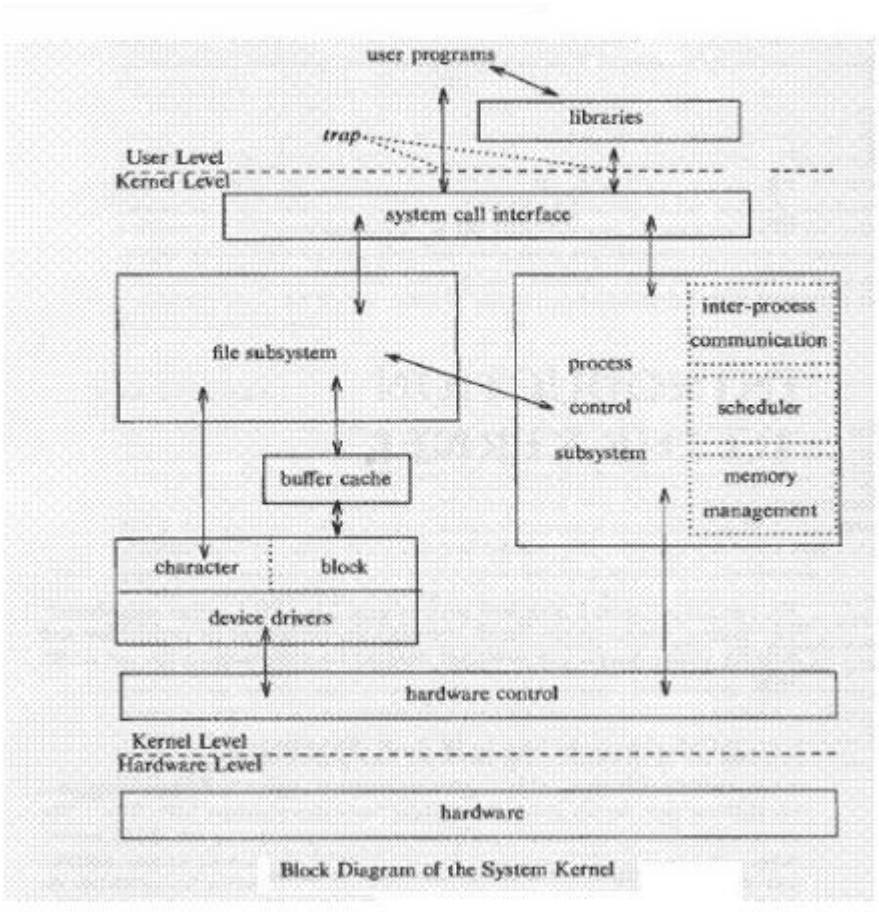
Unit 01

1) Draw and Explain the Block diagram of the UNIX kernel.

Introduction to kernel



Block Diagram of the System Kernel

The UNIX kernel is the core component of the UNIX operating system. It manages system resources, provides an interface for user programs, and handles hardware interactions. Here's a high-level explanation of the block diagram:

User Level:

At the top level, we have user programs (applications) on the left and libraries on the right.

These user-level components interact with the kernel through a system call interface (represented by dashed lines).

System calls allow user programs to request services from the kernel (e.g., file operations, process management).

Kernel Level:

The middle section represents the kernel itself.

Key components include:

File Subsystem: Responsible for managing files and directories.

Buffer Cache: Caches frequently accessed data from storage devices.

Process Control Subsystem: Manages processes (execution units) in the system.

Inter-Process Communication (IPC): Allows processes to communicate with each other.

Scheduler: Determines which process runs when (scheduling algorithms).

Memory Management: Allocates and manages memory for processes.

Kernel Level Hardware Level:

At the bottom, we have device drivers:

Character Device Drivers: Handle character-oriented devices (e.g., keyboards, terminals).

Block Device Drivers: Manage block-oriented devices (e.g., hard drives, SSDs).

These drivers interface with the actual hardware (represented by the “Hardware” box).

In summary, the UNIX kernel bridges the gap between user-level applications and hardware, providing essential services and managing system resources.

Explain with an example the Building Block Primitives.

Unix empowers users to develop small, modular programs that serve as fundamental building blocks for constructing more intricate programs. Two key primitives enhance the flexibility and functionality of these programs:

1. I/O Redirection:

• Unix commands often involve reading input and producing output. I/O redirection allows users to manipulate input and output sources, providing greater control over command execution.

• Standard Files:

• Unix defines three standard files:

• Standard Input (stdin)

• Standard Output (stdout)

• Standard Error (stderr)

• The terminal (monitor) typically serves as these three files, treating devices as files.

• I/O Redirection Examples:

• Output Redirection:

• ls: Lists files in the current directory.

• ls > output: Redirects the output of the ls command to a file named "output" instead of displaying it on the terminal.

• Input Redirection:

cat < test1: Takes each line from the file "test1" as input for the cat command, displaying it on the monitor.

Error Redirection:

• When an invalid command is entered, the shell displays an error message on the monitor, treating it as the standard error

file.

2. Pipes (|):

Pipes facilitate the flow of data between processes, allowing the output of one

command to serve as the input for another. This enables the creation of powerful and flexible command pipelines.

• Pipe Example:

• ls -l | wc -l: Counts and displays the total number of lines in the current directory by piping the output of ls -l (list files) to wc -l (word count).

• grep -n 'and' test: Searches for the word 'and' in the file "test" and displays the line numbers where it occurs.

• Functionality:

• A pipe consists of a reader process and a writer process.

• The output of the writer process becomes the input of the reader process.

2) Explain an algorithm for Buffer Allocation.

- To allocate a buffer for a disk block
  - Use 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;
 }
}}
```


The algorithm depicted in the image is titled "To use a buffer for a disk block." It outlines the steps for allocating a buffer to a disk block within a file system. Let's break down the key components of this algorithm:

1. **Input**:
   - The algorithm takes two inputs:
     - **File system number**: Identifies the specific file system.
     - **Block number**: Specifies the disk block that needs a buffer.
2. **Output**:
   - The desired outcome is a **locked buffer** associated with the specified disk block.
3. **Buffer Allocation Process**:
   - The algorithm proceeds as follows:

a. **Check Block Availability**:

- Verify that no other process is currently using the specified disk block.
- If the block is available, proceed to the next step.
- If the block is in use (e.g., locked by another process), wait for an event (such as the block becoming free).

b. **Allocate Buffer**:

- Mark the buffer as busy (indicating that it is in use).
- Remove the buffer from the free list (if it was previously available).

c. **Delayed Write Handling**:

- If the buffer was marked for delayed write (i.e., contains data that needs to be written back to disk), perform an asynchronous write operation to the disk.
- Continue to the next step.

d. **Update Hash Queue**:

- Move the buffer from its old hash queue (if any) to a new hash queue.
- The hash queue organizes buffers based on their associated disk blocks.
- Return a pointer to the buffer in the new hash queue.

4. **Use Cases**:
   - This algorithm is relevant in file systems where buffers are used to cache data read from or written to disk.
   - It ensures efficient buffer allocation and synchronization to prevent data corruption or contention.

In summary, the algorithm ensures that a buffer is appropriately allocated for a specific disk block, considering availability, delayed writes, and hash queue management.

3) Explain the advantages & disadvantages of buffer cache

Advantages:

1. Uniform Disk Access:

• The buffer cache allows for uniform disk access, meaning that data can be read from or written to disk through a standardized interface. This simplifies and standardizes disk interactions.

2. Eliminates Need for Special Alignment:

• The buffer cache eliminates the need for special alignment of user buffers. Data is copied from user buffers to system buffers, ensuring that alignment details are handled at the system level.

3. Reduces Disk Traffic:

• The use of a buffer cache reduces the overall amount of disk traffic. By caching frequently accessed data in memory, it minimizes the need for repeated disk accesses, leading to improved performance.

4. Enhances File System Integrity:

• The buffer cache helps ensure file system integrity. Each disk block is stored in only one buffer at a time, preventing inconsistencies that could arise from multiple buffers holding different versions of the same block.

Disadvantages:

1. Vulnerability to Crashes:

• The buffer cache can be vulnerable to crashes. If data is marked for delayed write but hasn't been written to disk, a crash may lead to data loss or inconsistency.

2. Extra Data Copy in Delayed Write:

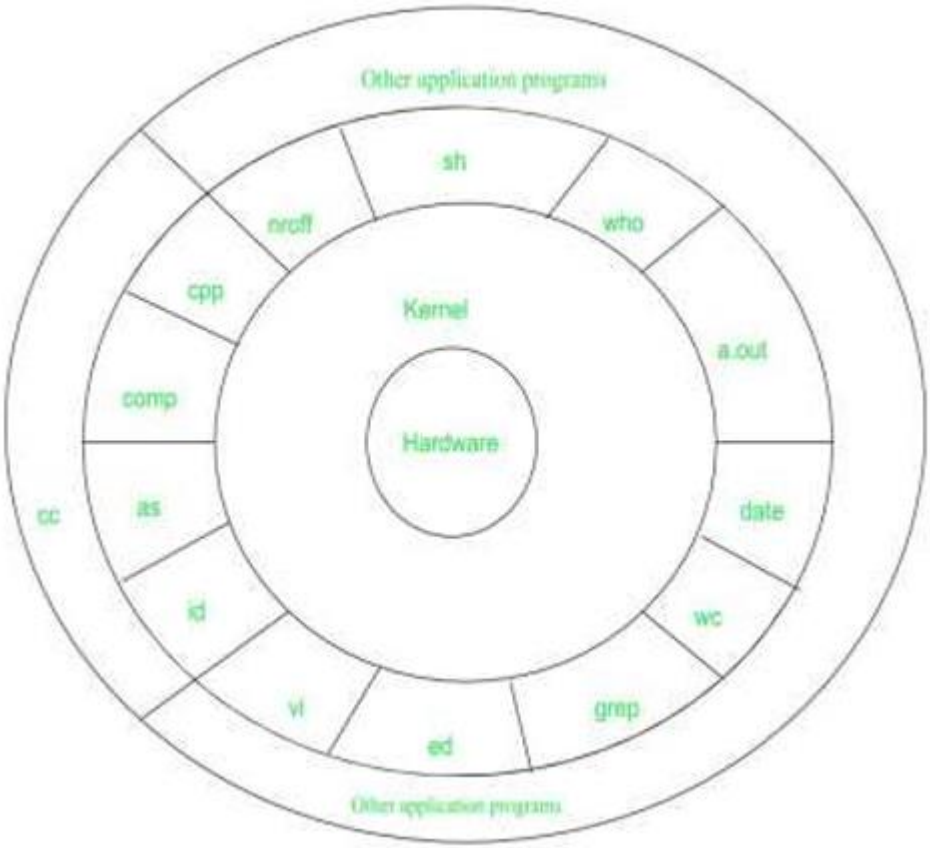
• In the case of delayed write operations, an extra data copy may be required. This additional step can introduce overhead and impact performance.

3. Reading and Writing to/from User Processes:

• Reading and writing to/from user processes can be less efficient. Copying data between user and system buffers introduces additional overhead, impacting the speed of data transfer.

4) Explain the architecture of the UNIX System.

Answer –



The UNIX operating system architecture is organized into layers, each serving a
specific purpose and contributing to the overall functionality. Let's delve into the
architecture of the UNIX system
UNIX is a family of multitasking, multiuser computer operating systems developed in the mid 1960s at Bell Labs. It was originally developed for mini
computers and has since been ported to various hardware platforms. UNIX has
a reputation for stability, security, and scalability, making it a popular choice for
enterprise-level computing.

The basic design philosophy of UNIX is to provide simple, powerful tools that can
be combined to perform complex tasks. It features a command-line interface
that allows users to interact with the system through a series of commands,
rather than through a graphical user interface (GUI).
Some of the key features of UNIX include:
1. Multiuser support: UNIX allows multiple users to simultaneously access
the same system and share resources.
2. Multitasking: UNIX is capable of running multiple processes at the same
t
ime.
3. Shell scripting: UNIX provides a powerful scripting language that allows
users to automate tasks.
4. Security: UNIX has a robust security model that includes file permissions,
user accounts, and network security features.
5. Portability: UNIX can run on a wide variety of hardware platforms, from
small embedded systems to large mainframe computers.
Hardware Layer (Layer-1):
• The lowest layer in the UNIX system architecture is the hardware layer, which encompasses all hardware-related components and information.
• It includes the physical components of the computer, such as the processor, memory, storage devices, and input/output devices.
• This layer is not considered part of the UNIX operating system but provides the underlying infrastructure on which the system operates.
2. Kernel (Layer-2):
• The kernel, also known as the system kernel, resides above the hardware layer and serves as the core of the UNIX operating system.
• Handles interaction with the hardware, including memory management and task scheduling.

• Executes system calls from user programs, providing essential services for program execution.
• Ensures isolation from user programs, facilitating portability across systems with the same kernel.
• User programs communicate with the kernel through system calls, instructing it to perform various operations.

3. Shell Commands (Layer-3):
• The shell commands layer sits on top of the kernel and is responsible for processing user requests and interpreting commands.
• The shell is a utility that processes user input, interpreting commands and calling the relevant programs.
• Various built-in commands and utilities, such as cp, mv, cat, grep, and more, are available for user interaction.
• Users interact with the system by typing commands into the shell, which then executes the corresponding programs or system utilities.

4. Application Layer (Layer-4):
• The outermost layer in the UNIX architecture is the application layer, where external applications are executed.
• Consists of user applications and higher-level programs built on top of the lower-level utilities and system calls.
• Users run their custom or third-party applications within this layer.
• Applications interact with the underlying layers through system calls and lower-level programs, leveraging the services provided by the kernel.

5) Explain the condition when Kernel wants a particular buffer and that buffer is currently busy.

When a kernel in a computing system requires access to a particular buffer, but that buffer is currently busy, it means that the buffer is being accessed or modified by another process or kernel instance. This situation can arise in various computing environments, including parallel computing systems, multi-threaded applications, and graphics processing units (GPUs).

Here's a more detailed explanation of this condition:

1. **Concurrency**: In parallel computing systems, multiple kernels or processes may run concurrently, each performing computations or operations on different parts of data. If multiple kernels attempt to access the same buffer simultaneously, it can lead to conflicts and potential data corruption.

2. **Resource Sharing**: Buffers are often shared among different kernels or processes to facilitate data exchange and communication. However, when one kernel is actively using a buffer, it may lock the buffer to prevent other kernels from modifying its contents until the operation is complete. This locking mechanism ensures data integrity and prevents race conditions.

3. **Synchronization**: Kernels typically use synchronization mechanisms such as locks, semaphores, or barriers to coordinate access to shared resources like buffers. If a kernel requests a buffer that is currently locked or in use by another kernel, it may need to wait until the buffer becomes available. This waiting period introduces latency and can impact overall system performance.

4. **Buffer Management**: Efficient buffer management strategies are essential to minimize contention and maximize resource utilization. Techniques such as double buffering, where multiple buffers are used alternately to overlap computation and data transfer, can help reduce the likelihood of buffers being busy when needed by kernels.

5. **Deadlocks**: In some cases, if proper synchronization is not implemented, deadlock situations may occur where two or more kernels are waiting for resources held by each other, resulting in a system-wide halt. Avoiding deadlocks requires careful design of synchronization protocols and resource allocation policies.

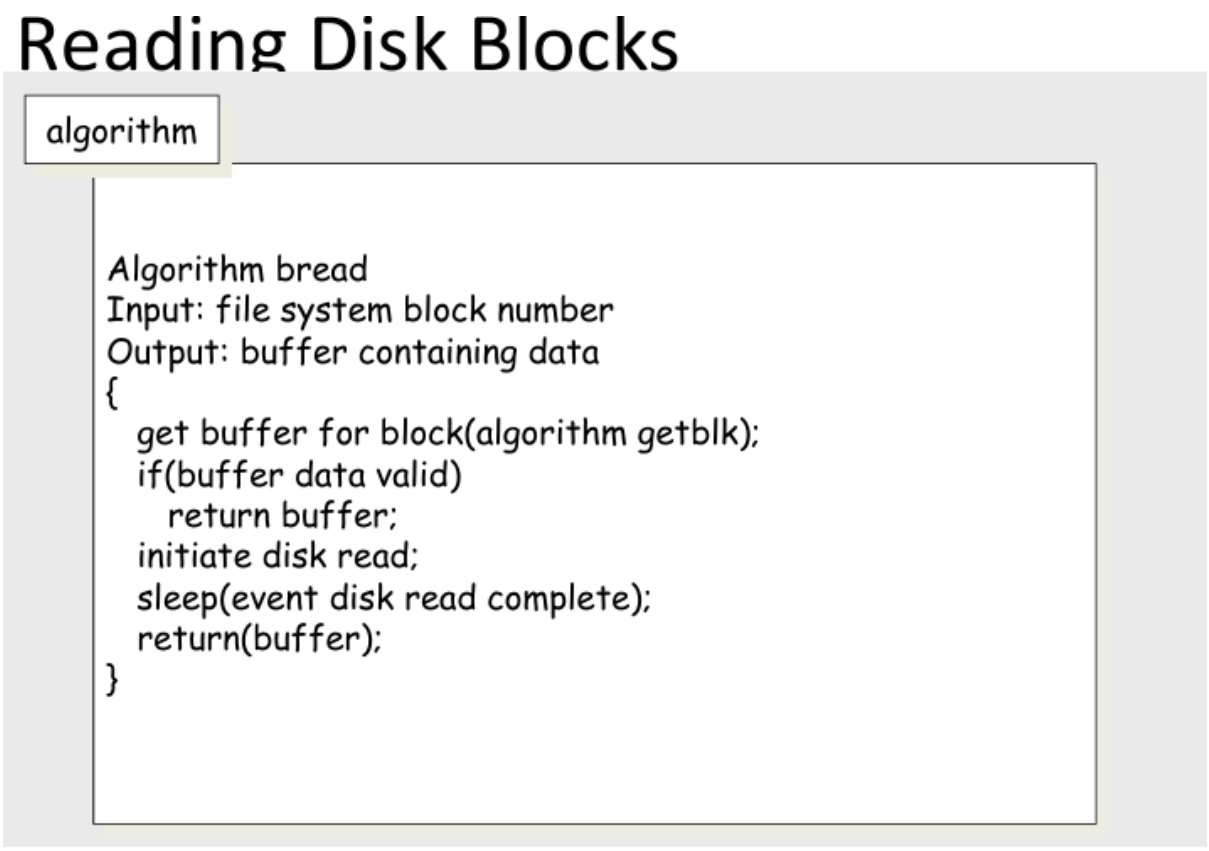
In summary, when a kernel requires access to a particular buffer but finds it busy, it indicates that the buffer is currently being utilized by another process or kernel. Proper synchronization and buffer

management are essential to ensure smooth operation and avoid contention in computing systems.

6) Explain the bread algorithm

Reading Disk Blocks

- Use bread()
- If the disk block is in buffer cache– How to know the block is in buffer cache
- Use getblk() – Return the data without disk access
- Else– Calls disk driver to schedule a read request – Sleep – When I/O complete, disk controller interrupts the Process – Disk interrupt handler awakens the sleeping process– Now process can use wanted data



"Reading Disk Blocks." The pseudocode outlines the steps for reading data from disk blocks. Here's a breakdown of the algorithm:

1. **Algorithm Name**: `bread` (short for "block read")
2. **Input**:
   - `buffer_system_block_number`: The block number for which data needs to be read.
3. **Output**:
   - Returns the data read from the specified disk block.

4. **Steps**:

   a. **Get Buffer Data (getblk)**:

   - The algorithm begins by calling a function named `getblk` with the input block number.
   - The purpose of `getblk` is to find or allocate a buffer associated with the specified block number.
   - If the buffer is already in memory, it is returned. Otherwise, a new buffer is allocated.

   b. **Initiate Disk Read**:

   - The algorithm initiates a read operation from the disk for the specified block.
   - The data from the disk will be loaded into the allocated buffer.

   c. **Sleep Until Disk Read Completes**:

   - The algorithm waits (sleeps) until the disk read operation completes.
   - This ensures that the buffer contains valid data before proceeding.

   d. **Return Buffer**:

   - Once the data is available in the buffer (after the disk read), the algorithm returns a pointer to the buffer.
   - The calling process can then access the data from the buffer.
5. **Purpose**:
   - The purpose of this algorithm is to efficiently read data from disk blocks while managing buffer allocation and synchronization.
   - It ensures that the data is consistent and available for subsequent processing.
6. **Use Cases**:
   - File systems, databases, and other storage systems use similar algorithms to read data blocks from storage devices.

Remember that this algorithm abstracts away low-level details related to disk I/O and buffer management, allowing higher-level processes to read data seamlessly. 🗂️🔍

What is a buffer? Explain the structure of the Buffer Header.

A buffer in computing refers to a temporary storage area used to hold data temporarily while it is being transferred from one place to another or while it is being processed. Buffers are widely used in various computing tasks, including input/output operations, network communication, and graphics rendering.

Buffer headers play a vital role in managing the buffer cache in UNIX, helping to

reduce the frequency of disk access. The buffer header contains essential

information about the associated buffer and facilitates efficient data management

1. Buffer Header:

• Device Number: Identifies the device associated with the buffer.

• Logical FS Number: Represents the logical file system number, not the physical device number.

• Block Number: Specifies the block number associated with the data in the buffer.

• Pointer to Data Array: Points to the memory array containing the actual data from the disk.

2. Memory Array (Buffer):

• Contains the data retrieved from or to be written to the disk.

• The size of the memory array corresponds to the size of the buffer.

3. States of Buffer:

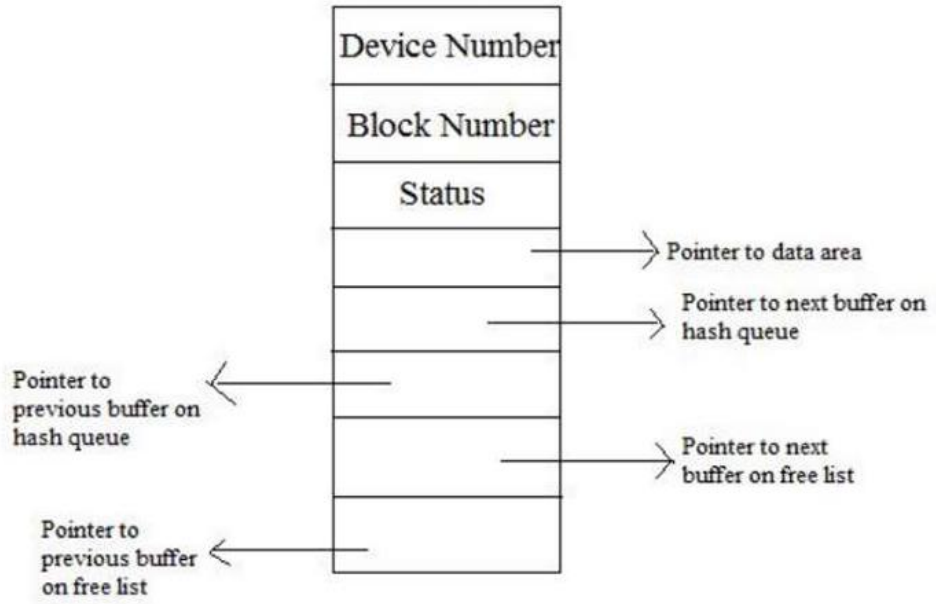
• Locked: Indicates that the buffer is currently being used or modified and is not available for other operations.

• Valid: Denotes that the data in the buffer is valid and consistent with the corresponding disk block.

• Delayed Write: Signifies that the changes to the buffer are scheduled for delayed writing to the disk.

• Reading/Writing: Indicates that the buffer is actively involved in a read or write operation.

• Waiting for Free: Implies that the buffer is in a queue, waiting for a free slot in the buffer cache.


Device Number
Block Number
Status
Pointer to data area
Pointer to next buffer on hash queue
Pointer to previous buffer on hash queue
Pointer to next buffer on free list
Pointer to previous buffer on free list