# K L Deemed to be University
## Department of Computer Science and Engineering

**19CS2106S & 19CS2106A**
**Operating Systems Design**

**Mr. Vishnuvardhan Mannava**
**Course Coordinator**

Topics

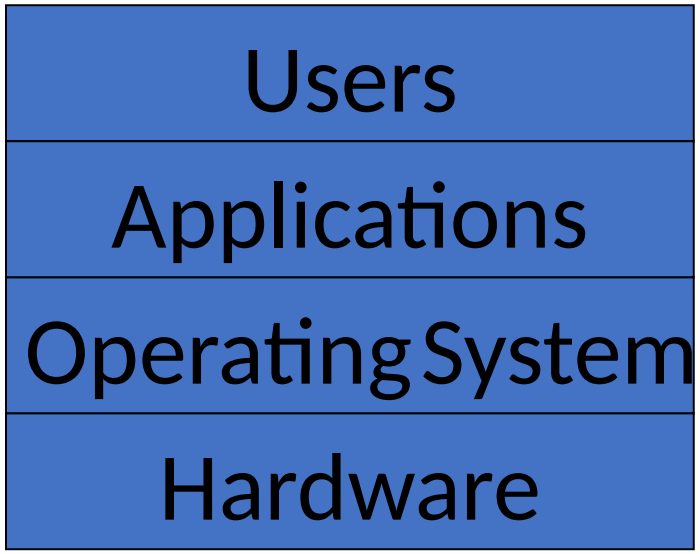
- Course Handout
- Introduction to Operating Systems
- Kernel Architecture

Questions answered in this lecture:

- What will you do in this course?
- What is an OS and why do you want one?
- Why study operating systems?
- What is a kernel?

# What is an Operating System?

Not easy to define precisely…




Operating System (OS):

Software that converts hardware into a useful form for applications

# What DOES OS Provide?

Role #1: Abstraction - Provide standard library for resources

What is a resource?

- Anything valuable (e.g., CPU, memory, disk)

What abstraction does modern OS typically provide for each resource?

- CPU:
  - process and/or thread
- Memory:
  - address space
- Disk:
  - files

Advantages of OS providing abstraction?

- Allow applications to **reuse** common facilities
- Make different devices **look the same**
- Provide higher-level or more **useful functionality**

Challenges

- What are the correct abstractions?
- How much of hardware should be exposed?

# What DOES OS PROVIDE?

Role #2: Resource management – **Share resources** well

Advantages of OS providing resource management?

- **Protect applications** from one another
- Provide **efficient access to resources** (cost, time, energy)
- Provide **fair access** to resources

Challenges

- What are the correct mechanisms?
- What are the correct policies?

# The Core Operating System: The Kernel

The term operating system is commonly used with two different meanings:

- To denote the entire package consisting of the central software **managing a computer's resources and all of the accompanying standard software tools**, such as command-line interpreters, graphical user interfaces, file utilities, and editors.
- More narrowly, to refer to the central software that **manages and allocates** computer resources (i.e., the CPU, RAM, and devices).
- The **kernel** is the central module of an operating system (OS). It is the part of the operating system that loads first, and it remains in main memory. Because it stays in memory, it is important for the kernel to be as small as possible while still providing all the **essential services required by other parts of the operating system** and applications. The kernel code is usually loaded into a protected area of memory to prevent it from being **overwritten by programs** or other parts of the operating system.

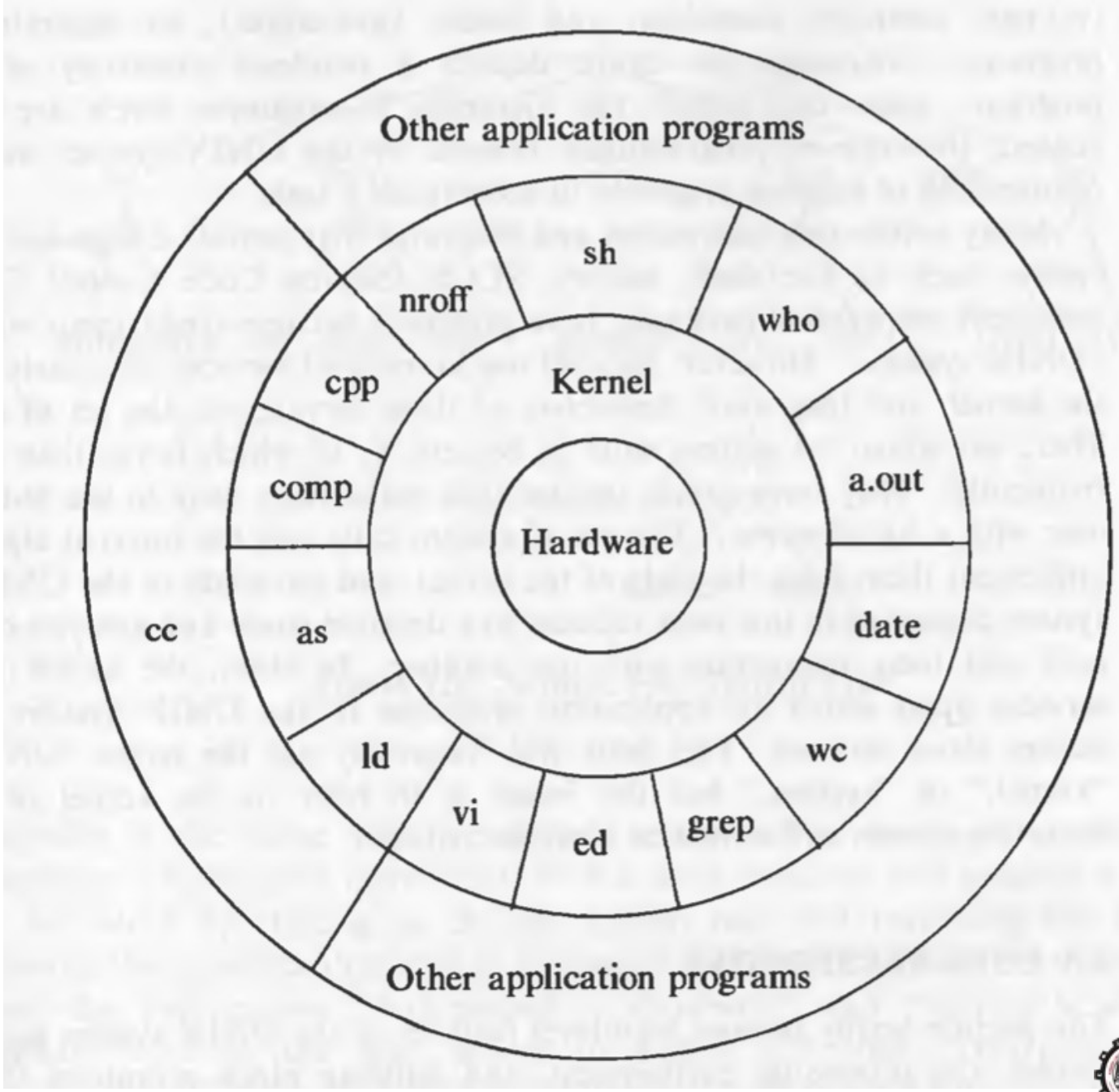
# The Core Operating System: The Kernel

- Typically, the kernel is responsible for **memory management, process and task management, and disk management.** The kernel connects the system hardware to the application software.
- Although it is **possible to run programs on a computer without a kernel**, the presence of a kernel greatly simplifies the writing and use of other programs and increases the power and flexibility available to programmers. The kernel does this by providing a software layer to manage the limited resources of a computer.
- The Linux kernel executable typically resides at the pathname **/boot/vmlinuz**, or something similar. The derivation of this filename is historical. On early UNIX implementations, the kernel was called **unix.** Later UNIX implementations, which implemented virtual memory, renamed the kernel as **vmunix**. On Linux, the filename mirrors the system name, with the z replacing the final x to signify that the kernel is a compressed executable.

# Unix Architecture

The operating systems interacts directly with the hardware, **providing common services to programs** and insulating them from hardware idiosyncrasies.

Programs such as the shell and editors shown in the outer layers **interact with the kernel by invoking a well defined set of *system calls***.



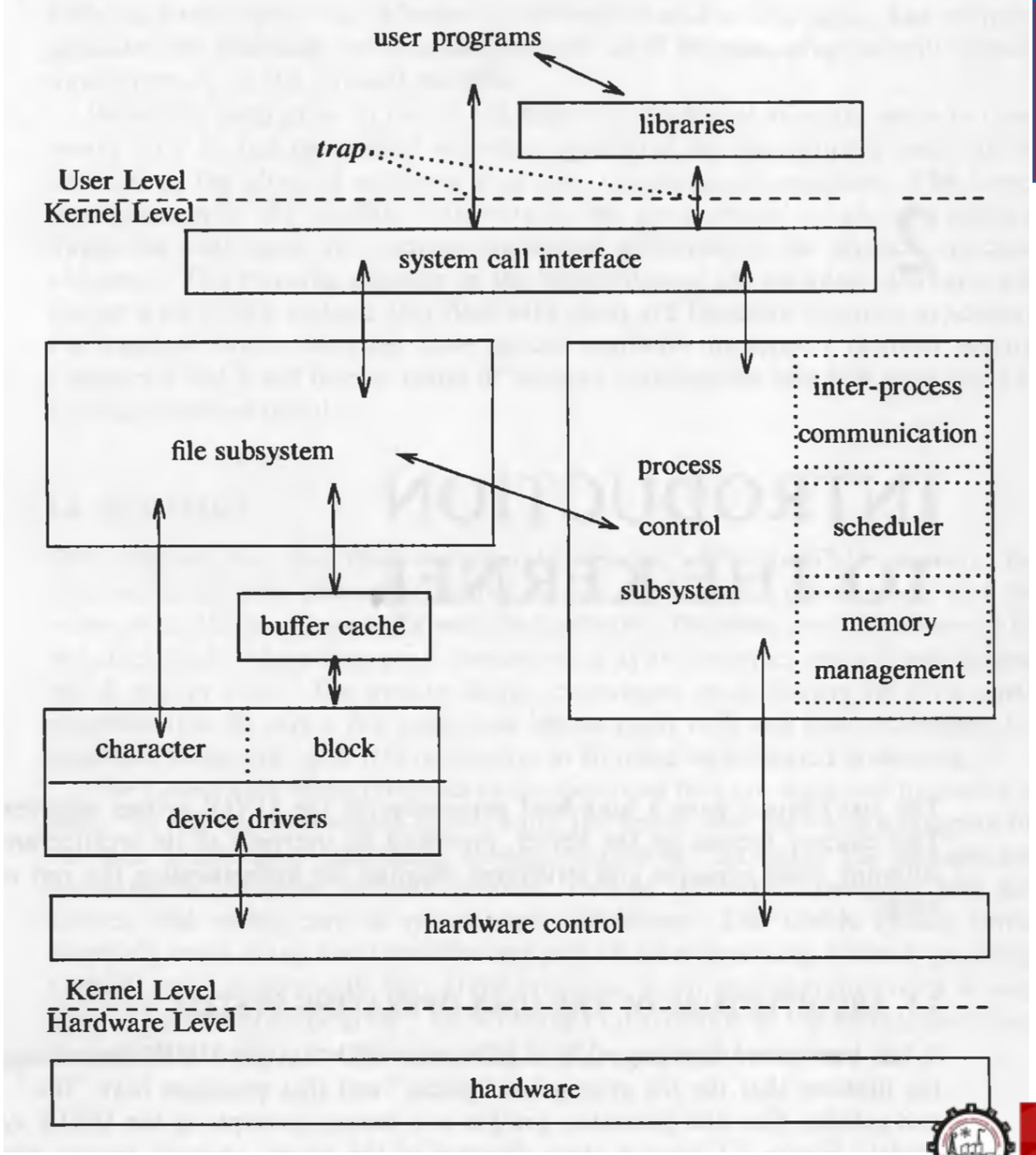
# Unix Family

- Unix and Unix-like operating systems are a family of computer operating systems that derive from the **original Unix System from Bell Labs** which can be traced back to 1965.
- Every operating system has a kernel. For example, the **Linux kernel is used** numerous operating systems including Linux, FreeBSD, Android and others. *Mac OS* X is based *on BSD* UNIX.
- Linux is nothing but a UNIX **clone** which is written Linus Torvalds from scratch with the help of some hackers across the globe.
- **Linux** is the most prominent example of a "real" **Unix** OS. It runs on anything and supports way more hardware than BSD or OS X.

# Unix system v6 Kernel

The two entities, **files and processes**, are the two central concepts in the UNIX system model.

- The *file subsystem* is on the left and the *process control subsystem* is on the right.
- The diagram shows **3 levels : user, kernel, and hardware**.
- The **system call and library interface represent the border between user programs and the kernel**.



# Tasks performed by the kernel: Process scheduling:

Among other things, the kernel performs the following tasks:

**Process scheduling:** A computer has one or more central processing units (CPUs), which **execute the instructions of programs**. Like other UNIX systems, Linux is a **preemptive multitasking operating system**, Multitasking means that multiple processes (i.e., running programs) can simultaneously reside in memory and each may receive use of the CPU(s). Preemptive means that the rules governing which processes receive use of the CPU and for how long are determined by the **kernel process scheduler (rather than by the processes themselves).**

# Tasks performed by the kernel: Memory management

**Memory management:** While computer memories are enormous by the standards of a decade or two ago, the size of software has also correspondingly grown, so that physical memory (RAM) remains a limited resource that the **kernel must share among processes in an equitable and efficient fashion**. Like most modern operating systems, Linux employs virtual memory management, a technique that confers two main advantages:

- Processes are **isolated from one another and from the kernel**, so that one process can't read or modify the memory of another process or the kernel.
- **Only part of a process needs to be kept in memory**, thereby lowering the memory requirements of each process and allowing more processes to be held in RAM simultaneously. This leads to better CPU utilization, since it increases the likelihood that, at any moment in time, there is at least one process that the CPU(s) can execute.

# Tasks performed by the kernel: UNIX treats all devices as files

**Provision of a file system:** The kernel provides a file system on disk, allowing files to be created, retrieved, updated, deleted, and so on.

**Access to devices:** The devices (mice, monitors, keyboards, disk and tape drives, and so on) attached to a computer allow communication of information between the computer and the outside world, permitting input, output, or both. **The kernel provides programs with an interface that standardizes and simplifies access to devices**, while at the same time arbitrating access by multiple processes to each device.

# Tasks performed by the kernel: Provision of a system call application programming interface (API)

- **Creation and termination of processes:** The kernel can load a new program into memory, providing it with the resources (e.g., CPU, memory, and access to files) that it needs in order to run. Such an **instance of a running program is termed a process**. Once a process has completed execution, the kernel ensures that the **resources it uses are freed for subsequent reuse by later programs**.
- **API:** Processes can request the kernel to perform various tasks using **kernel entry points known as system calls.**

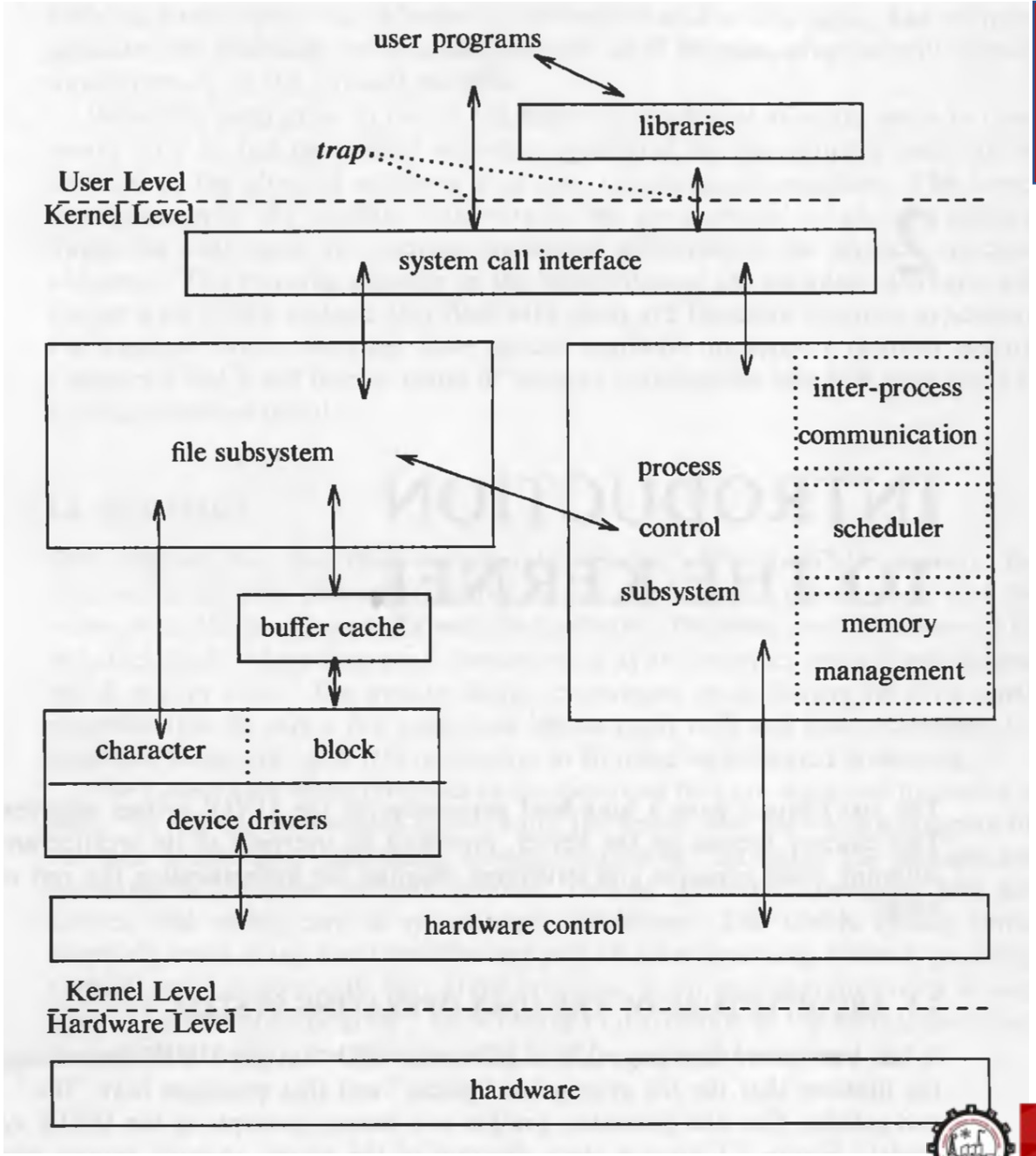
# Tasks performed by the kernel: Networking

**Networking:** The kernel transmits and receives network messages (packets) on behalf of user processes. This task includes **routing of network packets to the target system.**

# Unix system v6 Kernel

The two entities, files and processes, are the two central concepts in the UNIX system model.

- The *file subsystem* is on the left and the *process control subsystem* is on the right.
- The diagram shows 3 levels : user, kernel, and hardware.
- The system call and library interface represent the border between user programs and the kernel.



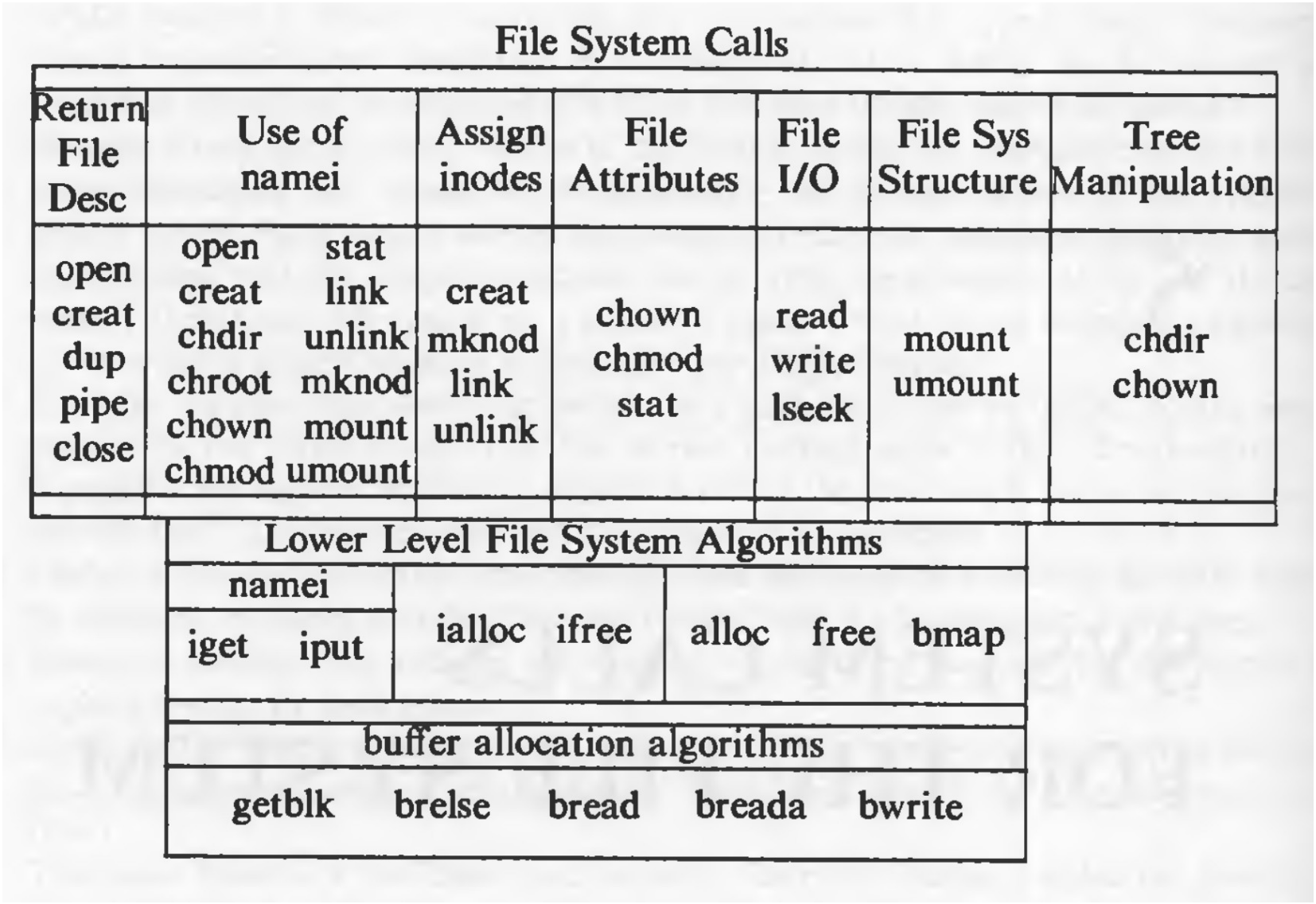
# Unix Kernel: system call interface

- A system call is a controlled entry point into the kernel, **allowing a process to request that the kernel perform some action on the process's behalf**. The kernel makes a range of services accessible to programs via the system call application programming interface (API). These services include, for example, creating a new process, performing I/O, and creating a pipe for interprocess communication. (The syscalls(2) manual page lists the Linux system calls.)
- System calls look like ordinary function calls in C programs, and **libraries map these function calls to the primitives** needed to enter the operating system.
- **Assembly language programs may invoke system calls directly without a system call library,** however. Programs frequently use other libraries such as the standard I/O library to provide a more sophisticated use of the system calls. The libraries are linked with the programs at compile time and are thus part of the user program for purposes of this discussion.

# Unix Kernel: file subsystem

- The **file subsystem** manages files, allocating **file space**, administering free space, controlling access to files and retrieving data for users.
- The file subsystem accesses file data using a **buffering mechanism** that regulates data flow between the **kernel and secondary storage devices**. The buffering mechanism interacts with block I/O device drivers to initiate data transfer to and from the kernel. **Device drivers** are the kernel modules that control the operation of peripheral devices. **Block I/O** devices are random access storage devices; alternatively, their device drivers make them appear to be random access storage devices to the rest of the system.
- Raw devices, sometimes called character devices, include all devices that are not block devices.

# Unix Kernel: File subsystem micro view



# Unix Kernel: process control subsystem

- The **process control subsystem** is responsible for process synchronization, interprocess communication, memory management, and process scheduling. The file subsystem and the process control subsystem interact **when loading a file into memory for execution**. the process subsystem reads executable files into memory before executing them.
- The **memory management** module controls the allocation of memory. If at any time the system does not have enough physical memory for all processes, the kernel moves them between main memory and secondary memory so that all processes get a fair chance to execute.
- The **scheduler** module allocates the CPU to processes. It schedules them to run in turn until they voluntarily relinquish the CPU while awaiting a resource or until the kernel preempts them when their recent run time exceeds a time quantum. The scheduler then chooses the highest priority eligible process to run; the original process will run again when it is the highest priority eligible process available.

# Unix Kernel: Process control sub system micro view

| System Calls Dealing with Memory Management | | | | System Calls Dealing with Synchronization | | | Miscellaneous | |
|---|---|---|---|---|---|---|---|---|
| fork | exec | brk | exit | wait | signal | kill | setpgrp | setuid |
| dupreg<br>attachreg | detachreg<br>allocreg<br>attachreg<br>growreg<br>loadreg<br>mapreg | growreg | detachreg | | | | | |

# Unix Kernel: interprocess communication

- A running Linux system consists of numerous processes, many of which operate **independently** of each other. Some processes, however, **cooperate to achieve their intended purposes**, and these processes need methods of communicating with one another and synchronizing their actions. One way for processes to communicate is by reading and writing information in disk files. However, for many applications, this is too slow and inflexible. There are several forms of interprocess communication, ranging from asynchronous signaling of events to synchronous transmission of messages between processes including the following: **signals, pipes, sockets, file locking, message queues, semaphores, shared memory**.

# Unix Kernel: hardware control

- Finally, the hardware control is responsible for **handling interrupts and for communicating with the machine**. Devices such as disks or terminals may interrupt the CPU while a process is executing. If so, the kernel may **resume execution** of the interrupted process after servicing the interrupt: Interrupts are *not* serviced by special processes but by **special functions in the kernel**, called in the context of the currently running process.

# To DO

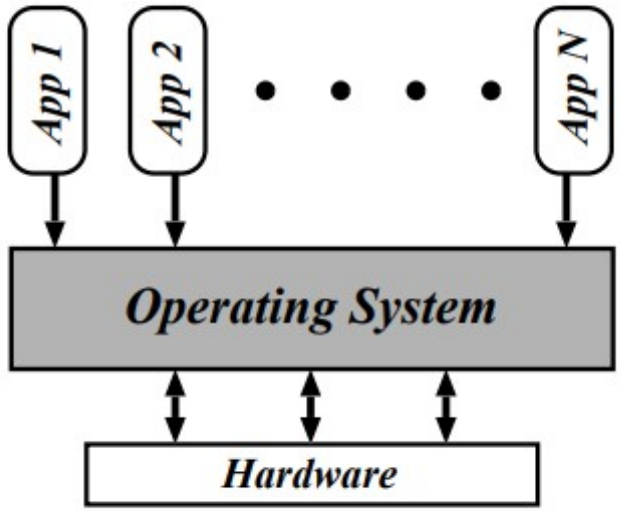
Take a look at course web page in LMS
Take a look at first lab and skilling Exercise
Watch video of next session before attending the class

# RECAP – SESSION-1:

**Questions answered :**

What will you do in this course?
What is an OS and why do you want one?
Why study operating systems?

- Acts as an intermediary between the user(s) and the computer
- System software which controls the components of computer system and coordinates the execution of all other programs (applications).
- The Operating System (OS) is a resource manager
- Hides the complexity and provide convenient environment
- For us, the OS ≈ the kernel.



- Command-line interface
- To provide GUI

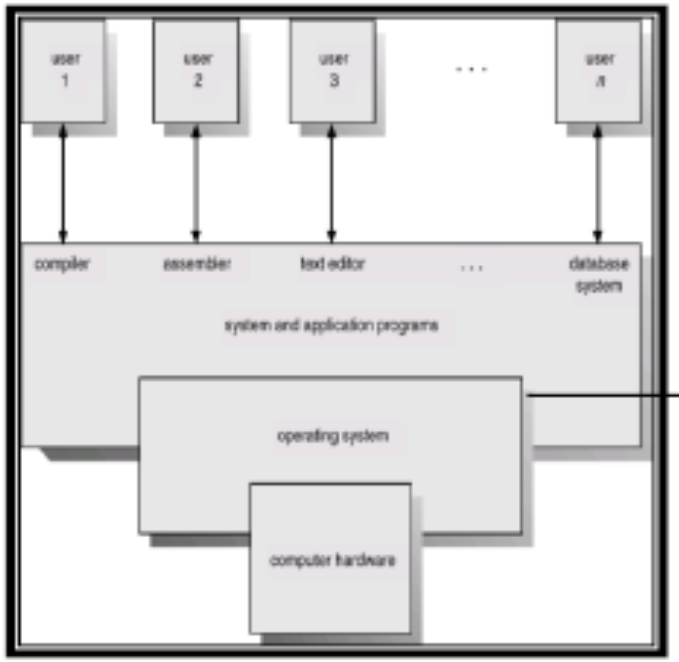
1

- Loading the program, executing, then storing the results back.

2

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

3




Abstract view of the components of a computer system

**Operating System Services**

1. User interface
2. Program execution
3. Process Management
4. Memory Management
5. I/O Device Management
6. File System Management
7. Disk Management
8. Error detection
9. Accounting.
10. Protection and Security

4

- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes (or parts thereof) and data to move into and out of memory
- Allocating and deallocating memory space as needed

5

- Managing the buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

6

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable (non volatile) storage media

7

- Free-space management
- Storage allocation
- Disk scheduling

8

- Identifying different types of errors
- Rectifying them to ensure smooth execution

9

- Recording usage of resources
- Billing for used resources

10

- To ensure that all access to system resources is controlled.
- Authentication mechanisms

**Abstract view of the components of a Computer System with Operating System services**

# Session 2

**Outcome :**

**Understand Operating Systems Design Approaches**

OS DESIGN APPROACHES

- ✓ Simple
- ✓ Monolithic
- ✓ Layered
- ✓ Microkernel
- ✓ Modular
- ✓ Hybrid
- ✓ iOS
- ✓ Android

# Operating System Design Approaches

- Internal structure of different Operating Systems can vary widely
- Start by defining goals and specifications
- Affected by choice of hardware, type of system(Portability)
- *User* goals and *System* goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

- Important principle to separate

  **Policy:** What will be done?
  **Mechanism:** How to do it?

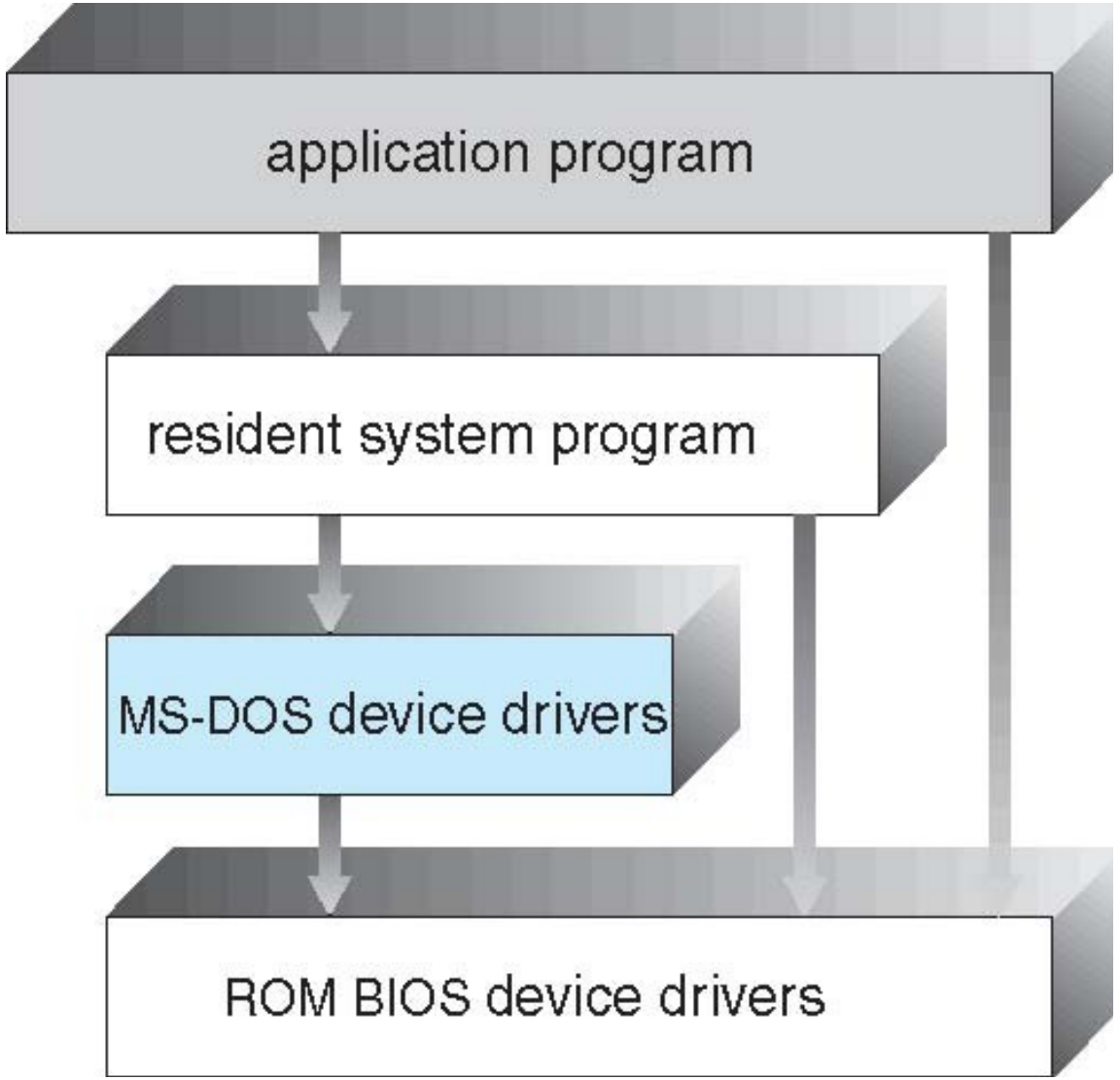
- Mechanisms determine how to do something; policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

# Simple Structure-

## MS-DOS Layer Structure

MS-DOS – written to provide the most functionality in the least space

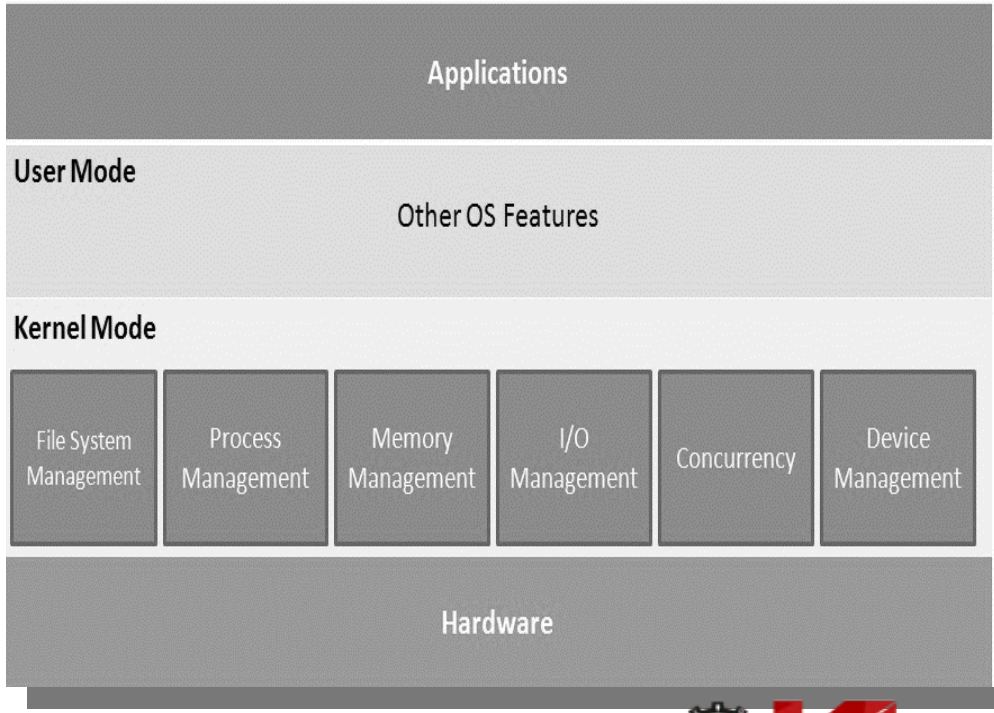
- Not divided into modules
- Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated



# Monolithic architecture

Monolithic kernel has all the operating system functions or services within a single kernel.

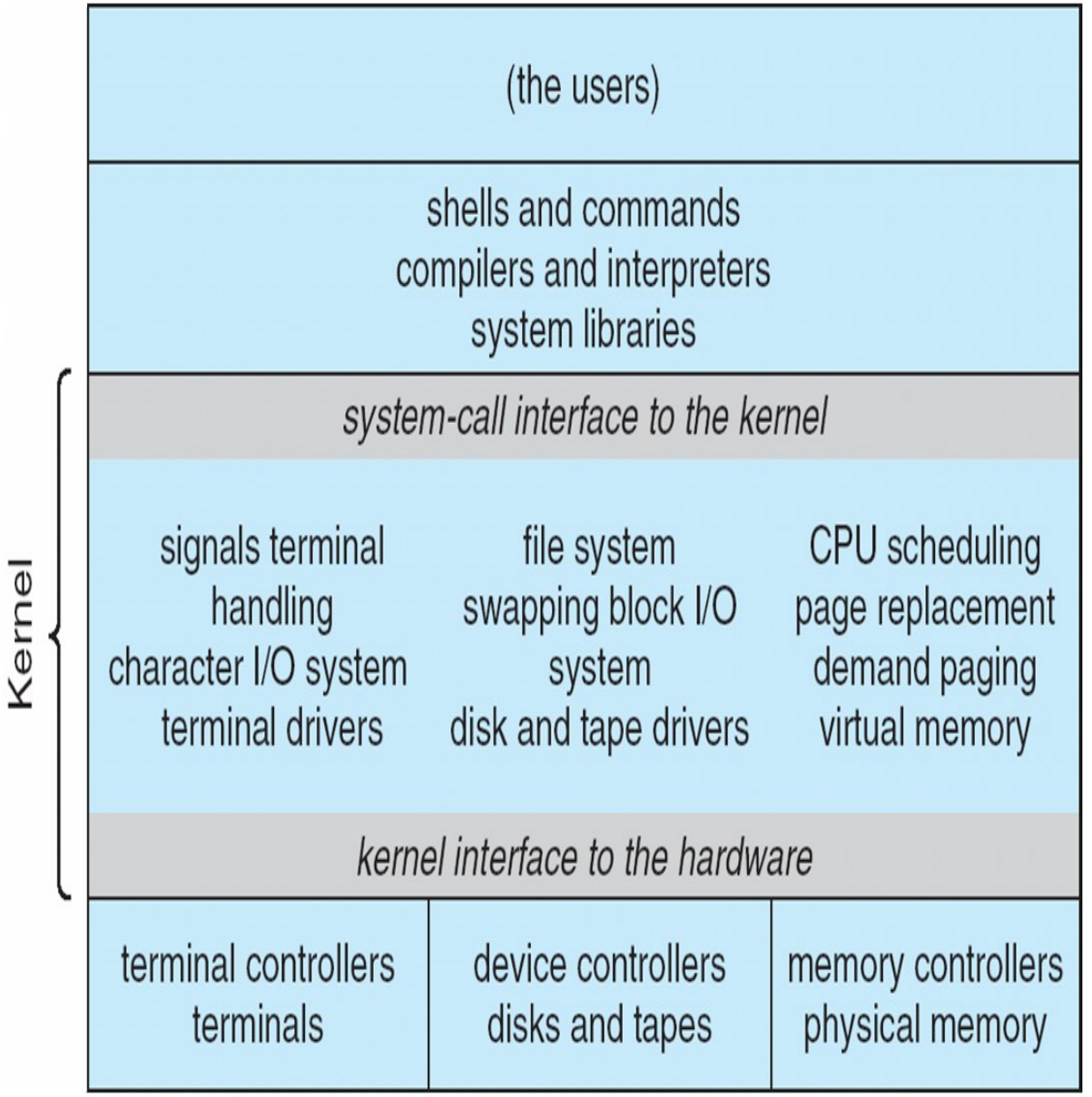
- Run as a single process in a single address space in memory.
- Less flexible for modifications
- Kernel mode and User mode
  - Kernel mode - has all the primary or core operation system features
  - User mode has the operating system features not added in the kernel mode
- Windows 95, 98, ME.
- All Linux distributions.
- Android uses a modified Linux Kernel



# Monolithic Structure – Original UNIX

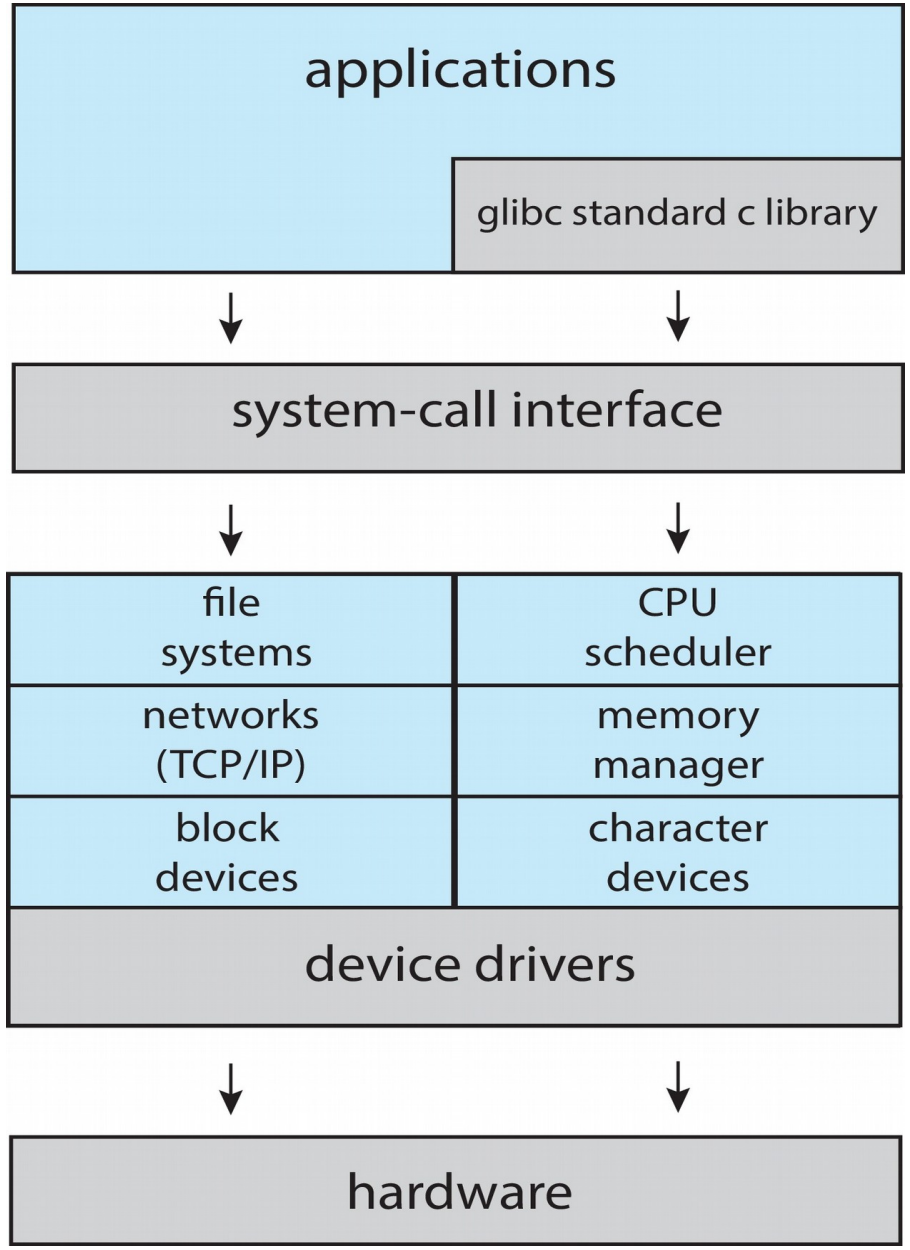
- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

Beyond simple but not fully layered



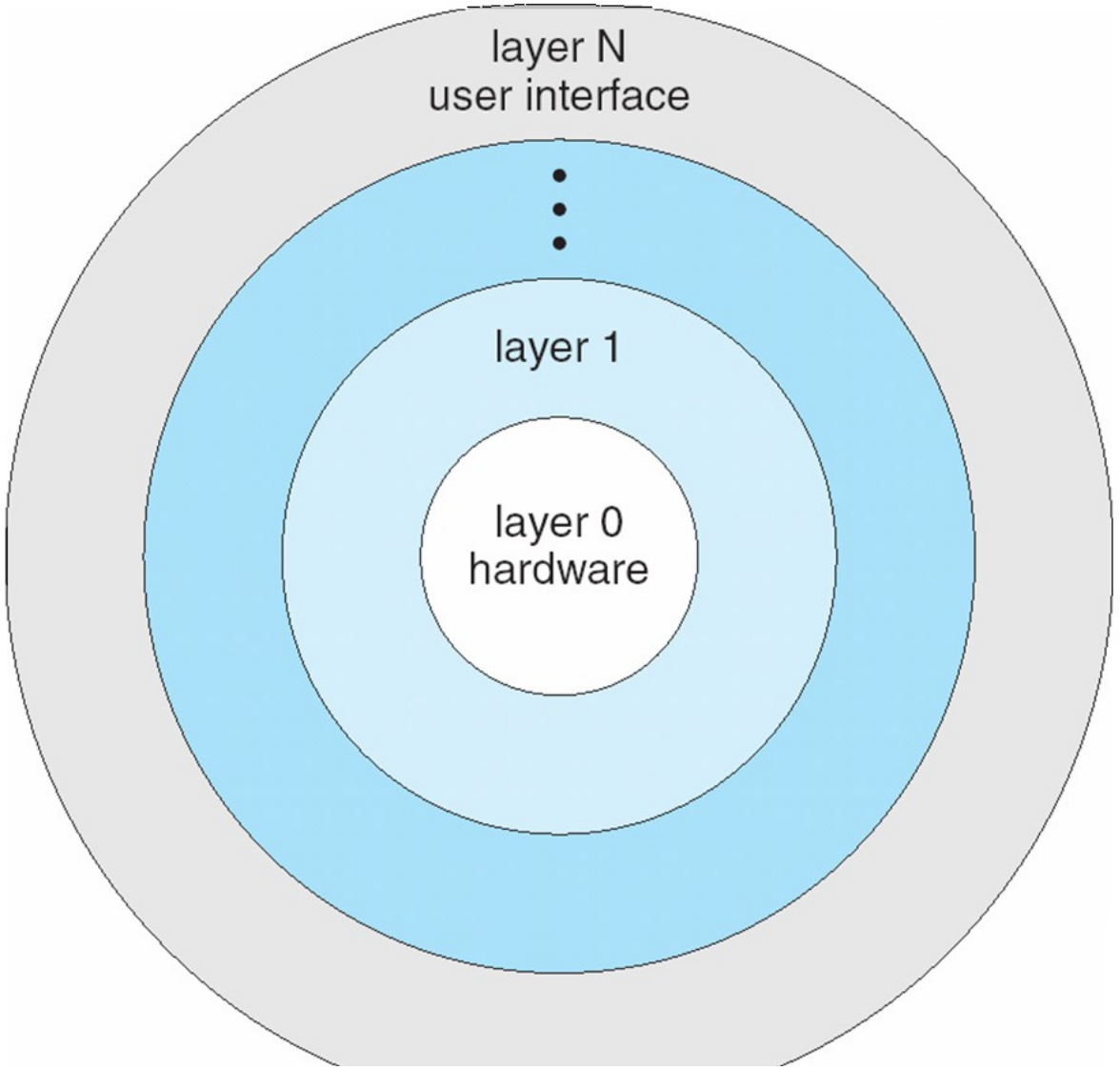
# Linux System Structure

Monolithic plus modular design



# Layered Approach

- A system can be made modular in many ways. One method is the **layered approach**, in which the operating system is broken up into a number of layers (levels).
- The bottom **layer** (**layer** 0) id the hardware; the highest (**layer** N) is the user interface. ... This **approach** simplifies debugging and system verification.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

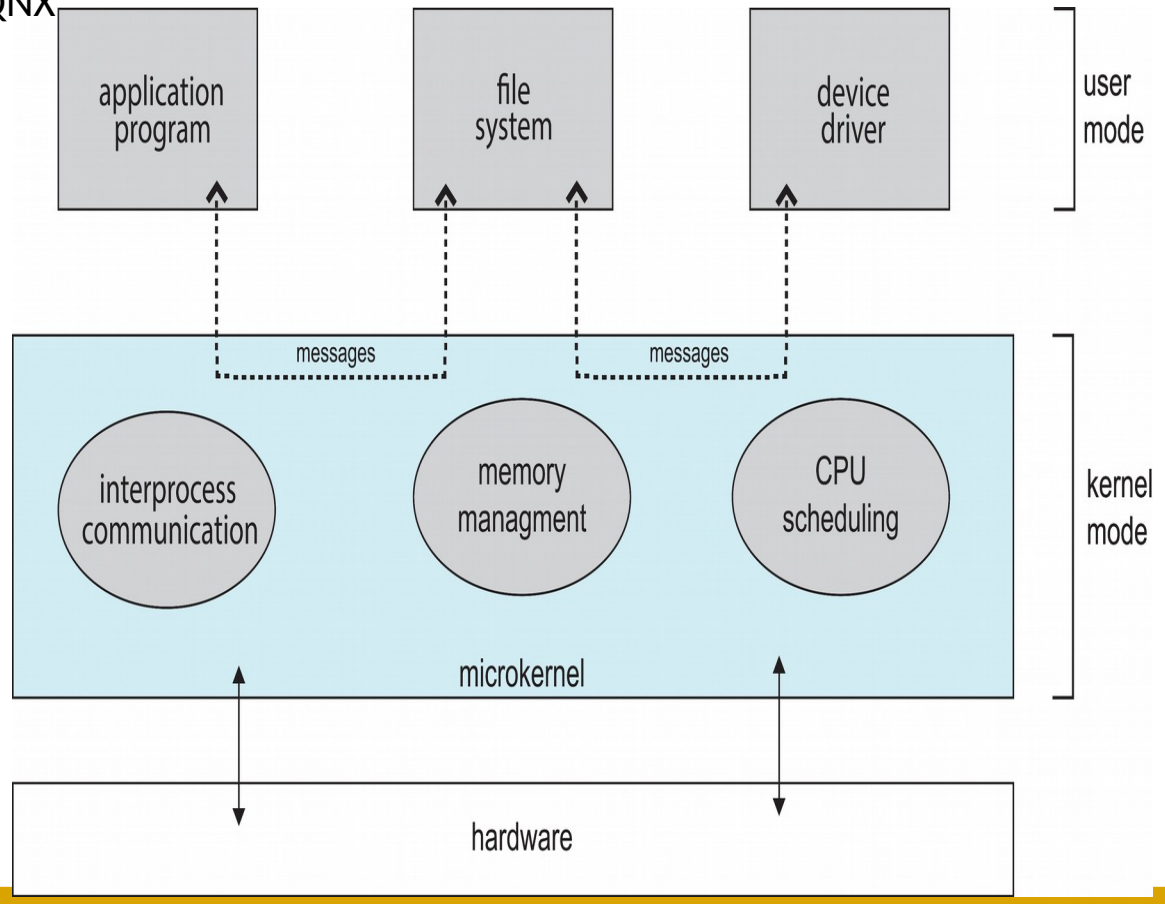


# Microkernel System Structure **(Mac OS X Structure)**

- Moves as much from the kernel into "*user*" space
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

Kernel have the basic interaction with hardware and the basic Inter-Process Communication mechanisms.
All the other Operating System services exist outside the Kernel.

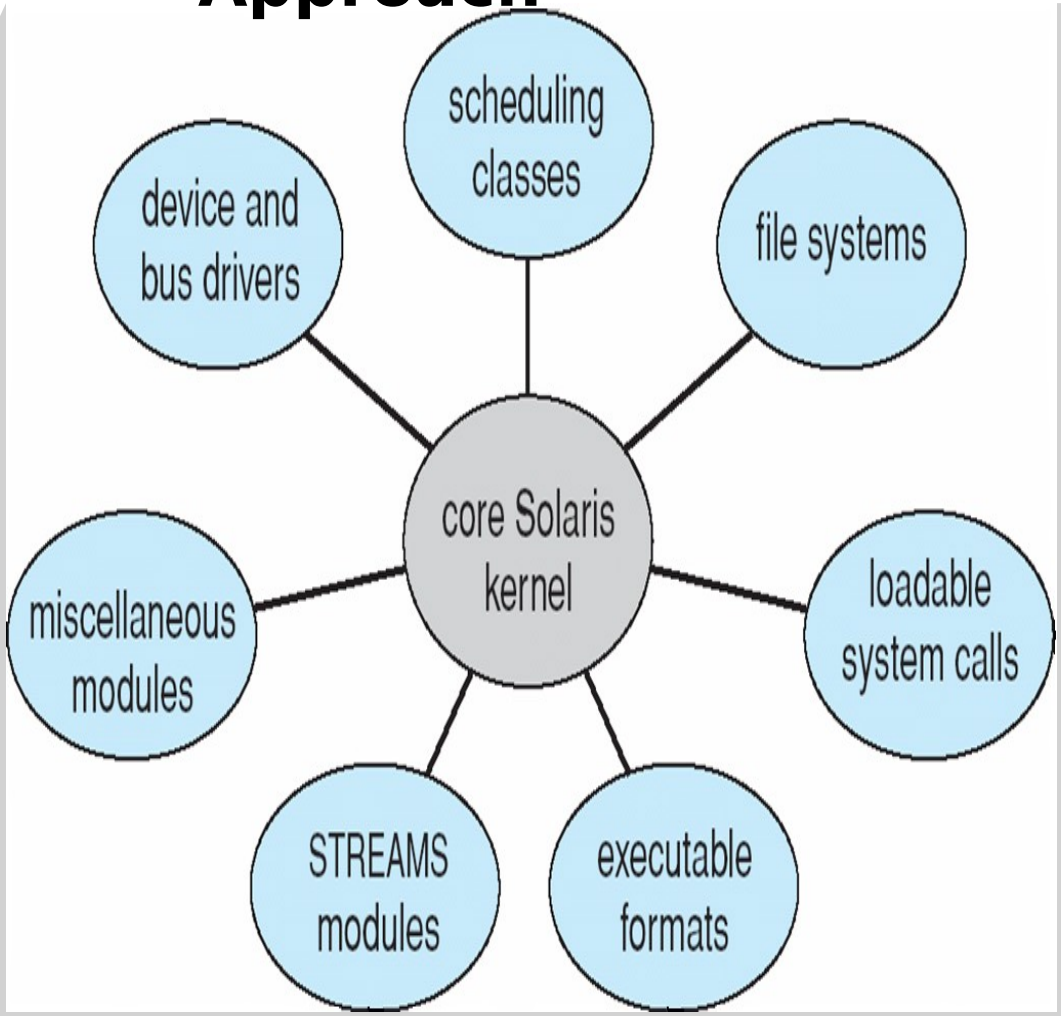
- Flexibility to add or modify features
- IPC, scheduling and memory management are the core services in a micro-kernel. Rest of the OS services exist as independent services.
- Examples: Eclipse IDE, Mach, OKL4, Codezero, Fiasco.OC, PikeOS, seL4, QNX



# Modules

- Most modern operating systems implement kernel modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible

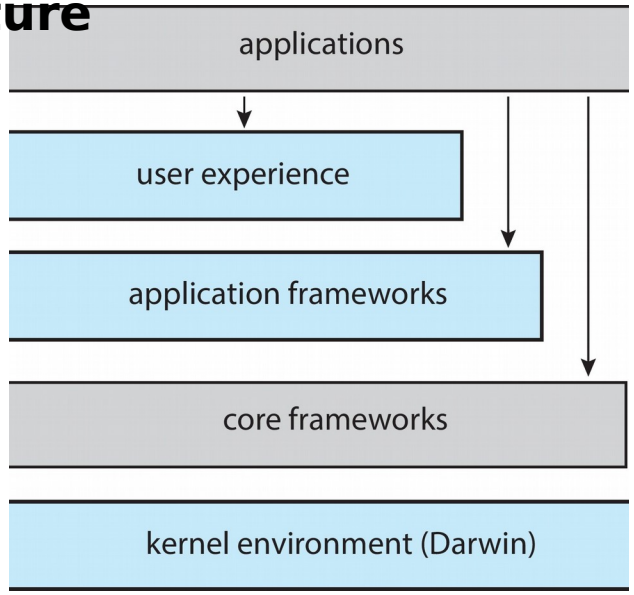
## Solaris Modular Approach



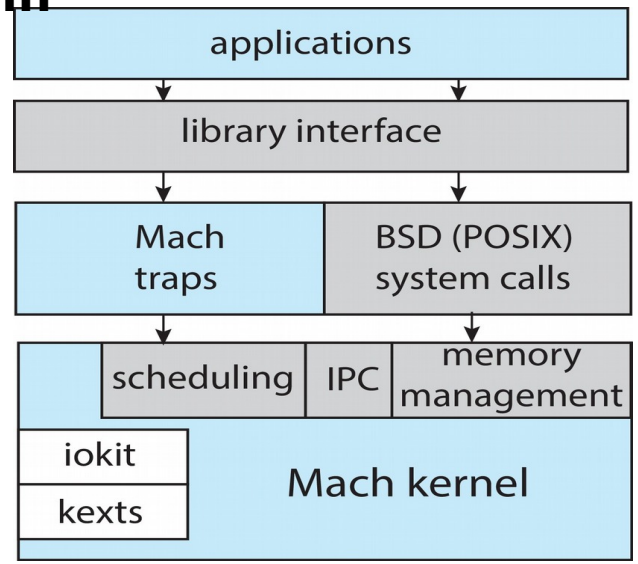
# Hybrid Systems

- Hybrid architecture tries to get the best features of both monolithic kernel as well as microkernel.
- Hybrid kernel aims to have an optimal performance and the flexibility to modify and upgrade kernel services.
- Most modern operating systems are not one pure model
- Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
- Windows mostly monolithic, plus microkernel for different subsystem ***personalities***
- Apple Mac OS X hybrid, layered, **Aqua** UI plus **Cocoa** programming environment. The kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called **kernel extensions**)

**macOS and iOS Structure**

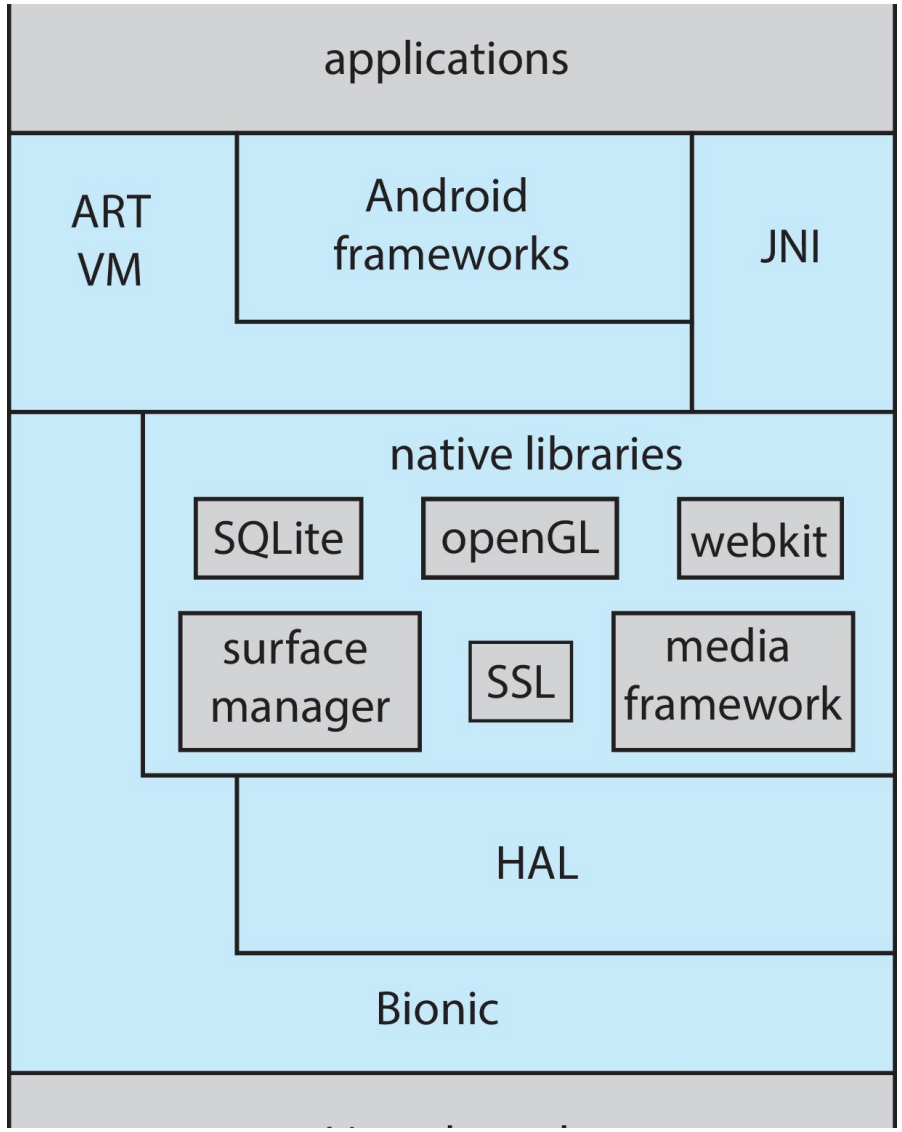



**Darwin**



# Android

- Uses Linux kernel for underlying functionalities such as threading, key security features and low-level memory management.
- HAL provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework.
- Android version 5.0 (API level 21) or higher, each app runs in its own process and with its own instance of the Android Runtime (ART).
- ART, HAL and other components use Native C/C++ libraries.
- The entire feature-set of the Android OS is available through APIs written in the Java language.
- Apps developed in Java plus Android API. Java class files compiled to Java bytecode then translated to executable then runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc



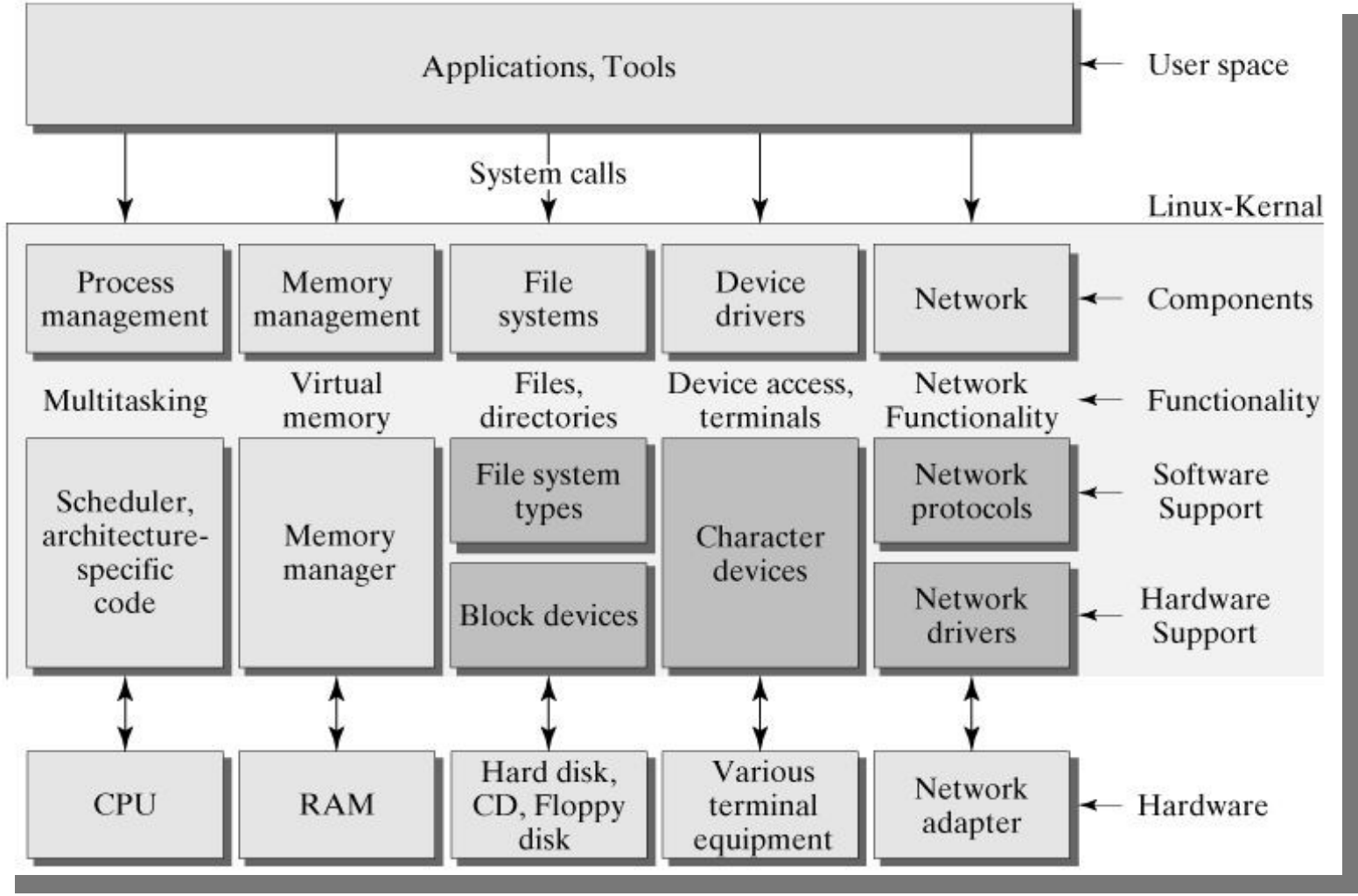
# How does the OS start?

- When OS is installed, it is stored on the hard disk, a non-volatile memory.
- Bootloader is a program that loads the kernel into RAM when the computer is started.
- Bootloader or bootstrap loader is stored in Read-only-Memory (ROM), EPROM or Flash memory.
- Firmware: Every device such as keyboard, mouse, disk drives have firmware stored in its own ROM storage.
  - Firmware is a type of logic that hardwires the devices and provides software, an interface to interact with hardware device.

# Kernel Architecture Design of Operating Systems

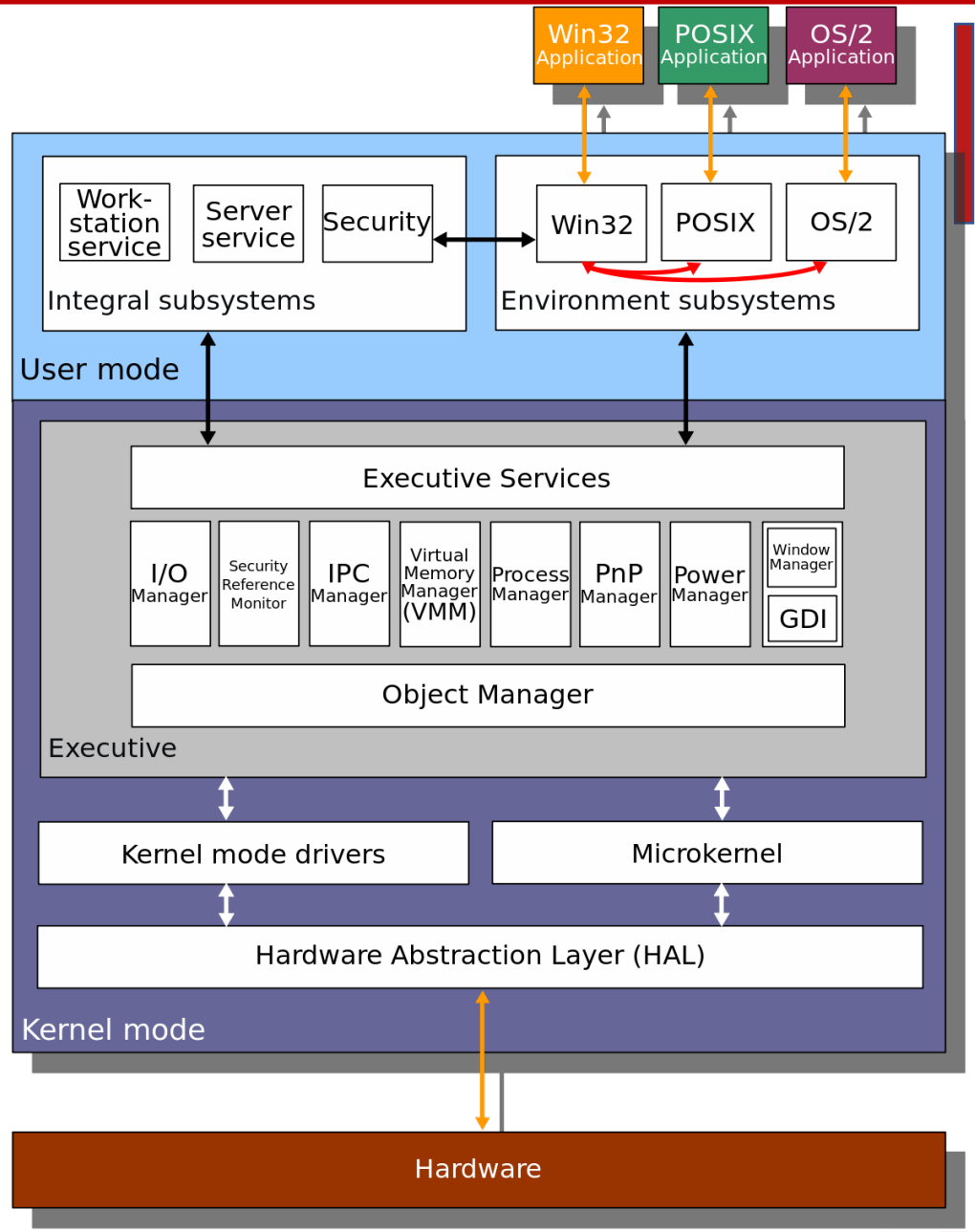
# Linux Architecture

- Linux has a monolithic Kernel.
- System libraries and Device drivers that help Software applications to pass instructions to the Kernel or the Device drivers.
- Device drivers are software's that interact (take input or provide output or both) with the firmware of the particular device.
- User mode has applications or tools which in turn interact with Kernel directly or indirectly via system libraries, device drivers.



# Windows NT Architecture

- **HAL** was designed to hide differences in hardware and provide a consistent platform on which the kernel is run.
- The **Executive interfaces**, with all the user mode subsystems, deal with I/O, object management, security and process management.
- The **kernel** sits between the hardware abstraction layer and the Executive to provide multiprocessor synchronization, thread and interrupt scheduling and dispatching, and trap handling and exception dispatching.
- The kernel is also responsible for initializing **device drivers** at bootup.

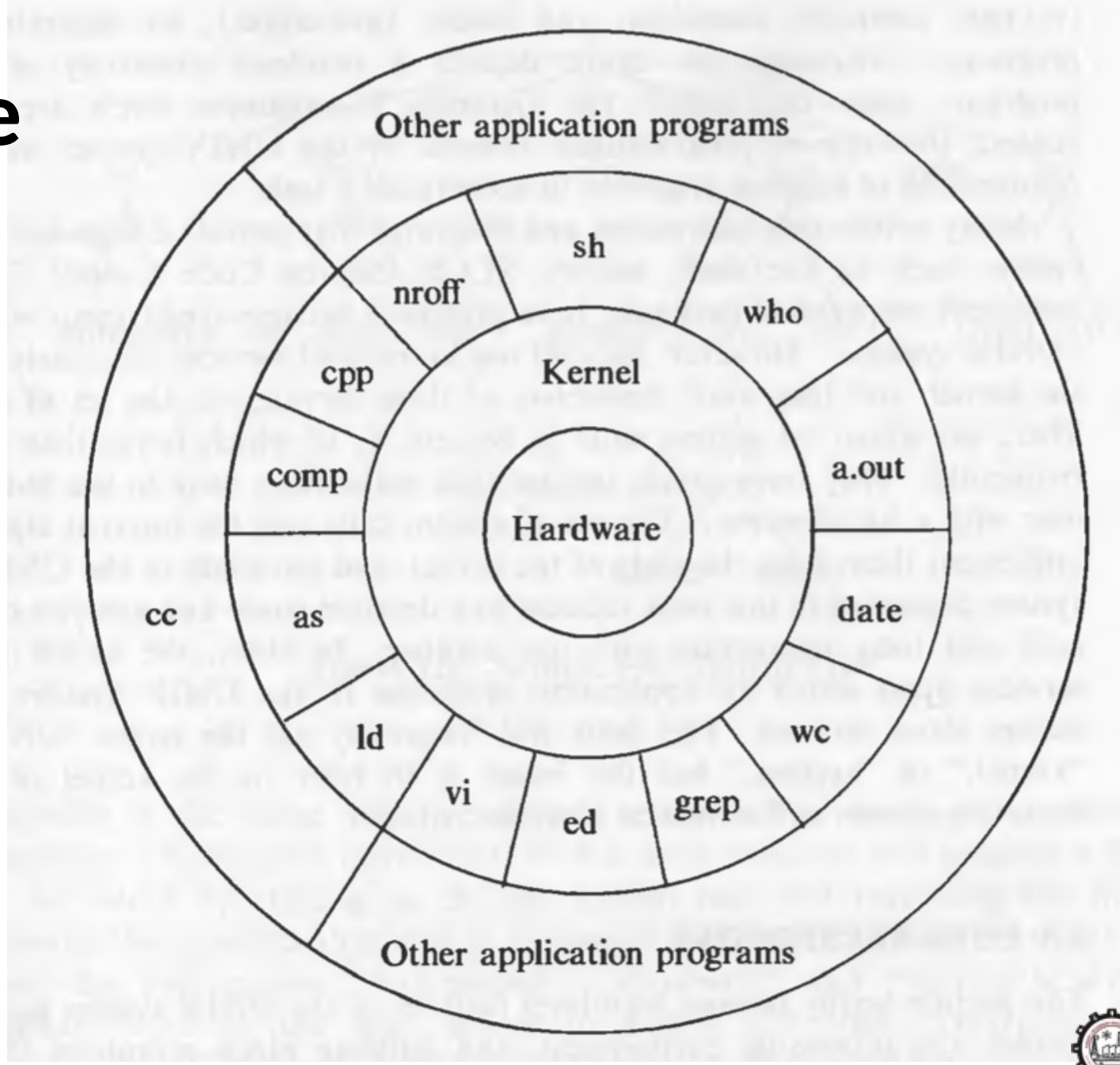


# Kernel Architecture Implementation in Course

# Unix Architecture

The operating systems interacts directly with the hardware, providing common services to programs and insulating them from hardware idiosyncrasies.

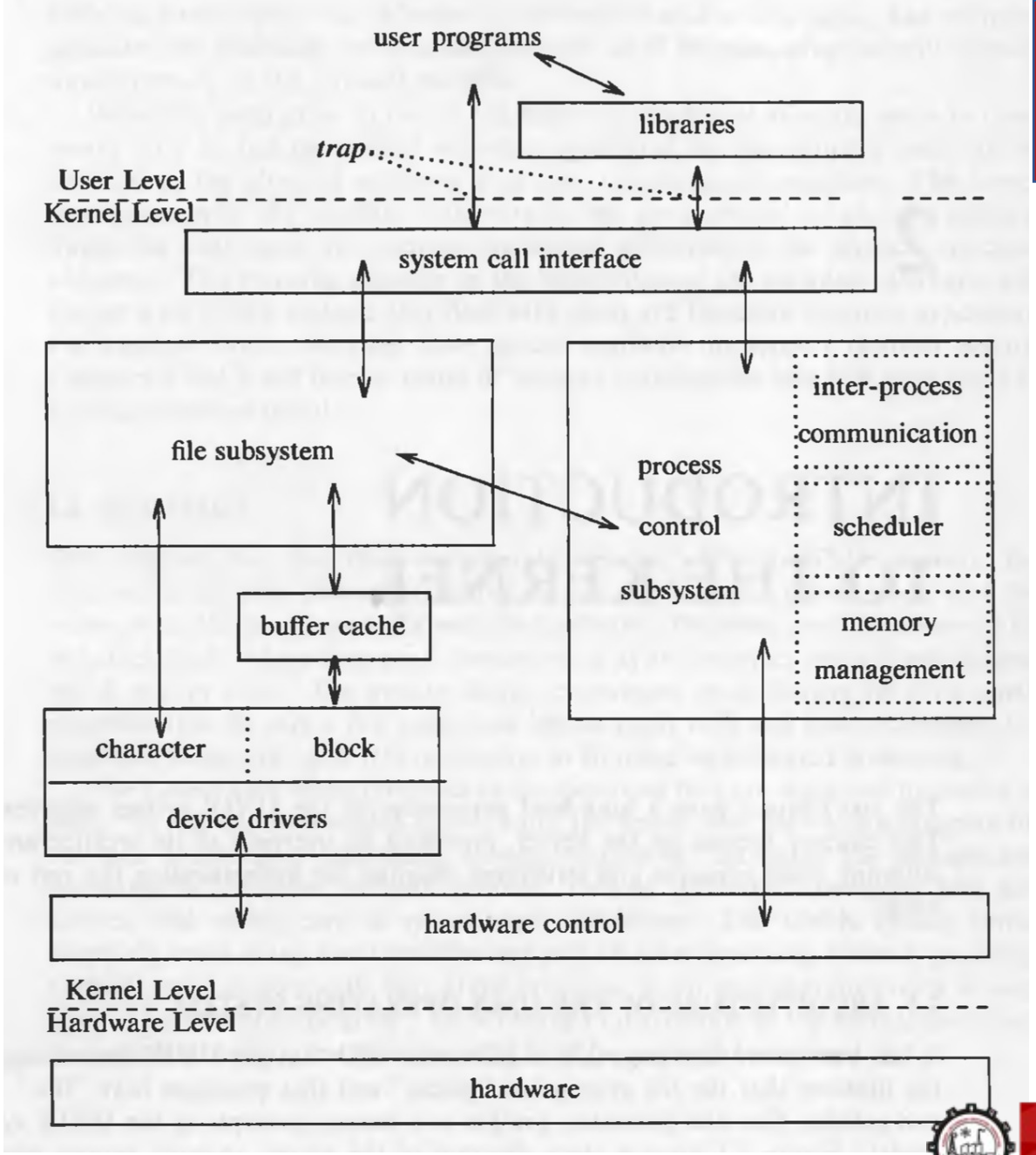
Programs such as the shell and editors shown in the outer layers interact with the kernel by invoking a well defined set of *system calls*.



# Unix Kernel

The two entities, files and processes, are the two central concepts in the UNIX system model.

- The *file subsystem* is on the left and the *process control subsystem* is on the right.
- The diagram shows 3 levels : user, kernel, and hardware.
- The system call and library interface represent the border between user programs and the kernel.



(DEEMED TO BE UNIVERSITY)

# 19CS2106R

# Operating Systems Design

## Session 3 : OS Interface

# Operating System User Interface

# Operating System User Interface

- A program that controls a display for the user (usually on a computer monitor) and that allows the user to interact with the system) .
- The user interface allows the user to communicate with the operating system.
- Types of User Interface
  - Command line interface
  - Graphical user interf

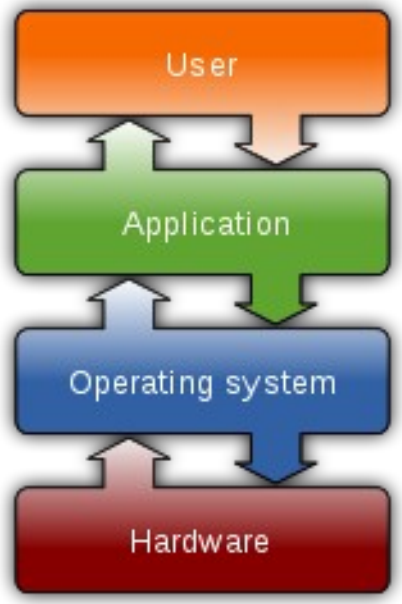
# Comparison

- Ease of Use/Learning
- Control on File System/Operating System
- Multitasking
- Speed of Operating

# Operating System Hardware Interface

# Abstracting physical resources

- A key requirement for an operating system is to support several activities at once.
- An operating system must fulfil three requirements:
  - Multiplexing
  - Isolation
  - Interaction.
- Operating System does this through Virtualization
- *Virtualization* refers to process involved in converting a physical view to a logical view.
  - The *logical* view is the view of the application programmer.
  - The *physical* view refers to the actual physical state of affairs.

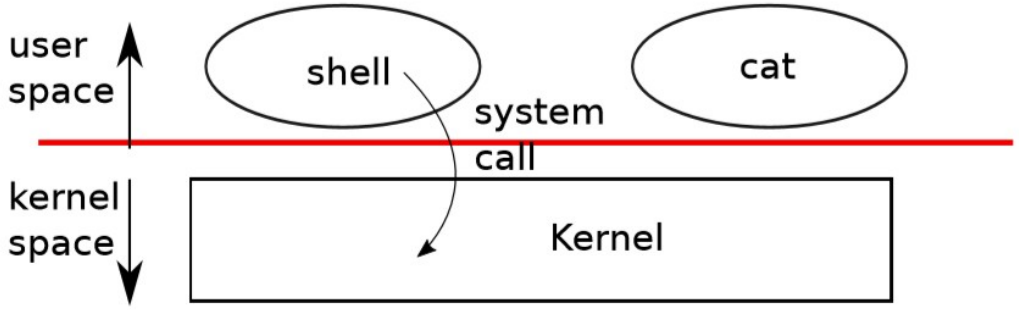


# Virtualization Mechanisms

- There are two general mechanisms for giving the appearance of multiple instances of a resource where there is physically only one resource.
- **Time-Shared Resources**
  - Processors, mice, keyboards, printers and network connections are time shared between processes.
- **Partitioned Resources**
  - Memory, displays, and disks are partitioned resources.
  - These resources are partitioned into "pieces" that are assigned to individual processes.

# User mode, Kernel mode and System calls

Strong isolation requires a hard boundary between applications and the operating system.




- An application can execute only user-mode instructions (e.g., adding numbers, etc.) and is said to be running in **user space**, .
- The software in kernel mode can also execute privileged instructions and is said to be running in **kernel space**.
- The software running in kernel space (or in kernel mode) is called the **kernel**.
- When a process needs to invoke a kernel service, it invokes a procedure call (system call) in the operating system interface.

# Process Management

- A process is the execution of a program.
- The kernel identifies each process by its process number, called the ***process ID* (PID)**.
- A process has Instructions, data and stack in User space.
  - The instructions implement the program's computation.
  - The data are the variables on which the computation acts.
  - The stack organizes the program's procedure calls.
- Kernel stores the State of every process in Kernel space.

# XV6 Interface Design Concepts

# XV6 Interface

- **Shell** is the primary user interface to traditional Unix-like systems including XV6.
- The shell is an ordinary program(user level) that reads commands from the user and executes them.
- The xv6 shell is a simple implementation of the essence of the Unix Bourne shell.
- XV6 Kernel
  - Process Management
  - I/O Management
  - File System Management

# XV6 System Calls

| System call | Description |
|---|---|
| fork() | Create a process |
| exit() | Terminate the current process |
| wait() | Wait for a child process to exit |
| kill(pid) | Terminate process pid |
| getpid() | Return the current process's pid |
| sleep(n) | Sleep for n clock ticks |
| exec(filename, *argv) | Load a file and execute it |
| sbrk(n) | Grow process's memory by n bytes |
| open(filename, flags) | Open a file; the flags indicate read/write |
| read(fd, buf, n) | Read n bytes from an open file into buf |
| write(fd, buf, n) | Write n bytes to an open file |
| close(fd) | Release open file fd |
| dup(fd) | Duplicate fd |
| pipe(p) | Create a pipe and return fd's in p |
| chdir(dirname) | Change the current directory |
| mkdir(dirname) | Create a new directory |
| mknod(name, major, minor) | Create a device file |
| fstat(fd) | Return info about an open file |
| link(f1, f2) | Create another name (f2) for the file f1 |
| unlink(filename) | Remove a file |

# Processes and memory

## fork() – Creating a Process

- A process on a UNIX system is the entity that is created by the *fork* system call.
- Fork creates a new process, called the **child process**, with exactly the **same memory contents** as the calling process, called the parent process.
- Every process except ***process 0*** is created when another process executes the *fork* system call.
- Process 0 is a special process that is created "by hand" when the system **boots**; after *fork*ing a child process (process 1), process 0 becomes the ***swapper* process**.
- Process 1, known as ***init*** is the **ancestor** of every other process.

# Processes and memory

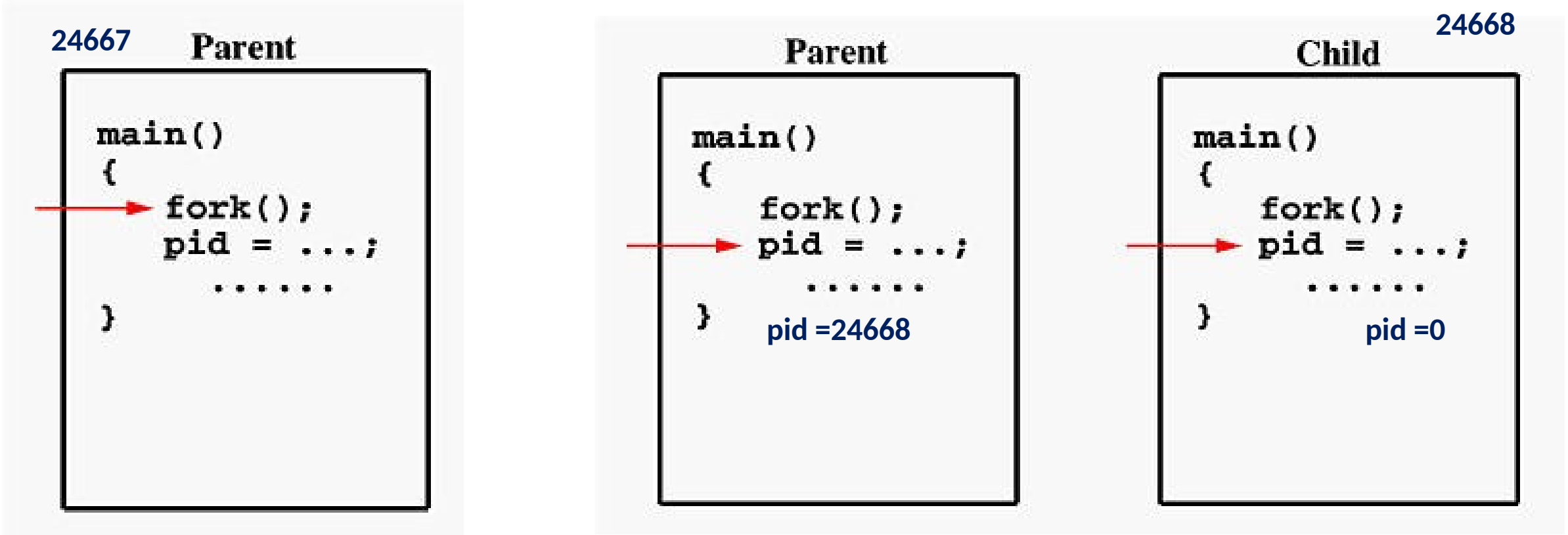
## fork() – Creating a Process

- **fork()** is used to create processes. It takes no arguments and returns a process ID
- The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller.
- After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call.
- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the ***process ID*** of the child process, to the parent.

# Processes and memory

## fork() – Creating a Process

### The fork() System Call



# Processes and memory

## wait() and exit()

- The system call **wait()** blocks the calling process until one of its child processes exits or a signal is received.
- The **exit** system call causes the calling process to stop executing and to release resources such as memory and open files.
- Exit takes an integer status argument, conventionally 0 to indicate success and 1 to indicate failure.

# Processes and memory

## fork(), wait() and exit() usage example

```
int pid = fork();
  if(pid > 0){
      printf("parent: child=%d\n", pid);
      pid = wait();
      printf("child %d is done\n", pid);
   } else if(pid == 0){
      printf("child: exiting\n");
      exit();
  } else {
      printf("fork error\n");
}
```

```
parent: child=1234
child: exiting
parent: child 1234 is done
```

# Processes and memory

## exec()

- The exec system call replaces the calling process's memory with a new memory image loaded from a file stored in the file system.
- The created child process does not have to run the same program as the parent process does.
- fork starts a new process which is a copy of the one that calls it, while exec replaces the current process image with another (different) one.
- Both parent and child processes are executed simultaneously in case of fork() while Control never returns to the original program unless there is an exec() error.

# I/O and File descriptors

- A file descriptor is a small integer representing a kernel-managed object that a process may read from or write to.
- A file descriptor interface abstracts away the differences between **files**, pipes, and devices, making them all look like streams of bytes.
- The **read** and **write** system calls read bytes from and write bytes to open files named by file descriptors.
- The call **read(fd, buf, n)** reads at most n bytes from the file descriptor **fd**, copies them into **buf**, and returns the number of bytes read.
- The call **write(fd, buf, n)** writes **n** bytes from **buf** to the file descriptor **fd** and returns the number of bytes written.
- The **close** system call releases a file descriptor, making it free.

# Pipes

- A pipe is a small kernel buffer exposed to processes as a pair of file descriptors, one for reading and one for writing.
- Writing data to one end of the pipe makes that data available for reading from the other end of the pipe.
- Pipes provide a way for processes to communicate

# File System

- The xv6 file system provides data files, which are uninterpreted byte arrays, and directories, which contain named references to data files and other directories.
- The directories form a tree, starting at a special directory called the root.
- Paths that don’t begin with / are evaluated relative to the calling process’s current directory, which can be changed with the **chdir** system call.
- There are multiple system calls to create a new file or directory: **mkdir** creates a new directory, open with the O_CREATE flag creates a new data file, and **mknod** creates a new device file.

# INTRODUCTION TO UNIX FILE SYSTEM

Session no :4

Operating System Design

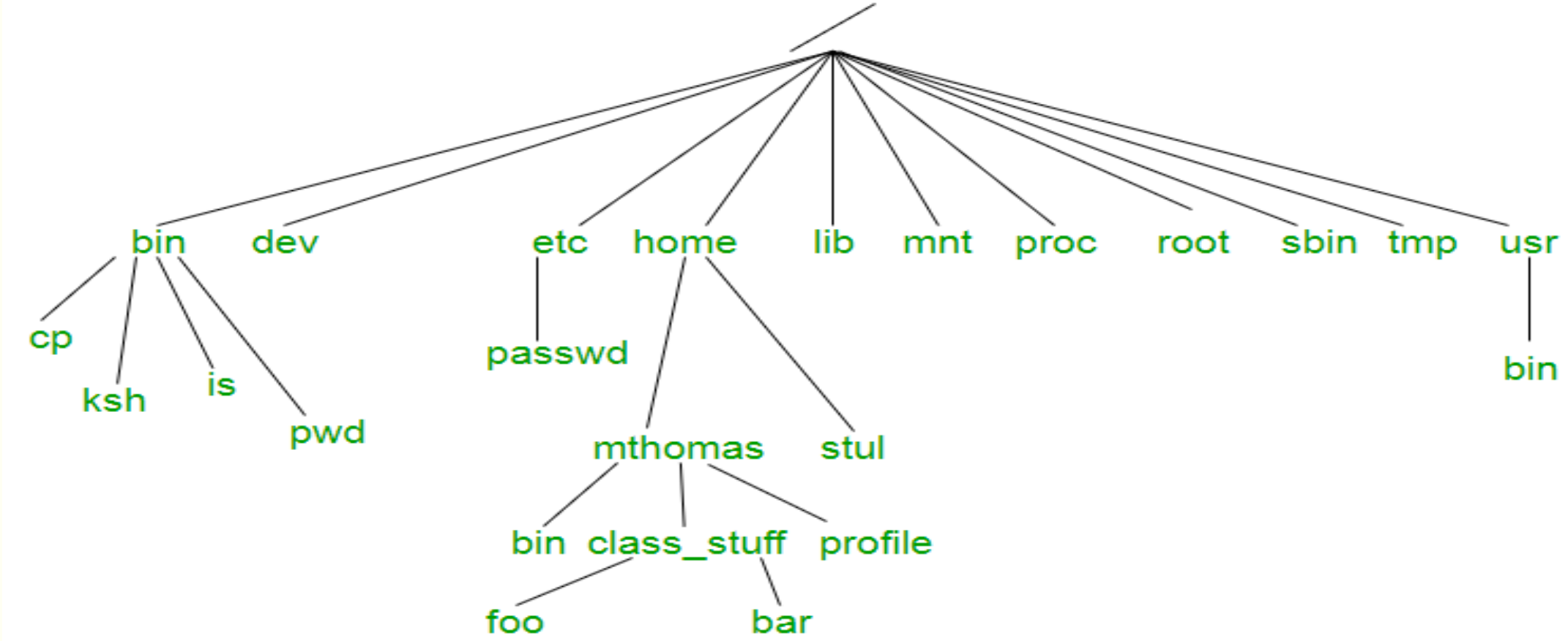
@KL University, 2020

(DEEMED TO BE UNIVERSITY)

# Understanding UNIX/Linux File Systems

- File: basic component for data storage
  - UNIX  considers everything to be a file
- A file system is UNIX/Linux's way of organizing files on mass storage devices
  - A physical file system is a section of the hard disk that has been formatted to hold files
- All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the file system.

# Unix File System



# Overview of the File system.

- The internal organization of the file is given by an Inode.
- The kernel contains two other data structures , the file table and the user file descriptor table .
- The file table  is a global kernel structure, but the user file descriptor  table is allocated per process.
- When a process opens or creates a file , the kernel allocates an entry from each table , corresponding to the file's inode.
- Entries in the three structures –user file descriptor table , file table  and inode table maintain the state of the file and the user's access to it.
- The file table keeps track of the byte offset in the file where the user's next read or write will start, and the access rights allowed to the opening process.
- The user file descriptor table identifies all open files for a process.

# File system.

- A file system  consists of a sequence of logical blocks (512/1024 byte etc.)
- Size of a logical block is homogeneous with in  a file system

- A file system has the following structure:

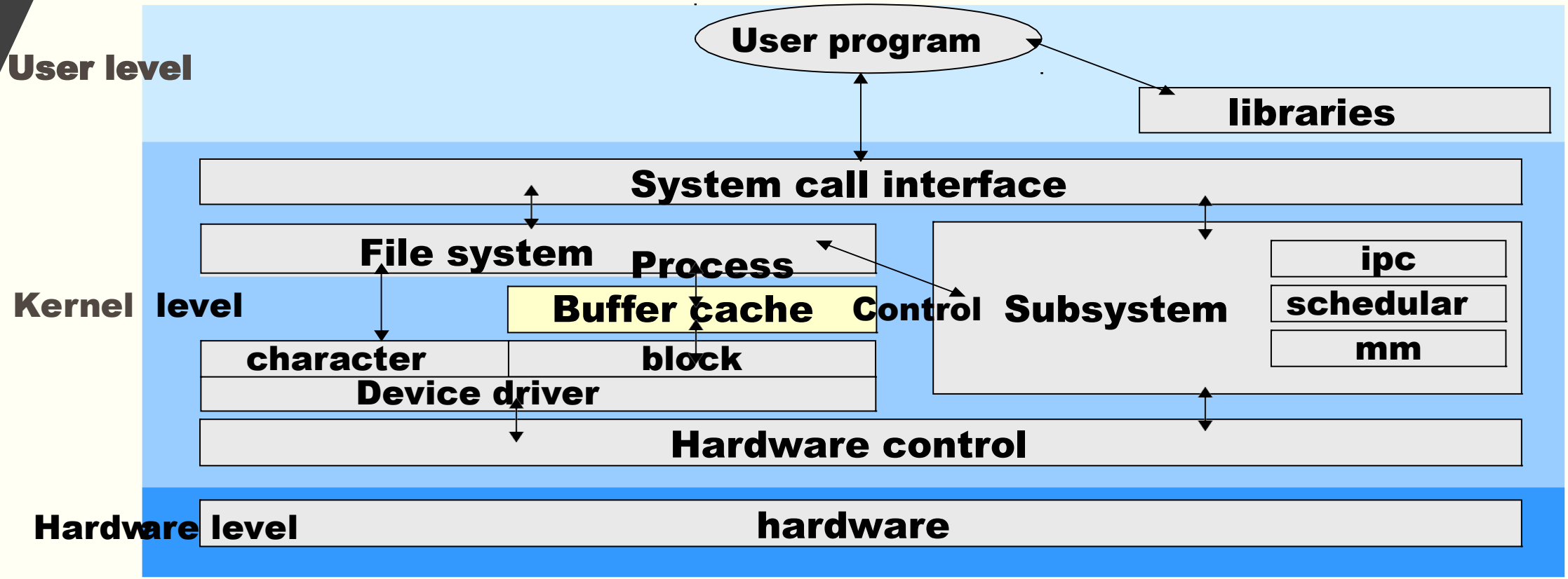
| Boot Block | Super Block | Inode List | Data Blocks |
|---|---|---|---|

File System Layout

# Why Buffer Cache

- Kernel maintains files on mass storage
- To access data ,kernel brings data into main memory
- System response time and throughput is poor due to slow disk transfer rate
- To minimize the frequency of disk accesses, maintains a pool of internal data buffers ,called the buffer cache.
- While reading data from the disk ,kernel attempts to read from the buffer cache.
- Data being written to disk is cached so that it will be there if the kernel later tries to read it.

# The Buffer cache




block diagram of the System Kernel

# Buffer Headers

- During system initialization , the kernel allocates space for a number of buffers ,configurable according to memory size & system performance constraints.
- Buffer
- Memory array: Contains data from disk
- Buffer header : Identifies the buffer
- The buffer is the in-memory copy of the disk block.
- A disk block can never map into more than one buffer at a time.
- The buffer header contains  a device number field and a block number field that specify the file system and block number of the data on disk and uniquely identifies the buffer.

# Status of buffer

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

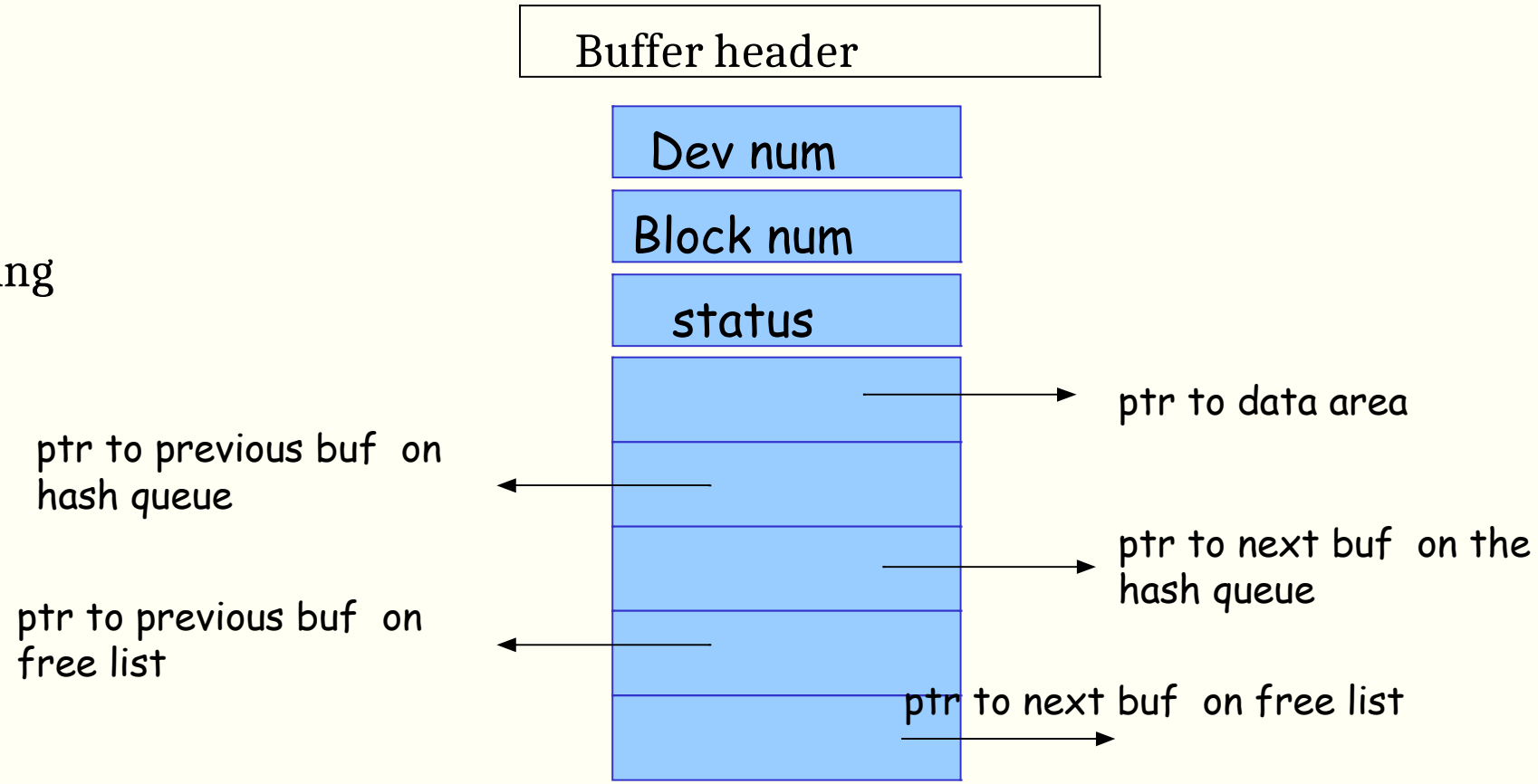
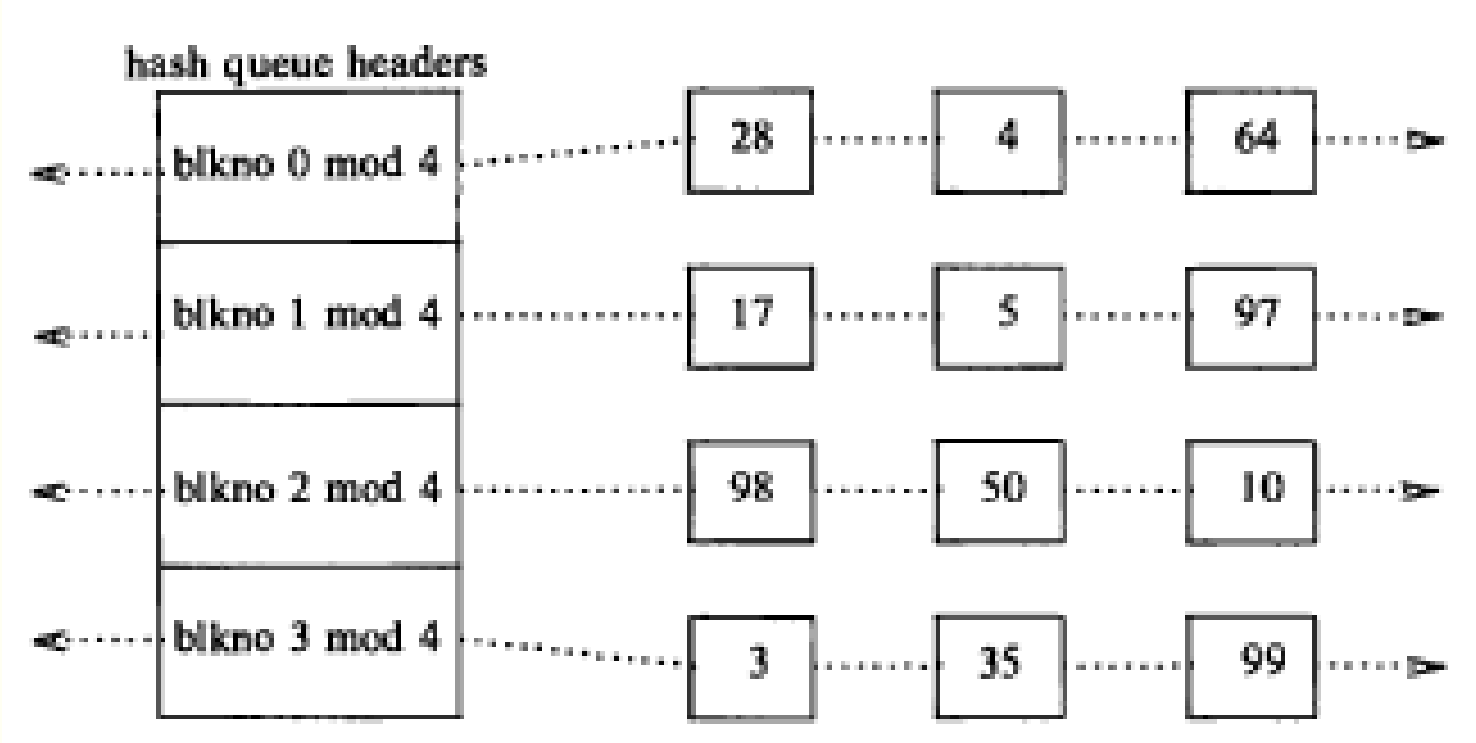



Figure : buffer header

# Structure of the Buffer Pool

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

# Buffers on the Hash Queues



# DESIGN

# Scenarios to allocate a buffer for a disk block

**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 (as in scenario 2), finds a buffer on the free list that has been marked "delayed write." The kernel must write the "delayed write" buffer to disk and allocate another buffer.

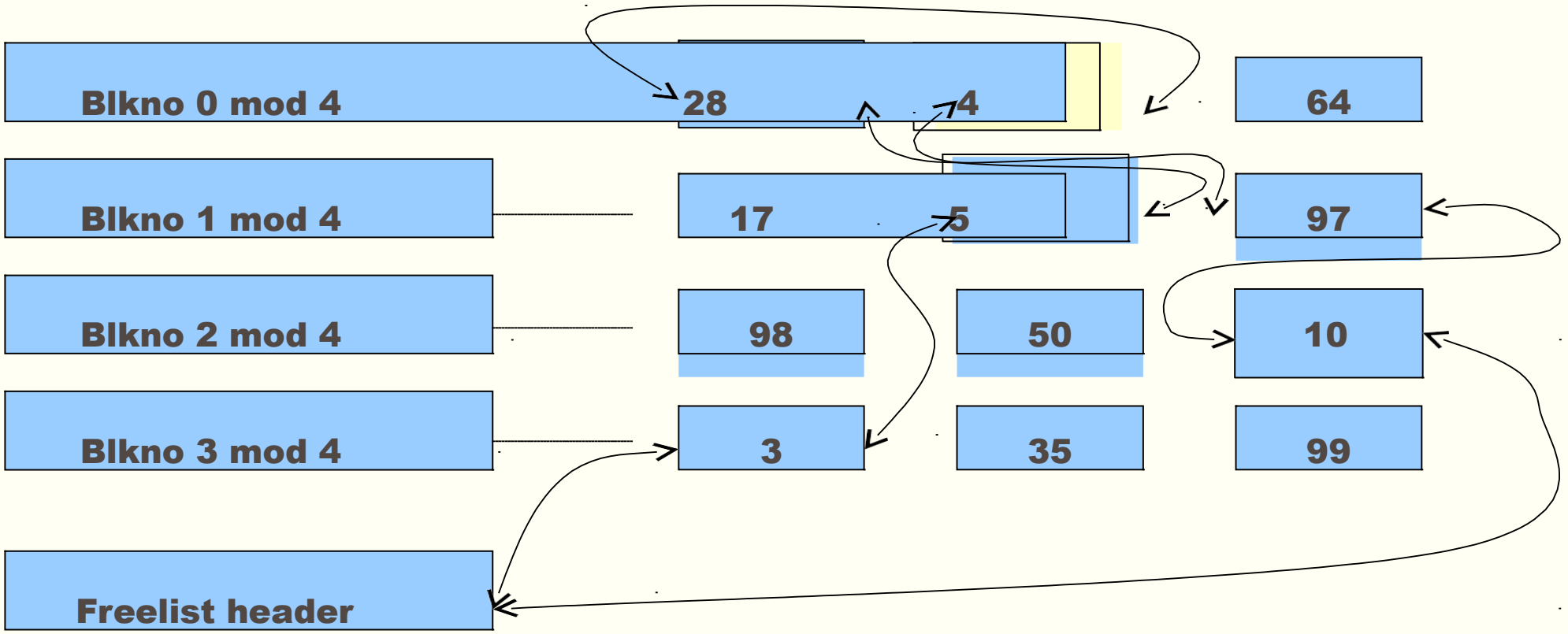
4. The kernel cannot find the block on the hash queue, and the free list of buffers is empty.

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

# 1st Scenario

Block is in hash queue, not busy

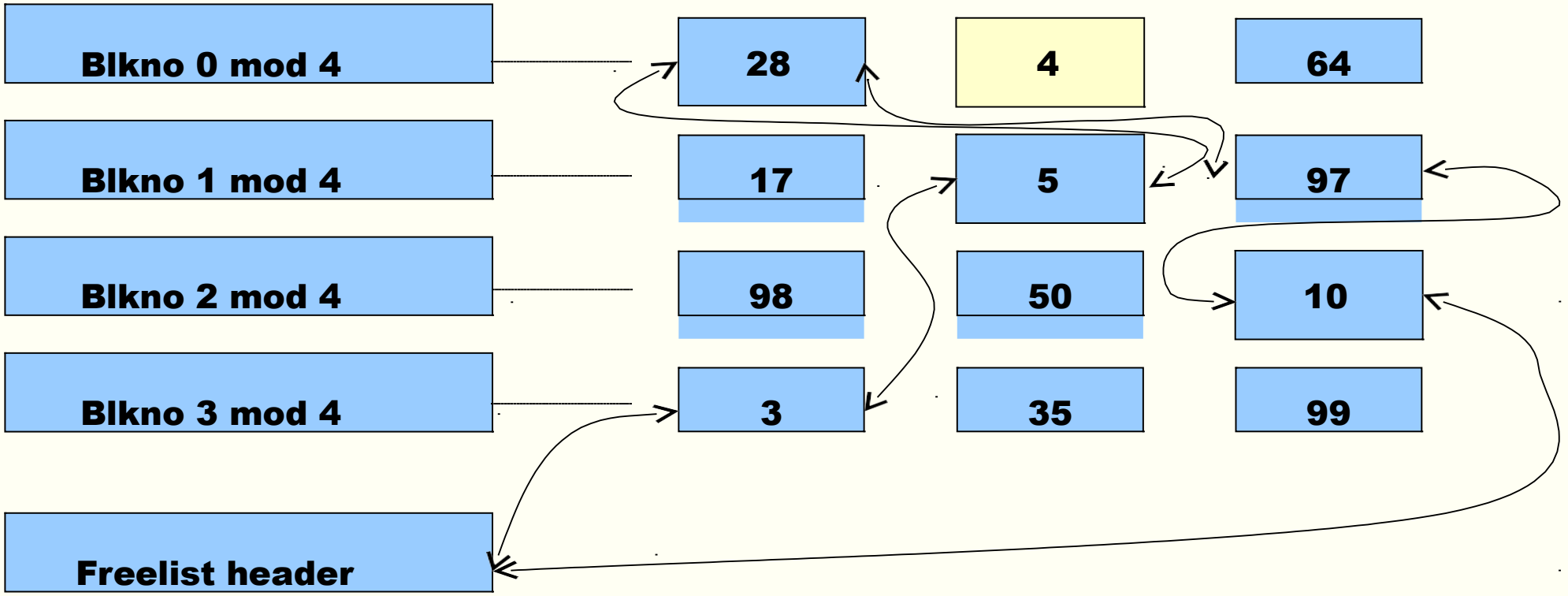
Choose that buffer




(a) Search for block 4 on first hash queue

# 1st Scenario

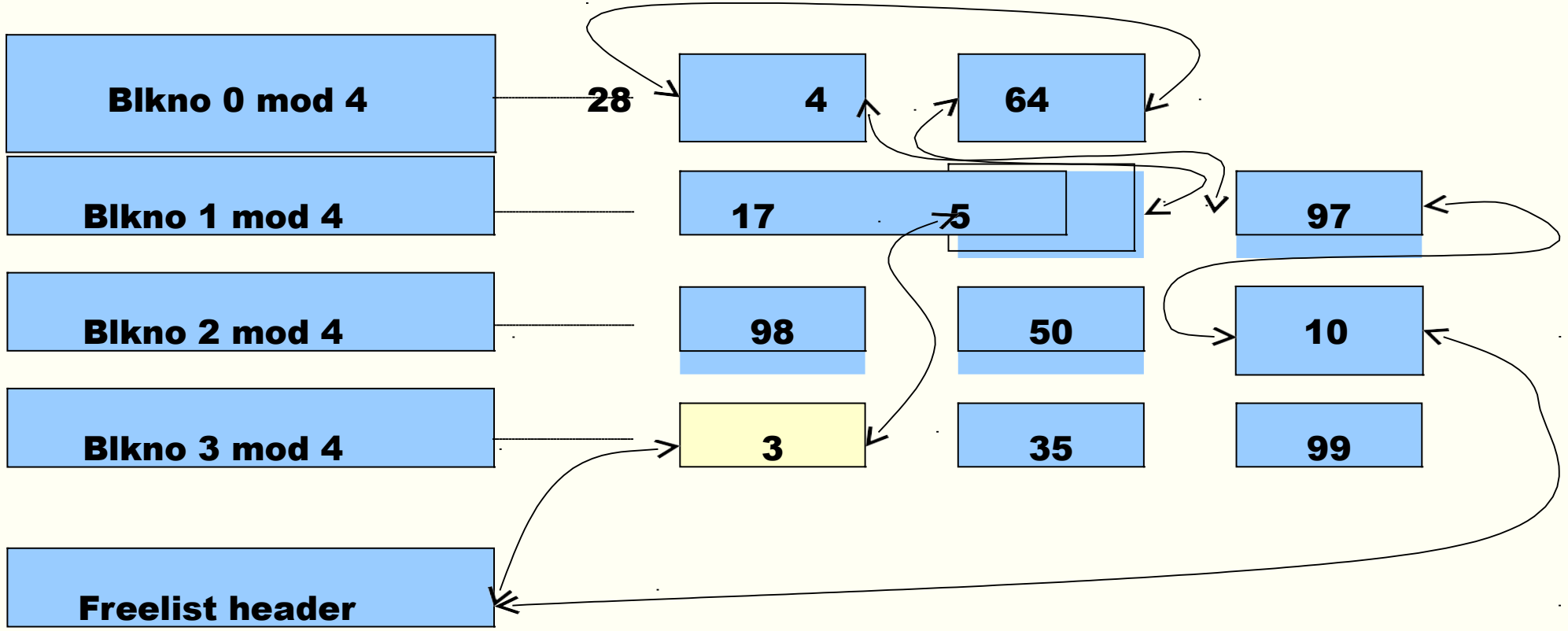
After allocating




(b) Remove block 4 from free list

# 2nd Scenario

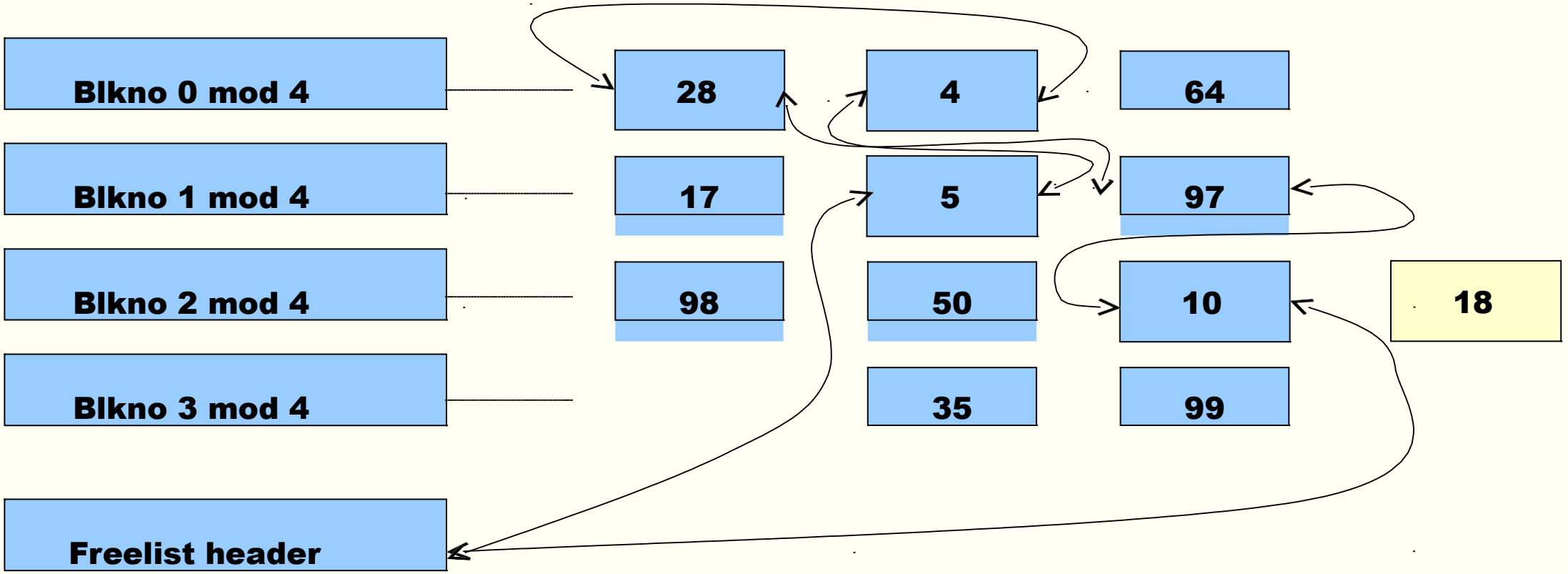
Not in the hash queue and exist in free buffer. Choose one buffer in front of free list.




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

# 2nd Scenario

After allocating

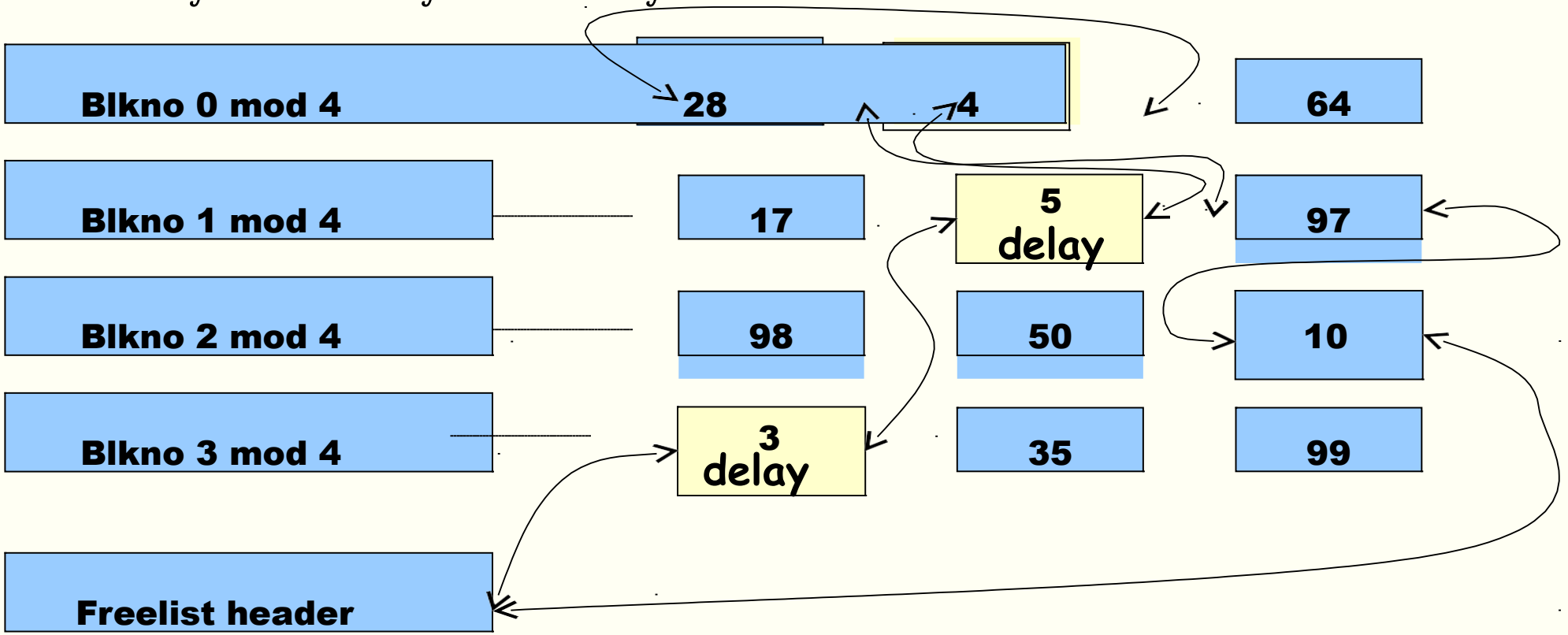



(b) Remove first block from free list, assign to 18

# 3rd Scenario

Not in the hash queue and there exists delayed write buffer in the front of free list

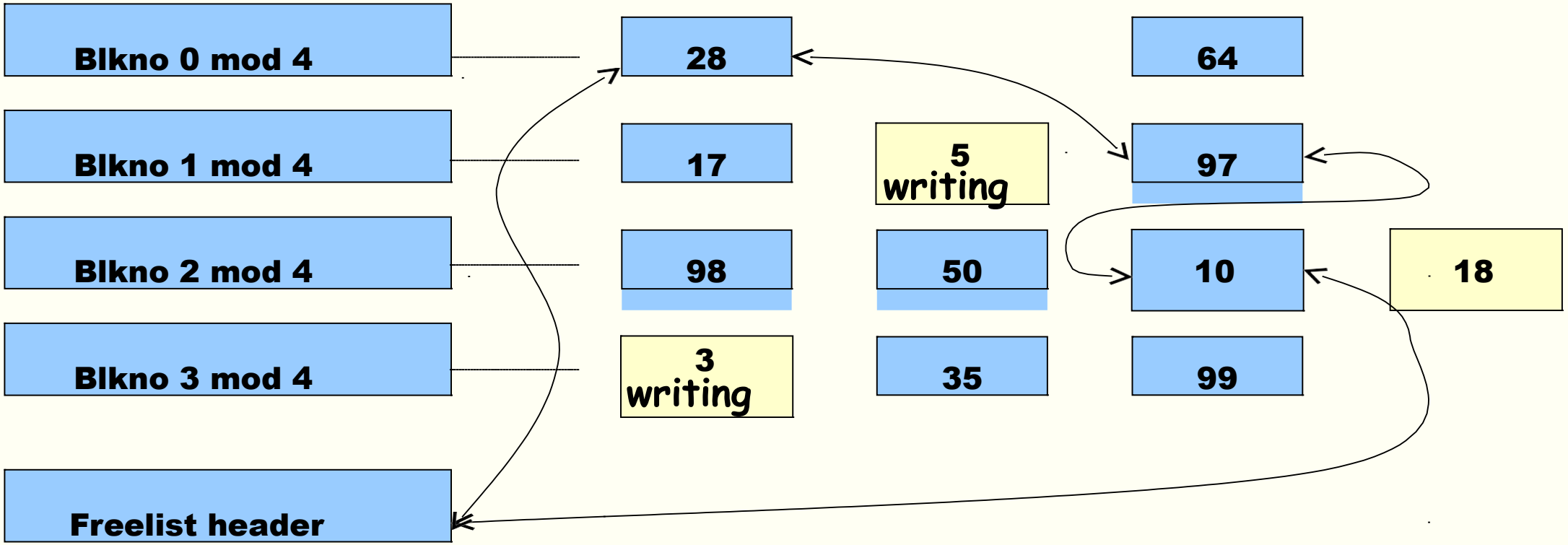
Write delayed buffer asynchronously and choose next




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

# 3rd Scenario

After allocating

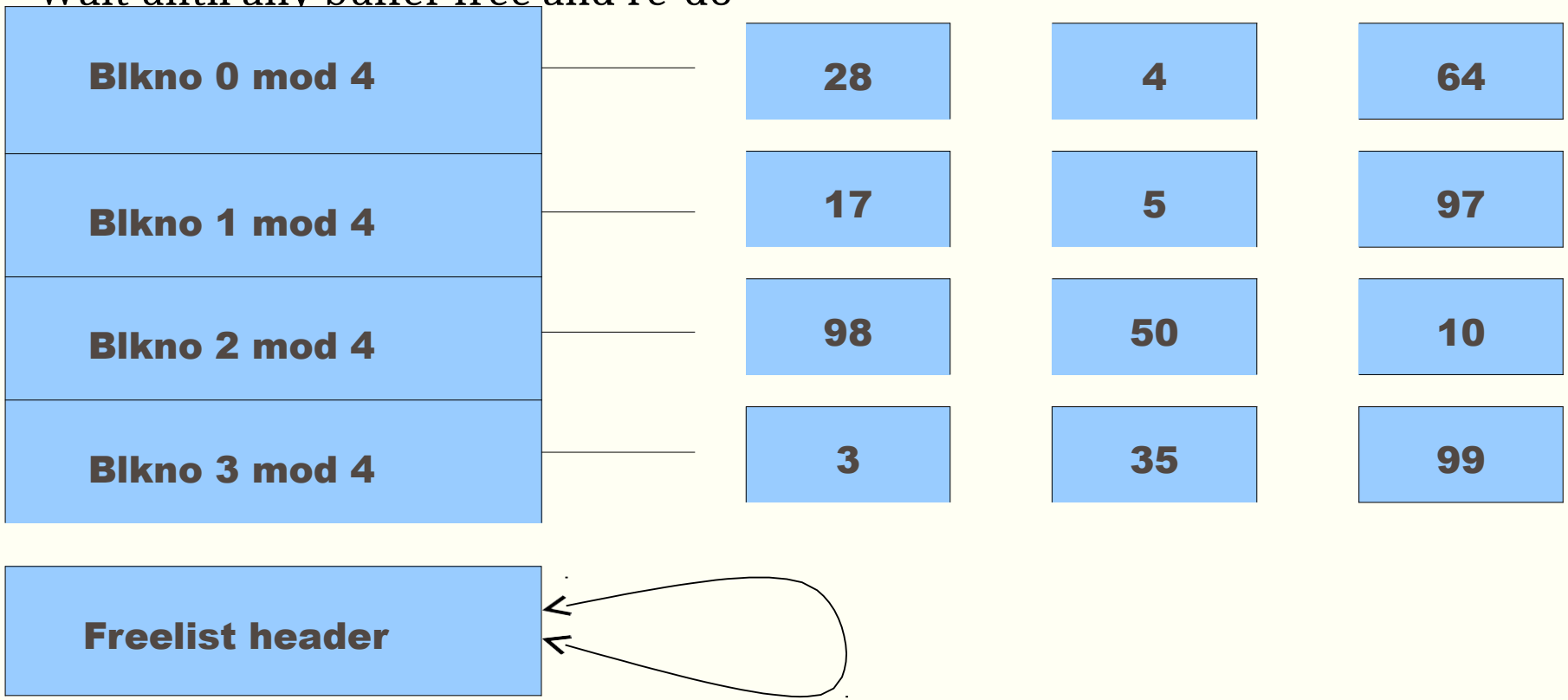



(b) Writing block 3, 5, reassign 4 to 18

# 4th Scenario

Not in the hash queue and no free buffer

Wait until any buffer free and re-do




(a) Search for block 18, empty free list

# 4th Scenario

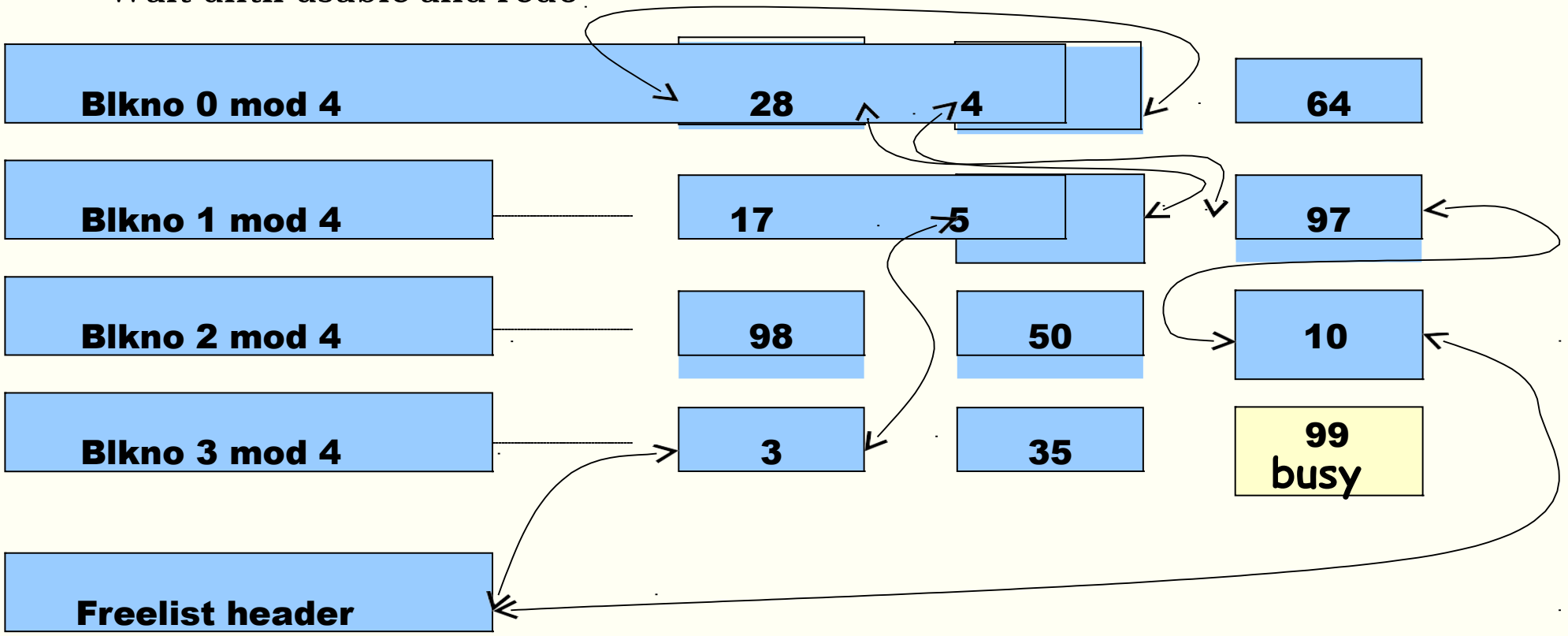
| Process A | Process B |
| --- | --- |
| Cannot find block b on the hash queue | |
| No buffer on free list | |
| Sleep | Cannot find block b on hash queue |
| | No buffers on free list Sleep |
| Somebody frees a buffer: brelse | |
| Takes buffer from free list Assign to block b | |

Figure 3.10 Race for free buffer

# 5th Scenario

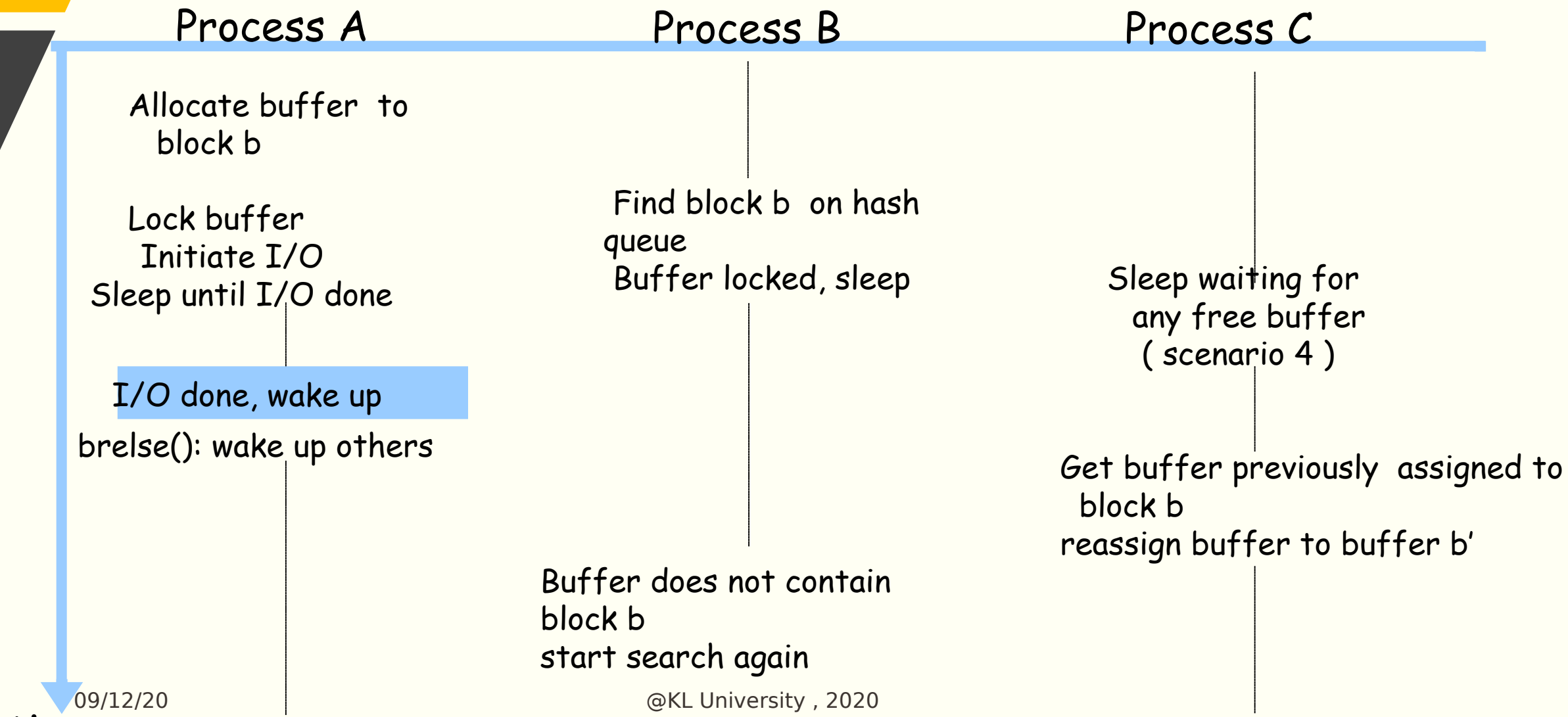
Block is in hash queue, but busy

Wait until usable and redo




(a) Search for block 99, block busy

# 5th Scenario



# Algorithm :getblk

```
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){ /*scenario 5*/  if(buffer busy){
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;

  }
  }}
```

# Algorithm :Brelse

```
algorithm brelse
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);
}
```

# IMPLEMENTATION

# Implementation XV6--bget

```
// Look through  buffer cache for block on device dev
// If not found, allocate a buffer.
// In either case, return locked buffer.
static struct buf*
bget(uint dev, uint blockno)
{
struct buf *b;
acquire(&bcache.lock);
// Is the block already cached?
for(b = bcache.head.next; b != &bcache.head; b = b->next)
{
  if(b->dev == dev && b->blockno == blockno)
  {
   b->refcnt++;
   release(&bcache.lock);
   acquiresleep(&b->lock);
   return b;
  }
}
```

```
// Not cached; recycle an unused buffer.
// Even if refcnt==0, B_DIRTY indicates a buffer is in use
// because log.c has modified it but not yet committed it.
for(b = bcache.head.prev; b != &bcache.head; b = b->prev)
{
    if(b->refcnt == 0 && (b->flags & B_DIRTY) == 0)
    {
        b->dev = dev;
        b->blockno = blockno;
        b->flags = 0;
        b->refcnt = 1;
        release(&bcache.lock);
        acquiresleep(&b->lock);
        return b;
    }
 }
panic("bget: no buffers");
}
```

# Implementation XV6--brelse

```
// Release a locked buffer.
// Move to the head of the MRU list
Void brelse(struct buf *b)
{
  if(!holdingsleep(&b->lock))
  panic("brelse");
  releasesleep(&b->lock);
  acquire(&bcache.lock);
  b->refcnt--;
  if (b->refcnt == 0)
  {
       // no one is waiting for it.
       b->next->prev = b->prev;
       b->prev->next = b->next;
       b->next = bcache.head.next;
       b->prev = &bcache.head;
       bcache.head.next->prev = b;
       bcache.head.next = b;
  }
release(&bcache.lock);
}
```

# BUFFER CACHE ALGORITHMS

Session no :5

Operating System Design

@KL University, 2020

# Recap of Session 4

- A file system is UNIX way of organizing files on mass storage devices.
- All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the file system.
- When a process wants to access data ,kernel brings data into main memory
- To minimize the frequency of disk accesses, maintains a pool of internal data buffers ,called the buffer cache.
- During system initialization , the kernel allocates space for a number of buffers ,configurable according to memory size & system performance constraints.
- Buffer allocation Algorithms : Getblk , Brelse.

# Reading Disk Blocks

- To read a disk block , a process uses the algorithm getblk to search for it in the buffer cache.

```
If  found  in the buffer cache
        Return the data without disk access
Else
        Calls disk driver to schedule a read request
        Sleep(awaiting  the event that I/O completes)
```

When I/O complete, disk controller interrupts the Processor and Disk interrupt handler awakens the sleeping process .

Some times we need to read the files sequentially , so we have to anticipate the need for reading the second disk block .

# Writing Disk Blocks

- Kernel informs the disk driver that , it has a buffer ,whose contents are to be written to the disk .
- Synchronous write
  - the calling process goes the sleep awaiting I/O completion and releases the buffer when awakens.
- Asynchronous write
  - the kernel starts the disk write. The kernel release the buffer when the I/O completes
- Delayed write
  - The kernel put off the physical write to disk until buffer reallocated

Release

- Use brelse()

# DESIGN

# Reading Disk Blocks

```
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);
}
```

Algorithm for Reading a Disk Block

# Reading Disk Blocks

Read Ahead:Improving performance

- Read additional block before request

Use breada()

```
Algorithm breada
Input: (1) file system block number for immediate read
       (2) file system block number for asynchronous read
Output: buffer containing data for immediate read
{
if (first block not in cache)
{
get buffer for first block(algorithm getblk);
 if(buffer data not valid)
initiate disk read;
}
if (second block not in cache)
{
get buffer for second block(algorithm getblk);
if(buffer data valid)
        release buffer(algorithm brelse);
else
        initiate disk read;
}
if(first block was originally in cache)
{
        read first block(algorithm bread)
        return buffer;
}
sleep(event first buffer contains valid data);
return buffer;
}
```

# Release Disk Block

algorithm

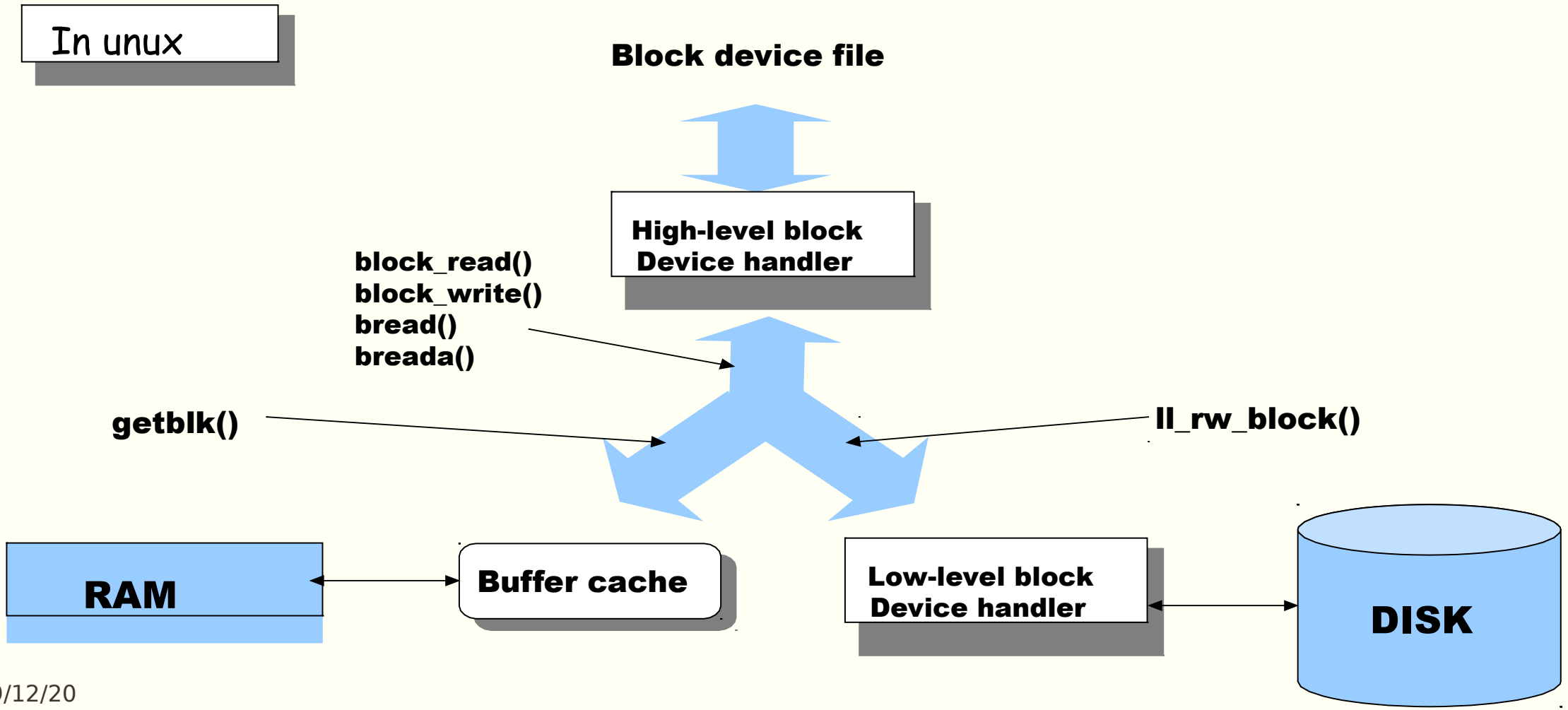
```
Algorithm brelse  Input: locked buffer  Output: none
{
wakeup all process; event, waiting for any buffer to become free;
wakeup all process; event, waiting for this buffer to become free;  raise
processor execution level to allow interrupts;
if( buffer contents valid and buffer not old)
        enqueue buffer at end of free list;
else
        enqueue buffer at beginning of free list
lower processor execution level to allow interrupts;  unlock(buffer);
}
```

# Writing Disk Blocks

algorithm

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

# Reading Disk Blocks



# IMPLEMENTATION

# Reading and Writing Disk Blocks

**Reading a Disk Block**

```
struct buf *
bread(uint dev, uint sector)
{
struct buf * b;
b=bget(dev,sector);
If(!(b->flags & B_valid))
iderw(b);
}
```

**Writing a Disk Block**

```
 Void bwrite(struct buf *b)
{
If (b->flags & B_BUSY)==0)
   panic("bwrite");
   b->flags |= B_DIRTY;
   iderw(b);
}
```

# Advantages and Disadvantages

Advantages

Allows uniform disk access

Eliminates the need for special alignment of user buffers

- by copying data from user buffers to system buffers,

Reduce the amount of disk traffic

- less disk access

Insure file system integrity

- one disk block is in only one buffer

Disadvantages

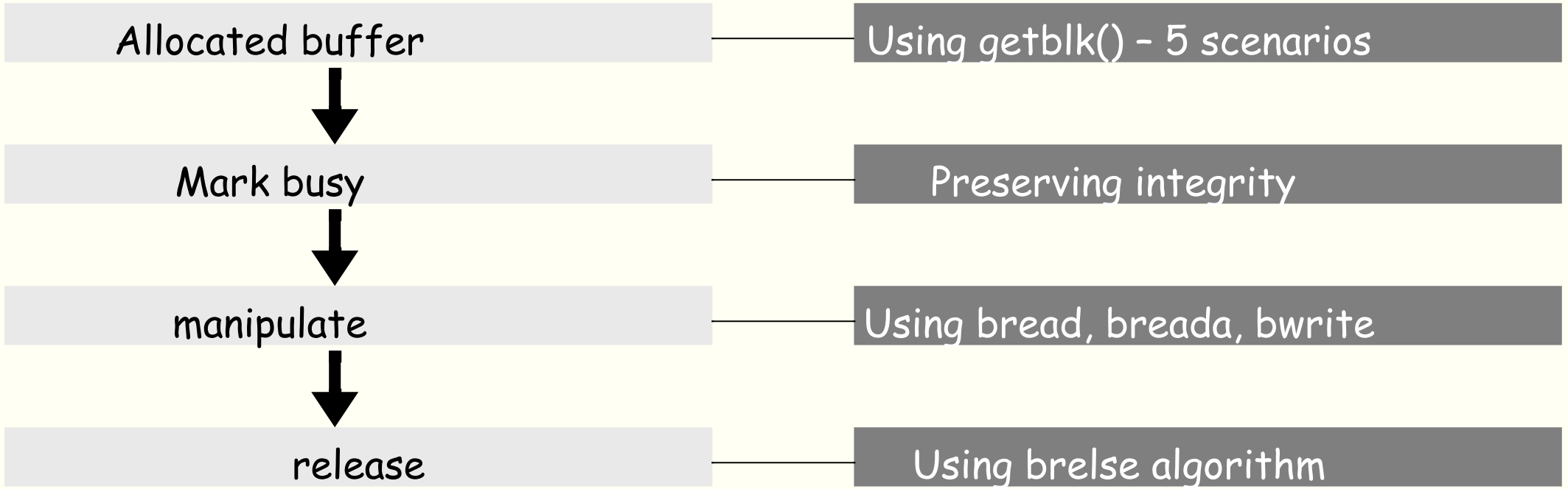
Can be vulnerable to crashes

- When delayed write

requires an extra data copy

- When reading and writing to and from user processes

# What happen to buffer until now



# Logging

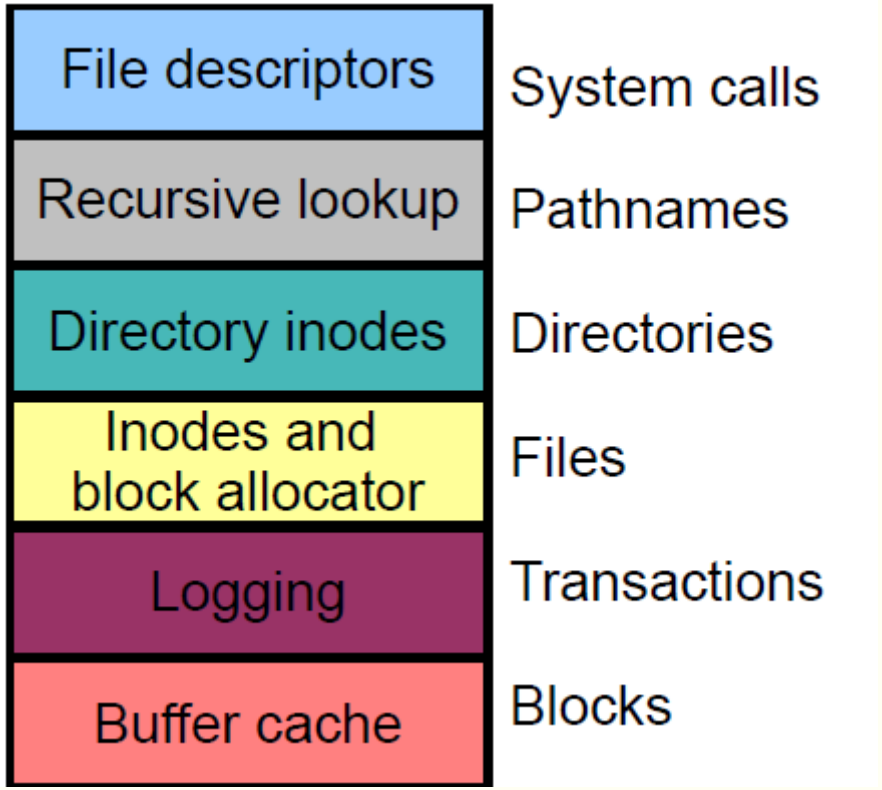
The purpose of a file system is to:

1 Organize and store data

2 Support sharing of data among users and applications

3 Ensure persistence of data after a reboot.

- The file system needs to support crash recovery
  - A restart must not corrupt the file system or leave it in an inconsistent state




XV6 File System Layers

# Logging Layer

One of the most interesting problems in file system design is crash recovery.

xv6 implements file system fault tolerance through a simple logging mechanism

- System calls do not directly write file system data structures
    - Instead:

1. A system call first writes a description of all the disk writes that it wishes to perform to a log on the disk
2. It then writes a special commit record to the log to specify that it contains a complete operation
3. Next it copies the required writes to the on-disk file system data structures
4. Finally, it deletes the log

# Recovery

- In case of a reboot, the file system performs recovery by looking at the log file .
- If the log contains the commit record, the recovery code copies the required writes to the on-disk data structures .
- If the log does not contain a complete operation, it is ignored and deleted.
- If the crash occurs before the commit record, the log will be ignored, and the state of the disk will stay unmodified .
- If the crash occurs after the commit record, then the recovery will replay all of the operation's writes, even repeating them if the crash occurred during the write to the on-disk data structure .
- In both cases, the correctness of the file system is preserved: Either all writes are reflected on the disk or none

# Log Design

- The log resides at a fixed location at the end of the disk
- It consists of a header block and a set of data blocks
- The header block contains
    1. An array of sector numbers, one for each of the logged data blocks
    2. Count of logged blocks
- The header block is written to after a commit
- The count is set to zero once all logged blocks have been reflected in the file system
    - The count will be zero in case of a crash before a commit
    - The count will be non-zero in case of a crash after a commit

# Log Design

- To allow concurrent execution of file system operations by different processes , the logging system can accumulate the writes of multiple system calls into one transaction.
- A transaction sequence is indicated by the start and end sequence of writes in the system call
- Only one system call can be in a transaction at any given time to ensure correctness
- The log holds at most one transaction at a time
- Only read system calls can execute concurrently with a transaction
- A fixed amount of space on the disk is dedicated to hold the log
- No system call can write more distinct blocks than the size of the log
  - Large writes are broken into multiple smaller writes so that each write can fit in the log

# Typical system call usage of log

```
begin_trans();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
commit_trans();
```

- begin_trans: Waits until it obtains exclusive use of the log •
- log_write:
  - Appends the block's new content to the log on the disk
  - Leaves the modified block in the buffer cache so that subsequent reads of the block during the transaction will yield the updated state
  - Records the block's sector number in memory to find out when a block is written multiple times during a transaction and overwrite the block's previous copy in the log •
- commit_trans:
  1. Writes the log's header block to disk, updating the count
  2. Calls install_trans to copy each block from the log to the relevant location on the disk
  3. Sets to count in the log header to zero

# Log_Write

```
begin_trans();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
commit_trans();
```

```
4922    void
        log_write(struct buf *b)
4923    {
4924      int i;
4925
4926      if (log.lh.n >= LOGSIZE || log.lh.n >= log.size − 1)
4927        panic("too big a transaction");
4928      if (log.outstanding < 1)
4929        panic("log_write outside of trans");
4930
4931      acquire(&log.lock);
4932    for (i = 0; i < log.lh.n; i++) {
4933    if (log.lh.block[i] == b−>blockno) // log absorbtion
4934      break;
4935      }
4936      log.lh.block[i] = b−>blockno;
4937    if (i == log.lh.n)
4938      log.lh.n++;
4939      b−>flags |= B_DIRTY; // prevent eviction
4940       release(&log.lock);
4941        }
```

# Operating Systems Design( 19CS2106S)
# Session-6
# Low-Level File System Algorithms
# (*Internal Representation of Files*)

**Text Book- The Design of the UNIX Operating System**

**----------Marice J.Bach**

## Review of Session-4

- Session-4 deals about Buffer Cache.
- Buffer Cache is small area in Main Memory(RAM).
- Similar to Cache Memory.
- Its purpose is to ***temporarily store*** the frequently accessed Blocks from Disk Memory (Secondary Memory) into the Buffers of the Buffer Cache ***by the Kernel.***
- ***That means, before allocating the buffers/blocks to various processes , the kernel keeps these disks blocks temporarily inside the buffers of this buffer cache.***
- In Buffer Cache, there are components like Buffer Header, and the other area where the buffers actually located in Hash Queues & Free Lists.

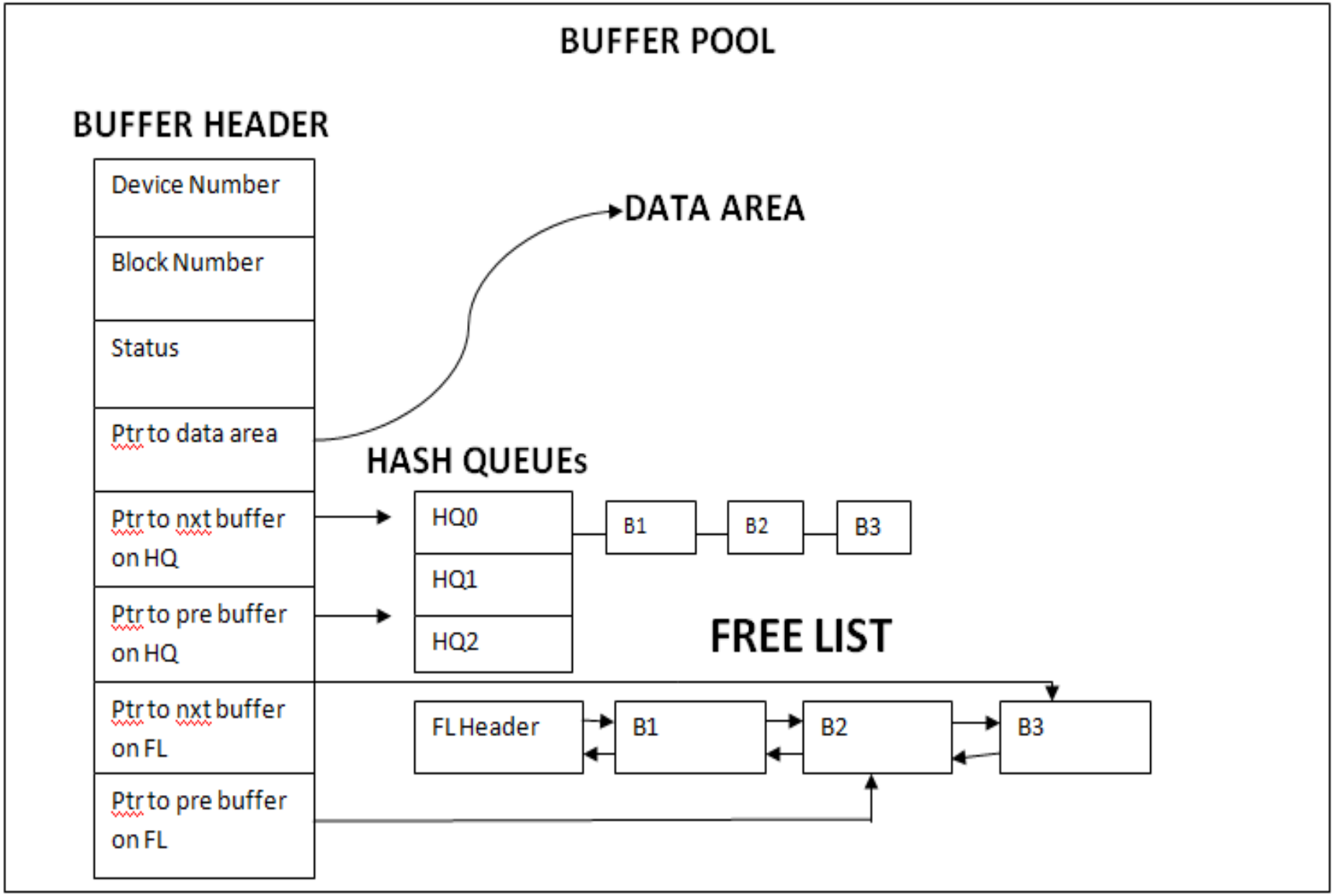
## Review of Session-4

- Buffer Header consists of various fields like device number, block number, status and various pointer fields
- By default, the kernel stores the disk blocks in the buffers of Hash Queues.
- These buffers will be allocated to various processes by the Kernel from time to time depending on the requirement.
- The Hash Queue stores functions based on a Hash function and this hash function is a ***device number mod block number*** combination

# Review of Session-4

- So, the blocks are mainly stored in Hash Queue buffers and these buffers the kernel allocates to various processes.
- But all the buffers stored in the HQs will not be allocated to processes, and few buffers will NOT be allocated to any processes and will be left FREE.
- These freely available blocks will be kept separately in a data structure called FREE LIST.

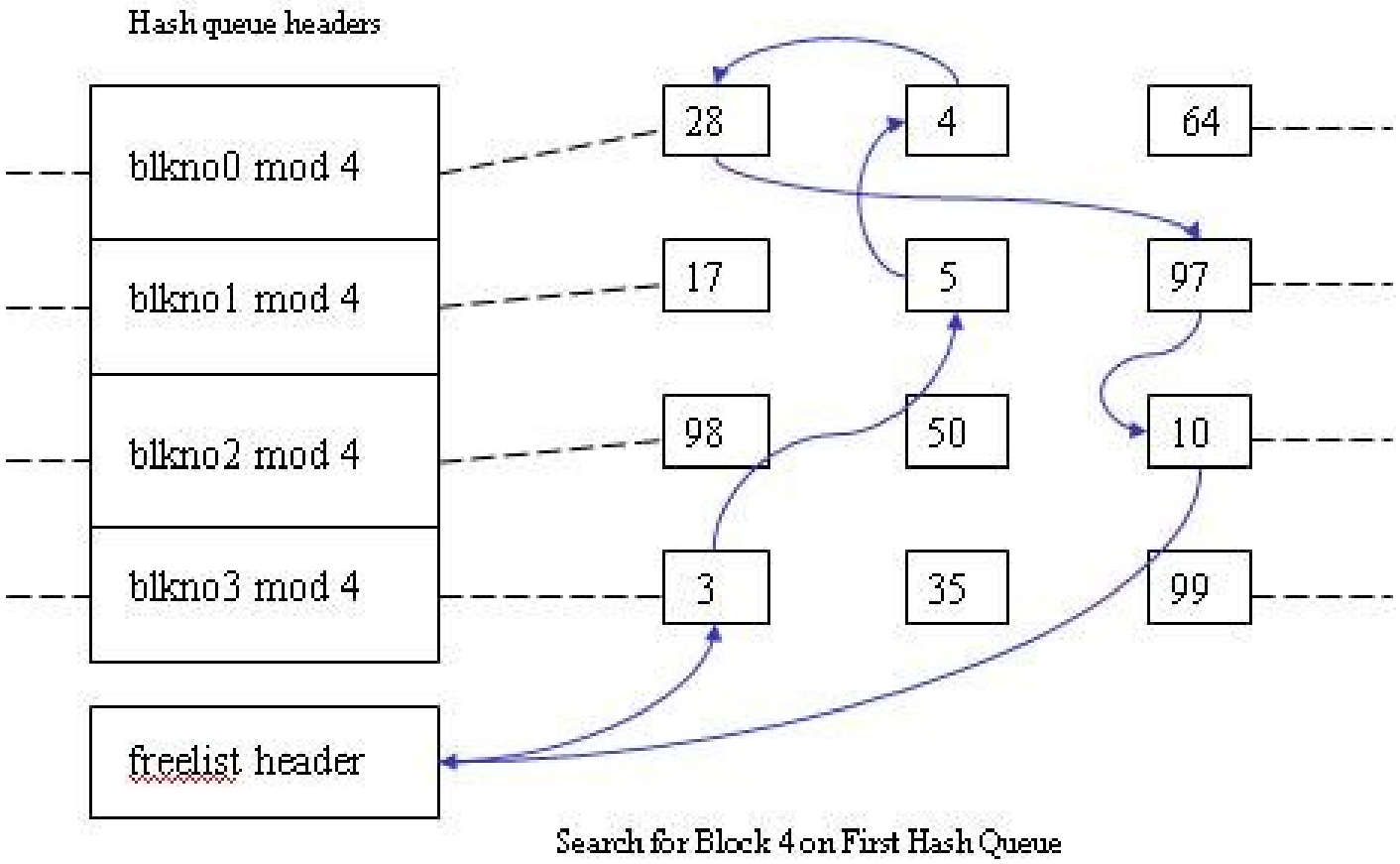
# Buffer Cache-Micro View



# Review of Session-4-FIVE SCENARIOS

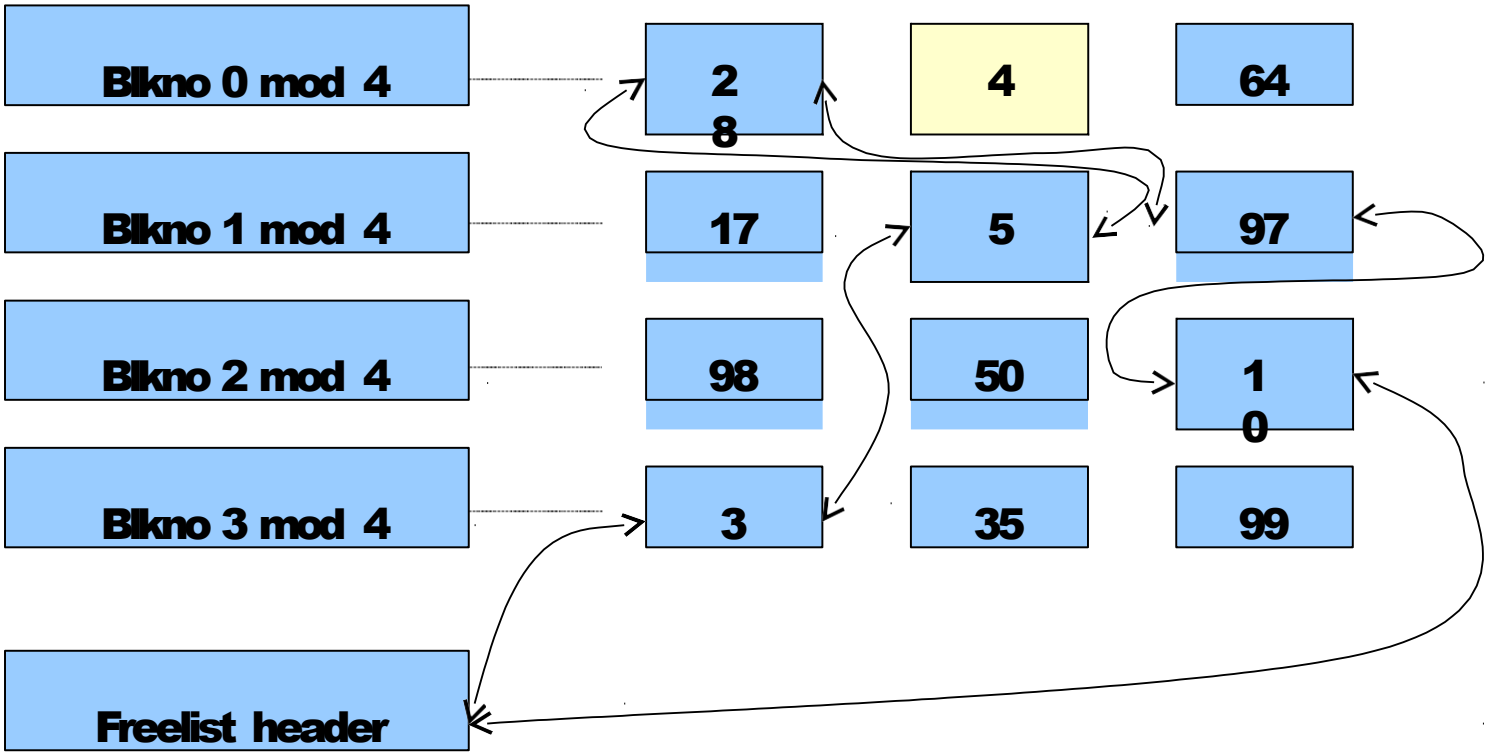
***While allocating the buffer blocks to various processes, the kernel faces FIVE different scenarios***

# Scenario-1


Hash queue headers
blkno0 mod 4
blkno1 mod 4
blkno2 mod 4
blkno3 mod 4
freelist header
28
4
64
17
5
97
98
50
10
3
35
99
Search for Block 4 on First Hash Queue

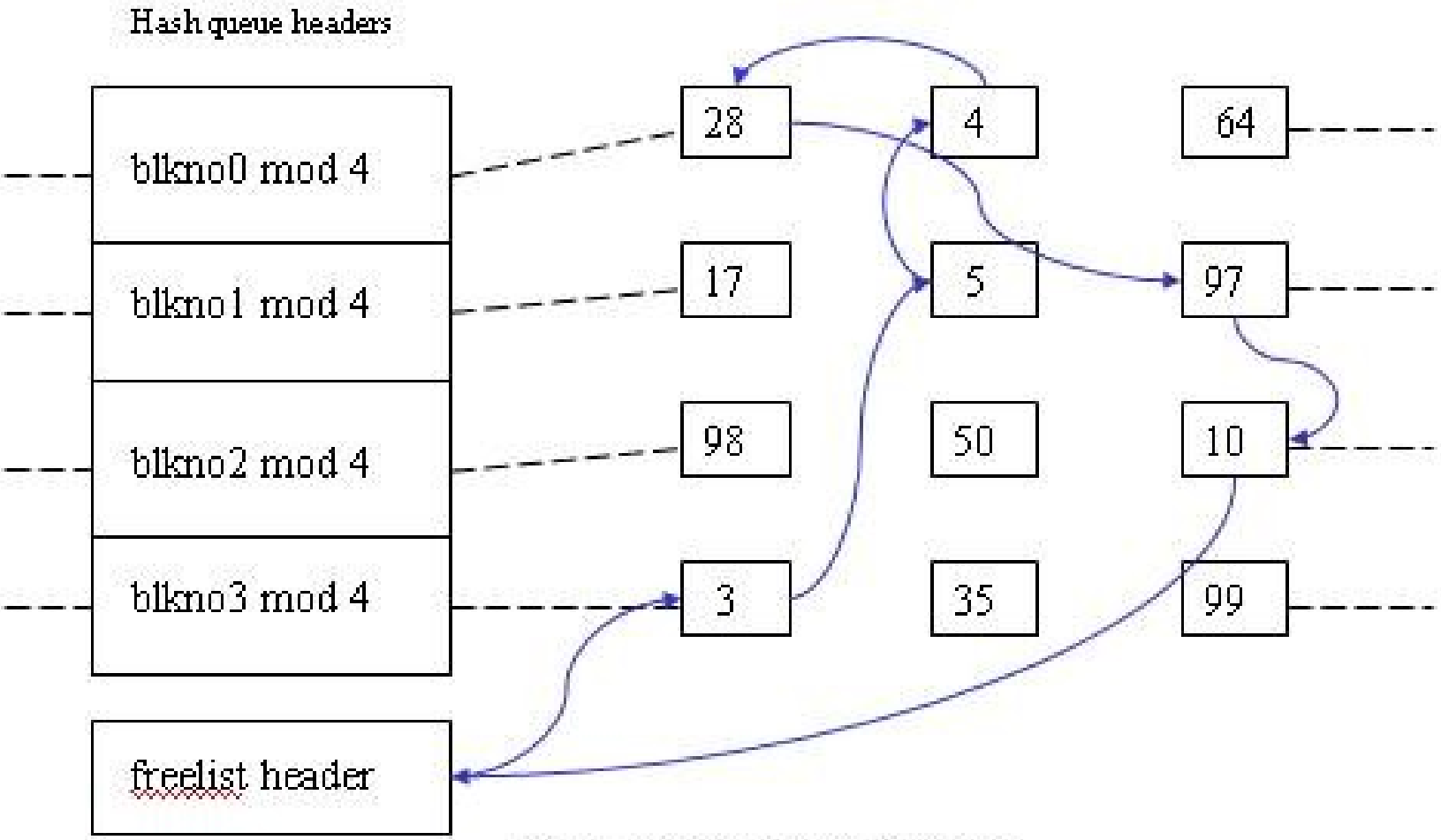
# 1st Scenario-After Allocation

After allocating



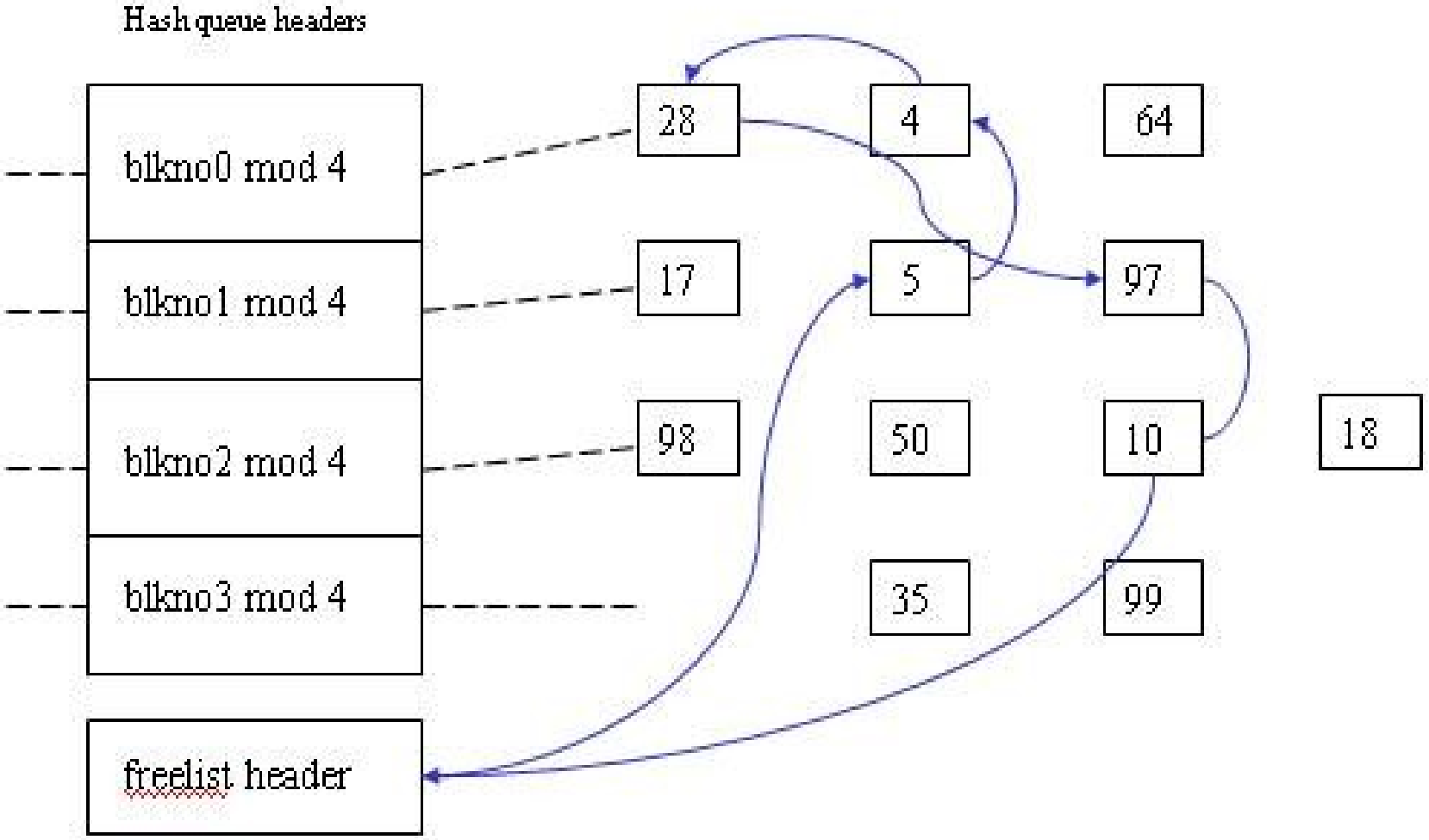

(b) Remove block 4 from free list

# Scenario-2



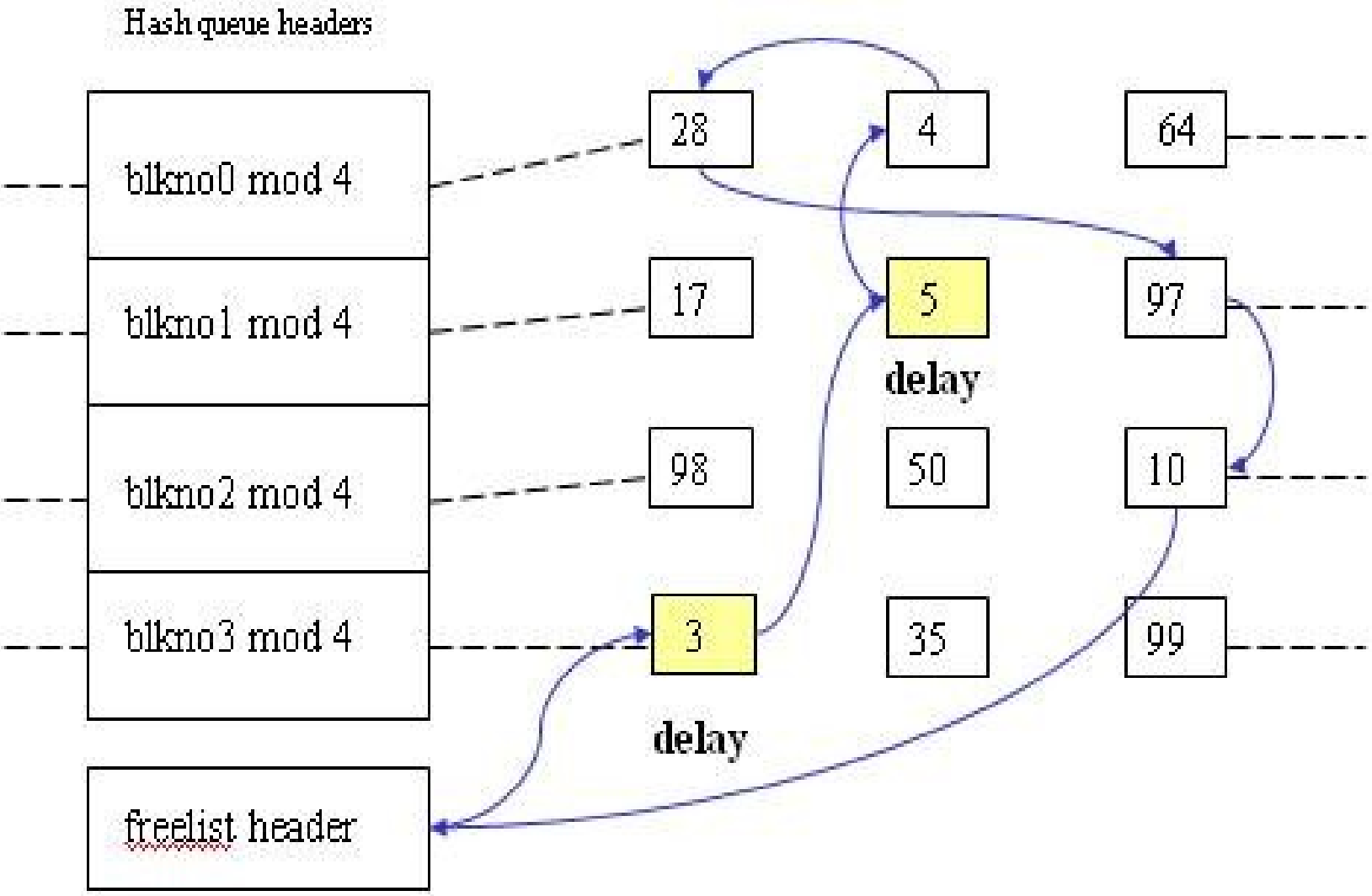

(a) Search for Block 18-Not in Cache

# Scenario-2-After Allocation



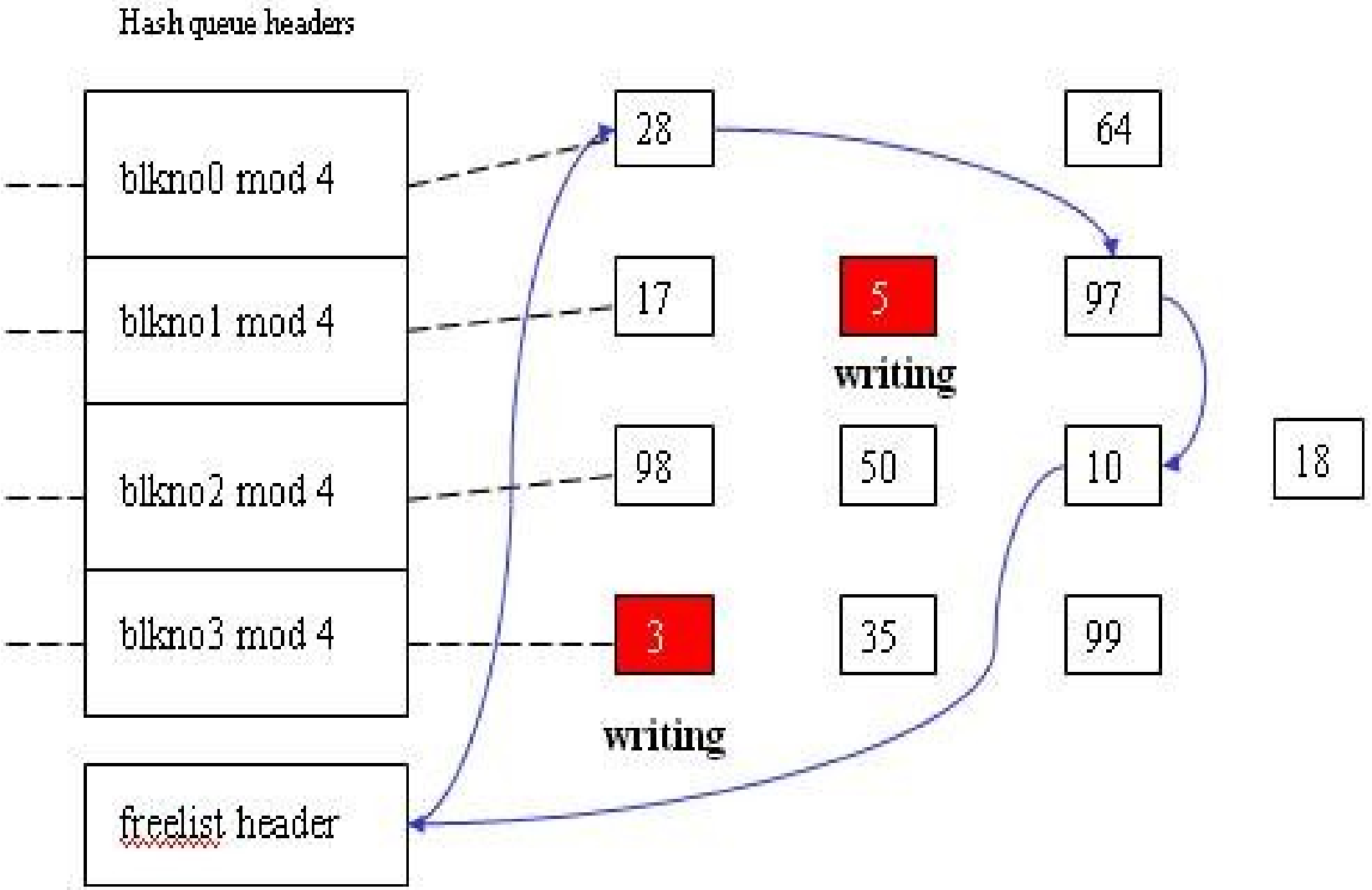

(b) Remove First Block from Free List, Assign to 18

# Scenario-3-with Delayed Write Buffers




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

# Scenario-3-After the Allocation




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

# What is Delayed-Write ?

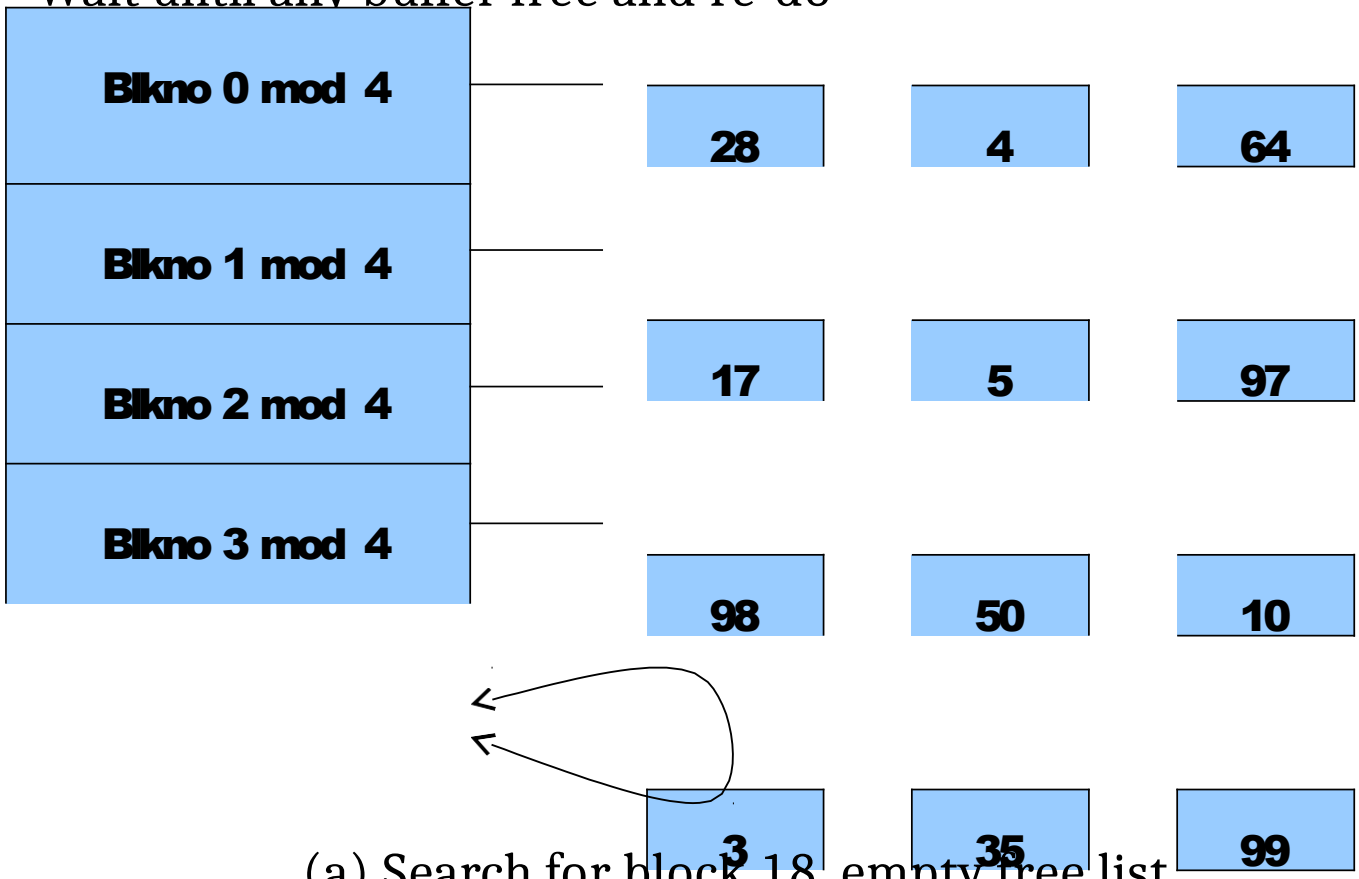
- It is mechanism of Kernel, intentionally delaying the write operation of a buffer block for some time , to increase the performance and throughput.
- Normally, the kernel has to perform the write operation synchronously **twice** i.e first in disk block and then in the buffer block.
- But here, the kernel after performing the write operation ,it delays/postpones the write operation on the buffer block intentionally , to speed up the operation.
- After some time, the kernel performs the write operation on the buffer block.

***For a buffer block, if the write operation is delayed or pending, it SHOULD NOT be allocated to any process, till the write operation completes.***

# 4th Scenario

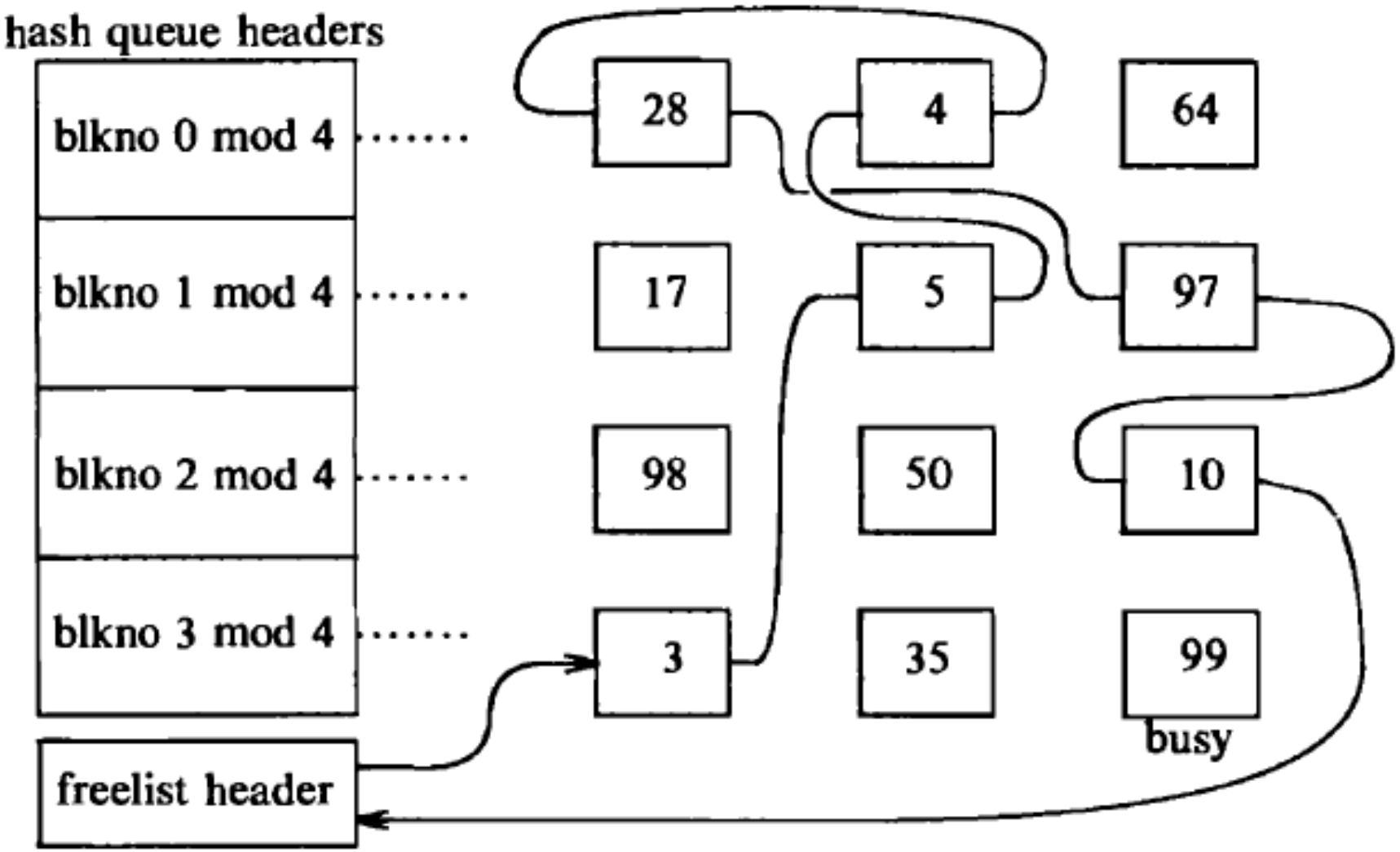
Nothing in the hash queue and no free buffer

Wait until any buffer free and re-do




(a) Search for block 18, empty free list

# Scenario-5




Search for Block 99, Block Busy

# Text Book

- ***The Design of UNIX Operating System***

***----Marice J.Bach***

# Review of Session-5-**Buffer Cache Algorithms**

- ***1.brelse***
- ***2.getblk***
- ***3.bread***
- ***4.breada***

# Review of Session 5:***brelse***

- When a buffer block is allocated to a process lets say, P1 ,by the kernel, that buffer block is marked as ***busy,*** so that it can not be accessed by any other process.
- If any other processes tries to access that buffer block, it will NOT be available for those processes because its marked busy
- Now  all those processes will go to sleep mode.
- Once that process P1 completes its hold on that buffer block, then the kernel releases that buffer block from the process P1, using this algorithm ***brelse.***
- That released buffer block is placed at end of the FREE LIST.
- Now the kernel awakens the sleeping processes and allocates the buffer blocks.

## Review of Session 5: *getblk*

- Used for Buffer Allocation.
- When about to **read data** from a particular disk block, the kernel checks whether the block is in the buffer pool and, if it is not there, assigns it a free buffer.
- When about to **write data** to a particular disk block, the kernel checks whether the block is in the buffer pool, and if not, assigns a free buffer for that block.
- The algorithms for reading and writing disk blocks use the algorithm ***getblk*** *t*o allocate buffers from the pool.
- Five Scenarios can be understood

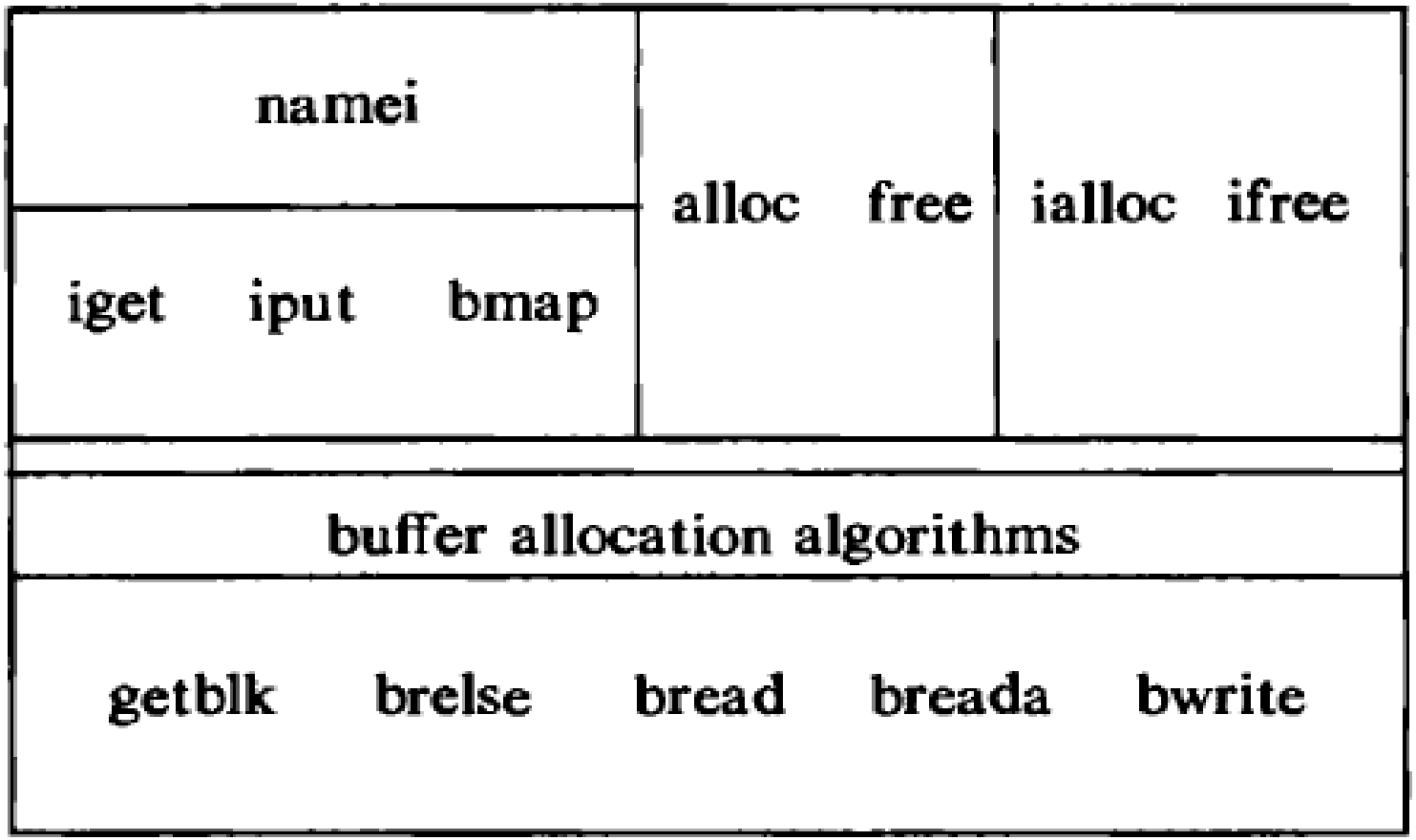
# Review of Session *5-bread & breada*

- ***bread*** stands for block read.
- This algorithm is used to read a disk block.
- ***breada*** stands for block read ahead.
- This algorithm also used for disk block reading but with a small difference.
- That is, ***bread*** is used for simple read operation of a single disk block where as ***breada*** is used for asynchronous reading of TWO disk blocks one after the other.

# Session-6

***Low Level file System Algorithms (Internal Representation of Files)***

# Lower Level File System Algorithms



# Review of ***inode***

- ***inode*** stands for index node.
- In Unix based operating system each file is indexed by a number called ***inode(index node)*** *or inode number.*
- It’s a name of a file in the form a number.
- The ***inode*** contains the information necessary for a process to access a file, such as file ownership, access rights, file size, and location of the file's data in the file system**.**
- inode contains the table of contents to locate a file's data on disk.
- Also called as ***disk inode.***

# inode number can be viewed using the *ls -il* command

```
bob@ubuntu:~$ ls -il
total 56
26117 drwxr-xr-x 2 bob  bob  4096 Apr 24 04:14 Desktop
26121 drwxr-xr-x 2 bob  bob  4096 Apr 24 04:14 Documents
26118 drwxr-xr-x 2 bob  bob  4096 May  5 04:57 Downloads
22852 -rw-r--r-- 1 bob  bob  8445 Apr 22 06:14 examples.desktop
26122 drwxr-xr-x 2 bob  bob  4096 Apr 24 04:14 Music
26123 drwxr-xr-x 2 bob  bob  4096 Apr 24 04:14 Pictures
26120 drwxr-xr-x 2 bob  bob  4096 Apr 24 04:14 Public
31628 -rwxr-xr-x 1 root root   30 May  8 07:44 script.sh
26119 drwxr-xr-x 2 bob  bob  4096 Apr 24 04:14 Templates
21303 -rw-rw-r-- 1 bob  bob  1799 Apr 28 04:55 textfile.txt
21181 -rw-rw-r-- 1 bob  bob    13 Apr 28 04:53 textfile.txt~
26124 drwxr-xr-x 2 bob  bob  4096 Apr 24 04:14 Videos
bob@ubuntu:~$
```

# Disk inode of a Sample File

owner mjb

group os

type regular file

perms rwxr-xr-x

accessed Oct 23 1984 1:45 P.M.

modified Oct 22 1984 10:30 A.M.

inode Oct 23 1984 1:30 P.M.

size 6030 bytes

disk addresses

# Concept of ***in-core inode***

- When the file is opened, then the kernel copies the inode(disk inode) into main memory area called inode cache, just likie the buffer cache.
- So, this copy of disk inode present in MM is called as ***in-core inode***.
- As the file changes, the ***in-core inode*** is updated usually more often than the on-disk copy.
- The in-core inode contains up-to-date information on the state of the file
- This is allocation is done using ***iget*** algorithm

## Accessing of ***inode***

- Accessing of inode is similar to the concept of accessing a buffer block in Buffer Cache.
- Here also the concept Hash Queue and FREE LIST will be there.
- The kernel maps the device number and inode number into a hash queue and searches the queue for the
- inode.
- If it cannot find the inode, it allocates one from the free list and locks it.
- The kernel then prepares to read the disk copy of the newly accessed inode into the in-core copy.

# Allocation of an in-core copy of actual disk inode *iget* Algorithm

```
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);
                                continue;       /* loop back to while */
                        }
                        /* special processing for mount points (Chapter 5) */
                        if (inode on inode free list)
                                remove from free list;
                        increment inode reference count;
                        return (inode);
                }
```

# Allocation of an in-core copy of actual disk inode

## *iget* Algorithm

```
        /* 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);
    }
}
```

- The kernel removes the in-core mode from the free list, places it on the correct hash queue, and sets its in-core ***reference count*** to 1.

- It copies the file type, owner fields, permission settings, link count, file size, and the table of contents from the disk inode to the in-core inode, and returns a locked inode.

# Releasing the inodes

- When the kernel releases an inode ,*it decrements its* in-core reference count.
- If the count drops to 0, the kernel writes the inode to disk if the in-core copy differs from the disk copy.
- They differ if the file data has changed, if the file access time has changed, or if the file owner or access permissions have changed.
- Now, the kernel places the inode on the free list of inodes, effectively caching the inode in case it is needed again soon.
- The kernel may also release all data blocks associated with the file and free the inode if the number of links to the file is 0.

# Releasing the inodes using ***iput*** algorithm

```
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, section 4.7);
                        set file type to 0;
                        free inode (algorithm ifree, section 4.6);
                }
                if (file accessed or inode changed or file changed)
                        update disk inode;
                put inode on free list;
        }
        release inode lock;
}
```

# Directories

- Directories are the set of files that gives the file system its hierarchical structure.
- A directory is a file whose data is a sequence of entries, each consisting of an inode number and the name of a file contained in the directory.
- A path name is a null terminated character string divided into separate components by the slash ("/") character.
- Each component except the last must be the name of a directory, but the last component may be a non-directory file

# Directory layout for /etc

| Byte Offset in Directory | Inode Number (2 bytes) | File Names |
|---|---|---|
| 0 | 83 | . |
| 16 | 2 | .. |
| 32 | 1798 | init |
| 48 | 1276 | fsck |
| 64 | 85 | clri |
| 80 | 1268 | motd |
| 96 | 1799 | mount |
| 112 | 88 | mknod |
| 128 | 2114 | passwd |
| 144 | 1717 | umount |
| 160 | 1851 | checklist |
| 176 | 92 | fsdb1b |
| 192 | 84 | config |
| 208 | 1432 | getty |
| 224 | 0 | crash |
| 240 | 95 | mkfs |
| 256 | 188 | inittab |

- Directory entries may be empty, indicated by an mode number of 0.
- For instance, the entry at address 224 in "/etc" is empty, although it once contained an entry for a file named "crash".

- Every directory contains the file names dot and dot-dot ("." and "..") whose inode numbers are those of the directory and its parent directory, respectively.
- The inode number of "." in "/etc“ is located at offset 0 in the file, and its value is 83.
- The mode number of ".." is located at offset 16, and its value is 2.

## Accessing a file in directory using path name

- The access to a file in a directory is by its path name,
- But the kernel works internally with inodes rather than with path names.
- Therefore, it converts the path names to inodes to access files.
- ***namei*** algorithm is used for Conversion of a Path Name to an mode.
- ***It follows a pathname until a terminal point is found***

# *namei* Algorithm

- The user ID of the process must match the owner or group ID of the file, and execute permission must be granted, or the file must allow search to all users.
- Otherwise the search fails.

# *namei* Algorithm

- ***namei*** *algorithm uses intermediate inodes as it parses a path name; call them working* inodes.
- The inode where the search starts is the first working mode.
- During each iteration of the ***namei*** *loop, the kernel makes sure that the working inode is indeed* that of a directory.
- Otherwise, the system would violate the assertion that non-directory files can only be leaf nodes of the file system tree.
- The process must also have permission to search the directory (read permission is insufficient).

# *namei* Algorithm

```
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;
                verify 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 algorithms
                        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);
                }
```

# ***namei*** Algorithm

```
        else    /* component not in directory */
            return (no inode);
    }
    return (working inode);
}
```

*Text Book to be followed :*

# *The Design of the UNIX Operating System*

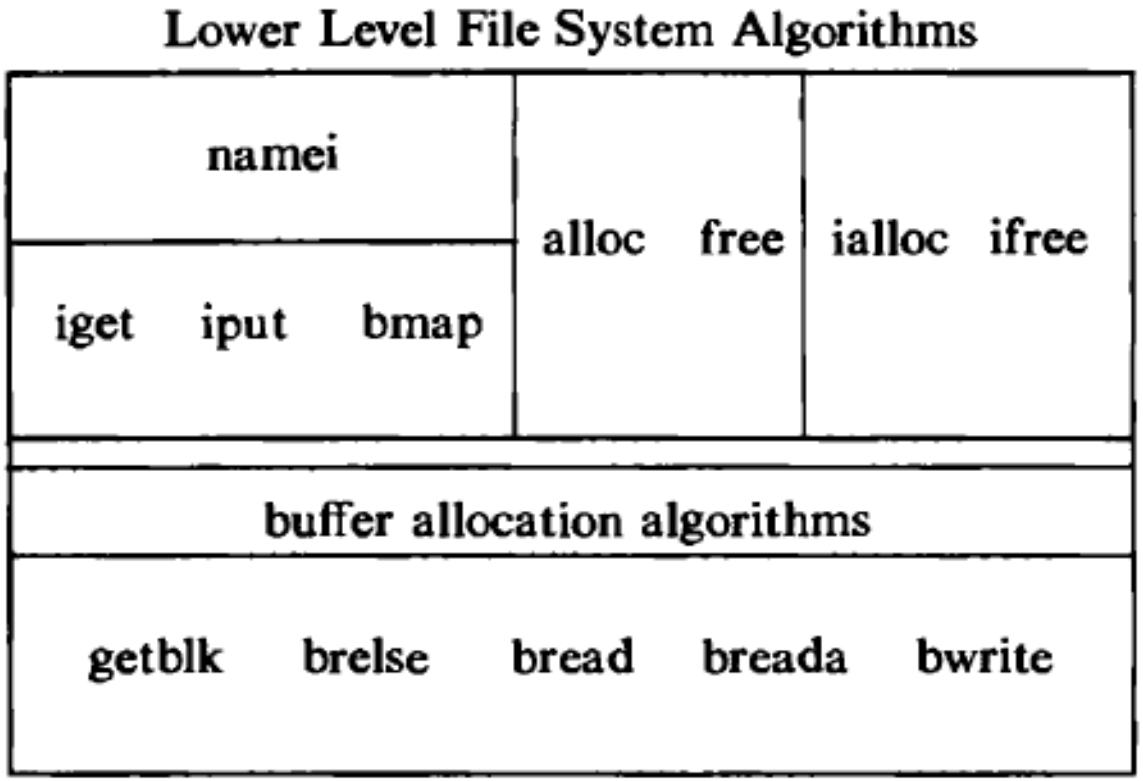
## *--Marice J.Bach*

# Operating Systems Design( 19CS2106S)
# Session-7
# Low-Level File System Algorithms
# (*Internal Representation of Files*)

**Text Book- The Design of the UNIX Operating System**

**----------Marice J.Bach**

# Review of Session-6



# Review of Session-6-***inode***

- ***inode*** stands for index node.
- In Unix based operating system each file is indexed by a number called ***inode(index node)*** *or inode number.*
- It’s a name of a file in the form a number.
- The ***inode*** contains the information necessary for a process to access a file, such as file ownership, access rights, file size, and location of the file's data in the file system**.**
- inode contains the table of contents to locate a file's data on disk.
- Also called as ***disk inode.***

# Review of Session-6- Concept of in-core inode

- When the file is opened, then the kernel copies the inode(disk inode) into main memory area called inode cache, just like the buffer cache.
- So, this copy of disk inode present in MM is called as ***in-core inode***.
- As the file changes, the ***in-core inode*** is updated usually more often than the on-disk copy.
- The in-core inode contains up-to-date information on the state of the file
- This is allocation is done using ***iget*** algorithm

## Review of Session-6- iget Algorith

- **iget** algorithm allocates an in-core copy of an inode
- It is almost identical to the algorithm ***getblk*** for finding a disk block in the buffer cache.
- Here, The kernel maps the device number and iode number into a hash queue and searches the queue for the inode.
- If it cannot find the inode, it allocates one from the free list and locks it.
- The kernel then prepares to read the disk copy of the newly accessed inode into the in-core copy.

# Review of Session-6-*iput* Algorithm

- ***iput*** algorithm is used to release the inodes
- When the kernel releases an inode ,*it decrements its* in-core reference count.
- If the count drops to 0, the kernel writes the inode to disk if the in-core copy differs from the disk copy.
- They differ if the file data has changed, if the file access time has changed, or if the file owner or access permissions have changed.
- Now, the kernel places the inode on the free list of inodes, effectively caching the inode in case it is needed again soon.

## Review of Session-6-***namei* Algorithm**

- ***namei*** algorithm is used for the conversion of a path name to an inode.
- It follows a pathname until a terminal point is found.
- In this *algorithm ,it uses intermediate inodes as it parses a path name; call them working* inodes.
- The inode where the search starts is the first working mode.
- During each iteration of the ***namei*** *loop, the kernel makes sure that the working inode is indeed* that of a directory.
- Otherwise, the system would violate the assertion that non-directory files can only be leaf nodes of the file system tree.

# Session-7

## Low Level File System Algorithms

- *ialloc & ifree*
- *alloc & free*

# *Lets start with concept of Super Block*

- When a partition or disk is formatted, the sectors in the hardisk is first divided into small groups. This groups of sectors is called as blocks.
- The block size is something that can be specified when a user formats a partition.
- The most simplest definition of Superblock is that, its the metadata of the file system.
- Similar to how i-nodes stores metadata of files, Superblocks store metadata of the file system

# Super Block

- A request to access any file requires access to the filesystem's superblock.
- If its superblock cannot be accessed, a filesystem cannot be *mounted* (i.e., logically attached to the main filesystem) and thus files cannot be accessed.

- ***The Super block contains an array to cache the numbers of free inodes in the file system.***

# Fields of Super Block

- size of the file system,
- the number of free blocks in the file system,
- a list of free blocks available on the file system,
- the index of the next free block in the free block list,
- the size of the inode list,
- the number of free inodes in the file system,
- a list of free inodes in the file system,
- the index of the next free inode in the free mode list,
- lock fields for the free block and free inode lists,
- a flag indicating that the super block has been modified.

# *ialloc* Algorithm

- ***ialloc*** algorithm allocates new inodes to various processes.

- We know that , the Super Block contains new free inodes.

# ialloc Algorithm

- The kernel first verifies that no other processes have locked access to the super block free inode list.
- If the list of inode numbers in the super block is not empty, the kernel assigns the next mode number, allocates a free in-core inode for newly assigned disk inode using algorithm ***iget*** *(reading the inode from disk if* necessary), copies the disk inode to the in-core copy, initializes the fields in the inode, and returns the locked inode.

# *ialloc* Algorithm

- If the super block list of free inodes is empty, the kernel searches the disk and places as many free inode numbers as possible into the super block.
- The kernel reads the inode list on disk, block by block, and fills the super block list of inode
- numbers to capacity, remembering the highest-numbered inode that it finds.
- Call that inode the "remembered" inode;
- it is the last one saved in the super block.

# *ialloc* Algorithm

```
algorithm ialloc        /* allocate inode */
input:   file system
output: locked inode
{
        while (not done)
        {
                if (super block locked)
                {
                        sleep (event super block becomes free);
                        continue;       /* while loop */
                }
                if (inode list in super block is empty)
                {
                        lock super block;
                        get remembered inode for free inode search;
                        search disk for free inodes until super block full,
                                or no more free inodes (algorithms bread and brelse);
                        unlock super block;
                        wake up (event super block becomes free);
                        if (no free inodes found on disk)
                                return (no inode);
                        set remembered inode for next free inode search;
                }
```

# *ialloc* Algorithm

```
        /* there are inodes in super block inode list */
        get inode number from super block inode list;
        get inode (algorithm iget);
        if (inode not free after all)        /* !!! */
        {
                write inode to disk;
                release inode (algorithm iput);
                continue;        /* while loop */
        }
        /* inode is free */
        initialize inode;
        write inode to disk;
        decrement file system free inode count;
        return (inode);
    }
}
```

# *ifree* Algorithm

- After incrementing the total number of available inodes in the file system, the kernel checks the lock on the super block.
- If locked, it avoids race conditions by returning immediately: The inode number is not put into the super block, but it can be found on disk and is available fo reassignment.

# *ifree* Algorithm

- If the list is not locked, the kernel checks if it has room for more inode numbers and, if it does, places the inode number in the list and returns.
- If the list is full, the kernel may not save the newly freed mode there: It compares the number of the freed inode with that of the remembered inode.
- If the freed inode number is less than the remembered inode number, it "remembers" the newly freed inode number, discarding the old remembered mode number from the super block.

# *ifree* Algorithm

```
algorithm ifree        /* inode free */
input:   file system inode number
output: none
{
        increment file system free inode count;
        if (super block locked)
                return;
        if (inode list full)
        {
                if (inode number less than remembered inode for search)
                        set remembered inode for search = input inode number;
        }
        else
                store inode number in inode list;
        return;
}
```

## Allocation & Free of Disk Blocks

- When a process writes data to a file, the kernel must allocate disk blocks from the file system for direct data blocks and, sometimes, for indirect blocks.
- The file system super block contains an array that is used to cache the numbers of free disk blocks in the file system.
- This allocation can be done using ***alloc*** algorithm

# *alloc Algorithm*

- When the kernel wants to allocate a block from a file system, it allocates the next available block in the super block list.
- Once allocated, the block cannot be reallocated until it becomes free.
- If the allocated block is the last available block in the super block cache, the kernel treats it as a pointer to a block that contains a list of free blocks.

# *alloc* Algorithm

- It reads the block, populates the super block array with the new list of block numbers, and then proceeds to use the original block number.
- It allocates a buffer for the block and clears the buffer's data (zeros it).
- The disk block has now been assigned, and the kernel has a buffer to work with.
- If the file system contains no free blocks, the calling process receives an error

# *alloc* Algorithm

```
algorithm alloc     /* file system block allocation */
input:   file system number
output: buffer for new block
{
        while (super block locked)
                sleep (event super block not locked);
        remove block from super block free list;
        if (removed last block from free list)
        {
                lock super block;
                read block just taken from free list (algorithm bread);
                copy block numbers in block into super block;
                release block buffer (algorithm brelse);
                unlock super block;
                wake up processes (event super block not locked);
        }
        get buffer for block removed from super block list (algorithm getblk);
        zero buffer contents;
        decrement total count of free blocks;
        mark super block modified;
        return buffer;
}
```

## ***free*** Algorithm

- The algorithm ***free*** *for freeing a block is the reverse of the one for allocating a* block.
- If the super block list is not full, the block number of the newly freed block is placed on the super block list.
- If, however, the super block list is full, the newly freed block becomes a link block; the kernel writes the super block list into the block and writes the block to disk.
- It then places the block number of the newly freed block in the super block list: That block number is the only member of the list.

*Text Book to be followed :*

# *The Design of the UNIX Operating System*

## *--Marice J.Bach*

# Operating Systems Design
# 19CS2106S

## Session-8
## File System Calls

# Session Plan

(DEEMED TO BE UNIVERSITY)

**SESSION NUMBER** : 8

**Session Outcome: 1** Understand and Explore the Design of File System Calls: Open, read, write, close

| Time(min) | Topic | BTL | Teaching- Learning Methods | Active Learning Methods |
|---|---|---|---|---|
| 5 | Attendance/Poll/Pop Question | 1 | Talk | --- NOT APPLICABLE --- |
| 20 | Algorithms for File System Calls: open, read, write, close | 3 | PPT | --- NOT APPLICABLE --- |
| 5 | Ask for any doubts through Public chat/ Break | 1 | Talk | --- NOT APPLICABLE --- |
| 10 | Xv6 functions: sys_open, filealloc, sys_read, fileread, sys_write, filewrite, sys_close, fileclose | 3 | PPT | Case Study |
| 10 | Design and Implementation of open, read, write, close, file.c, sysfile.c | 3 | LTC | --- NOT APPLICABLE --- |

# Learning Outcomes

- ***To understand open ( ), read ( ), write ( ) and close ( ) systems calls.***
- ***To explore the above system calls in xv6 as case study.***
- ***To understand the design and implementation of sysfile.c***

## Basic Operations on Files

- Opening Files
- Reading and Writing Files
- Closing Files

- All the above operations can be implemented using ***Open ( ), Read ( ), Write( ), Close ( )*** File System Calls

# Review of basic terminology with Files


- File Descriptor
- Concept of ***inode***
- ***inode*** table
- File Table

# Lets start with File Descript

- A ***file descriptor*** is just an integer, private per process, and is used in UNIX systems to access files.

- It is dynamically generated by the kernel when you call open( ) system call (or certain other system calls).

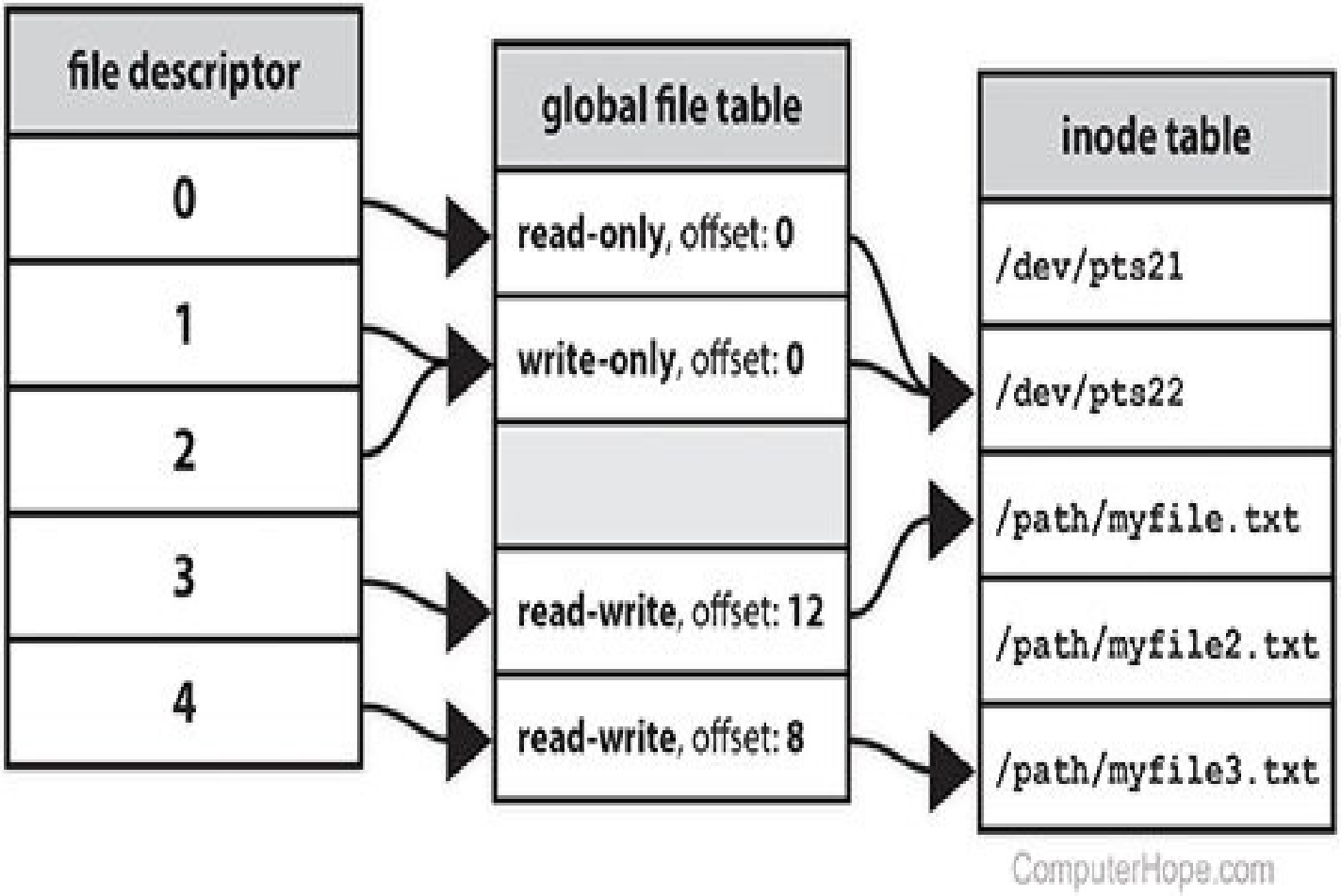
- When a file is opened, the file descriptor is used to read or write the file assuming you have permission to do so.

- The descriptor is identified by a ***unique non-negative integer***, such as 0, 12, or 567.

- At least one file descriptor exists for every open file on the system

- The first three user file descriptors (0, 1, and 2) are called the standard input, standard output, and standard error file descriptors.

- Processes on UNIX systems conventionally use the standard input descriptor to read input data, the standard output descriptor to write output data, and the standard error descriptor to write error data (messages).



- When a process makes a successful request to open a file, the kernel returns a ***number***(file descriptor) which points to an entry in the kernel's global file table.
- The file table entry contains information such as the ***inode*** of the file, ***byte offset***, and the access restrictions for that data stream (read-only, write-only
- ***Byte offset*** is the number of **bytes** from the beginning of the **file**.
- Read and write operations normally start at the current **offset** and cause the **offset** to be incremented the number of **bytes** read or written, etc.).

# Syntax


*fd =open(pathname, flags, modes);*

*Or*

*int fd = open(pathname , flags , modes);*

# *Here*

***pathname*** is a file name, ***flags*** indicate the type of open (such as for reading or writing), and ***modes*** give the file permissions if the file is being created.

# *open ( )* file system Call

- The *open system call is the first step a process must take to access the data in a* file.

- The syntax for the *open system call is*

  ***fd =open(pathname, flags, modes);***

- Here ***pathname*** *is a file name,* ***flags*** *indicate the type of open (such as for reading* or writing), and ***modes*** *give the file permissions if the file is being created.*

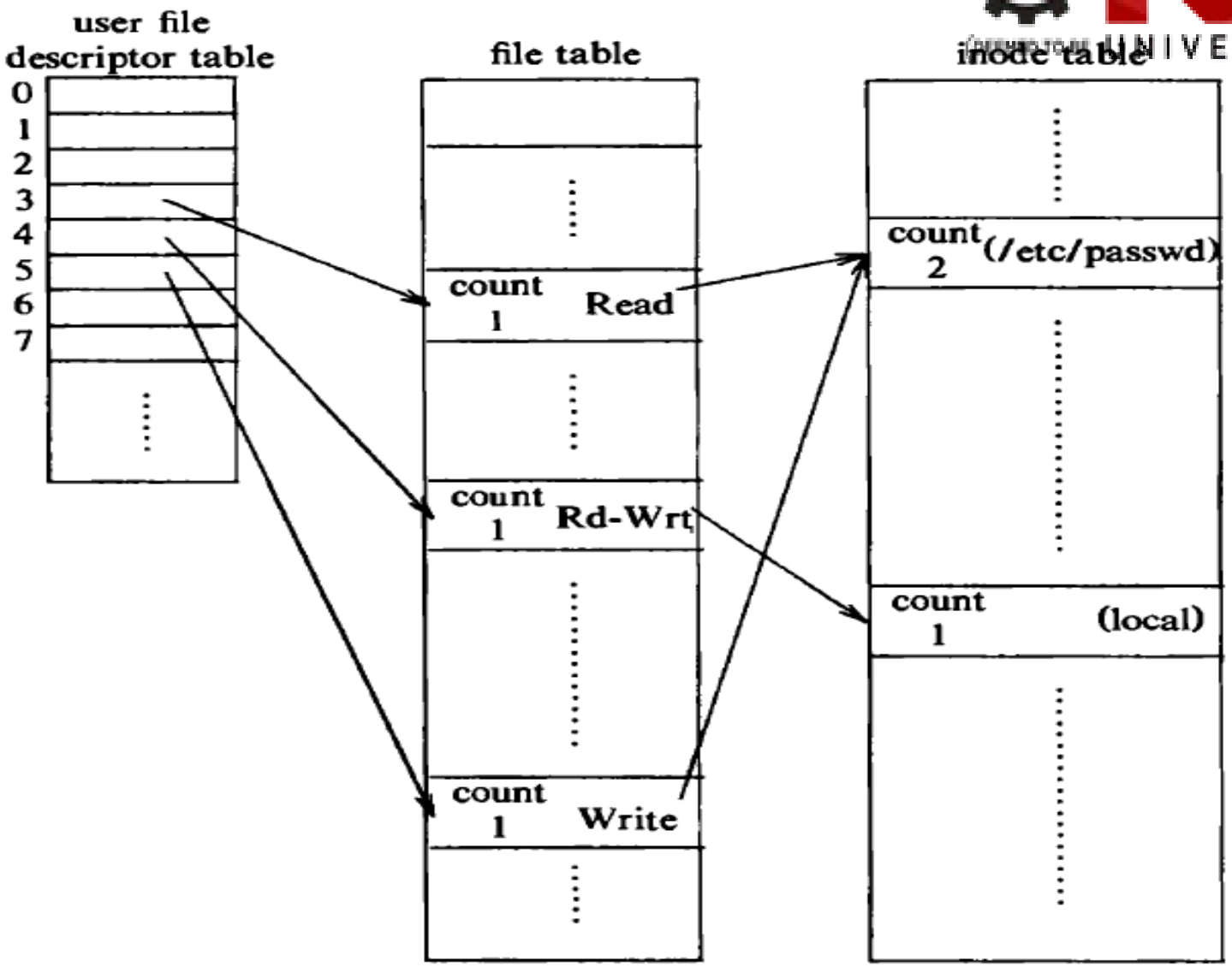
- **Algorithmic Design-open ( ) file system call**

# Algorithmic Design for open ( ) system call

```
algorithm open
inputs:  file name
         type of open
         file permissions (for creation type of open)
output: file descriptor
{
        convert file name to inode (algorithm namei);
        if (file does not exist or not permitted access)
                return(error);
        allocate file table entry for inode, initialize count, offset;
        allocate user file descriptor entry, set pointer to file table entry;
        if (type of open specifies truncate file)
                free all file blocks (algorithm free);
        unlock(inode);                  /* locked above in namei */
        return(user file descriptor);
}
```

- The kernel searches the file system for the file name parameter using algorithm ***namei*** .
- It checks permissions for opening the file after it finds the in-core mode and allocates an entry in the file table for the open file.
- The file table entry contains a pointer to the mode of the open file and a field that indicates the byte offset in the file where the kernel expects the next read or write to begin.
- The kernel initializes the offset to 0 during the open call, meaning that the initial read or write starts at the beginning of a file by default

# Data Structures after O



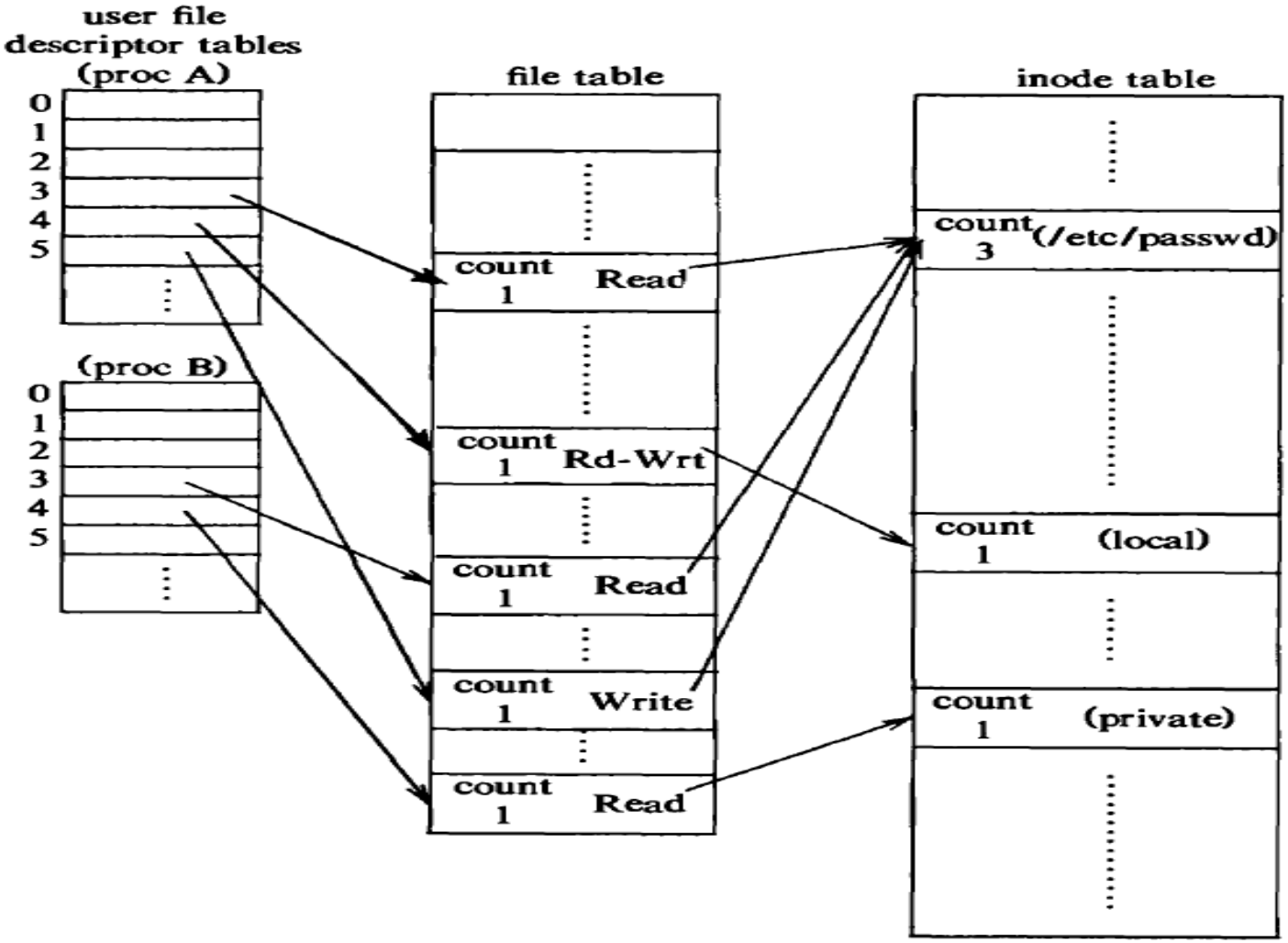
- Suppose a process executes the following code, opening the file letc/passwd“ twice, once read-only and once write-only, and the file "local" once, for reading and writing.

- ***fdl = open(letc/passwd", O_RDONLY);***
- ***fd2 = open("local", 0 RDWR);***
- ***fd3 = open(letc/passwd", O_WRONLY);***

- Here, each open returns a file descriptor to the process, and the corresponding entry in the user file descriptor table points to a unique entry in the kernel file table even though one file ("/etc/passwd") is opened twice.

- The file table entries of all instances of an open file point to one entry in the in-core mode table.

# Data Structures after TWO Processes opens a File



# *read ( )* File System Call

- The syntax of the read system call is

  ***number = read(fd, buffer, count)***

  where

**fd** is the file descriptor returned by open

**buffer** is the address of a data structure in the user process that will contain the read data on successful completion of the call

**count** is the number of bytes the user wants to read, and number is the number of bytes actually read.

- **Algorithmic  Design for read ( ) system call**

# Algorithm for reading a file


```
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);
```

# Contd…..

```
        copy data from system buffer to user address;
        update u area fields 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 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 ( )* system call

- Syntax :

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

where the meaning of the variables *fd, buffer, count, and number are the same as* they are for the *read system call*

# close ( ) system call

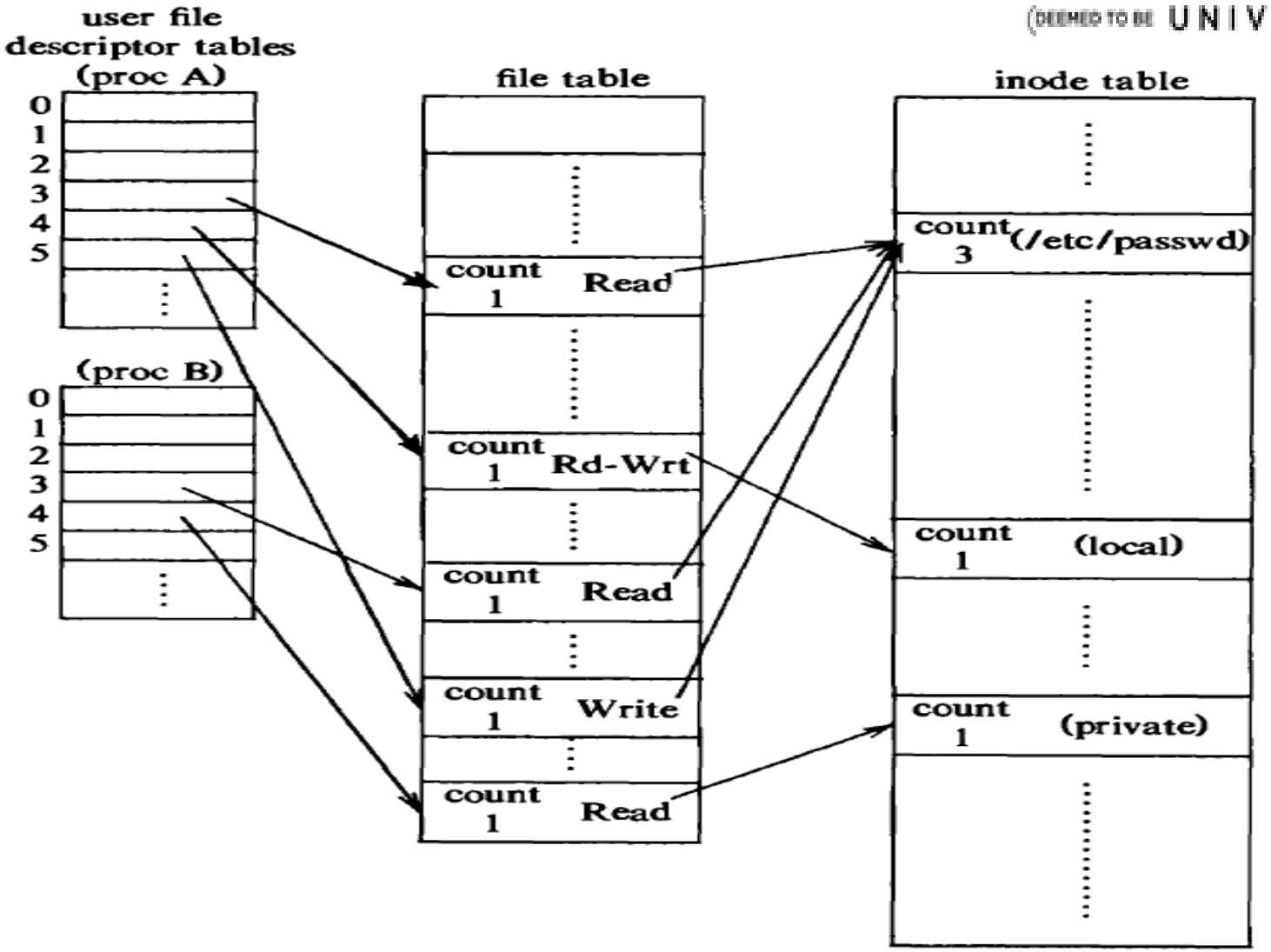
- Syntax:

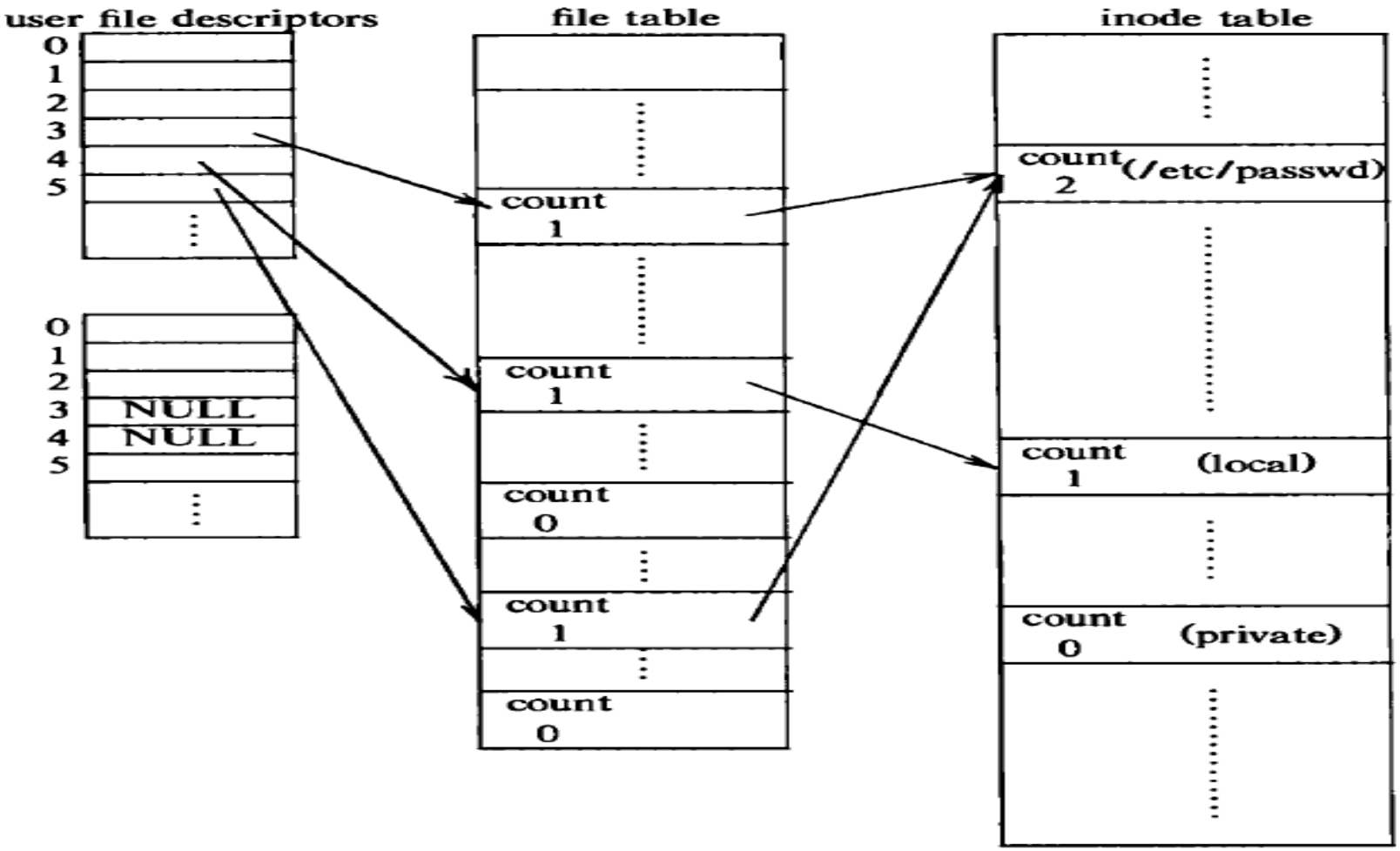
  ***close (fd);***

where *fd* is the file descriptor for the open file.

- The kernel does the *close* operation by manipulating the file descriptor and the corresponding file table and mode table entries.

- When the close system call completes, the user file descriptor table entry is empty.
- Attempts by the process to use that file descriptor result in an error until the file descriptor is reassigned as a result of another system call.
- When a process exits, the kernel examines its active user file descriptors and internally closes each one.
- Hence, no process can keep a file open after it terminates.




user file descriptors
0
1
2
3
4
5
0
1
2
3 NULL
4 NULL
5
file table
count 1
count 1
count 0
count 1
count 0
inode table
count 2 (/etc/passwd)
count 1 (local)
count 0 (private)

- The entries for file descriptors 3 and 4 in the user file descriptor table are empty.
- The count fields of the file table entries are now 0, and the entries are empty.
- The mode reference count for the files "ietc/passwd" and "private" are also decremented.
- The mode entry for "private“ is on the free list because its reference count is 0, but its entry is not empty.

# xv6 Case Study

- *sys_open*
- *filealloc*
- *sys_read, fileread*
- *sys_write, filewrite*
- *sys_close, fileclose*

# *Design and Implementation of sysfile.c*

- Its an xv6 file that consists of functions of various file system calls.
- Its available from Sheet No. 60-65 of xv6 code manual.
- It contains the file related functions like ***sys_open***, ***filealloc***, ***sys_read***, ***fileread***, ***sys_write***, ***filewrite***, ***sys_close***, ***fileclose***

## *sys_open*

- ***sys_open()*** system call opens the specified filename, using the program associated with the corresponding file type of filename.
- The behaviour of this command is the same as that of double clicking on filename in the Windows Explorer.
- For example, if filename is " c:\mydata\ sales.xls " and the .xls extension is associated with Microsoft Excel, sys_open("c:\mydata\ sales.xls") will open Excel and load the file.

# Xv6 Function of sys_open


```
6400 int
6401 sys_open(void)
6402 {
6403   char *path;
6404   int fd, omode;
6405   struct file *f;
6406   struct inode *ip;
6407
6408   if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
6409     return -1;
6410
6411   begin_op();
6412
6413   if(omode & O_CREATE){
6414     ip = create(path, T_FILE, 0, 0);
6415     if(ip == 0){
6416       end_op();
6417       return -1;
6418     }
6419   } else {
6420     if((ip = namei(path)) == 0){
6421       end_op();
6422       return -1;
6423     }
6424     ilock(ip);
6425     if(ip->type == T_DIR && omode != O_RDONLY){
6426       iunlockput(ip);
6427       end_op();
6428       return -1;
6429     }
6430   }
6431
6432   if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
6433     if(f)
6434       fileclose(f);
6435     iunlockput(ip);
6436     end_op();
6437     return -1;
6438   }
6439   iunlock(ip);
6440   end_op();
6441
6442   f->type = FD_INODE;
6443   f->ip = ip;
6444   f->off = 0;
6445   f->readable = !(omode & O_WRONLY);
6446   f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
6447   return fd;
6448 }
```

## *filealloc*

- All the open files in the system are kept in a global file table, the ftable.

- ***filealloc*** is a function present in file table ftable to allocate a file

# Xv6 Function of *filealloc*


```
5874 // Allocate a file structure.
5875 struct file*
5876 filealloc(void)
5877 {
5878   struct file *f;
5879
5880   acquire(&ftable.lock);
5881   for(f = ftable.file; f < ftable.file + NFILE; f++){
5882     if(f->ref == 0){
5883       f->ref = 1;
5884       release(&ftable.lock);
5885       return f;
5886     }
5887   }
5888   release(&ftable.lock);
5889   return 0;
5890 }
```

# *sys_read*



```
6131 int
6132 sys_read(void)
6133 {
6134   struct file *f;
6135   int n;
6136   char *p;
6137
6138   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6139     return -1;
6140   return fileread(f, p, n);
6141 }
```

# *sys_write*


```
6150 int
6151 sys_write(void)
6152 {
6153   struct file *f;
6154   int n;
6155   char *p;
6156
6157   if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
6158     return -1;
6159   return filewrite(f, p, n);
6160 }
```

# *sys_close*


```
6162  int
6163  sys_close(void)
6164  {
6165    int fd;
6166    struct file *f;
6167
6168    if(argfd(0, &fd, &f) < 0)
6169      return -1;
6170    myproc()->ofile[fd] = 0;
6171    fileclose(f);
6172    return 0;
6173  }
```

# Operating Systems Design
# 19CS2106R
# Session-9
# File System Calls:
# *creat, mknod, chdir and stat*

# Session Plan

SESSION NUMBER : 9

**Session Outcome:** 1 Understand and Explore the Design of File System Calls: creat, mknod, chdir, stat

| Time(min) | Topic | BTL | Teaching- Learning Methods | Active Learning Methods |
|---|---|---|---|---|
| 5 | Attendance/Poll/Pop Question | 1 | Talk | --- NOT APPLICABLE --- |
| 20 | Algorithms for File System Calls: creat, mknod, chdir, stat | 3 | PPT | --- NOT APPLICABLE --- |
| 5 | Ask for any doubts through Public chat/ Break | 1 | Talk | --- NOT APPLICABLE --- |
| 10 | xv6 functions: create, sys_mknod, sys_mkdir, sys_chdir, sys_fstat, filestat, stat.h | 3 | PPT | Case Study |
| 10 | Design and Implementation of creat, mknod, chdir, stat, file.c, sysfile.c | 3 | LTC | --- NOT APPLICABLE --- |

## Learning Outcomes

At the end of the session, the student will be able to:

- ***Understand the file system calls creat, mknod, chdir and stat.***
- ***Understand above system calls as part of xv6 case study.***
- ***Design and Implementation of the above file.c***

# Creating a NEW File using *creat( )*

- The open system call gives a process access to an existing file, but the ***creat( )*** system call creates a new file in the system.

- The syntax for the ***creat( )*** system call is

  ***fd = creat(pathname, modes);***

Here, the variables ***pathname***, ***modes***, *and* ***fd*** *mean the same as they do in the open system call.*

- If no such file previously existed, the kernel creates a new file with the specified name and permission modes;

- If the file already existed, the kernel truncates the file (releases all existing data blocks and sets the file size to 0) subject to suitable file access permissions.

# Algorithmic Design of *creat*


```
algorithm creat
input:   file name
         permission settings
output: file descriptor
{
        get inode for file name (algorithm namei);
        if (file already exists)
        {
                if (not permitted access)
                {
                        release inode (algorithm iput);
                        return(error);
                }
        }
        else        /* file does not exist yet */
        {
                assign free inode from file system (algorithm ialloc);
                create new directory entry in parent directory: include
                        new file name and newly assigned inode number;
        }
        allocate file table entry for inode, initialize count;
        if (file did exist at time of create)
                free all file blocks (algorithm free);
        unlock(inode);
        return(user file descriptor);
}
```

# Creating Special Files using *mknod( )*

- The system call ***mknod ( )*** creates special files in the system, including named pipes, device files, and directories.
- It is similar to ***creat*** in that the kernel allocates an inode for the file.
- The syntax of the ***mknod ( )***system call is

  ***mknod(pathname, type and permissions, dev)***

  where ***pathname*** is the name of the node to be created, ***type and permissions*** give the node type (directory, for example) and access permissions for the new file to be created, and ***dev*** specifies the major and minor device numbers for block and character special files

# Algorithmic Design of *mkno*


```
algorithm make new node
inputs:  node (file name)
         file type
         permissions
         major, minor device number (for block, character special files)
output: none
{
        if (new node not named pipe and user not super user)
                return(error);
        get inode of parent of new node (algorithm namei);
        if (new node already exists)
        {
                release parent inode (algorithm iput);
                return(error);
        }
        assign free inode from file system for new node (algorithm ialloc);
        create new directory entry in parent directory: include new node
                name and newly assigned inode number;
        release parent directory inode (algorithm iput);
        if (new node is block or character special file)
                write major, minor numbers into inode structure;
        release new node inode (algorithm iput);
}
```

## Changing current dierectory using *chdir()*

- When the system is first booted, process 0 makes the file system root its current directory during initialization.
- It executes the algorithm iget on the root mode, saves it in the u area as its current directory, and releases the mode lock.
- When a new process is created via the fork system call, the new process inherits the current directory of the old process in its u area, and the kernel increments the mode reference count accordingly.

- The syntax for the *chdir ( ) system call is*

  ***chdir (pathname) ;***

  where *pathname is the directory that becomes the new current directory of the* process.

# Algorithmic Design of *chdir*


```
algorithm change directory
input:   new directory name
output: none
{
        get inode for new directory name (algorithm namei);
        if (inode not that of directory or process not permitted access to file)
        {
                release inode (algorithm iput);
                return(error);
        }
        unlock inode;
        release "old" current directory inode (algorithm iput);
        place new inode into current directory slot in u area;
}
```

## *stat & fstat*

- The system calls ***stat*** and ***fstat*** allow processes to query the status of files and returning information such as the file type, file owner, access permissions, file size, number of links, mode number, and file access times

# Syntax of *stat ( ) & fstat (*

- ***stat (pathname, statbuffer) ;***
- ***fstat (fd, statbuffer);***

where pathname is a file name, fd is a file descriptor returned by a previous open call, and ***statbuffer*** is the address of a data structure in the user process that will contain the status information of the file on completion of the call.

- The system calls simply write the fields of the mode into ***statbuffer*.**

# Implementation of *stat ( ) & fstat ( )*


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

main(argc, argv)
        int argc;
        char *argv[];
{
        int fd;
        char buf[1024];
        struct stat statbuf;

        if (argc != 2)                  /* need a parameter */
                exit();
        fd = open(argv[1], O_RDONLY);
        if (fd == -1)                   /* open fails */
                exit();
        if (unlink(argv[1]) == -1)          /* unlink file just opened */
                exit();
        if (stat(argv[1], &statbuf) == -1)          /* stat the file by name*/
                printf("stat %s fails as it should\n", argv[1]);
        else
                printf("stat %s succeeded!!!!\n", argv[1]);
        if (fstat(fd, &statbuf) == -1)                  /* stat the file by fd */
                printf("fstat %s fails!!!\n", argv[1]);
        else
                printf("fstat %s succeeds as it should\n", argv[1]);
        while (read(fd, buf, sizeof(buf)) > 0)      /* read open/unlinked file */
                printf("%1024s", buf);          /* prints 1K byte field */
}
```

- Case Study of xv6 Functions: ***creat, sys_mknod, sys_mkdir, sys_chdir and sys_fstat.***

# *creat* in xv6

- The function create (6357)creates a new name for a new inode.

- It is a generalization of the three file creation system calls: open with the O_CREATE flag makes a new ordinary file, ***mkdir*** makes a new directory, and ***mkdev*** makes a new device file.

- Using create, it is easy to implement **s*ys_open***, ***sys_mkdir,*** and ***sys_mknod***.

# *chdir* and *sys_chdir* in xv6

- In xv6, The directories form a tree, starting at a special directory called the ***root***.

- A path like /a/b/c refers to the file or directory named c inside the directory named b inside the directory named a in the root directory /.

- Paths that don't begin with / are evaluated relative to the calling process's current directory, which can be changed with the ***chdir*** system call.

# *chdir in xv6*



```
chdir("/a");
chdir("b");
open("c", O_RDONLY);

open("/a/b/c", O_RDONLY);
```

# *mkdir, mknod* and *sys_mkdir & sys_mknod* in xv6


- There are multiple system calls to create a new file or directory:

- ***mkdir*** creates a new directory, open with the O_CREATE flag creates a new data file and
- ***mknod*** creates a new device file.

```
mkdir("/dir");
fd = open("/dir/file", O_CREATE|O_WRONLY);
close(fd);
mknod("/console", 1, 1);
```

- Here, ***mknod*** creates a file in the file system, but the file has no contents.

- Instead, the file's metadata marks it as a device file and records the major and minor device numbers (the two arguments to ***mknod***), which uniquely identify a kernel device.

- When a process later opens the file, the kernel diverts read and write system calls to the kernel device implementation instead of passing them to the file system.

# *fstat* in xv6

- ***fstat*** retrieves information about the object a file descriptor refers to.
- In xv6, it will be handled as follows:

```
#define T_DIR 1 // Directory
#define T_FILE 2 // File
#define T_DEV 3 // Device
struct   stat {
short  type; // Type of file
Int  dev; // File system's disk device
uint  ino; // Inode number
short  nlink; // Number of links to file
uint  size; // Size of file in bytes
};
```

# Design and implementation of *file.c*

- ***https://github.com/mit-pdos/xv6-public/blob/master/file.c***

***file.c*** is a github repository on file related activities and operations.

- Consists of 157 lines of xv6 code.

***file.c*** consists of xv6 code implementations of

a) ***File Descriptors***

b) ***Allocation of File Structures***

c) ***Incrementing the ref.count for file f***

d) ***Close operation of file f***

e) ***Getting metadata about file f***

f) ***Reading from file f***

g) ***Writing to file f***

# Closing a file f


```
55  // Close file f.  (Decrement ref count, close when reaches 0.)
56  void
57  fileclose(struct file *f)
58  {
59    struct file ff;
60
61    acquire(&ftable.lock);
62    if(f->ref < 1)
63      panic("fileclose");
64    if(--f->ref > 0){
65      release(&ftable.lock);
66      return;
67    }
68    ff = *f;
69    f->ref = 0;
70    f->type = FD_NONE;
71    release(&ftable.lock);
72
73    if(ff.type == FD_PIPE)
74      pipeclose(ff.pipe, ff.writable);
75    else if(ff.type == FD_INODE){
76      begin_op();
77      iput(ff.ip);
78      end_op();
79    }
80  }
```

# Reading from a file

```
 95  // Read from file f.
 96  int
 97  fileread(struct file *f, char *addr, int n)
 98  {
 99    int r;
100
101    if(f->readable == 0)
102      return -1;
103    if(f->type == FD_PIPE)
104      return piperead(f->pipe, addr, n);
105    if(f->type == FD_INODE){
106      ilock(f->ip);
107      if((r = readi(f->ip, addr, f->off, n)) > 0)
108        f->off += r;
109      iunlock(f->ip);
110      return r;
111    }
112    panic("fileread");
113  }
```

# Writing to file f

```
116   // Write to file f.
117   int
118   filewrite(struct file *f, char *addr, int n)
119   {
120     int r;
121
122     if(f->writable == 0)
123       return -1;
124     if(f->type == FD_PIPE)
125       return pipewrite(f->pipe, addr, n);
126     if(f->type == FD_INODE){
127       // write a few blocks at a time to avoid exceeding
128       // the maximum log transaction size, including
129       // i-node, indirect block, allocation blocks,
130       // and 2 blocks of slop for non-aligned writes.
131       // this really belongs lower down, since writei()
132       // might be writing a device like the console.
133       int max = ((MAXOPBLOCKS-1-1-2) / 2) * 512;
134       int i = 0;
135       while(i < n){
136         int n1 = n - i;
137         if(n1 > max)
138           n1 = max;
139
140         begin_op();
141         ilock(f->ip);
142         if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
143           f->off += r;
144         iunlock(f->ip);
145         end_op();
146
147         if(r < 0)
148           break;
149         if(r != n1)
150           panic("short filewrite");
151         i += r;
152       }
153       return i == n ? n : -1;
154     }
155     panic("filewrite");
156   }
157
```

# Operating Systems Design
# 19CS2106R
# Session-10

## File System Calls:
## *pipe, dup,link and unlink*

# Session Plan

(DEEMED TO BE UNIVERSITY)

**SESSION NUMBER** : 10

**Session Outcome: 1** Understand and Explore the Design of File System Calls: Pipe, dup, link, unlink

| Time(min) | Topic | BTL | Teaching- Learning Methods | Active Learning Methods |
|---|---|---|---|---|
| 5 | Attendance/Poll/Pop Question | 1 | Talk | --- NOT APPLICABLE --- |
| 20 | Algorithms for File System Calls : Pipe, dup, link, unlink | 2 | PPT | --- NOT APPLICABLE --- |
| 5 | Ask for any doubts through Public chat/ Break | 1 | Talk | --- NOT APPLICABLE --- |
| 10 | Xv6: sys_pipe, pipealloc, piperead, pipewrite,sys_dup, filedup, sys_link, dirlink, sys_unlink | 3 | PPT | Case Study |
| 10 | Design and Implementation of Pipe, dup, link, unlink, sysfile.c, pipe.c | 3 | LTC | --- NOT APPLICABLE --- |

## Learning Outcomes

- At the end of the session, the student will

***Understand the system calls pipe ( ), dup ( ), link ( ) and unlink ( ).***

- ***Understand the xv6 case study of the above system calls.***

- ***To explore the design and implementation of the above in sysfile.c***

## Concept of Pipes

- Pipes are the IPC mechanism used for communicating between two or more process in Linux i.e to pipe the data between TWO processes.
- It is a form of redirection (transfer of standard output to some other destination).
- Their implementation allows processes to communicate even though they do not know what processes are on the other end of the pipe.

- Two types of pipes:

**Unnamed pipes** and **Named pipes**

- Pipe is used to combine two or more commands, and in this, the output of one command acts as input to another command, and this command's output may act as input to the next command and so on.
- Piping mechanism is denoted by the '|'character.
- The command line programs that do the further processing are referred to as ***filters***.

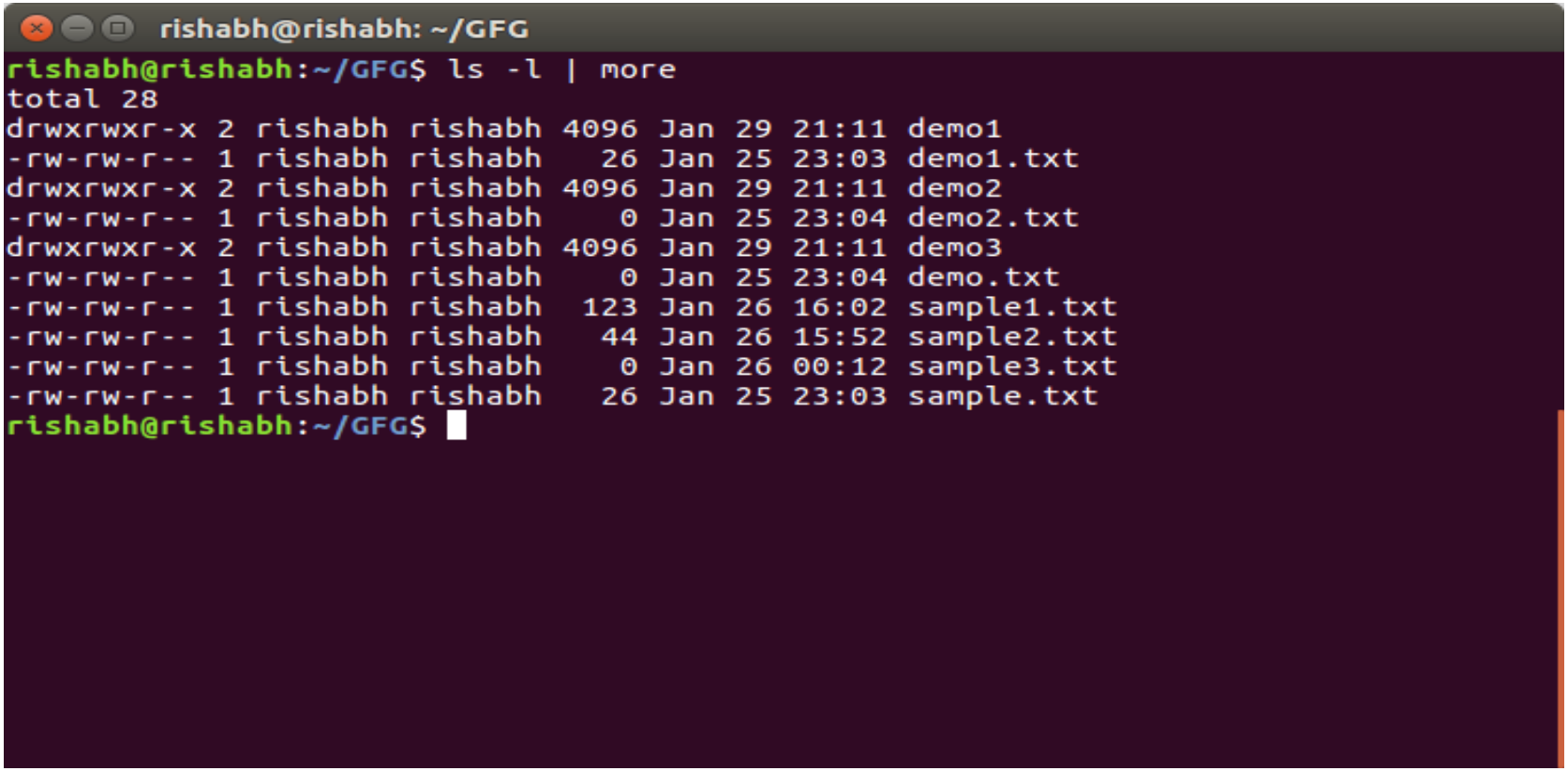
# General Syntax:

- Command_1 | command_2 | command_3 | .... | command_N
- Eg : **Listing all files and directories and give it as input to more command.**

  **$** ls -l | more .

Here two commands namely, ***ls-l*** and more are piped. ***ls*** command lists and ***more*** command displays the content one page at a time.

The ***more*** command takes the output of ***$ ls -l*** as input. The net effect of this command is that the output of ls -l is displayed one screen at a time. The pipe acts as a container which takes the output of ls -l and gives it to more as input.



## Unnamed Pipes

- 1) These are created by the shell automatically.
  2) They exists in the kernel.
  3) They can not be accesses by any process, including the process that creates it.
  4) They are opened at the time of creation only.
  5) They are unidirectional.

# Named Pipes( also called FIFO, First In First Out)

- They have “names” and exist as special files within a file system.
- They are created programmatically using the command ***mkfifo*** or using ***mknod()*** system call.
- They exist until they are removed with ***rm*** or ***unlink()*** **.**
- ***Normally used to communicate between TWO totally unrelated processes.***
- ***These are bi-directional.***

# Syntax of *pipe ( )* system c

- The syntax for creation of a pipe is

  ***pipe (fdptr) ;***

  where ***fdptr*** is the pointer to an integer array that will contain the two file descriptors for reading and writing the pipe.

# Algorithmic Design Approach of Creation of *pipe( )* system call-unnamed

```
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;
}
```

# *dup ( )* System call

- 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.
- The syntax of the system call is

  ***newfd = dup (fd);***

  where fd is the file descriptor being duped and newfd is the new file descriptor that references the file.

  Because dup duplicates the file descriptor, it increments the count of the corresponding file table entry, which now has one more file descriptor entry that points to it.

# *dup ( )* System call-C Proce

```
#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));
}
```

# Creating new link to an existing file using *link( )* system call

- The link system call links a file to a new name in the file system directory structure, creating a new directory entry for an existing mode.
- The syntax for the link system call is

  ***link(source file name, target file name);***

  where source file name is the name of an existing file and target .file name is the new (additional) name the file will have after completion of the link call

# Algorithmic Design for link () system call

- **Algorithm link**
- **input: existing file name**
- **new file name**
- **output: none**
- **{**
- **get inode for existing file name (algorithm namei);**
- **if (too many links on file or linking directory without super user permission)**
- **{**
- **release inode (algorithm iput);**
- **return (error);**
- **}**
- **increment link count on inode;**
- **update disk copy of inode;**
- **unlock inode;**
- **get parent inode for directory to contain new file name (algorithm namei);**
- **if (new file name already exists or existing file, new file on**
- **different file systems)**
- **{**
- **undo update done above;**
- **return (error); }**

# Contnd….

- Create new directory entry in parent directory of new file name:

  include new file name, inode number of existing file name;

- Release parent directory inode (algorithm iput);

release inode of existing file (algorithm iput);

}

# Removing a directory entry for a file using Unlink ( ) system call

- The unlink system call removes a directory entry for a file.

- The syntax for the unlink call is

  ***unlink (pathname) ;***

  where pathname identifies the name of the file to be unlinked from the directory hierarchy.

- If a process unlinks a given file, no file is accessible by that name until another directory entry with that name is created.

# Algorithmic Design of *unlink*


```
algorithm unlink
input:   file name
output: none
{
        get parent inode of file to be unlinked (algorithm namei);
        /* if unlinking the current directory... */
        if (last component of file name is ".")
                increment inode reference count;
        else
                get inode of file to be unlinked (algorithm iget);
        if (file is directory but user is not super user)
        {
                release inodes (algorithm iput);
                return(error);
        }
        if (shared text file and link count currently 1)
                remove from region table;
        write parent directory:  zero inode number of unlinked file;
        release inode parent directory (algorithm iput);
        decrement file link count;
        release file inode (algorithm iput);
                /* iput checks if link count is 0:  if so,
                 * releases file blocks (algorithm free) and
                 * frees inode (algorithm ifree);
                 */
}
```

- Xv6 Case Study : ***sys_pipe, pipealloc, piperead, pipewrite, sys_dup, filedup, sys_link and sys_unlink.***

# Function of pipe ( ) system call using xv6

```
6551 sys_pipe(void)
6552 {
6553   int *fd;
6554   struct file *rf, *wf;
6555   int fd0, fd1;
6556
6557   if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
6558     return -1;
6559   if(pipealloc(&rf, &wf) < 0)
6560     return -1;
6561   fd0 = -1;
6562   if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
6563     if(fd0 >= 0)
6564       myproc()->ofile[fd0] = 0;
6565     fileclose(rf);
6566     fileclose(wf);
6567     return -1;
6568   }
6569   fd[0] = fd0;
6570   fd[1] = fd1;
6571   return 0;
```

# Function of link ( ) using


```
6202 sys_link(void)
6203 {
6204   char name[DIRSIZ], *new, *old;
6205   struct inode *dp, *ip;
6206
6207   if(argstr(0, &old) < 0 || argstr(1, &new) < 0)
6208     return -1;
6209
6210   begin_op();
6211   if((ip = namei(old)) == 0){
6212     end_op();
6213     return -1;
6214   }
6215
6216   ilock(ip);
6217   if(ip->type == T_DIR){
6218     iunlockput(ip);
6219     end_op();
6220     return -1;
6221   }
6222
6223   ip->nlink++;
6224   iupdate(ip);
6225   iunlock(ip);
6226
6227   if((dp = nameiparent(new, name)) == 0)
6228     goto bad;
6229   ilock(dp);
6230   if(dp->dev != ip->dev || dirlink(dp, name, ip->inum) < 0){
6231     iunlockput(dp);
6232     goto bad;
6233   }
6234   iunlockput(dp);
6235   iput(ip);
6236
6237   end_op();
6238
6239   return 0;
6240
6241 bad:
6242   ilock(ip);
6243   ip->nlink--;
6244   iupdate(ip);
6245   iunlockput(ip);
6246   end_op();
6247   return -1;
6248 }
```

# Function of of *unlink( )* in xv6


```
6301 sys_unlink(void)
6302 {
6303   struct inode *ip, *dp;
6304   struct dirent de;
6305   char name[DIRSIZ], *path;
6306   uint off;
6307
6308   if(argstr(0, &path) < 0)
6309     return -1;
6310
6311   begin_op();
6312   if((dp = nameiparent(path, name)) == 0){
6313     end_op();
6314     return -1;
6315   }
6316
6317   ilock(dp);
6318
6319   // Cannot unlink "." or "..".
6320   if(namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
6321     goto bad;
6322
6323   if((ip = dirlookup(dp, name, &off)) == 0)
6324     goto bad;
6325   ilock(ip);
6326
6327   if(ip->nlink < 1)
6328     panic("unlink: nlink < 1");
6329   if(ip->type == T_DIR && !isdirempty(ip)){
6330     iunlockput(ip);
6331     goto bad;
6332   }
6333
6334   memset(&de, 0, sizeof(de));
6335   if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
6336     panic("unlink: writei");
6337   if(ip->type == T_DIR){
6338     dp->nlink--;
6339     iupdate(dp);
6340   }
6341   iunlockput(dp);
6342
6343   ip->nlink--;
6344   iupdate(ip);
6345   iunlockput(ip);
6346
6347   end_op();
6348
6349   return 0;
```

```
6350 bad:
6351    iunlockput(dp);
6352    end_op();
6353    return -1;
6354 }
6355
```

# Function of *dup ( )* using system call u

```
6118 sys_dup(void)
6119 {
6120   struct file *f;
6121   int fd;
6122
6123   if(argfd(0, 0, &f) < 0)
6124     return -1;
6125   if((fd=fdalloc(f)) < 0)
6126     return -1;
6127   filedup(f);
6128   return fd;
6129 }
```

- Exploring the Design and Implementation of ***sysfile.c***

***https://github.com/mit-pdos/xv6-public/blob/master/sysfile.c***

***sysfile.c*** is a github repository on various file system calls.

- Consists of 444 lines of xv6 code.

# *sysfile.c* file consists of xv6 implementations of

- *fdalloc( )---for file descriptor allocation*
- *Sys_dup---dup system call*
- *Sys_open—open system call*
- *Sys_read---read system call*
- *Sys_write---write system call*
- *Sys_close---close system call*
- *Sys_fstat----fstat system call*
- *Sys_link----link system call*
- *Sys_unlink---unlink system call*
- *Sys_mknod---mknod system call*
- *Sys_chdir---Chdir System call*
- *Sys_pipe-----Pipe System Call*

# *sys-pipe*


```
422  int
423  sys_pipe(void)
424  {
425    int *fd;
426    struct file *rf, *wf;
427    int fd0, fd1;
428
429    if(argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0)
430      return -1;
431    if(pipealloc(&rf, &wf) < 0)
432      return -1;
433    fd0 = -1;
434    if((fd0 = fdalloc(rf)) < 0 || (fd1 = fdalloc(wf)) < 0){
435      if(fd0 >= 0)
436        myproc()->ofile[fd0] = 0;
437      fileclose(rf);
438      fileclose(wf);
439      return -1;
440    }
441    fd[0] = fd0;
442    fd[1] = fd1;
443    return 0;
444  }
```

# sys_chdir


```
371  int
372  sys_chdir(void)
373  {
374    char *path;
375    struct inode *ip;
376    struct proc *curproc = myproc();
377
378    begin_op();
379    if(argstr(0, &path) < 0 || (ip = namei(path)) == 0){
380      end_op();
381      return -1;
382    }
383    ilock(ip);
384    if(ip->type != T_DIR){
385      iunlockput(ip);
386      end_op();
387      return -1;
388    }
389    iunlock(ip);
390    iput(curproc->cwd);
391    end_op();
392    curproc->cwd = ip;
393    return 0;
394  }
```

# Sys_open


```
285     int
286     sys_open(void)
287     {
288        char *path;
289        int fd, omode;
290        struct file *f;
291        struct inode *ip;
292
293        if(argstr(0, &path) < 0 || argint(1, &omode) < 0)
294           return -1;
295
296        begin_op();
297
298        if(omode & O_CREATE){
299           ip = create(path, T_FILE, 0, 0);
300           if(ip == 0){
301              end_op();
302              return -1;
303           }
304        } else {
305           if((ip = namei(path)) == 0){
306              end_op();
307              return -1;
308           }
309           ilock(ip);
310           if(ip->type == T_DIR && omode != O_RDONLY){
311              iunlockput(ip);
312              end_op();
313              return -1;
314           }
315        }
316
317        if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
318           if(f)
319              fileclose(f);
320           iunlockput(ip);
321           end_op();
322           return -1;
323        }
324        iunlock(ip);
325        end_op();
326
327        f->type = FD_INODE;
328        f->ip = ip;
329        f->off = 0;
330        f->readable = !(omode & O_WRONLY);
331        f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
332        return fd;
333     }
```

- **Conduction of ALMs in Breakout Session**

- ***Understand the internal algorithms behind the design of various xv6 file system calls.***
- ***Perform the given tests related to file system.***
- ***Customize the usertests.c given in xv6 source code base and execute.***
- ***Submit the output of all the tests***

# ALM-1



```
1.  // does the error path in open() for attempt to write a directory call iput() in a tra...tion
    void openiputtest(void)
2.  // simple file system tests
    void opentest(void)
3.  //small file test
    void writetest(void)
4.  //big files test
    void writetest1(void)
5.  //many creates, followed by unlink test
    void createtest(void)
6.  //mkdir test
    void dirtest(void)
7.  // simple fork and pipe read/write
    void pipe1(void)
8.  // four processes write different files at the same
    // time, to test block allocation.
    void fourfiles(void)
9.  // More file system tests
    // two processes write to the same file descriptor
    // is the offset shared? does inode locking work?
    void sharedfd(void)
10. // four processes create and delete different files in same directory
    void createdelete(void)
11. // can I unlink a file and still read it?
    void unlinkread(void)
12. //linktest
    void linktest(void)
13. // test concurrent create/link/unlink of the same file
    void concreate(void)
14. //subdir test
    void subdir(void)
```

# ALM-2

- Assume that a process A executes the following three function calls:

  fdl = open(“/etc/passwd”, O_RDONLY);

  fd2 = open(“local”, O_RDWR);

  fd3 = open(“/etc/passwd”, O_WRONLY);

  For process A, show the relationship between the inode table, file table, and user file descriptor data structures.

  Suppose a second process B executes the following code.

  fdl — open("/etc/passwd", 0_RDONLY);

  fd2 — open("private", 0_RDONLY);

***Draw the resulting picture that shows the relationship between the appropriate data structures while both processes (and no others) have the files open.***

# ALM-3

- The following sequence of code has been observed in various programs:

```
dup2(fd, 0);
dup2(fd, 1);
dup2(fd, 2);
if (fd > 2)
close(fd);
```

***To see why the if test is needed, assume that fd is 1 and draw a picture of what happens to the three descriptor entries and the corresponding file table entry with each call to dup2. Then assume that fd is 3 and draw the same picture.***

# ALM-4

- ***Write a program that create two pipes, send filename from command line to child process. In child read that file and send it back using pipe. Parent process should print the file. if error occur in child process error must be send to parent process***