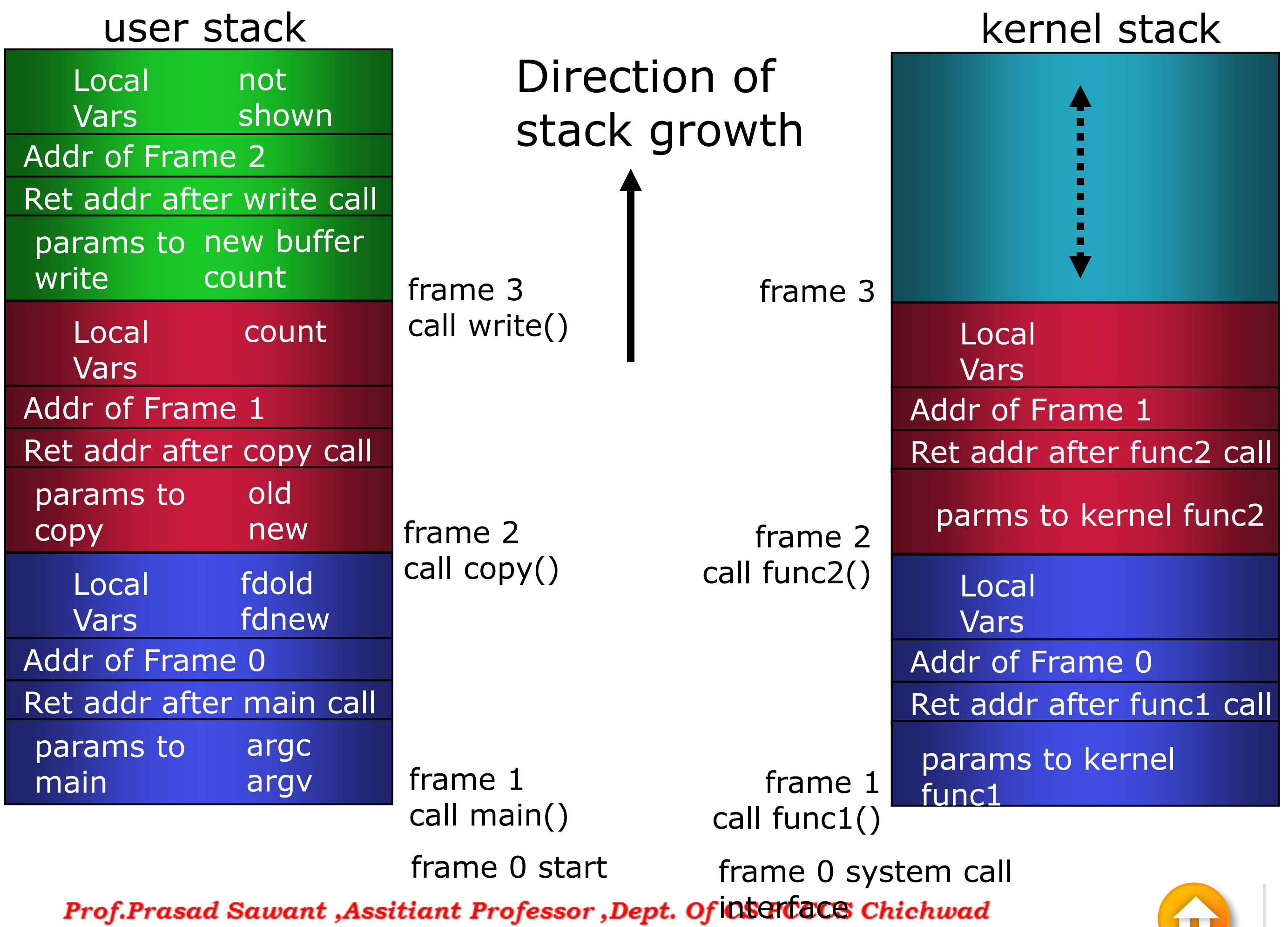
data block

- contain file data and administrative data

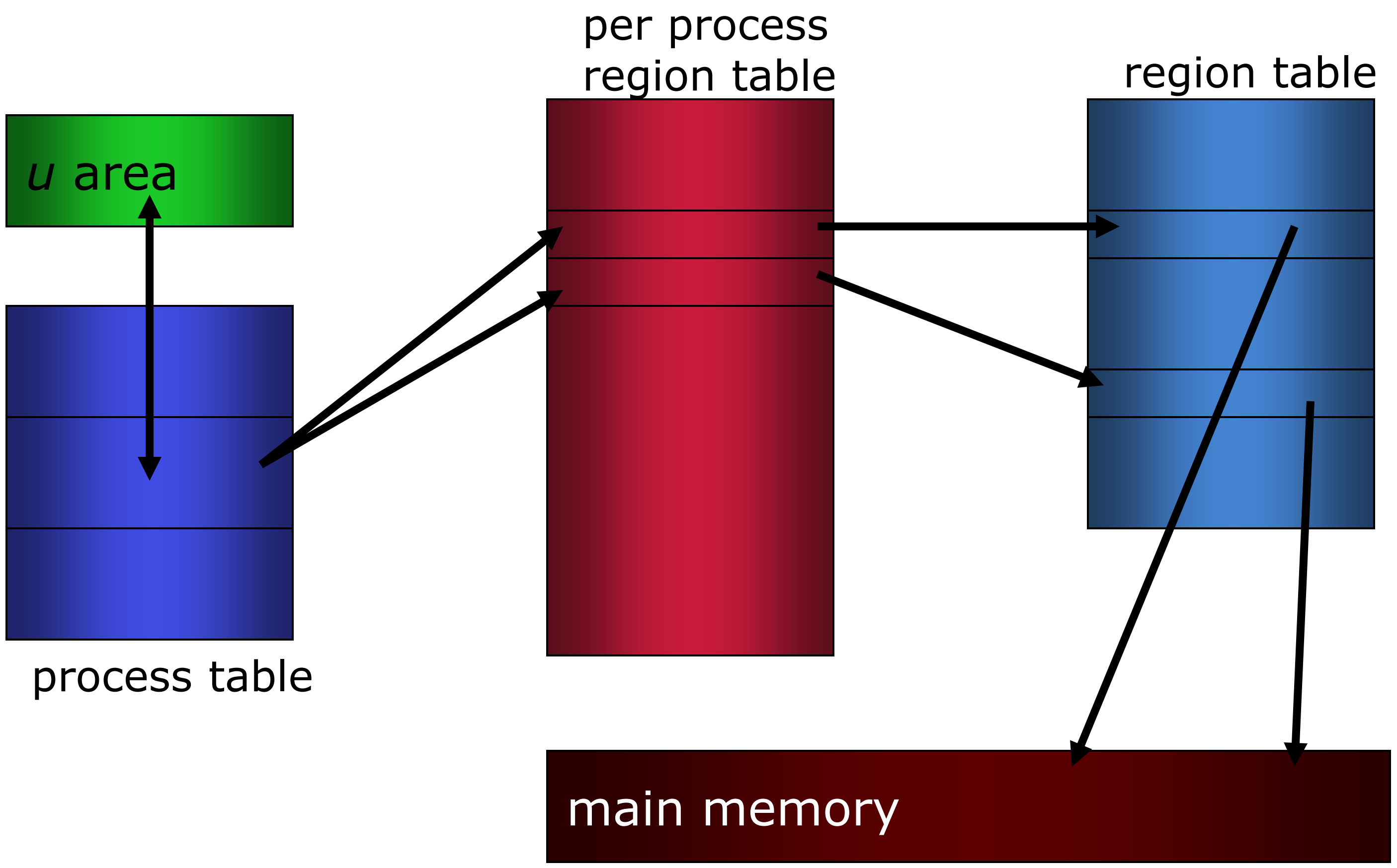


```
#include  <fcntl.h>
char buffer[2048];
int version = 1;          /* Chapter 2 explains this */

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

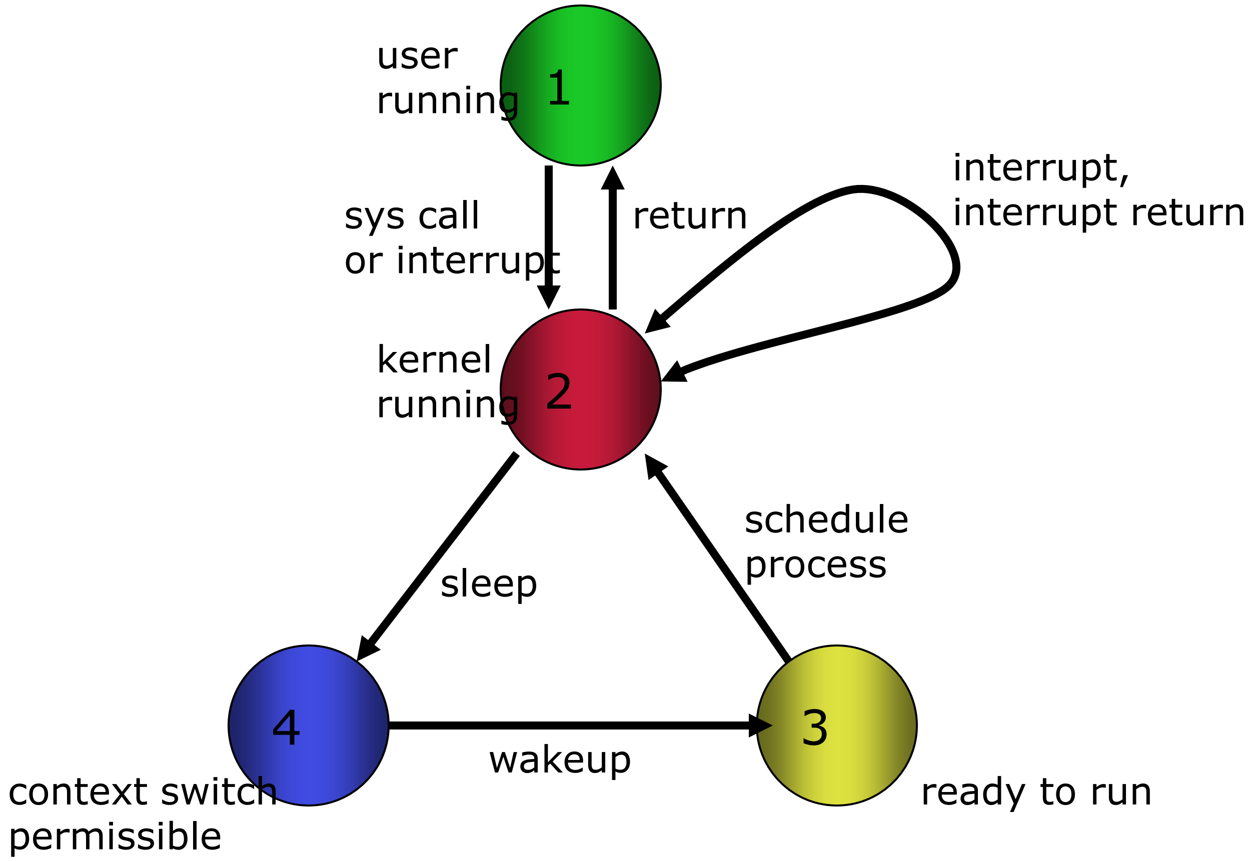
Ready to run

- soon as the scheduler chooses it.

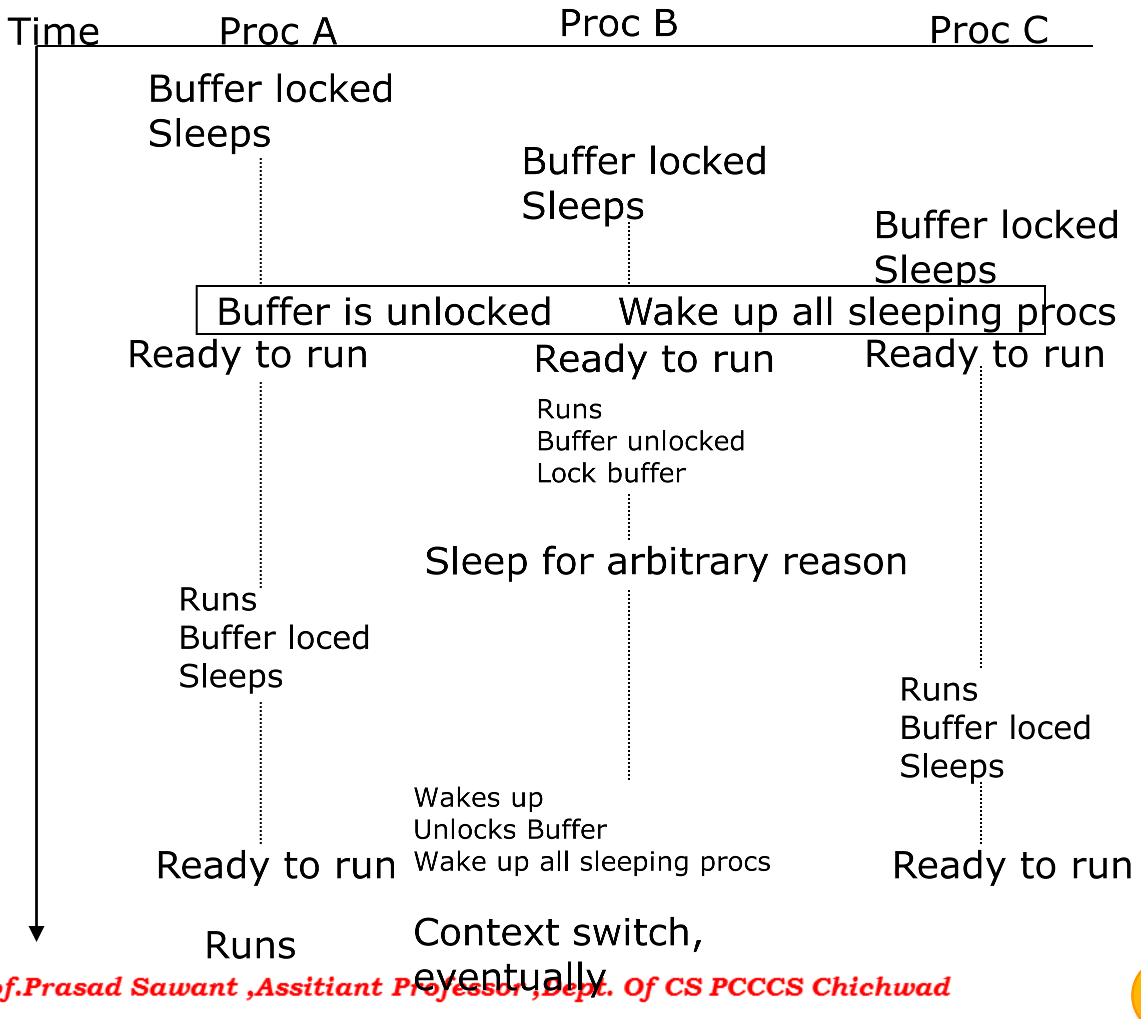
Sleeping

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

PROCESS TRANSITION



MULTIPLE PROCESES SLEEPING ON A LOCK



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

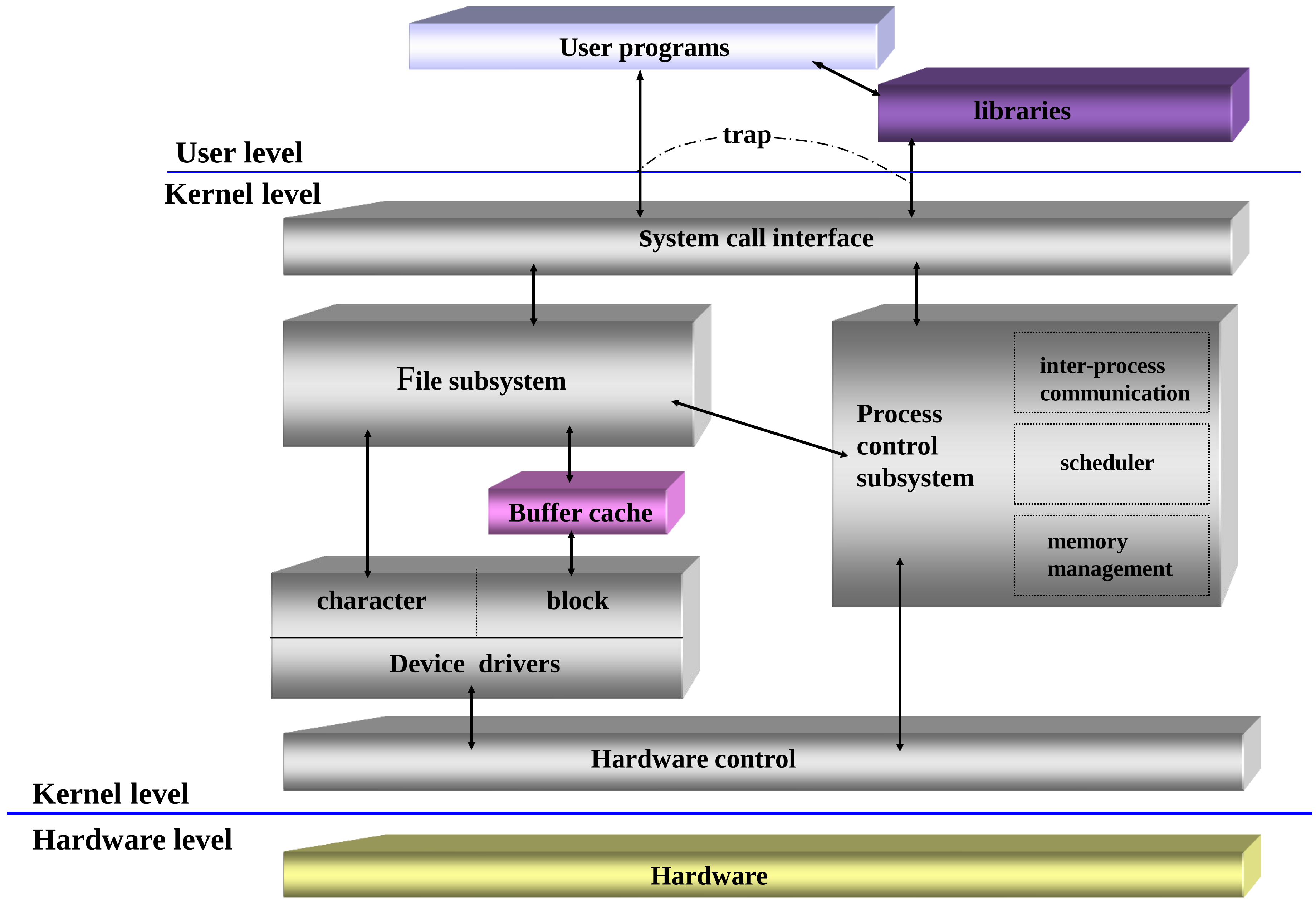
Kernel minimize the frequency of disk access

- 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
Process control subsystem
inter-process communication
scheduler
memory management
Buffer cache
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

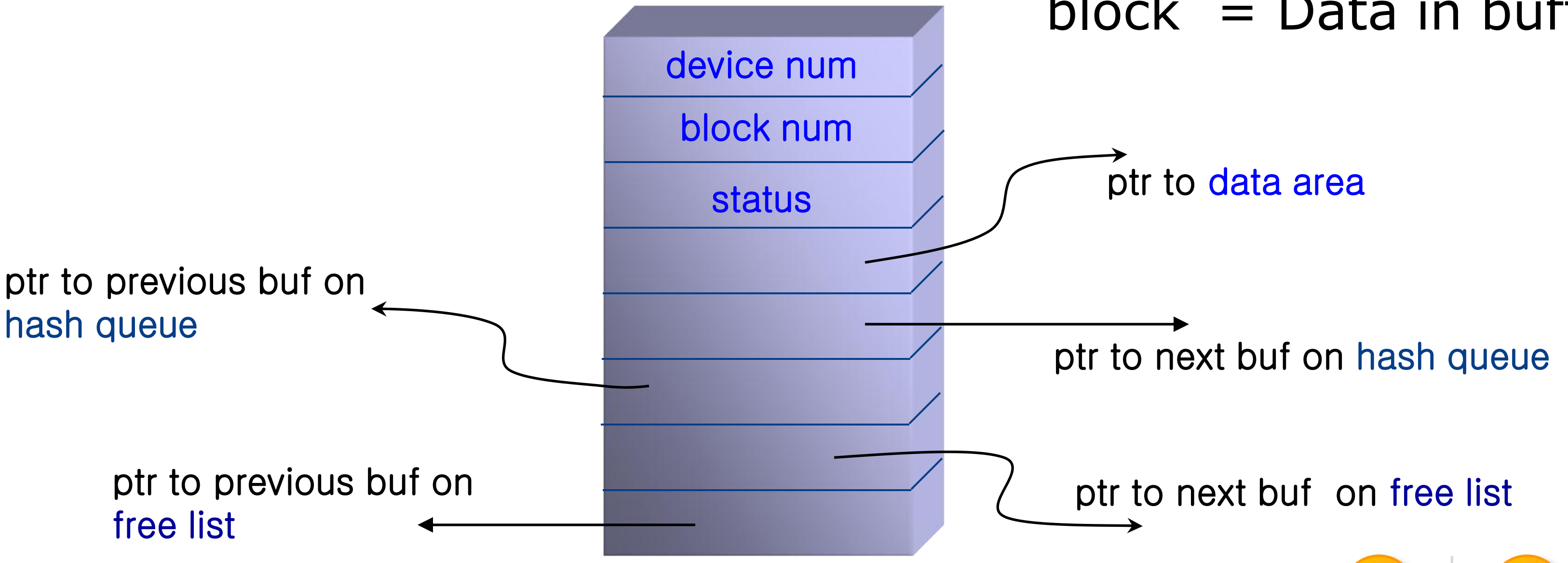

device num
block num
status
ptr to data area
ptr to previous buf on hash queue
ptr to next buf on hash queue
ptr to previous buf on free list
ptr to next buf on free list

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

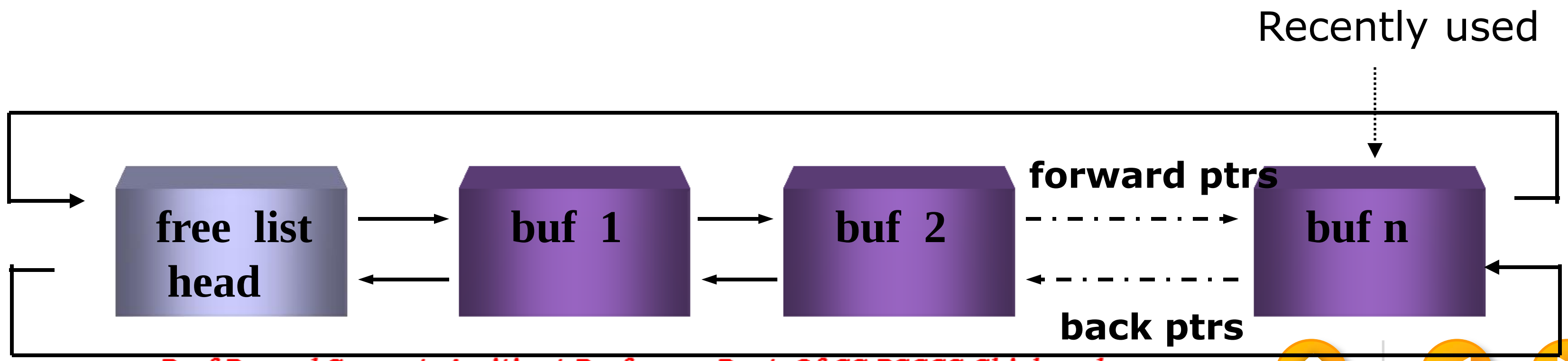
- The buffer is currently locked.
- The buffer contains valid data.
- “delayed-write” as condition
- The kernel is currently reading or writing the contents of buffer to disk.
- A process is currently waiting for the buffer to become free.

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

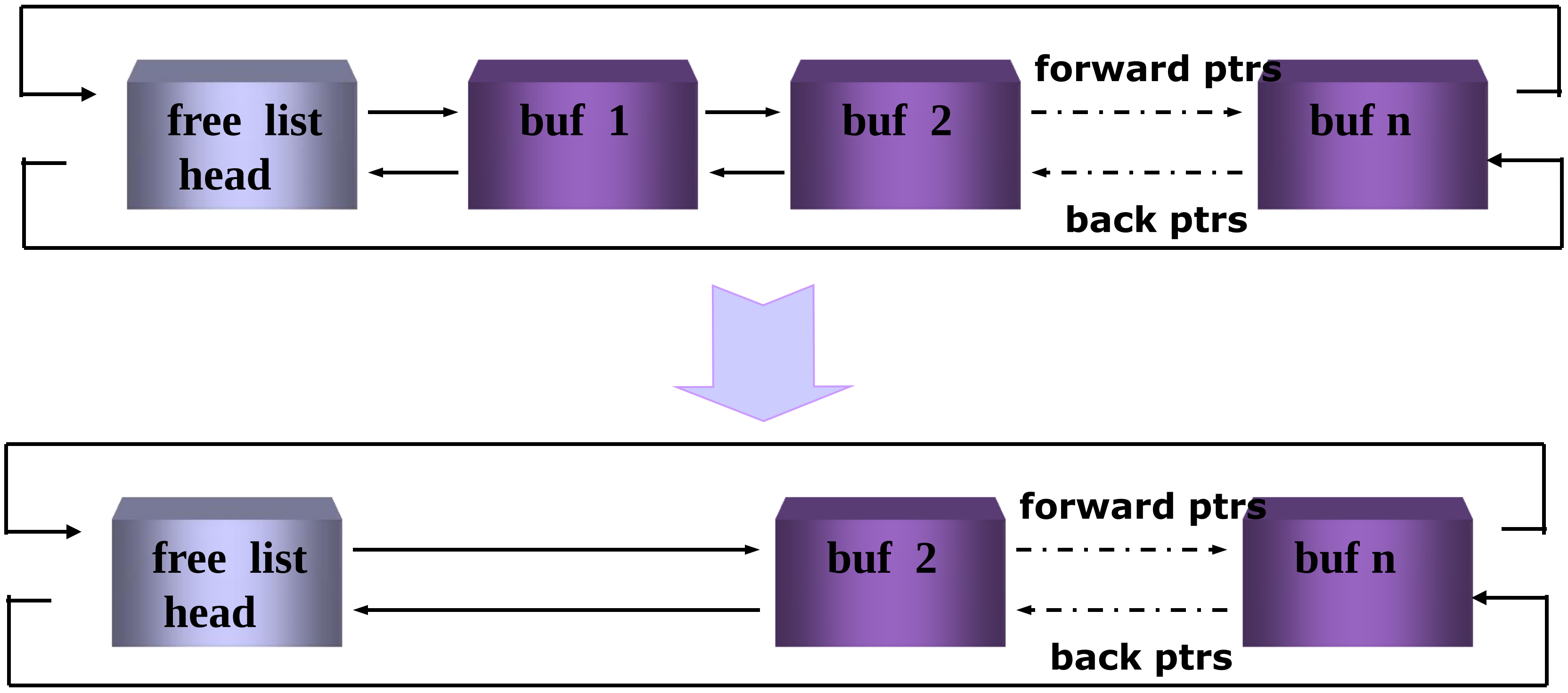


Figure 3.2. Free list of Buffers

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

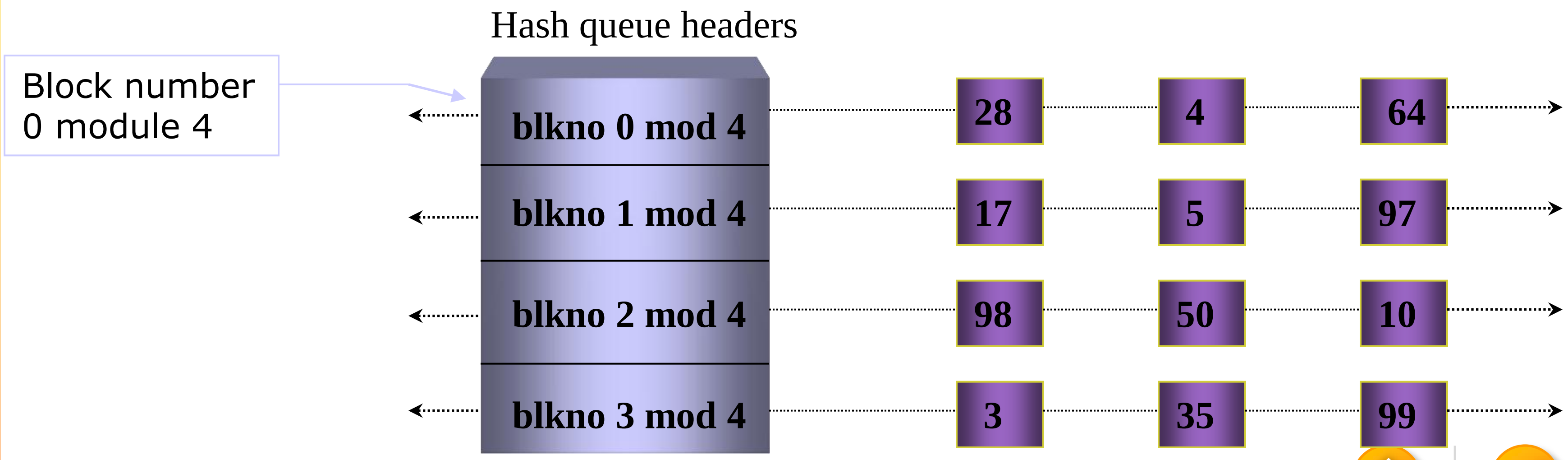


Figure 3.3 Buffers on the Hash Queues

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

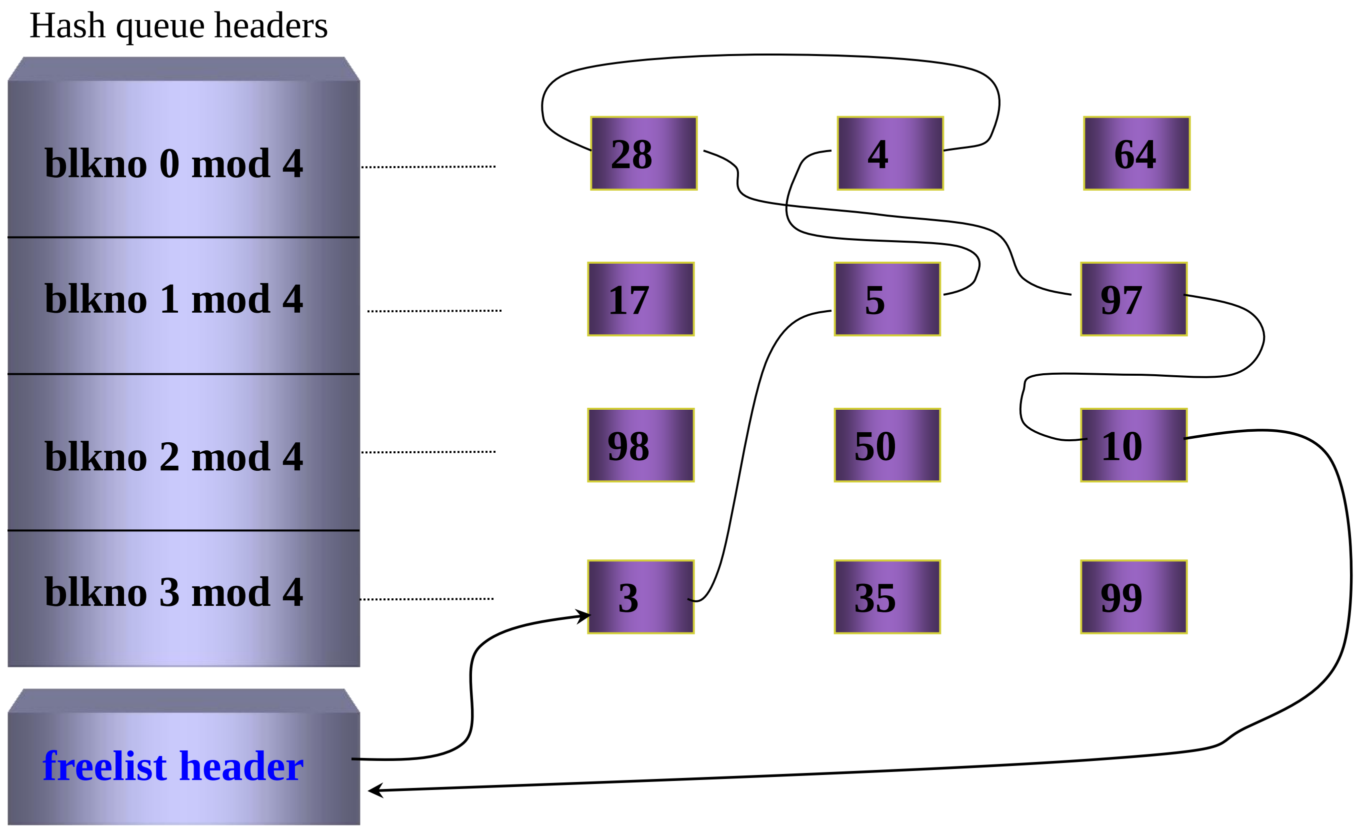
Output: locked buffer that can now be used for block

```
{
   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;
      }
   }
```

```
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

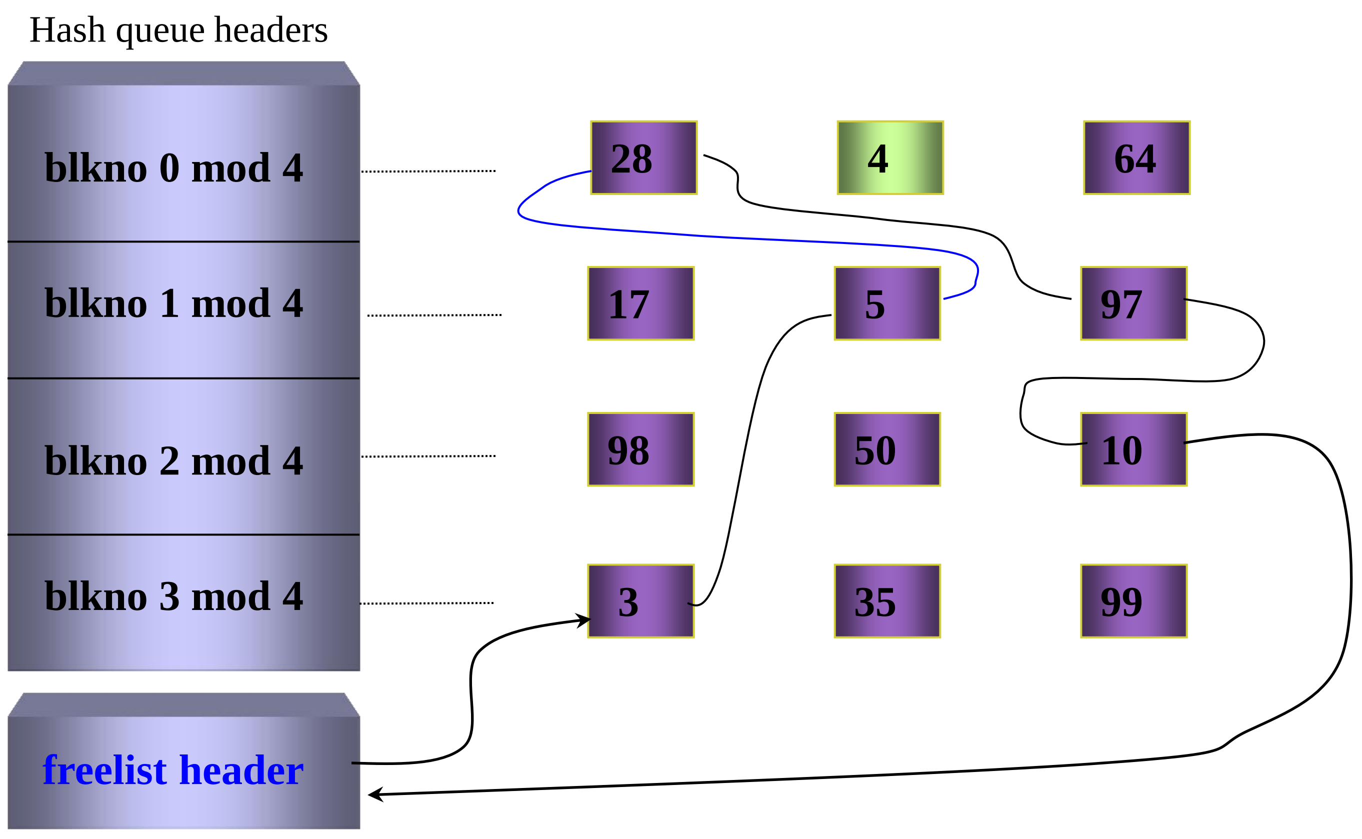
FIRST SCENARIO IN FINDING A BUFFER: BUFFER ON HASH QUEUE (A)



(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

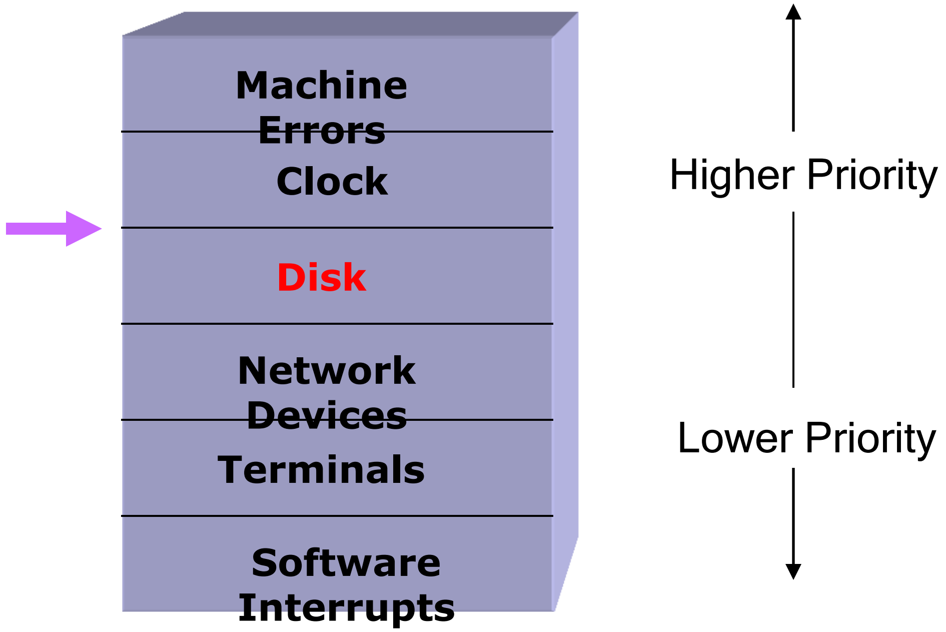
```
Algorithm  brelse
Input: locked  buffer
{
          wakeup all process: event, waiting for any buffer to become free;
          wakeup all process: event, waiting for this buffer to become free;
          raise processor execution level to block interrupts;
          if (buffer contents valid and 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**

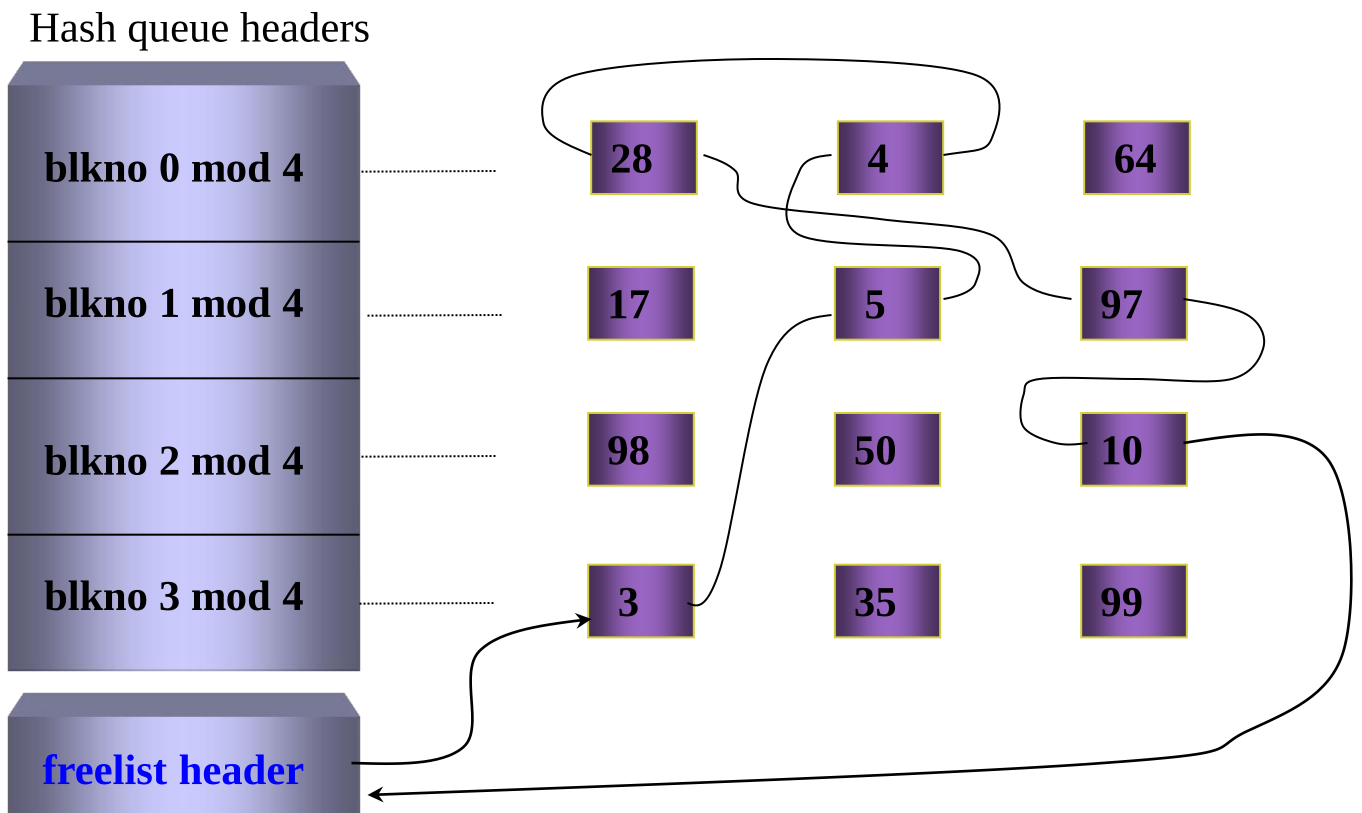
Input: file system number
          block number

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

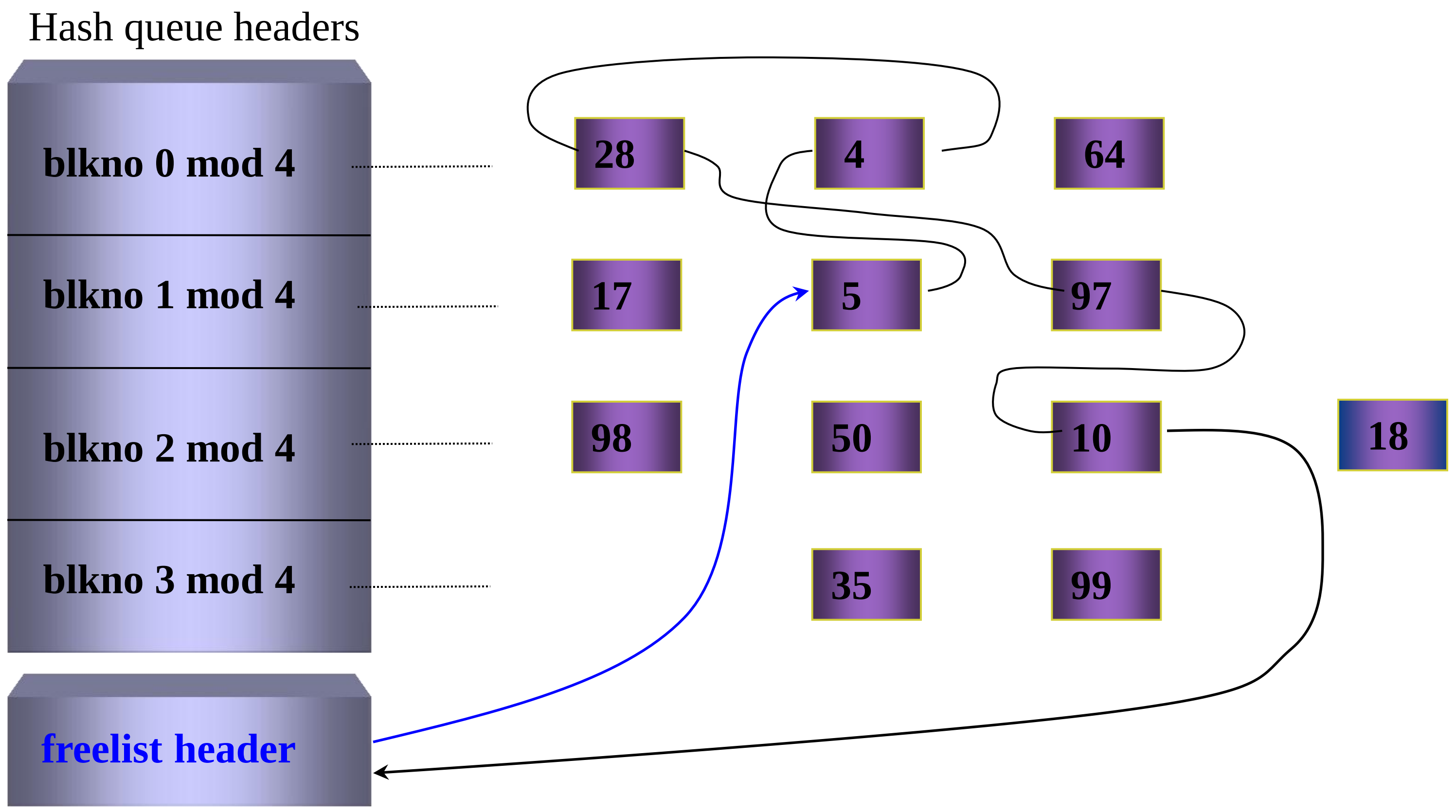
SECOND SCENARIO FOR BUFFER ALLOCATION (A)



(a) Search for Block 18 – Not in Cache

SCENARIOS FOR RETRIEVAL OF A BUFFER

SECOND SCENARIO FOR BUFFER ALLOCATION (B)

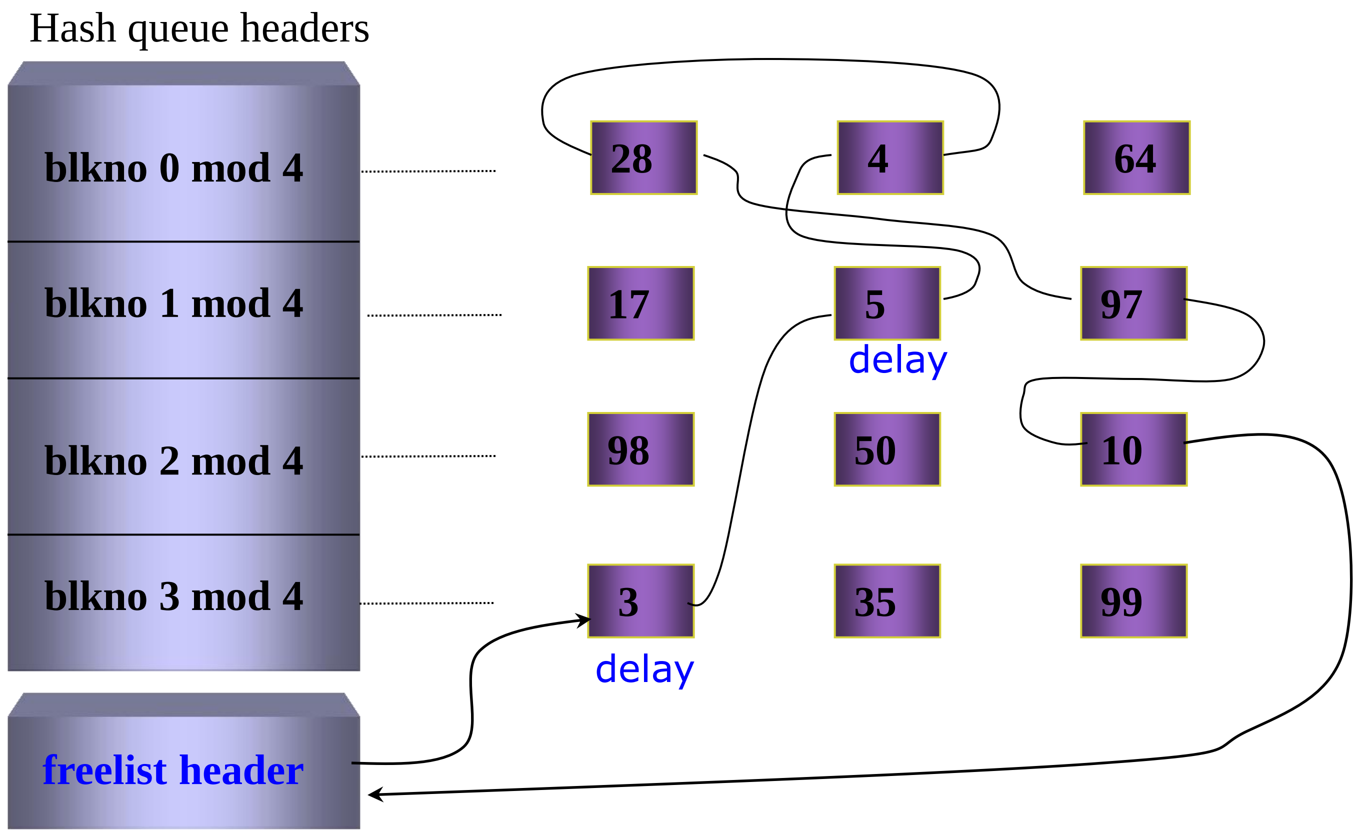


(b) Remove First Block from Free list, Assign to 18

```
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

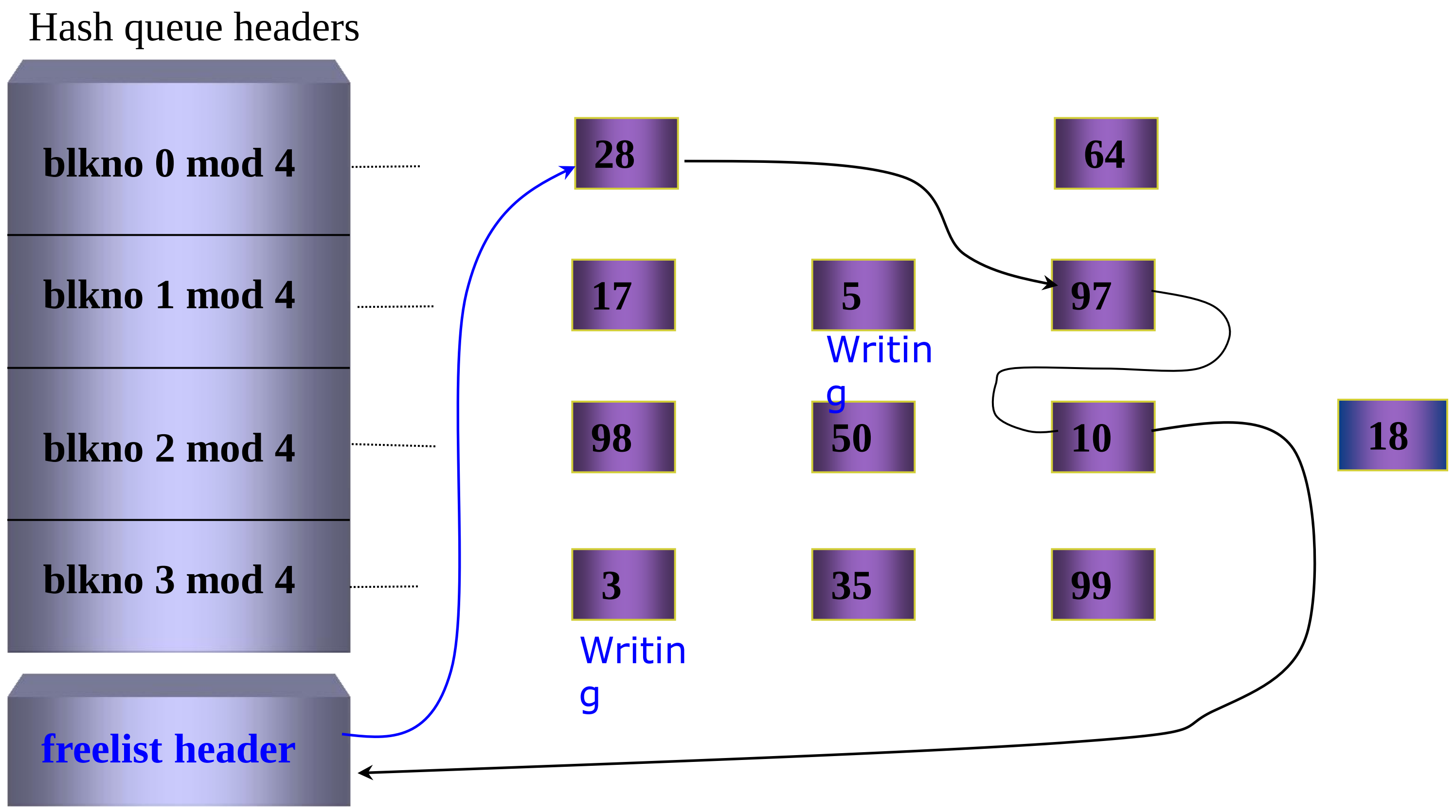
THIRD SCENARIO FOR BUFFER ALLOCATION (A)



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

SCENARIOS FOR RETRIEVAL OF A BUFFER

THIRD SCENARIO FOR BUFFER ALLOCATION (B)

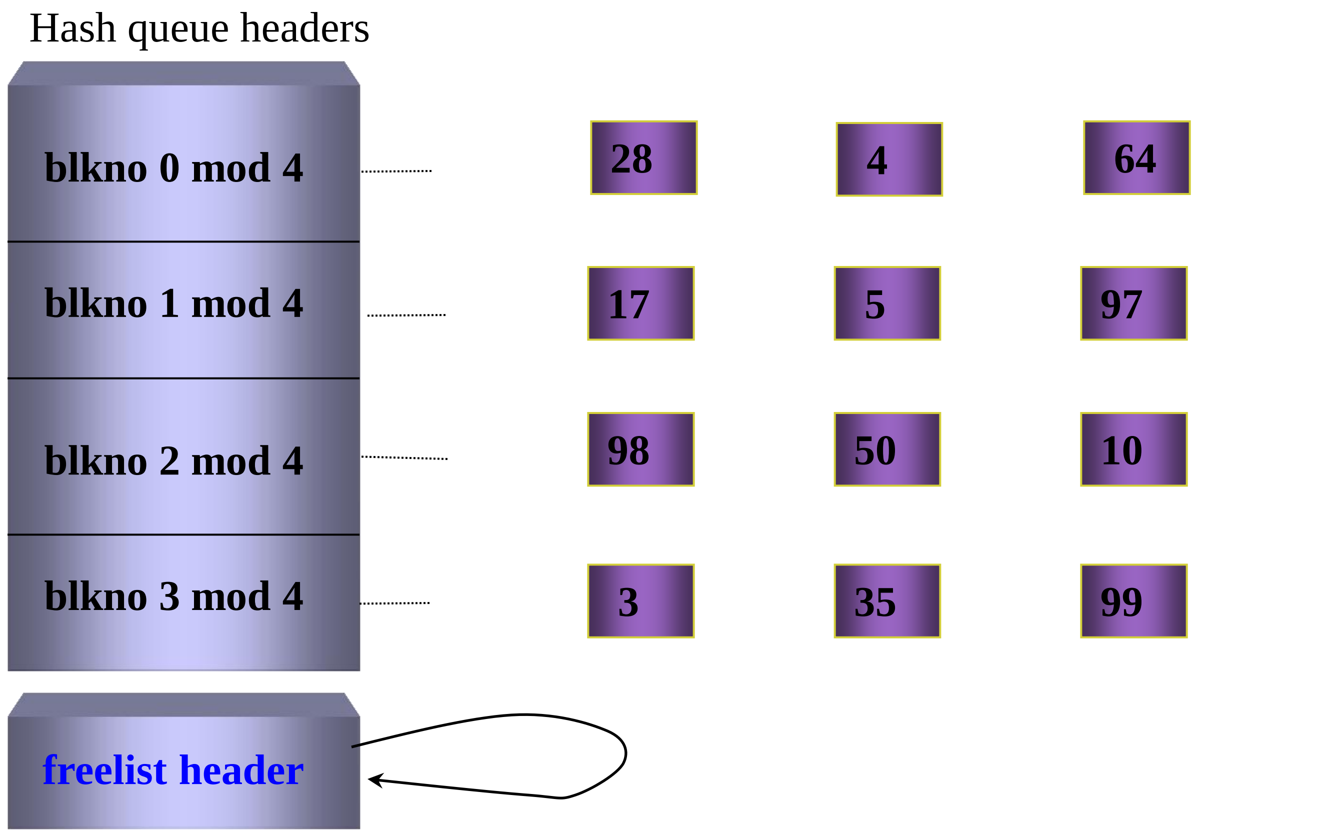


(b) Writing Blocks 3, 5, Reassign 4 to 18

```
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

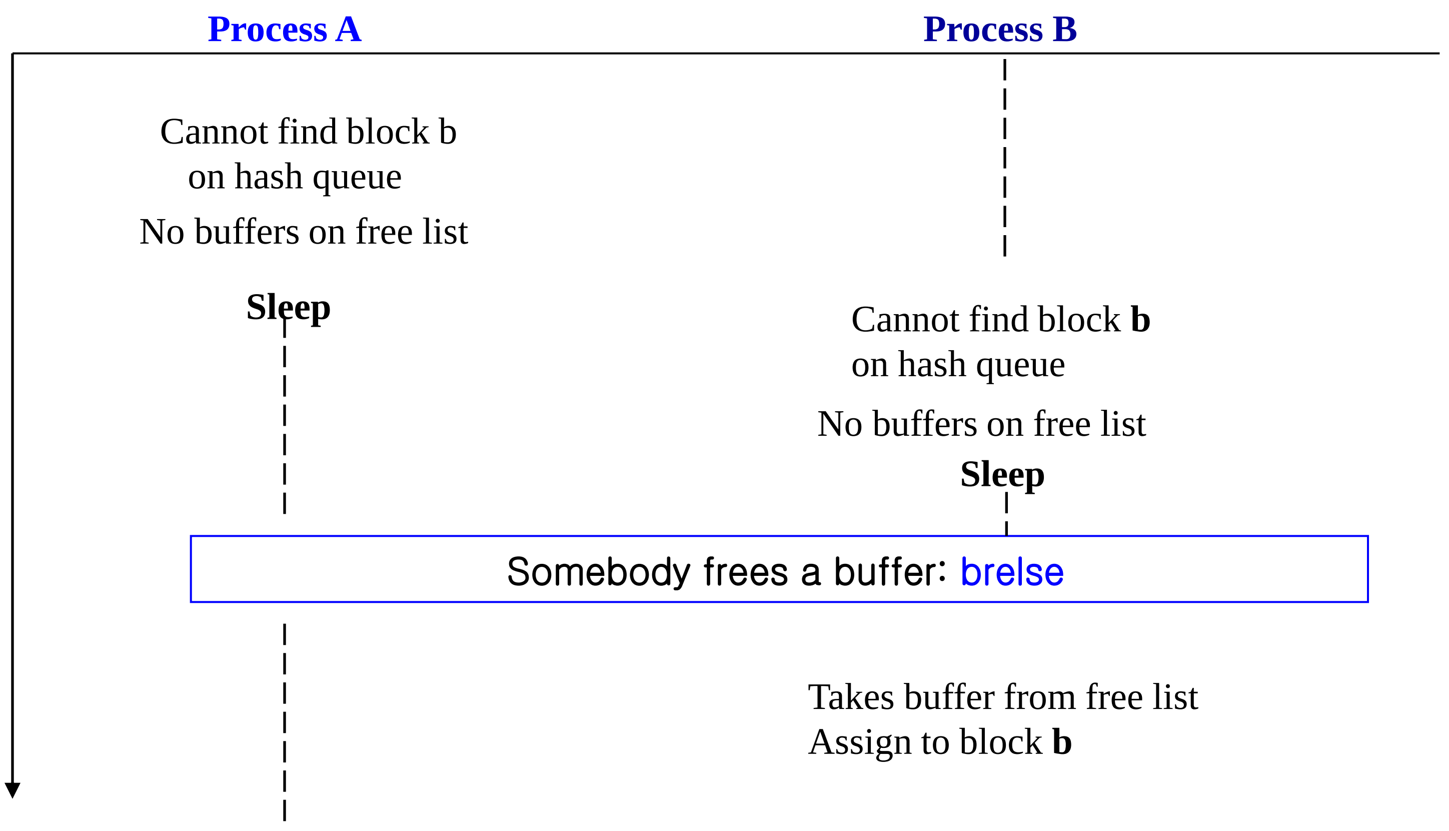
FOURTH SCENARIO FOR ALLOCATING BUFFER



Search for Block 18, Empty Free list

SCENARIOS FOR RETRIEVAL OF A BUFFER

RACE FOR FREE BUFFER



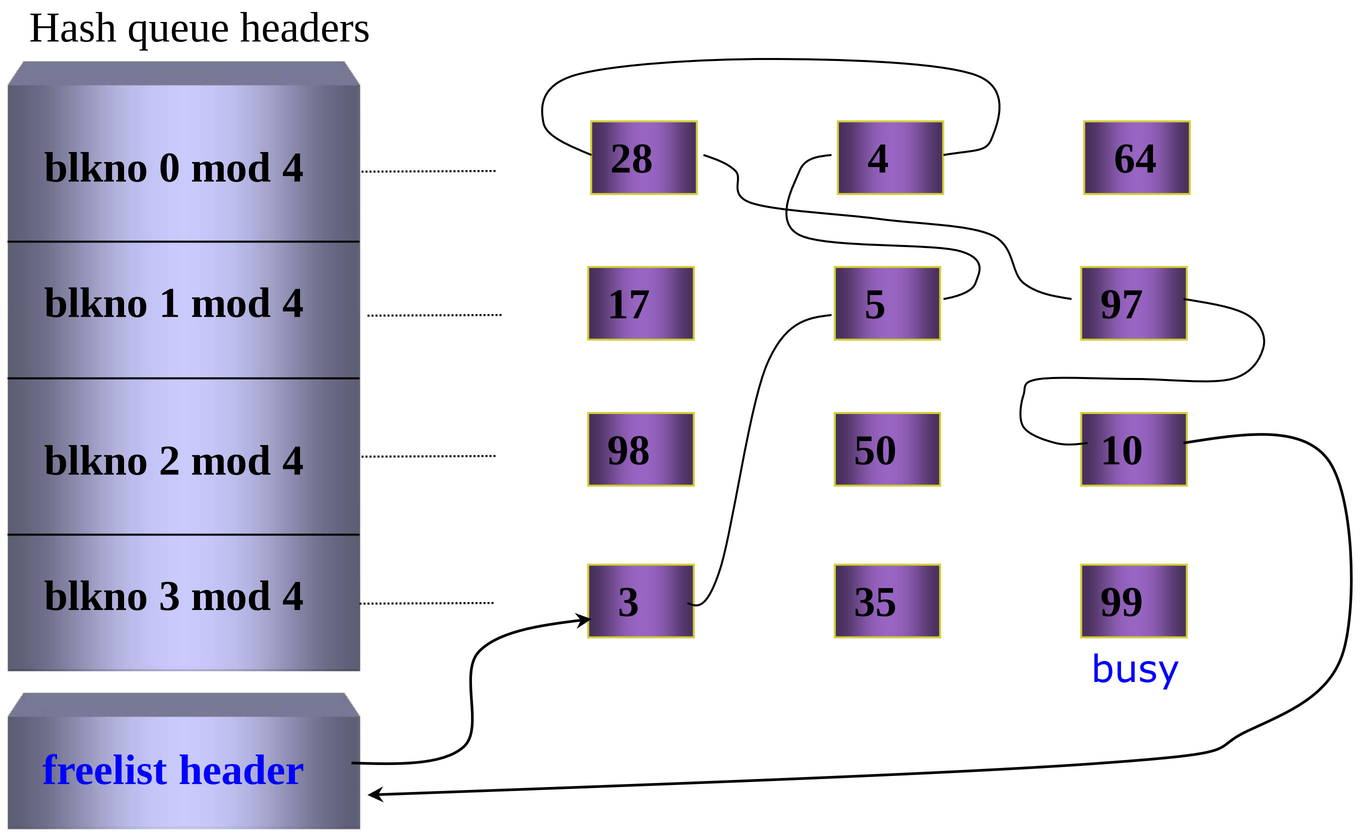
Process A
Process B
Cannot find block b on hash queue
No buffers on free list
Sleep
Cannot find block **b** on hash queue
No buffers on free list
Sleep
Somebody frees a buffer: brelse
Takes buffer from free list
Assign to block **b**

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

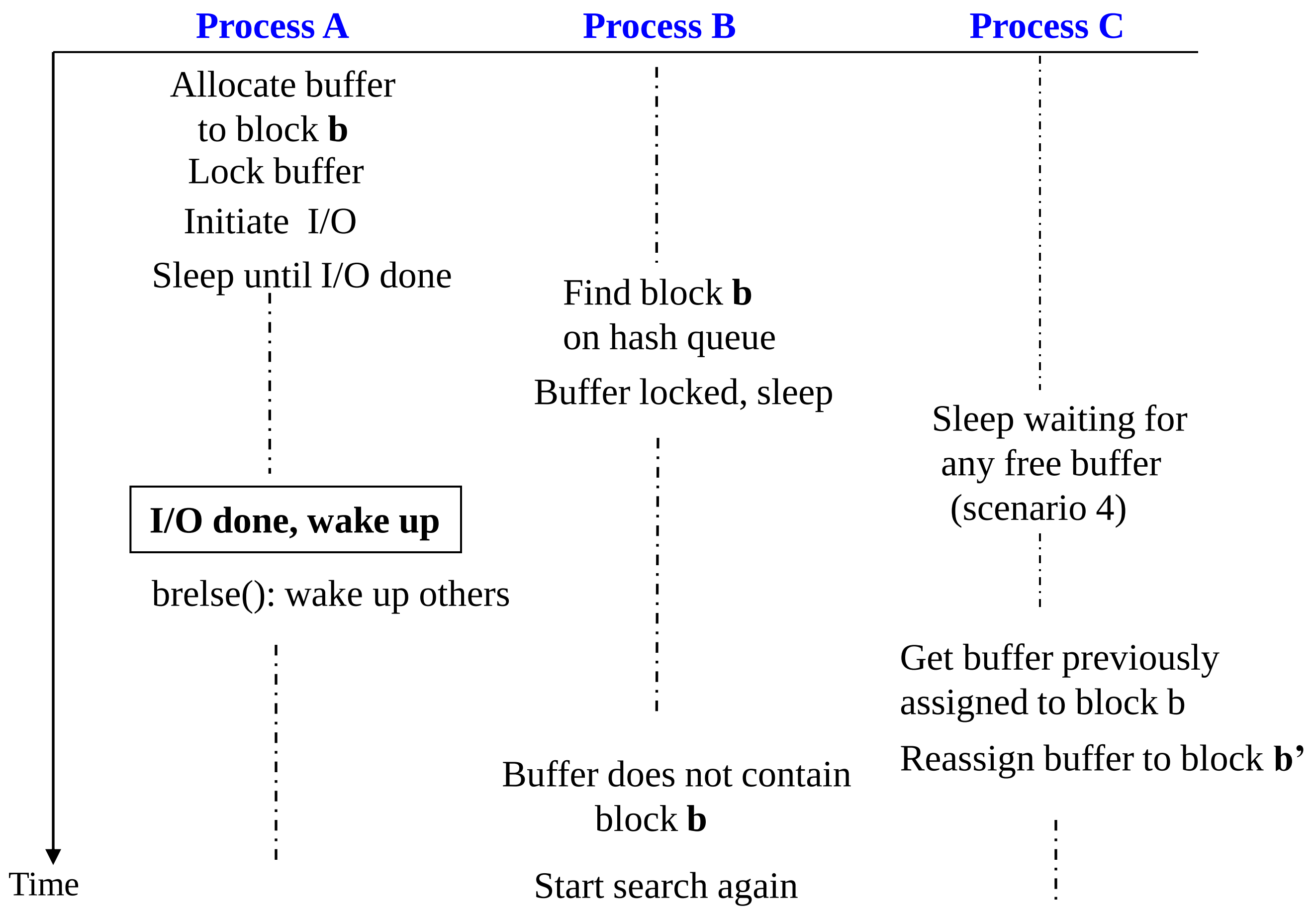
FIFTH SCENARIO FOR BUFFER ALLOCATION



Search for Block 99, Block busy

SCENARIOS FOR RETRIEVAL OF A BUFFER

RACE FOR A LOCKED BUFFER



Process A
Process B
Process C
Allocate buffer to block **b**
Lock buffer
Initiate I/O
Sleep until I/O done
Find block **b** on hash queue
Buffer locked, sleep
Sleep waiting for any free buffer (scenario 4)
I/O done, wake up
brelse(): wake up others
Get buffer previously assigned to block b
Reassign buffer to block **b'**
Buffer does not contain block **b**
Start search again
Time

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);
            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;
}
```

Out line of Session

1. *Inodes*
2. *Structure of a regular file*

Definition Of Inodes

- Every file has a **unique inode**
- Contain the **information** necessary for a process to access a file
- Exist in a **static form** on disk
- **Kernel** reads them into an **in-core inode** to manipulate them.

Contents Of Disk Inodes

1. File owner identifier (individual/group owner)
2. File type (regular, directory,..)
3. File access permission (owner,group,other)
4. File access time
5. Number of links to the file
6. Table of contents for the disk address of data in a file (byte stream vs discontiguous disk blocks)
7. File size
8. ***** Inode does **not specify** the **path name** that access the file

SAMPLE DISK INODE

File owner identifier

File type

File access permission

File access time

Number of links to the file

Table of contents for the disk address of data in a file

File size

Owner PMS
Group os
Type regular file
Perms rwxr-xr-x
Accessed Oct 23 2013 1:45 P.M
Modified Oct 22 2013 10:3 A.M
Inode Oct 23 2013 1:30 P.M
Size 6030 bytes
Disk addresses

Distinction Between Writing Inode And File

1. File change only when writing it.Inode change when changing the file, or when changing its owner, permisson,or link settings.
2. Changing a file implies a change to the inode,But, changing the inode does not imply that the file change.

Contents Of The In-core Copy Of The Inode

Fields of the disk inode

- Status of the in-core inode,
  - Inode is locked
  - Process is waiting for the inode to become unlocked
  - Differ from the disk copy as a result of a change to **the data in the inode**
  - Differ from the disk copy as a result of a change to **the file data**
  - File is a mount point

Contents Of The In-core Copy Of The Inode

2. Logical device number of the file system
3. Inode number (linear array on disk, disk inode not need this field)
4. Pointers to other in-core inodes
5. Reference count

In-core Inode Vs Buffer Header

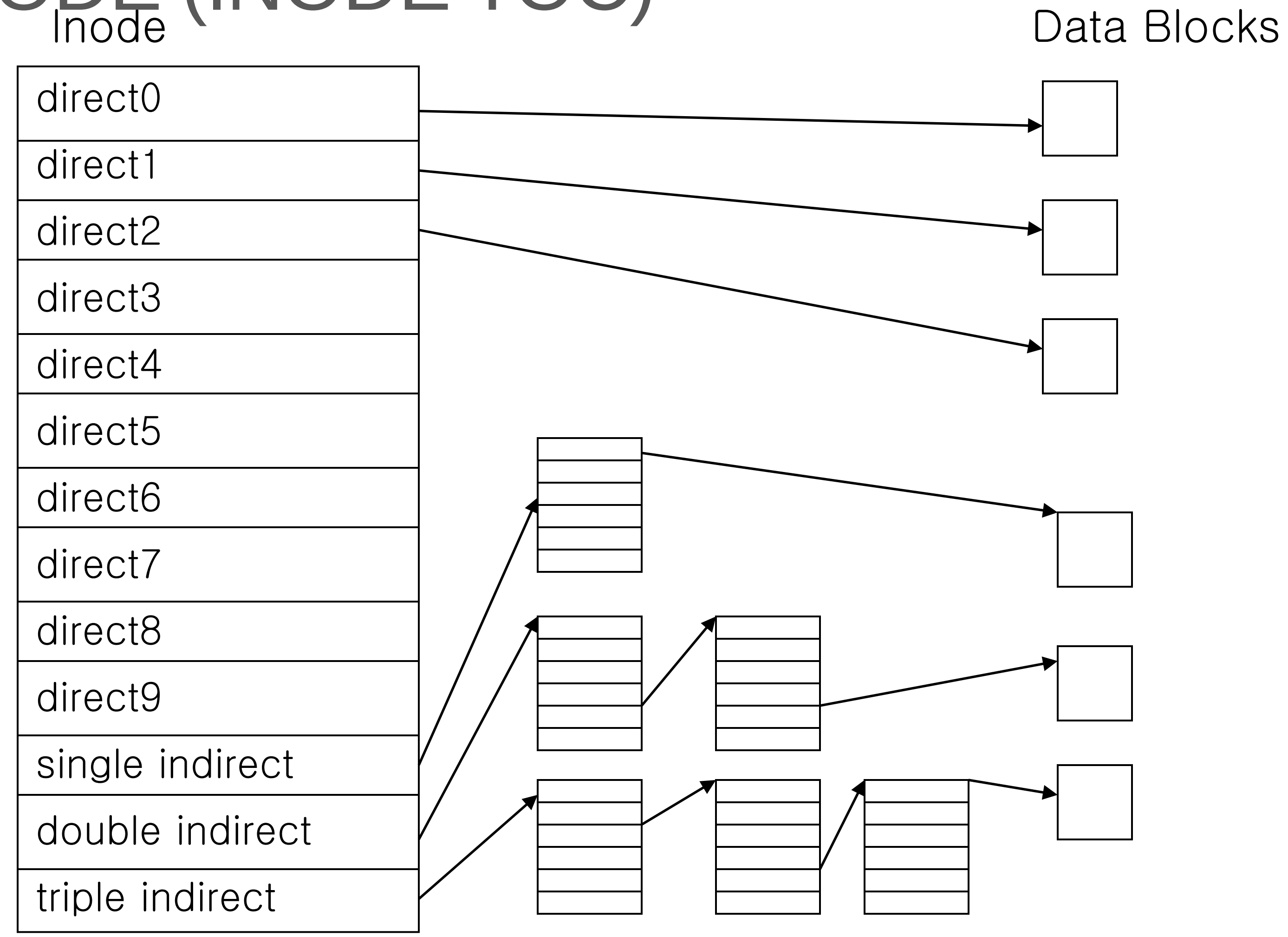
In-core Inode

- An inode is on the free list only if its reference count is 0
- Kernel can reallocate the in-core inode to another disk inode

Buffer header

- No reference count
- It is on the free list if and only if it is unlocked

DIRECT AND INDIRECT BLOCKS IN INODE (INODE TOC)



INODE TOC

Every blk size 1kb

Address of blk 4bytes

Every blk contain 256 pointer

D.I. pointer max size of file 10kb

S.I. pointer max size of file 256kb X 256kb

T.I. pointer max size of file 256kb X 256kb X 256 kb

Max size =256kb X 256kb X 256kb X 10 kb

$= 2^{8} \times 2^{8} \times 2^{8} \times 2^{10}$

QUESTION (JUSTIFY T/F)

Unix permit file size 16gb but it can only access file size of 4gb

ACCESSING INODES

1. Kernel identifies inodes by their file system and inode number
2. Allocate in-core inodes at the request of higher-level algorithms (in-core inode, by iget algorithm)
3. Kernel maps the device number & inode number into a hash queue
4. Search the queue for the inode

…

ALGORITHM FOR ALLOCATION OF IN-CORE INODES

```
algorithm iget
input: file system inode number
output: locked inode
{
        while(not done){
                if(inode in inode cache){
                        if(inode locked){
                                sleep(event inode becomes unlocked);
                                coninue;
                        }
                        if(inode on inode free list) remove from free list;
                        increment inode reference count
                        reutrn(inode);
                }
```

ALGORITHM FOR ALLOCATION OF IN-CORE INODES

```
   /*inode not in inode cache*/
           if(no inodes on free list)
              return(error);
           remove new inode from free list;
           reset inode number and file system;
           remove inode from old hash queue,place on new one;
           read inode from disk(algorithm bread);
           initialize inode (e.g. reference count to 1);
           return(inode);
           }
}
```

BLOCK NUMBER & BYTE OFFSET

Computing logical disk block number

- Block number

= ((inode number –1) / number of inodes per block)

+ start block inode list

Computing byte offset of the inode in the block

- ((inode number –1) mod (number of inodes per block))

* size of disk inode

Inode Lock And Reference Count

Kernel manipulates Inode Lock And Reference Count independently

Inode lock

- Set during execution of a system call to prevent other processes from accessing the inode while it is in use.
- Kernel releases the lock at the conclusion of the system call
- Inode is never locked across system calls.

Reference count

- Kernel increase/decrease when reference is active/inactive
- Prevent the kernel from reallocating an active in-core inode

QUESTION (JUSTIFY T/F)

1. *Inode is never access system call*
2. *Reference count is set to access system call*

RELEASING AN INODE

```
algorithm iput  /* release (put) access to in-core inode */
input: pointer to in-core inode
output: none
{
        lock inode if not already locked;
        decrement inode reference count;
        if(reference count ==0)
        {
                if(inode link count ==0)
                {
                        free disk blocks for file (algorithm free,);
                        set file type to 0;
                        free inode (algorithm ifree,);
                }
                if(file accessed or inode changed or file changed) update disk inode;
                put inode on free list;
        }
        release inode lock;
}
```

File System Calls and Relation to Other Algorithms

File System Calls

Return File Desc	Use of namei		Assign inodes	File Attributes	File I/O	File Sys Structure	Tree Manipulation
open creat dup pipe close	open creat chdir chroot chown chmod	stat link unlink mknod mount umount	creat mknod link unlink	chown chmod stat	read write lseek	mount umount	chdir chown

Lower Level File System Algorithms		
namei	ialloc ifree	alloc free bmap
iget iput		

Buffer allocation algorithms				
getblk	brelse	bread	breada	bwirte

```
Algorithm namei                 /* convert path-name to inode */
Input           : path name
Output          : locked inode
{
        if (path name starts from root)
                working inode = root inode (algorithm iget);
        else
                working inode = current directory inode ( algorithm iget);

        while (there is more path name)
        {
                read next path name component from input;
                varify that working inode is of directory, access permissions OK;
                if (working inode is of root and component is "..")
                        continue;       /* loop back to while */
                read directory ( working inode ) by repeated use of alogrithms
                                bmap, bread and brelse;
                if (component matches an entry in directory (working inode))
                {
                        get inode number for matched component;
                        release working inode (algorithm iput);
                        working inode = inode of matched component (algorithm iget);
                }
```

OPEN

First step a process must take to access the data in a file

Syntax

- fd = open(pathname, flags, modes)
    - pathname : file name
    - flags : type of open (ex. reading, writing)
    - modes : file permissions if the file is being created
    - fd : the user file descriptor , integer

Algorithm for Opening a File

Namei file name -> inode

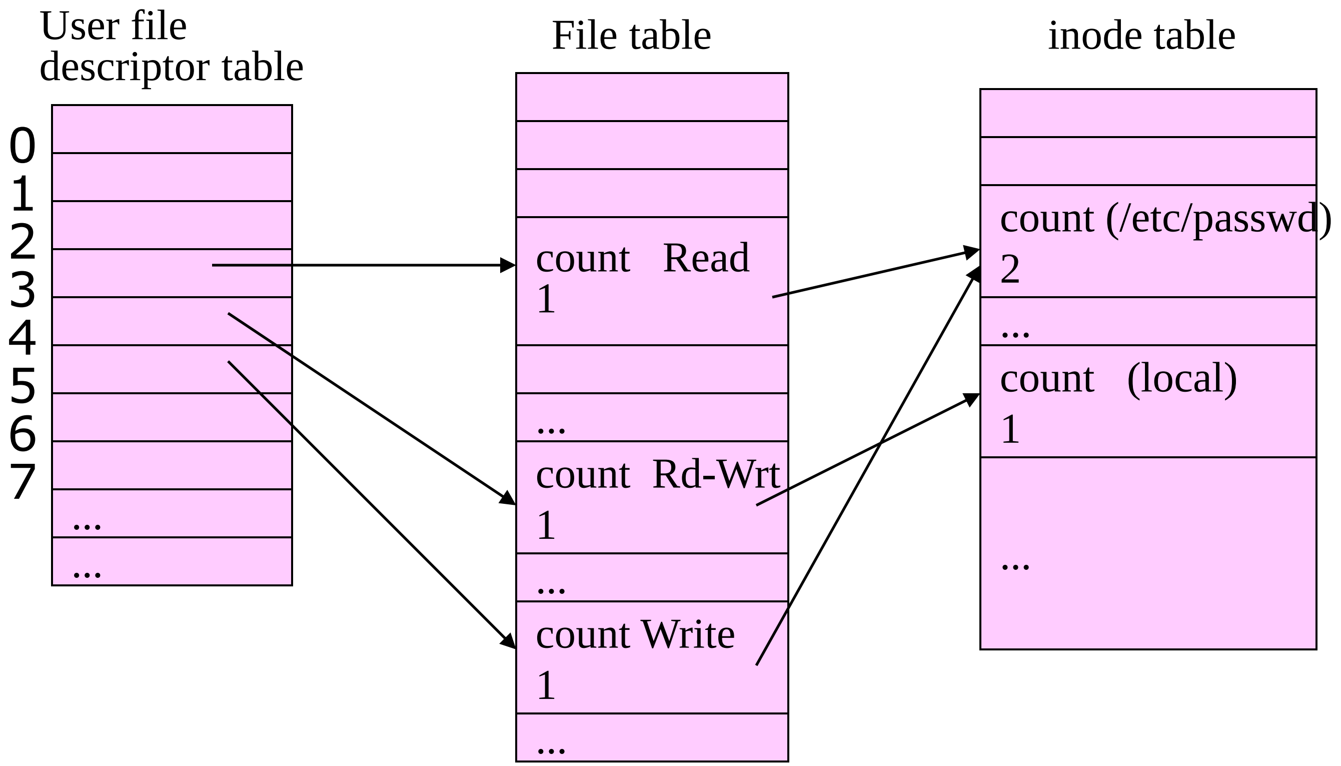
File permission file table entry

Pointer : inode

- Field : byte offset (0 or write-append mode)

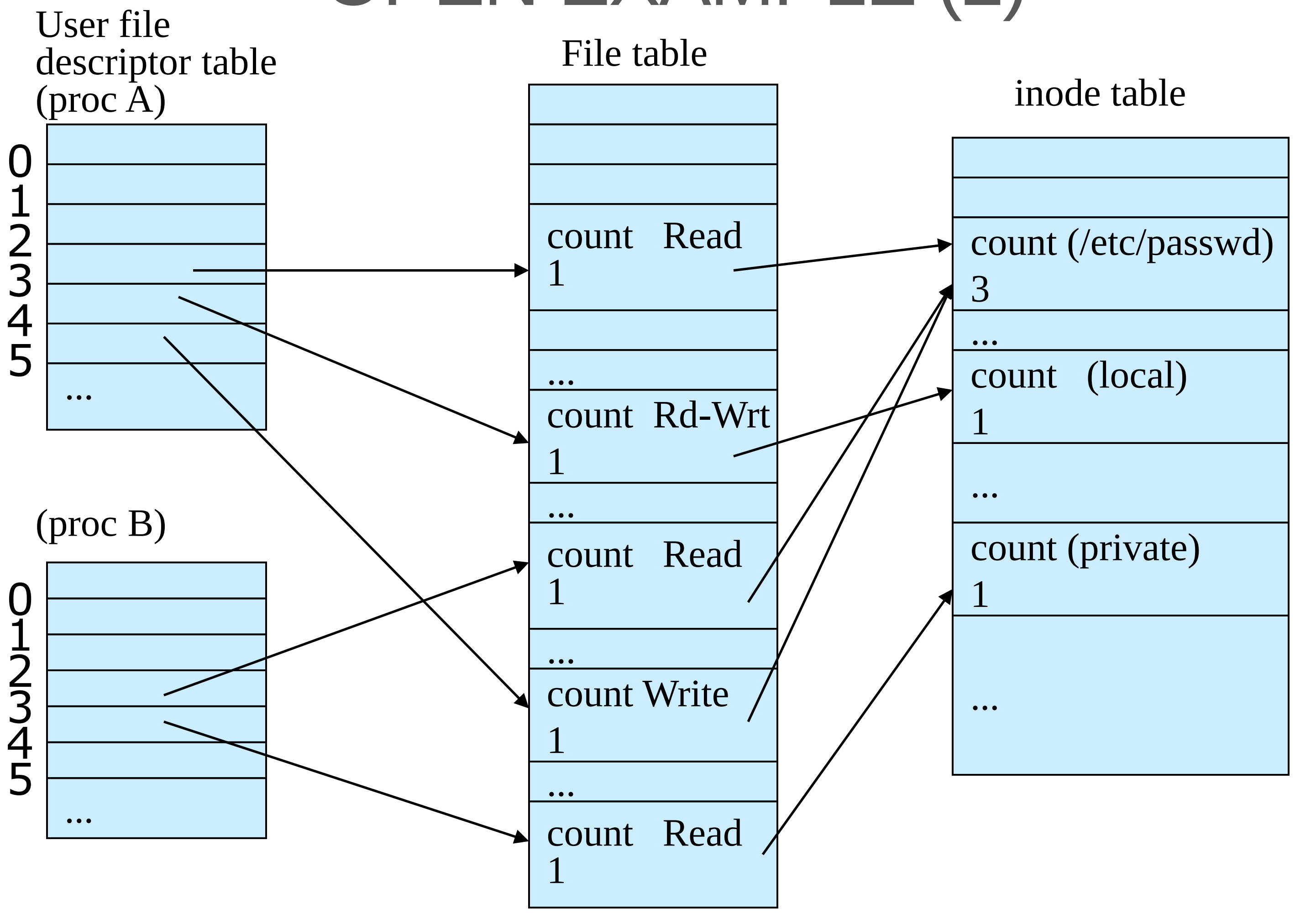
User file descriptor table entry

OPEN EXAMPLE (1)



```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("local", O_RDWR);
fd3 = open("/etc/passwd", O_WRONLY);
```

OPEN EXAMPLE (2)



```
fd1= open("/etc/passwd", O_RDONLY);
fd2 = open("private", O_RDONLY);
```

READ

Syntax

- number = read(fd, buffer, count);
    - fd : file descriptor returned by open
    - buffer : address of a data structure that will contain data
    - count : number of bytes the user want to read
    - number : number of bytes actually read

U –area

- mode : indicates read or write
- count : count of bytes to read or write
- offset : byte offset in file
- address : target address to copy data, in user or kernel memory
- flag : indicates if address is in user or kernel memory

ALGORITHM FOR READING A FILE

Get file table entry from user file descriptor

Set parameters in u area

Get inode from file table and lock inode

Repeat loop until user request is satisfied

- Converting the file byte offset to a block number
- Reading the block from disk to a system buffer
- Copying data from the buffer to the user process
- Releasing the buffer
- Updating I/O parameters in the u area

```
Algorithm read
Input : user file descriptor
        address of buffer in user process
        number of bytes to read
Output : count of bytes copied into user space
{
    get file table entry from user file descriptor;
    check file accessibility;
    set parameters in u area for user address, byte count, I/O to user;
    get inode from file table;
    lock inode;
    set byte offset in u area from file table offset;
    while (count not satisfied)
    {
        convert file offset to disk block (algorithm bmap);
        calculate offset into block, number of bytes to read ;
        if (number of bytes to read is 0)          /* trying to read end of file */
            break;                                 /* out of loop */
        read block (algorithm breada if with read ahead, algorithm bread otherwise);
        copy data from system buffer to user address;
        update u area field for file byte offset, read count, address to write into user space;
        release buffer;                            /* locked in bread */
    }
    unlock inode;
    update file table offset for next read;
    return(total number of bytes read);
}
```

SAMPLE PROGRAM FOR READING A FILE

```
#include <fcntl.h>
main()
{
        int fd;
        char lilbuf[20], bigbuf[1024];
        fd = open(“/etc/passwd”, O_RDONLY);

        read(fd, lilbuf, 20);
        read(fd, bigbuf, 1024);
        read(fd, lilbuf, 20);
}
```

WRITE

Syntax

- number = write(fd, buffer, count);

Algorithm

- If the file does not contain a block that corresponds to the byte offset to be written, the kernel allocate a new block
- The inode is locked
- Update the file size entry in the inode

Delayed write

- Use to write the data to disk
- caching

CLOSE

Close an open file when it no longer wants to access it

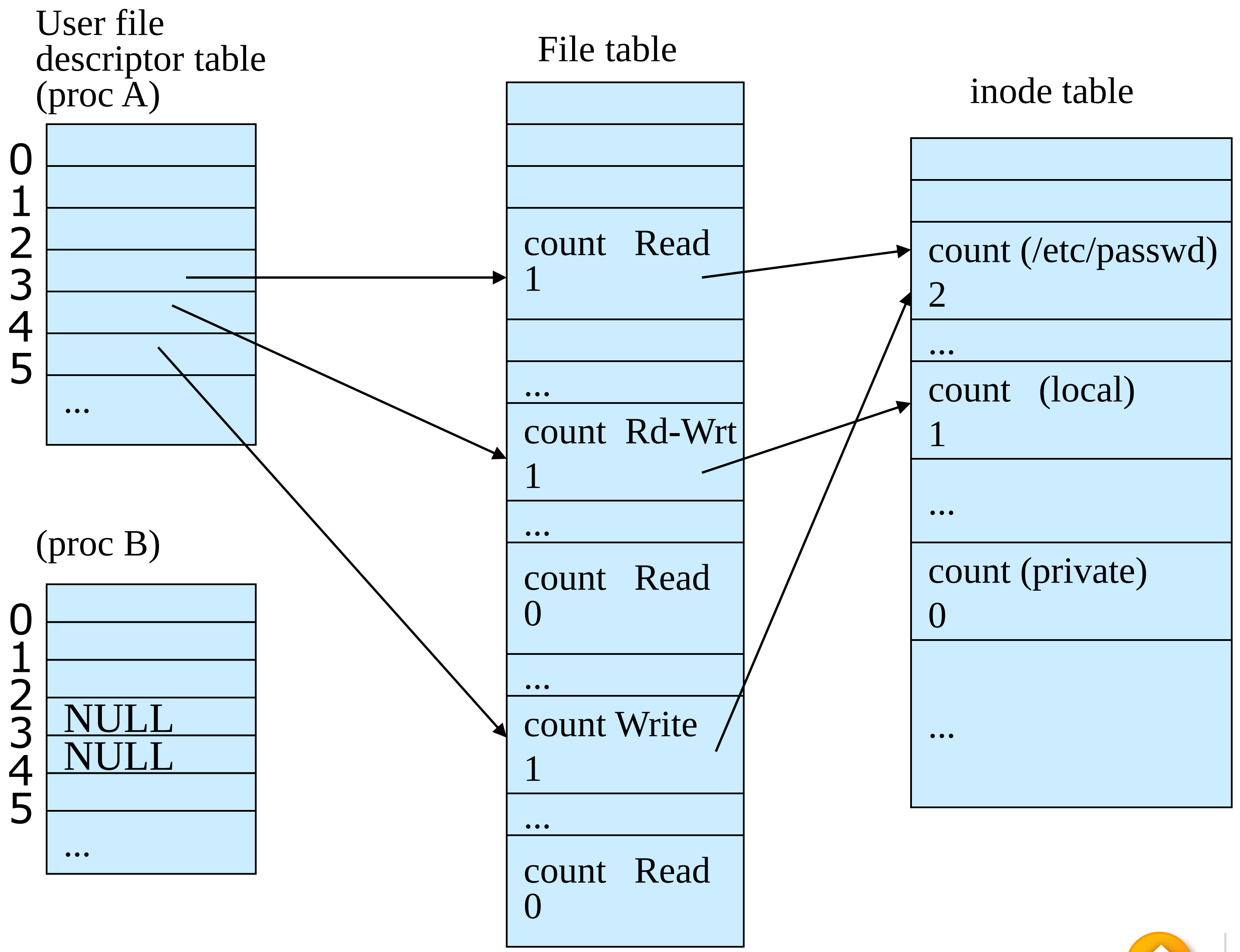
Syntax

- close(fd);
    - fd : file descriptor for the open file

Algorithm

- File descriptor, file table entry , inode table entry
    - reference count > 1
    - reference count = 1
    - if other processes still reference the inode
    - inode reference count = 0
- No process can keep a file open after it terminates

CLOSE Example



PIPES

pipe

- Transfer of data between processes in a FIFO
- Synchronization of process execution
- Traditionally use to store the data

named pipe vs. unnamed pipe

- Process use the open system call for named pipes, but pipe system call create unnamed pipe.

PIPE SYSTEM CALL

Creation of a pipe

Syntax

- pipe (fdptr);
    - fdptr : two file descriptors for reading and writing the pipe

Algorithm

- Assign an inode for a pipe from the pipe device
    - Pipe device : a file system from which the kernel can assign inodes and data block for pipes
- Allocate two file table entries for the read and write descriptor
- Update the information in the in-core inode
    - Count : 1
    - Inode reference count : 2
- Record byte offsets in the inode

  -> FIFO access

```
Algorithm pipe
Input : none
Output : read file descriptor
         write file descriptor
{
   assign new inode from pipe device (algorithm ialloc);
   allocate file table entry for reading, another for writing;
   initialize file table entries to point to new inode;
   allocate user file descriptor for reading, another for writing,
        initialize to point to respective file table entries;
   set inode reference count to 2;
   initialize count of inode readers, writers to 1;
}
```

NAMED PIPE

Semantics are the same as those of unnamed pipe

- Have a directory entry and be accessed by a path name

A process that opens the named pipe for reading will sleep until another process opens the named pipe for writing

- Open a named pipe for reading and a writing exist
- No delay option

READING AND WRITING PIPES

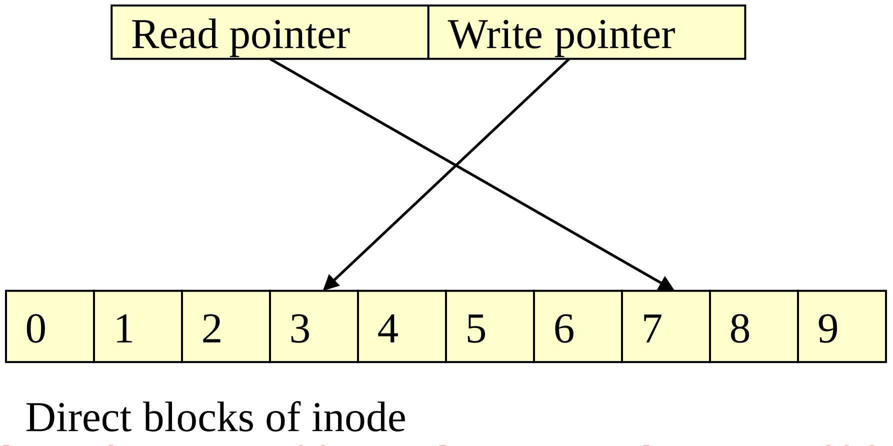
Process access data from a pipe in FIFO manner

Difference :

- Use only the direct blocks of the inode for greater efficiency

Circular queue

- to maintain Read and write pointers internally to preserve the FIFO order



READING AND WRITING PIPES

Four cases

- Writing a pipe that has room for the data being written
- Reading from a pipe that contains enough data to satisfy the read
  - Check the pipe is empty
  - Not empty – as to read regular file
- Reading from a pipe that does not contain enough data to satisfy the read
  - Pipe empty -> sleep
- Writing a pipe that does not have room for the data being written
  - Kernel marks the inode -> sleep

CLOSING PIPES

Same procedure for closing a regular file

Decrements the number of pipe readers or writers according to file descriptor type

PIPES EXAMPLE

```
char string[] = “hello”;
main()
{
        char buf[1024];
        char *cp1, *cp2;
        int fds[2];

        cp1 = string;    cp2 = buf;
        while (*cp1) *cp2++ = *cp1++;

        pipe(fds);
        for (;;) {
            write(fds[1], buf, 6);
            read(fds[0], buf, 6);
          }
 }
```

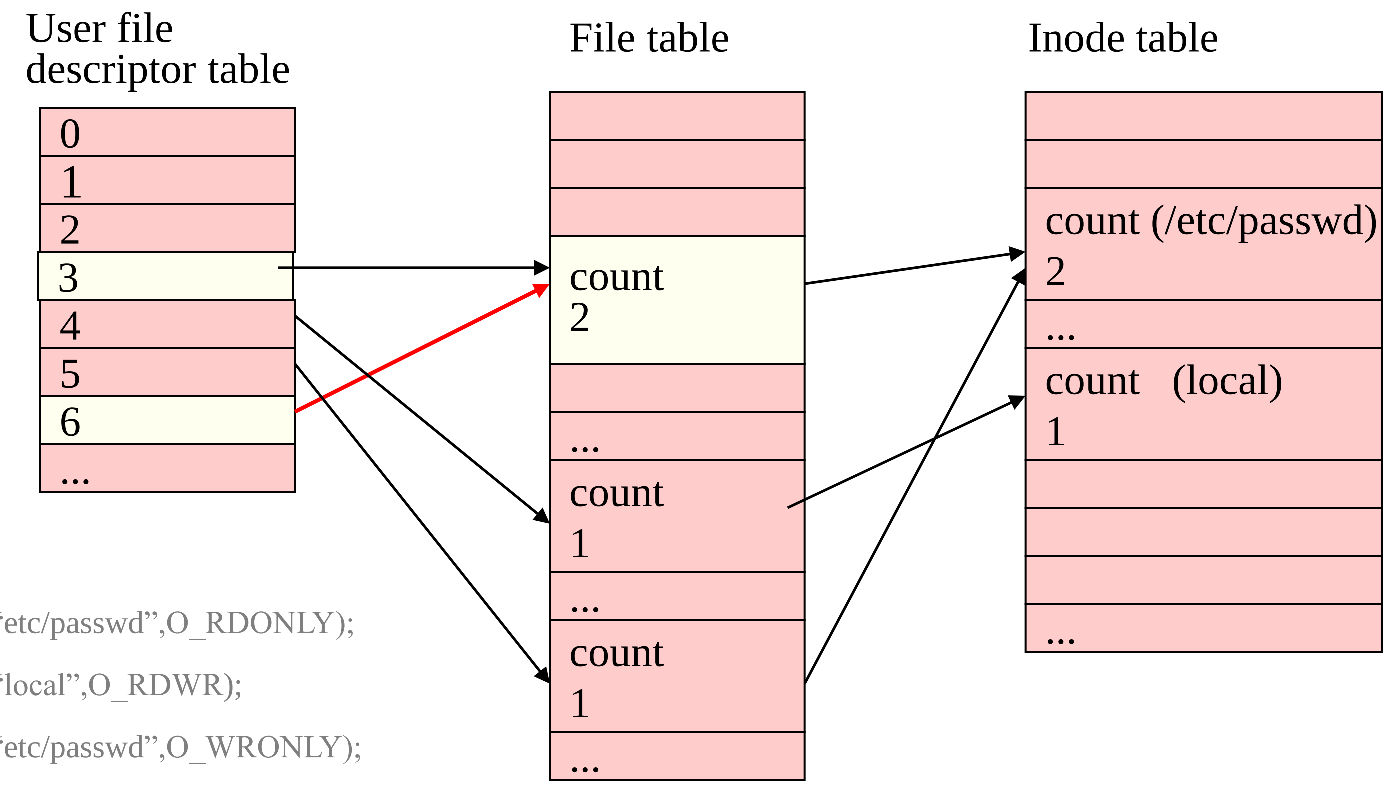
DUP

The dup system call copies a file descriptor into the first free slot of the user file descriptor table ,returning the new file descriptor to the user

Syntax

- newfd = dup(fd);
    - fd : file descriptor being duped
    - newfd : new file descriptor that references the file

DUP Example



Fd1=open("etc/passwd",O_RDONLY);

Fd2=open("local",O_RDWR);

Fd3=open("etc/passwd",O_WRONLY);

Fd4=dup(Fd1);

DUP EXAMPLE

```
#include <fcntl.h>
main()
{
        int i, j;
        char buf1[512], buf2[512];
        i = open("/etc/passwd", O_RDONLY);
        j = dup(i);
        read(i, buf1, sizeof(buf1));
        read(j, buf2, sizeof(buf2));
        close(i);
        read(j, buf2, sizeof(buf2));
}
```

SESSION OUT LINE

open, creat, file sharing, atomic operations, dup2, sync, fsync, and fdatasync, fcntl, /dev/fd, stat, fstat, lstat, file types, Set-User-ID and Set-Group-ID, file access permissions, ownership of new files and directories, access function, umask function, chmod and fchmod, sticky bit,chown, fchown, and lchown, file size, file truncation, file systems, link, unlink, remove, and rename functions, symbolic links, symlink and readlink functions, file times, utime, mkdir and rmdir, reading directories, chdir, fchdir, and getcwd, device special files

OPEN

A file is opened or created by calling the open function.

```
#include <fcntl.h>
int open(const char *pathname, int oflag, ... /* mode_t mode */ );
```

Returns: file descriptor if OK, 1 on error

Flag	Dec.	Implementation
O_RDONLY	Open for reading only.	0
O_WRONLY	Open for writing only.	1
O_RDWR	Open for reading and writing	2

OPEN

O_APPEND	**Append to the end of file on each write.**
O_CREAT	Create the file if it doesn't exist. This option requires a third argument to the open function, the mode, which specifies the access permission bits of the new file.
O_EXCL	Generate an error if O_CREAT is also specified and the file already exists. This test for whether the file already exists and the creation of the file if it doesn't exist is an atomic operation.
O_TRUNC	If the file exists and if it is successfully opened for either write-only or readwrite, truncate its length to 0.
O_NOCTTY	If the pathname refers to a terminal device, do not allocate the device as the controlling terminal for this process
O_NONBLOCK	If the pathname refers to a FIFO, a block special file, or a character special file, this option sets the nonblocking mode for both the opening of the file and subsequent I/O.

creat function

A new file can also be created by calling the creat function.

```
#include <fcntl.h>

 int creat(const char *pathname, mode_t mode);
```

Returns: file descriptor opened for write-only if OK, 1 on error

Note that this function is equivalent to

open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);

There was no way to open a file that didn't already exist. Therefore, a separate system call, creat, was needed to create new files. With the O_CREAT and O_TRUNC options now provided by open, a separate creat function is no longer needed.

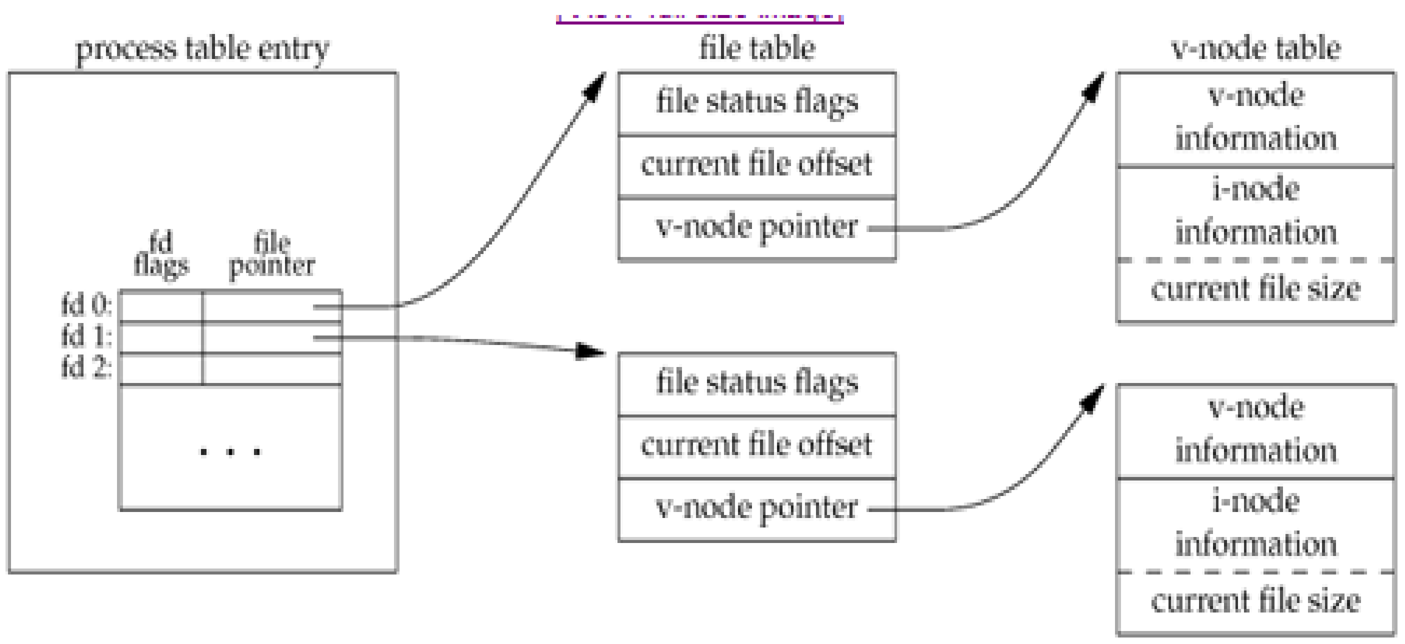
File Sharing

The UNIX System supports the sharing of open files among different processes

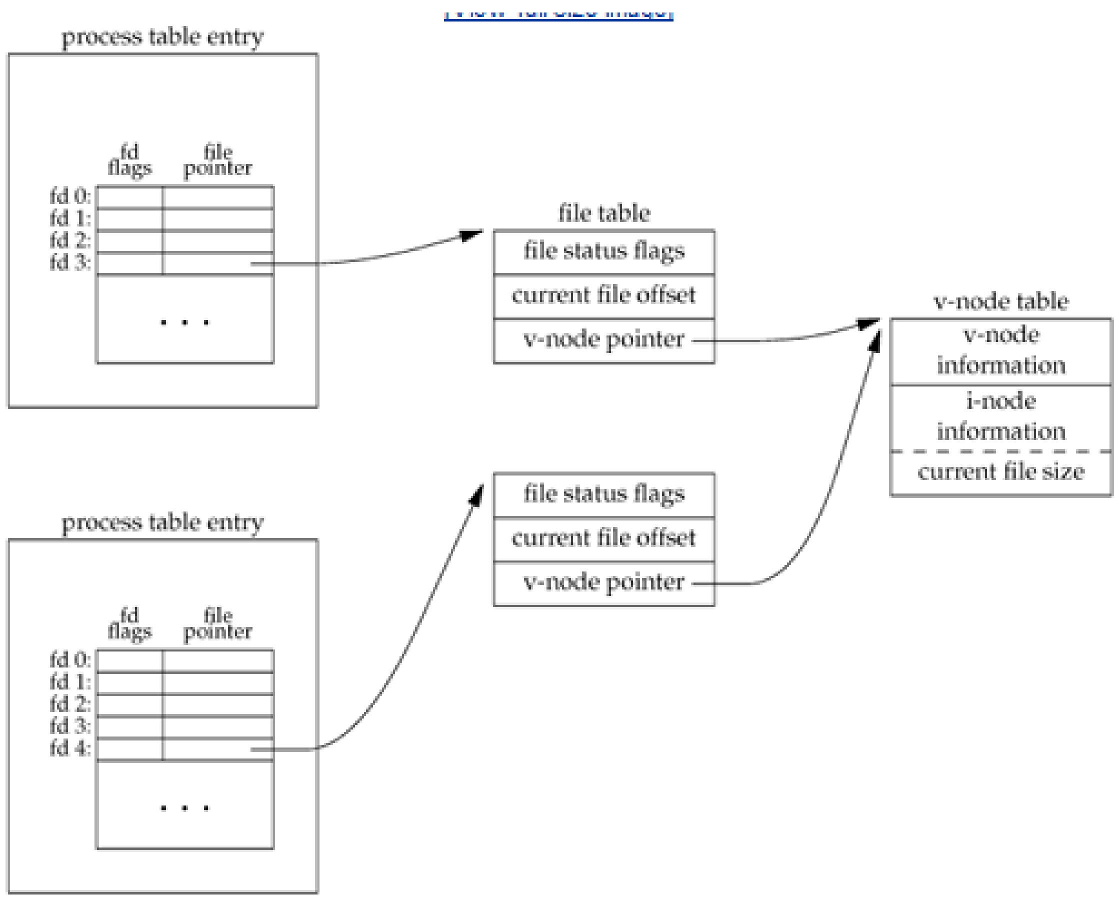
The kernel uses three data structures to represent an open file, and the relationships among them determine the effect one process has on another with regard to file sharing.

1. Every process has an entry in the process table. Within each process table entry is a table of open file descriptors, which we can think of as a vector, with one entry per descriptor. Associated with each file descriptor are
   - The file descriptor flags
   - A pointer to a file table entry
2. The kernel maintains a file table for all open files. Each file table entry contains
   - The file status flags for the file, such as read, write, append, sync, and nonblocking;
   - The current file offset
   - A pointer to the v-node table entry for the file
3. Each open file (or device) has a v-node structure that contains information about the type of file and pointers to functions that operate on the file. For most files, the v-node also contains the i-node for the file. This information is read from disk when the file is opened, so that all the pertinent information about the file is readily available. For example, the i-node contains the owner of the file, the size of the file, pointers to where the actual data blocks for the file are located on disk, and so on.

Kernel Data Structures For Open Files



Kernel Data Structures For Open Files



ATOMIC OPERATIONS

Consider a single process that wants to append to the end of a file. Older versions of the UNIX System didn't support the O_APPEND option to open, so the program was coded as follows:

```
if (lseek(fd, 0L, 2) < 0)            /* position to EOF */
   err_sys("lseek error");
if (write(fd, buf, 100) != 100)         /* and write */
   err_sys("write error");
```

PREAD AND PWRITE FUNCTIONS

The Single UNIX Specification includes XSI extensions that allow applications to seek and perform I/O atomically. These extensions are pread and pwrite.

```
#include <unistd.h>

ssize_t pread(int filedes, void *buf, size_t  nbytes, off_t offset);
```

Returns: number of bytes read, 0 if end of file, 1 on error

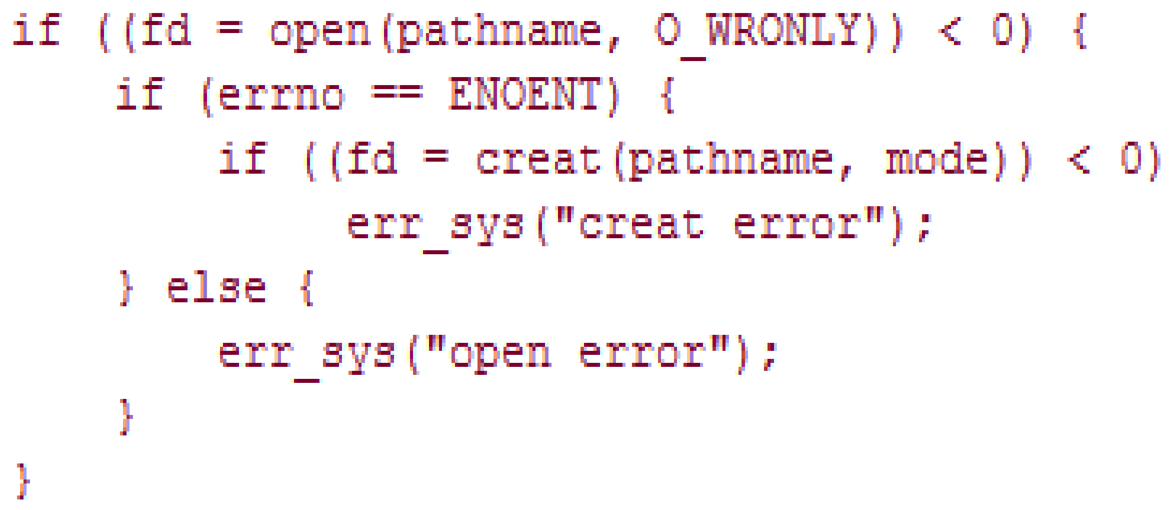
```
ssize_t pwrite(int filedes, const void *buf,  size_t nbytes, off_t offset);
```

Returns: number of bytes written if OK, 1 on error

Calling pread is equivalent to calling lseek followed by a call to read, with the following exceptions.

1. There is no way to interrupt the two operations using pread.
2. The file pointer is not updated.

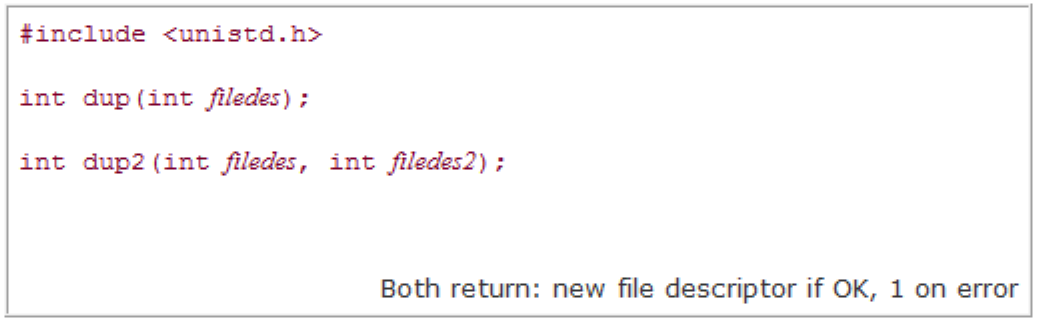
CREATING A FILE



```
if ((fd = open(pathname, O_WRONLY)) < 0) {
    if (errno == ENOENT) {
        if ((fd = creat(pathname, mode)) < 0)
            err_sys("creat error");
    } else {
        err_sys("open error");
    }
}
```

DUP AND DUP2 FUNCTIONS

An existing file descriptor is duplicated by either of the following functions

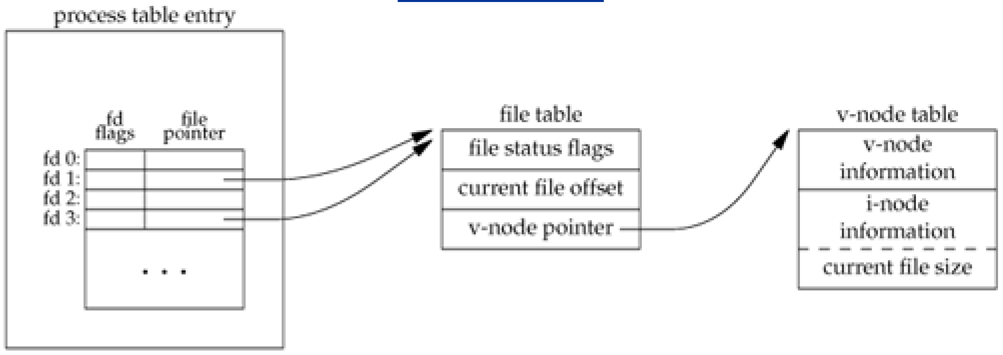


```
#include <unistd.h>

int dup(int filedes);

int dup2(int filedes, int filedes2);
```

Both return: new file descriptor if OK, 1 on error

Kernel Data Structures After Dup(1)

[View full size image]



SYNC, FSYNC, AND FDATASYNC FUNCTIONS

```
#include <unistd.h>
 int fsync(int filedes);
 int fdatasync(int filedes);
```

Returns: 0 if OK, 1 on error

```
void sync(void);
```

Fcntl Function

The fcntl function can change the properties of a file that is already open.

```
#include <fcntl.h>
 int fcntl(int filedes, int cmd, ... /* int arg */ );
```

Returns: depends on cmd if OK (see following), 1 on error

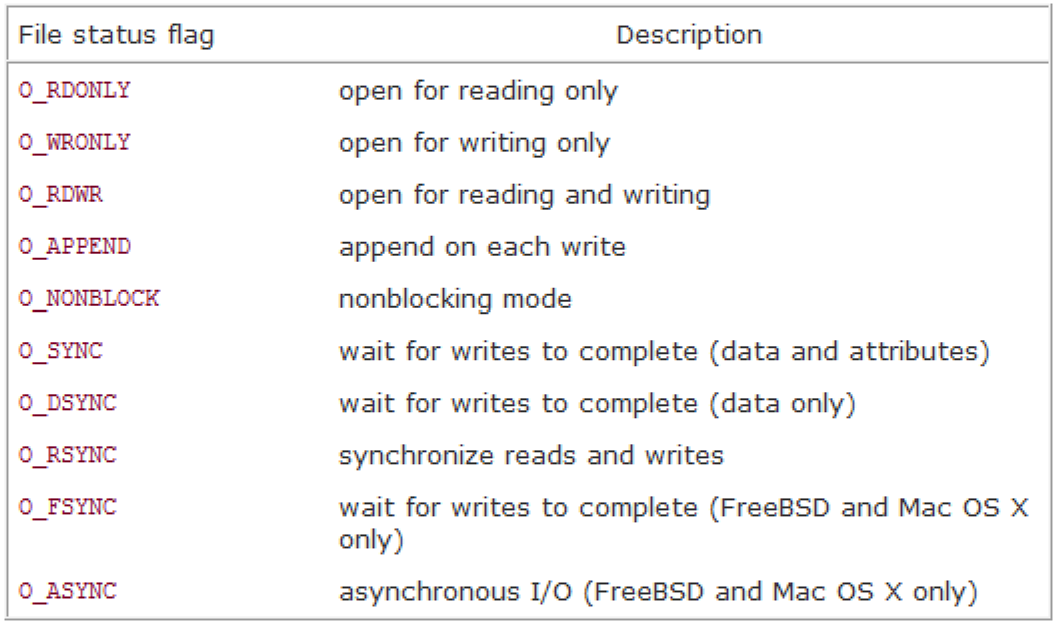
The fcntl function is used for five different purposes.

1. Duplicate an existing descriptor (cmd = F_DUPFD)
2. Get/set file descriptor flags (cmd = F_GETFD or F_SETFD)
3. Get/set file status flags (cmd = F_GETFL or F_SETFL)
4. Get/set asynchronous I/O ownership (cmd = F_GETOWN or F_SETOWN)
5. Get/set record locks (cmd = F_GETLK, F_SETLK, or F_SETLKW)

FCNTL FUNCTION

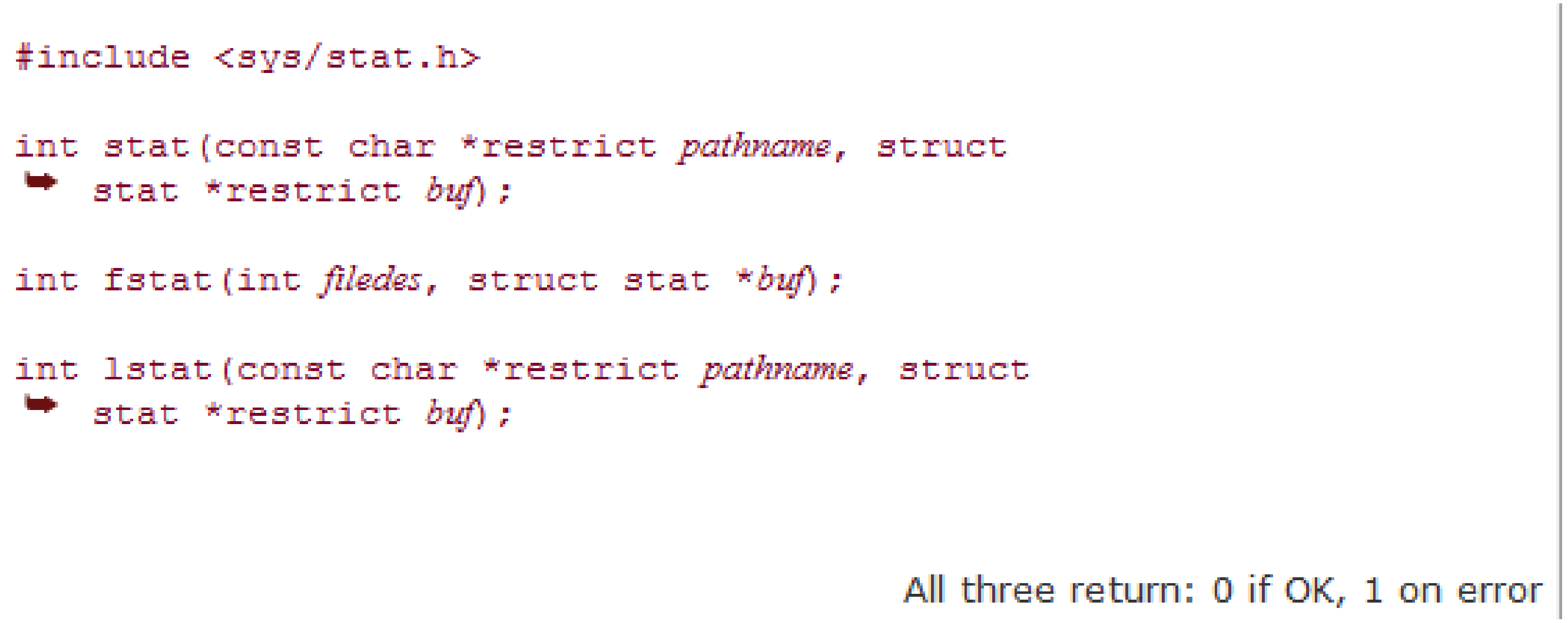
F_DUPFD	**Duplicate the file descriptor filedes.**
F_GETFD	Return the file descriptor flags for filedes as the value of the function. Currently, only one file descriptor flag is defined: the FD_CLOEXEC flag.
F_SETFD	Set the file descriptor flags for filedes. The new flag value is set from the third argument (taken as an integer).
F_GETFL	Return the file status flags for filedes as the value of the function.

FCNTL FUNCTION **FILE STATUS FLAGS FOR FCNTL**



File status flag	Description
O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_APPEND	append on each write
O_NONBLOCK	nonblocking mode
O_SYNC	wait for writes to complete (data and attributes)
O_DSYNC	wait for writes to complete (data only)
O_RSYNC	synchronize reads and writes
O_FSYNC	wait for writes to complete (FreeBSD and Mac OS X only)
O_ASYNC	asynchronous I/O (FreeBSD and Mac OS X only)

STAT, FSTAT, AND LSTAT FUNCTIONS



```
#include <sys/stat.h>

int stat(const char *restrict pathname, struct
    stat *restrict buf);

int fstat(int filedes, struct stat *buf);

int lstat(const char *restrict pathname, struct
    stat *restrict buf);
```

All three return: 0 if OK, 1 on error

STAT, FSTAT, AND LSTAT FUNCTIONS

The stat function returns a structure of information about the named file.

The fstat function obtains information about the file that is already open on the descriptor filedes.

The lstat function is similar to stat, but when the named file is a symbolic link, lstat returns information about the symbolic link, not the file referenced by the symbolic link

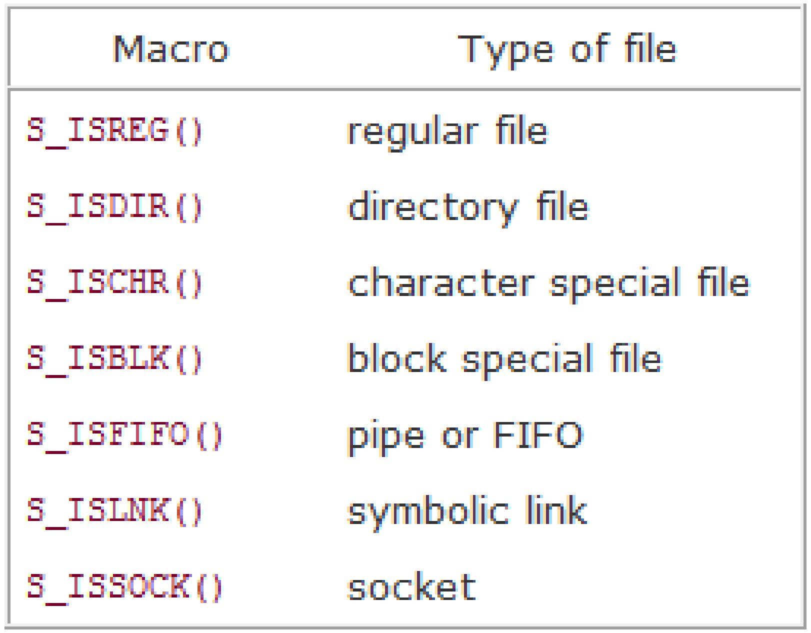
```
struct stat {
  mode_t    st_mode;      /* file type & mode (permissions) */
  ino_t     st_ino;       /* i-node number (serial number) */
  dev_t     st_dev;       /* device number (file system) */
  dev_t     st_rdev;      /* device number for special files */
  nlink_t   st_nlink;     /* number of links */
  uid_t     st_uid;       /* user ID of owner */
  gid_t     st_gid;       /* group ID of owner */
  off_t     st_size;      /* size in bytes, for regular files */
  time_t    st_atime;     /* time of last access */
  time_t    st_mtime;     /* time of last modification */
  time_t    st_ctime;     /* time of last file status change */
  blksize_t st_blksize;   /* best I/O block size */
  blkcnt_t  st_blocks;    /* number of disk blocks allocated */
};
```

FILE TYPES

1. Regular file.
2. Directory file
3. Block special file
4. Character special file
5. FIFO
6. Socket
7. Symbolic link

FILE TYPE MACROS IN

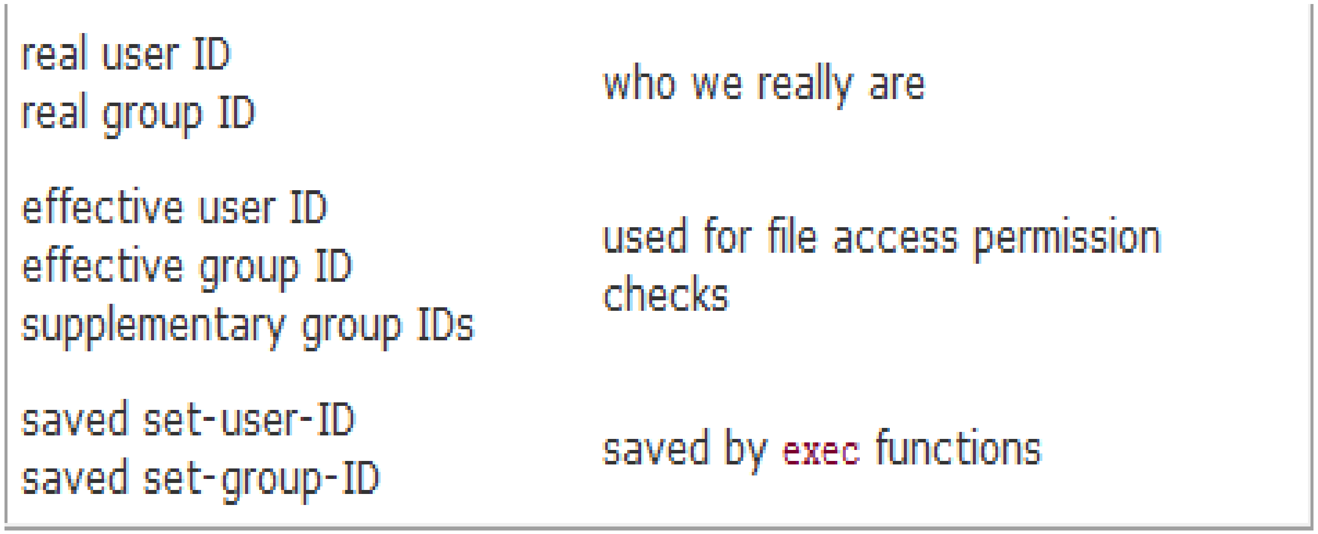
<sys/stat.h>



Macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link
S_ISSOCK()	socket

SET-USER-ID AND SET-GROUP-ID

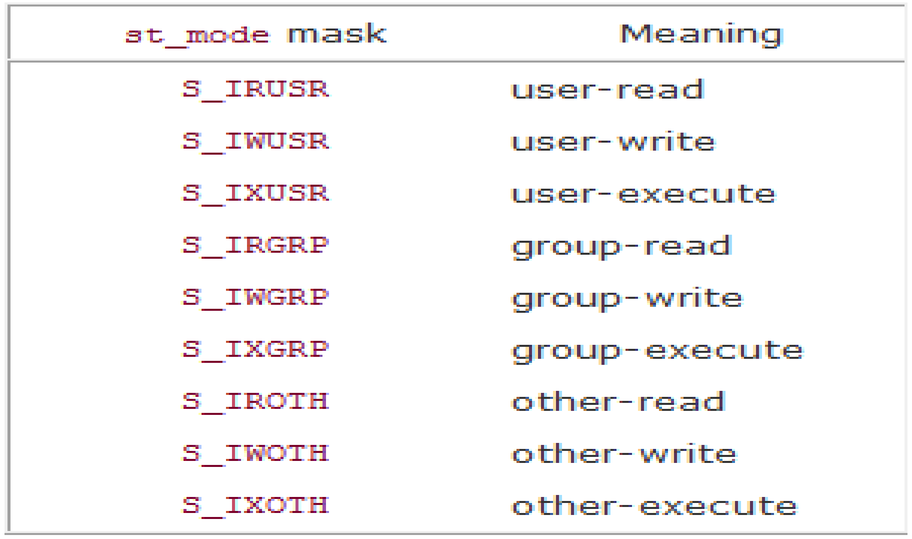
Every process has six or more IDs associated with it.



real user ID real group ID	who we really are
effective user ID effective group ID supplementary group IDs	used for file access permission checks
saved set-user-ID saved set-group-ID	saved by exec functions

FILE ACCESS PERMISSIONS

- All the file types have permissions
- There are nine permission bits for each file, divided into three categories



st_mode mask	Meaning
S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	user-execute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	group-execute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

RULES FOR FILE PERMISSION

1. The first rule is that whenever we want to open any type of file by name, we must have execute permission in each directory mentioned in the name, including the current directory, if it is implied. This is why the execute permission bit for a directory is often called the search bit.
2. The read permission for a file determines whether we can open an existing file for reading: the O_RDONLY and O_RDWR flags for the open function.
3. The write permission for a file determines whether we can open an existing file for writing: the O_WRONLY and O_RDWR flags for the open function.
4. We must have write permission for a file to specify the O_TRUNC flag in the open function.
5. We cannot create a new file in a directory unless we have write permission and execute permission in the directory.
6. To delete an existing file, we need write permission and execute permission in the directory containing the file. We do not need read permission or write permission for the file itself.
7. Execute permission for a file must be on if we want to execute the file using any of the six exec functions The file also has to be a regular file.

OWNERSHIP OF NEW FILES AND DIRECTORIES

The user ID of a new file is set to the effective user ID of the process. POSIX.1 allows an implementation to choose one of the following options to determine the group ID of a new file.

1. The group ID of a new file can be the effective group ID of the process.
2. The group ID of a new file can be the group ID of the directory in which the file is being created.

ACCESS FUNCTION

The access function bases its tests on the real user and group IDs.

```
#include <unistd.h>
 int access(const char *pathname, int mode);
```

Returns: 0 if OK, 1 on error

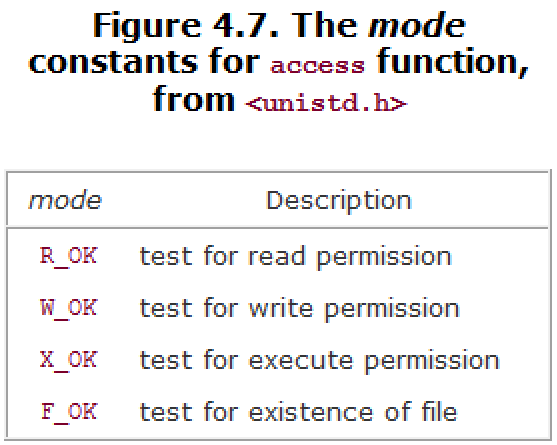


Figure 4.7. The *mode* constants for `access` function, from `<unistd.h>`

mode	Description
`R_OK`	test for read permission
`W_OK`	test for write permission
`X_OK`	test for execute permission
`F_OK`	test for existence of file

UMASK FUNCTION

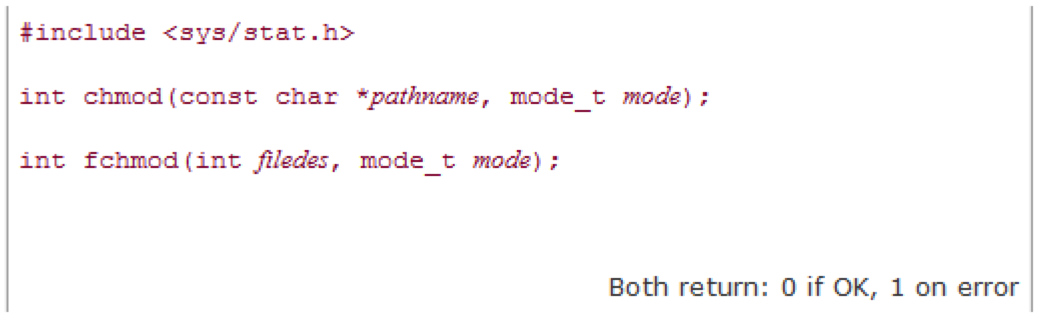
The umask function sets the file mode creation mask for the process and returns the previous value.

```
#include <sys/stat.h>
 mode_t umask(mode_t cmask);
```

Returns: previous file mode creation mask

CHMOD AND FCHMOD FUNCTIONS

These two functions allow us to change the file access permissions for an existing file.



```
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);

int fchmod(int filedes, mode_t mode);
```

Both return: 0 if OK, 1 on error

THE MODE CONSTANTS FOR CHMOD FUNCTIONS, FROM <SYS/STAT.H>

mode	Description
S_ISUID	set-user-ID on execution
S_ISGID	set-group-ID on execution
S_ISVTX	saved-text (sticky bit)
S_IRWXU	read, write, and execute by user (owner)
S_IRUSR	read by user (owner)
S_IWUSR	write by user (owner)
S_IXUSR	execute by user (owner)
S_IRWXG	read, write, and execute by group
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXO	read, write, and execute by other (world)
S_IROTH	read by other (world)
S_IWOTH	write by other (world)
S_IXOTH	execute by other (world)

STICKY BIT

If the bit is set for a directory, a file in the directory can be removed or renamed only if the user has write permission for the directory and one of the following:

Owns the file

Owns the directory

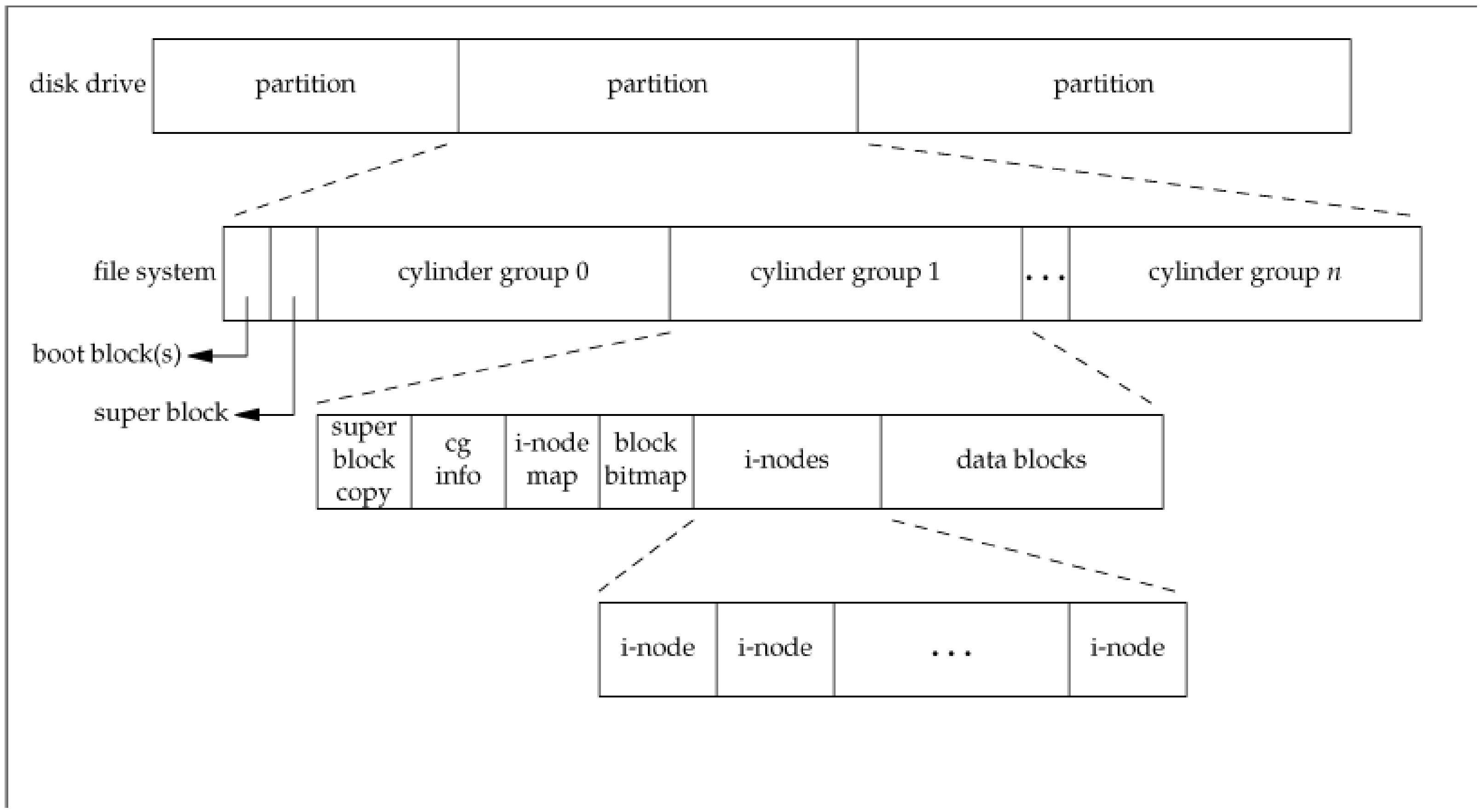
Is the superuser

CHOWN, FCHOWN, AND LCHOWN FUNCTIONS

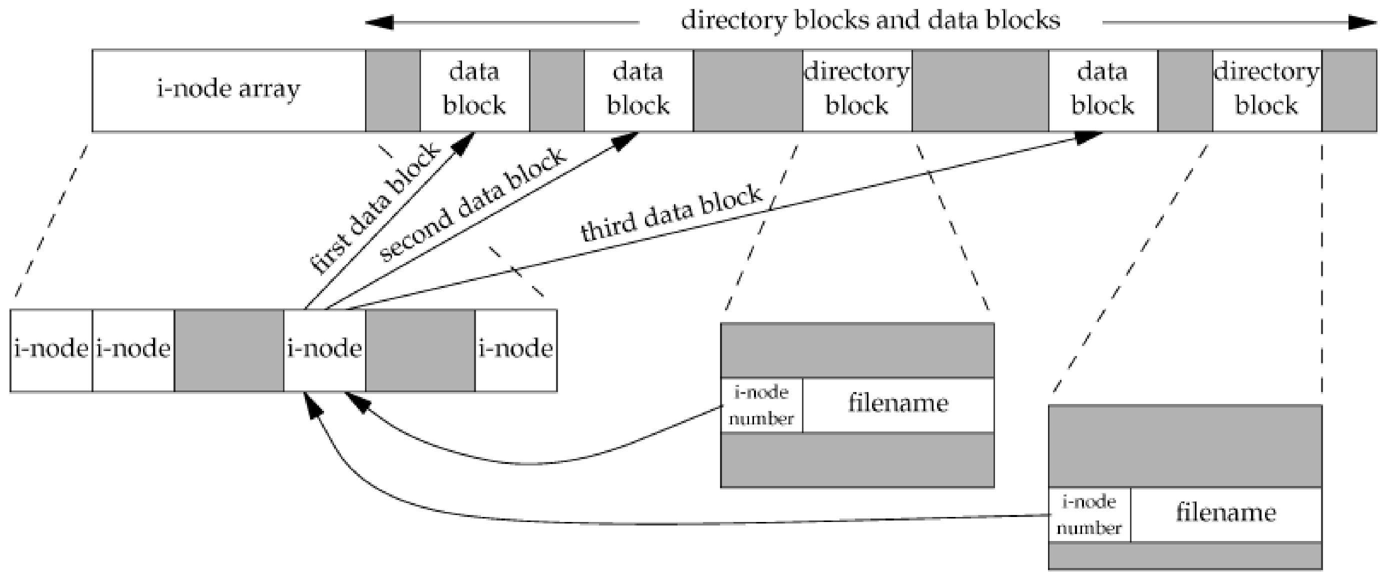
```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t
➥  group);

int fchown(int filedes, uid_t owner, gid_t group);

int lchown(const char *pathname, uid_t owner,
➥  gid_t group);
```

All three return: 0 if OK, 1 on error

FILE SYSTEMS
DISK DRIVE, PARTITIONS, AND A FILE SYSTEM

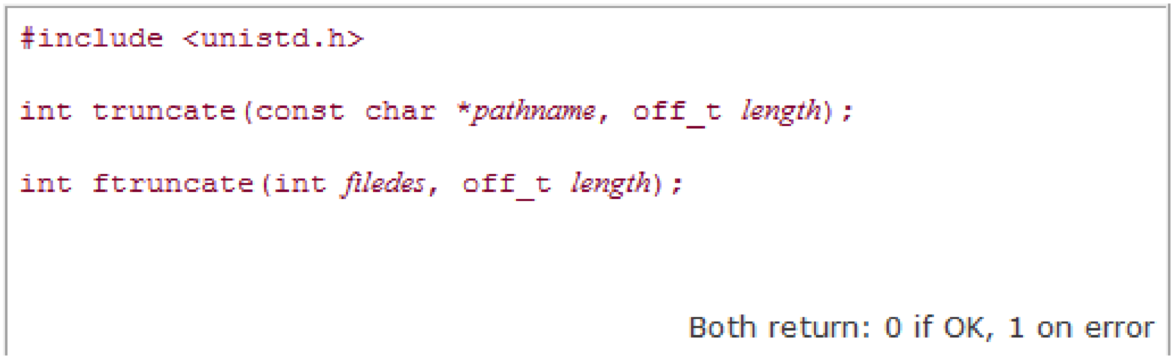


CYLINDER GROUP'S I-NODES AND DATA BLOCKS IN MORE DETAIL



FILE TRUNCATION

There are times when we would like to truncate a file by chopping off data at the end of the file. Emptying a file, which we can do with the O_TRUNC flag to open, is a special case of truncation.



```
#include <unistd.h>

int truncate(const char *pathname, off_t length);

int ftruncate(int filedes, off_t length);
```

Both return: 0 if OK, 1 on error

LINK, UNLINK, REMOVE, AND RENAME FUNCTIONS

```
#include <unistd.h>
int link(const char *existingpath, const char *newpath);
                                        Returns: 0 if OK, 1 on error
int unlink(const char *pathname);
                                        Returns: 0 if OK, 1 on error
#include <stdio.h>
int remove(const char *pathname);
                                        Returns: 0 if OK, 1 on error
                 int rename(const char *oldname, const char *newname);
                                        Returns: 0 if OK, 1 on error
```

SYMBOLIC LINKS

A symbolic link is an indirect pointer to a file, unlike the hard links from the previous section, which pointed directly to the i-node of the file. Symbolic links were introduced to get around the limitations of hard links.

1. Hard links normally require that the link and the file reside in the same file system
2. Only the superuser can create a hard link to a directory

SYMLINK FUNCTION

A symbolic link is created with the symlink function.

```
#include <unistd.h>
 int symlink(const char *actualpath, const char *sympath);
```

Returns: 0 if OK, 1 on error

READLINK FUNCTIONS

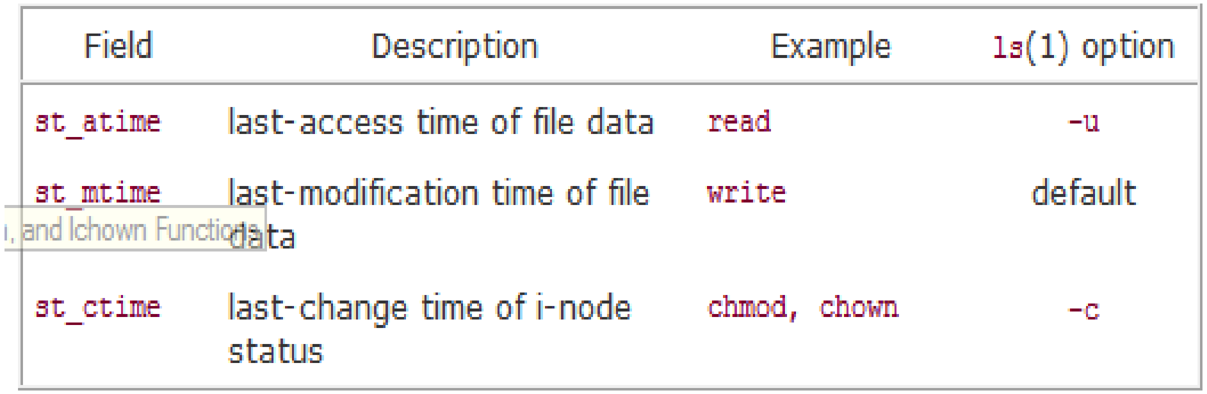
```
#include <unistd.h>

 ssize_t readlink(const char* restrict pathname, char *restrict
buf, size_t bufsize);
```

Returns: number of bytes read if OK, 1 on error

FILE TIMES

Three time fields are maintained for each file.

Note the difference between the modification time (st_mtime) and the changed-status time (st_ctime). The modification time is when the contents of the file were last modified. The changed-status time is when the i-node of the file was last modified



Field	Description	Example	ls(1) option
st_atime	last-access time of file data	read	-u
st_mtime	last-modification time of file data	write	default
st_ctime	last-change time of i-node status	chmod, chown	-c

UTIME FUNCTION

The access time and the modification time of a file can be changed with the utime function.

```
#include <utime.h>
int utime(const char *pathname, const struct utimbuf *times);
```

Returns: 0 if OK, 1 on error

The structure used by this function is

```
struct utimbuf
{
  time_t actime; /* access time */
 time_t modtime; /* modification time */
}
```

UTIME FUNCTION

The operation of this function, and the privileges required to execute it, depend on whether the times argument is NULL.

- If times is a null pointer, the access time and the modification time are both set to the current time. To do this, either the effective user ID of the process must equal the owner ID of the file, or the process must have write permission for the file.
- If times is a non-null pointer, the access time and the modification time are set to the values in the structure pointed to by times. For this case, the effective user ID of the process must equal the owner ID of the file, or the process must be a superuser process. Merely having write permission for the file is not adequate.

Note that we are unable to specify a value for the changed-status time, st_ctime the time the i-node was last changed as this field is automatically updated when the utime function is called.

MKDIR

Directories are created with the mkdir function and deleted with the rmdir function.

```
#include <sys/stat.h>
int mkdir(const char *pathname, mode_t mode);
```

Returns: 0 if OK, 1 on error

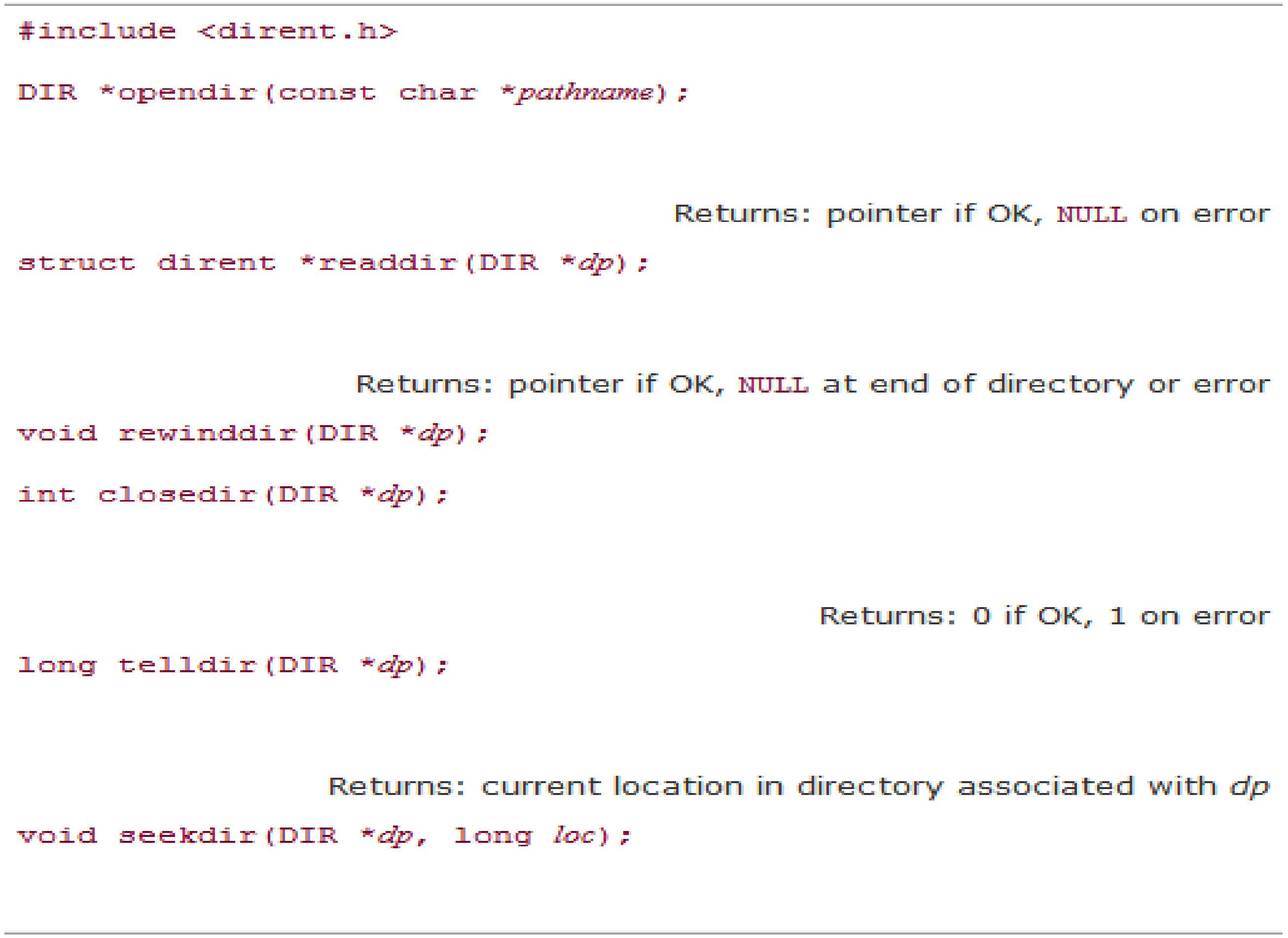
RMDIR

An empty directory is deleted with the rmdir function.

```
#include <unistd.h>
 int rmdir(const char *pathname);
```

Returns: 0 if OK, 1 on error

READING DIRECTORIES



```
#include <dirent.h>

DIR *opendir(const char *pathname);
```

Returns: pointer if OK, NULL on error

```
struct dirent *readdir(DIR *dp);
```

Returns: pointer if OK, NULL at end of directory or error

```
void rewinddir(DIR *dp);

int closedir(DIR *dp);
```

Returns: 0 if OK, 1 on error

```
long telldir(DIR *dp);
```

Returns: current location in directory associated with *dp*

```
void seekdir(DIR *dp, long loc);
```