Introduction and Buffer Cache

Draw and Explain the Block diagram of the UNIX kernel.

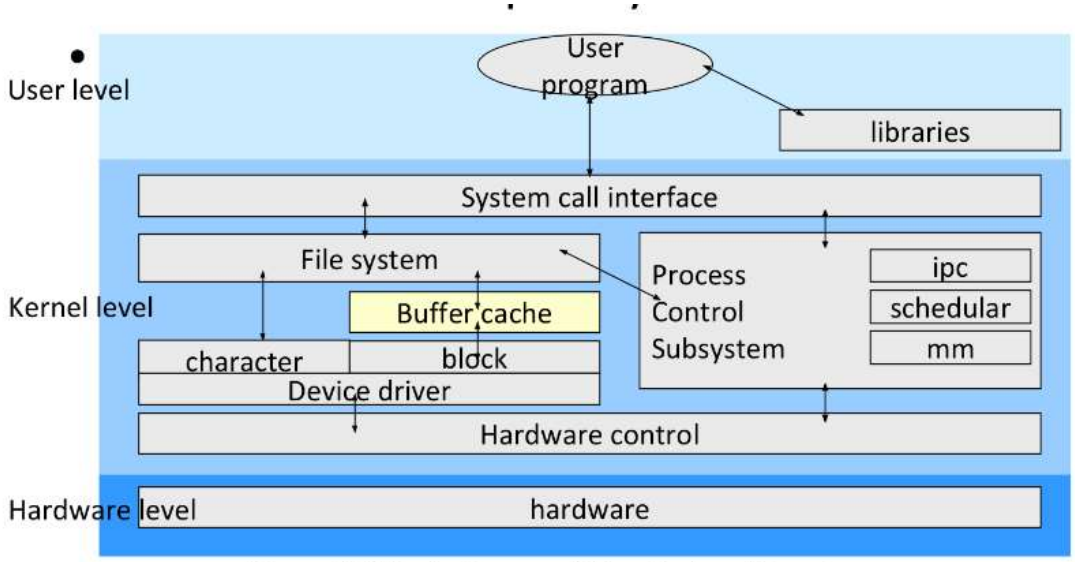


Figure 2.1 block diagram of the System Kernel

Prepared By : Prof. S. P. Kakade

The block diagram of the UNIX kernel provides an overview of the various modules and their relationships within the kernel. Let's break down the components and their functionalities:

Levels of the Kernel:

- **User Level:** Represents the interface where user programs interact with the kernel through system calls and libraries.
- **Kernel Level:** Consists of the core components of the operating system responsible for managing resources and providing services to user programs.
- **Hardware Level:** Represents the physical hardware components of the system.

Central Concepts:

- **Files and Processes:** These are the two central concepts in the UNIX system model, forming the foundation of its operation.

Major Components:

1. **File Subsystem:**
    - Manages files, including allocating file space, administering free space, controlling access, and retrieving data for users.
    - Interacts with processes through specific system calls such as open, close, read, write, and stat.
    - Accesses file data using a buffering mechanism that regulates data flow between the kernel and secondary storage devices.
    - Interacts with block I/O device drivers for data transfer.

2. **Process Control Subsystem:**
    - Responsible for process synchronization, interprocess communication, memory management, and process scheduling.
    - Memory management module controls memory allocation, managing processes between main memory and secondary memory.
    - Scheduler module allocates CPU to processes, scheduling them to run in turn or preempting them based on priority and time quantum.
    - Interacts with the file subsystem during the loading of files into memory for execution.
3. **Hardware Control:**
    - Handles interrupts and communicates with hardware devices such as disks or terminals.
    - Ensures proper handling of interrupts, allowing the kernel to resume execution of interrupted processes after servicing the interrupt.

Key Interactions:

- **System Call and Library Interface:** Acts as the boundary between user programs and the kernel, facilitating communication through system calls and libraries.
- **File Subsystem and Process Control Subsystem Interaction:** Collaboration occurs during file loading into memory for execution, demonstrating the interconnected nature of kernel components.

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.

1. The pipe, a mechanism that allows a stream of data to be passed between reader and writer processes. Processes can redirect their standard output to a pipe to be read by other processes that have redirected their standard input to come from the pipe.
2. The data that the first processes write into the pipe is the input for the second processes. The second processes could also redirect their output, and so on, depending on programming need. Again, the processes need not know what type of file their standard output is; they work regardless of whether their standard output is a regular file, a pipe, or a device

- ***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.

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;
 }
}}
```

Prepared By : Prof. S. P. Kakade

Scenario 1: Finding a Buffer - Buffer on Hash Queue

1. **Mark Buffer as Busy:** Mark the buffer associated with the requested block as busy.
2. **Remove from Free List:** Remove the buffer from the free list.
3. **Return Buffer to Caller:** Return the buffer to the caller.

Scenario 2: Buffer Not Present in Hash Queue

1. **Remove Buffer from Free List:** Select a buffer from the free list.
2. **Asynchronous Write (if needed):** Initiate an asynchronous write operation if the buffer was marked for delayed write.
3. **Remove from Old Hash Queue:** Detach the buffer from its old hash queue.
4. **Put onto New Hash Queue:** Place the buffer into the new hash queue.
5. **Return Buffer to Caller:** Return the allocated buffer to the caller.

Scenario 3: Delayed Write Buffer Present

1. **Remove Delayed Write Buffer from Free List:** Identify and remove the delayed write buffer from the free list.
2. **Asynchronous Write of Delayed Buffer:** Asynchronously write the delayed buffer to disk.

3. **Remove Next Buffer from Free List:** Select the next buffer from the free list.
4. **Remove from Old Hash Queue:** Detach the selected buffer from its old hash queue.
5. **Put onto New Hash Queue:** Insert the buffer into the new hash queue.
6. **Return Buffer to Caller:** Return the allocated buffer to the caller.

Scenario 4: No Free Buffers Available

1. **Sleep on Event for Free Buffer:** Process waits for a signal indicating the availability of a free buffer.
2. **Awakening on Buffer Release:** Process awakens when another process releases a buffer.
3. **Resuming Execution and Repeating Steps:** Resumes execution and repeats steps if necessary.
4. **Buffer Allocation or Repeating Second Scenario:** Process either successfully allocates a buffer or repeats steps from scenario 2.

Key Considerations:

- Involves waiting for an indefinite amount of time if no free buffers are available.
- Mitigated by maintaining an adequate number of buffers in the free list.

Scenario 5: Busy Buffer in Hash Queue

1. **Sleep on Event for Free Buffer:** Process waits for the buffer to become free.
2. **Awakening on Buffer Release:** Process awakens when the buffer becomes available.
3. **Resuming Execution and Repeating Steps:** Process evaluates the buffer allocation scenario again.
4. **Marking Buffer as Busy and Returning:** Process marks the buffer as busy, removes it from the free list, and returns it.
5. **Buffer Reassigned by Another Process:** Process restarts its search if the buffer is no longer found in the hash queue.

   Key Considerations:

   - Involves waiting for the buffer to become free, potentially experiencing multiple iterations.
   - Buffer cache design minimizes this scenario by ensuring efficient buffer utilization.

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.

Explain the architecture of the UNIX System.

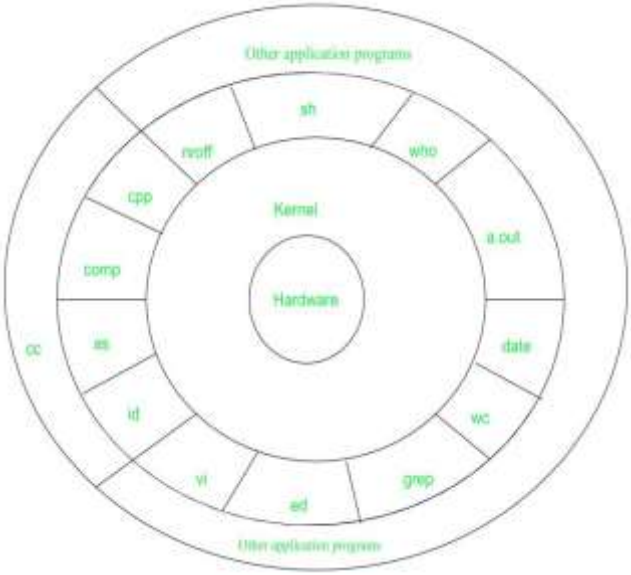
The UNIX operating system architecture is organized into layers, each serving a specific purpose and contributing to the overall functionality. Let's delve into the architecture of the UNIX system

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 time.
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.

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

The fifth scenario for buffer allocation occurs when the requested block is present in the hash queue, but the buffer containing it is currently busy or locked by another process. In this case, the algorithm goes through specific steps to handle this scenario:

Algorithm Steps:

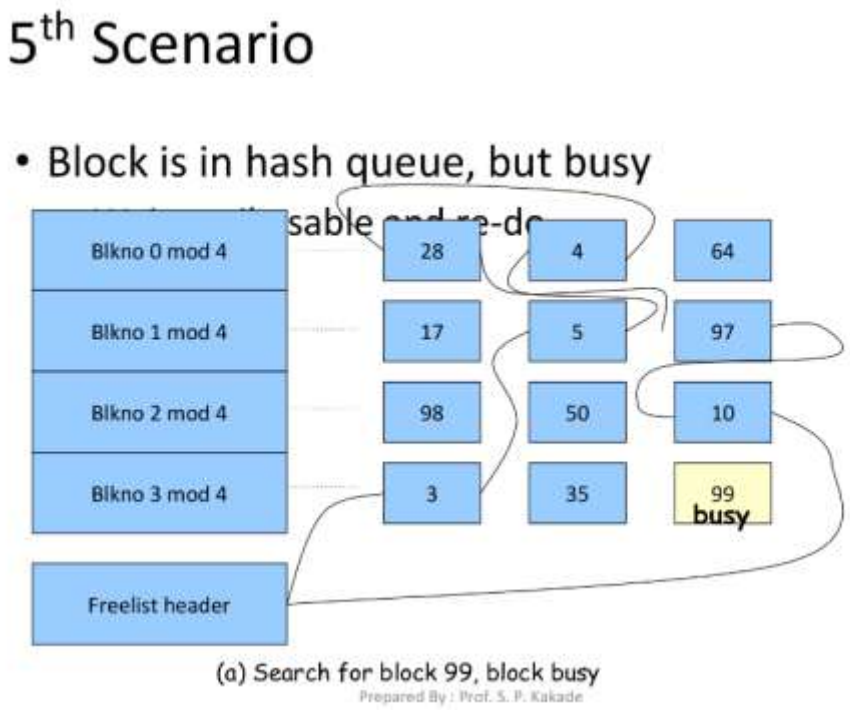
1. **Sleep on Event for Free Buffer:**
    - The process, upon finding that the buffer for the requested block is in the hash queue but is currently busy, sleeps on the event of the buffer becoming free. This means that the process suspends its execution and waits for a signal indicating that the buffer is no longer busy.
2. **Awakening on Buffer Release:**
    - When another process releases the buffer, marking it as not busy, the waiting process is awakened. It receives a signal indicating that the buffer is now available for use.
3. **Resuming Execution and Repeating Steps:**
    - The awakened process resumes its execution and reevaluates the buffer allocation scenario. If the buffer for the requested block is still found in the hash queue and is not busy, the process proceeds to use the buffer.
4. **Marking Buffer as Busy and Returning:**
    - If the buffer is not busy, the process marks the buffer as busy, removes it from the free list, and returns the buffer to the caller.
5. **Buffer Reassigned by Another Process:**
    - If, during the process's search, the buffer is no longer found in the hash queue, it indicates that the buffer has been reassigned to another block by another process. In this case, the process starts its search again.

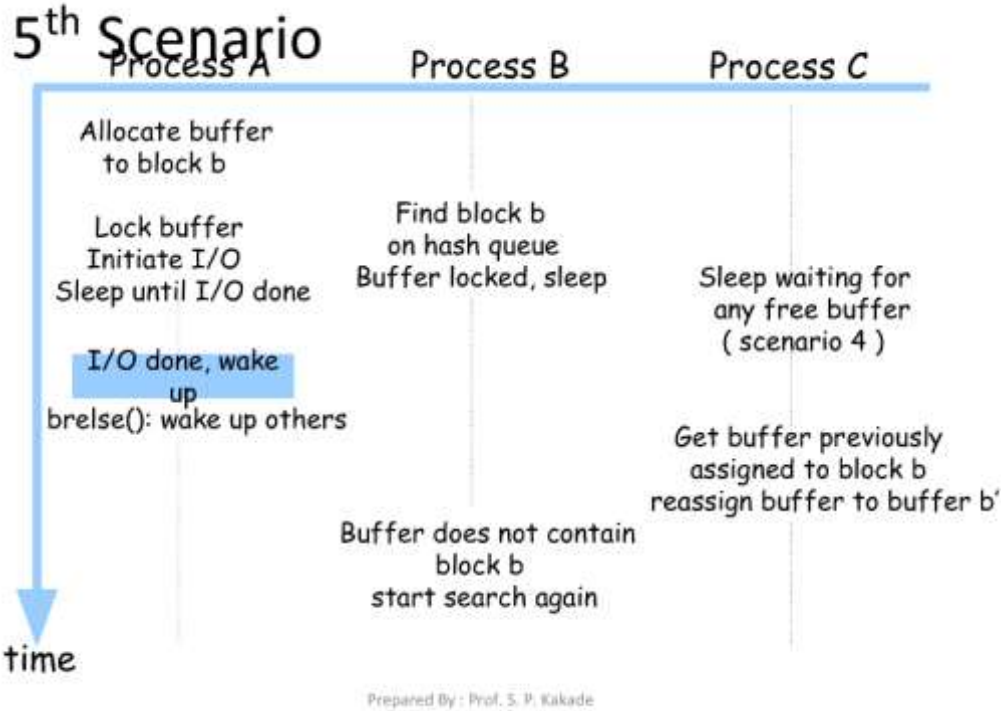
Key Considerations:

- This scenario involves some overhead and potential delay as the process may need to wait for the buffer to become free.
- The process might experience multiple iterations of searching for a free buffer in the hash queue.
- The buffer cache is designed to minimize the occurrence of this scenario by using a fair scheduling policy, avoiding long locks on buffers, and ensuring efficient buffer utilization.

Reminder:

Processes releasing buffers should promptly notify and wake up waiting processes to prevent unnecessary delays in buffer allocation





Explain the bread algorithm.

The algorithm for reading a disk block involves using the **bread()** function, which utilizes the **getblk()** function to manage buffers in the buffer cache. The goal is to retrieve the data from the buffer cache if the disk block is already present; otherwise, initiate a disk read operation, sleep until the read is complete, and return the buffer containing the data.

Algorithm: bread()

Input: File system block number

Output: Buffer containing data

1. **Get Buffer for Block (using getblk()):**
    - Check if the buffer for the specified block is already present in the buffer cache.
    - If the buffer is found and its data is valid, return the buffer immediately.
2. **Initiate Disk Read:**
    - If the buffer is not in the cache or its data is not valid, initiate a disk read operation for the specified block.
3. **Sleep on Disk Read Complete Event:**
    - Suspend the execution of the process by sleeping until the disk read operation is complete.
    - This waiting state is triggered by an event signaling the completion of the disk read.
4. **Buffer Validity Check:**
    - Upon awakening, check whether the buffer's data is now valid.
5. **Return Buffer:**
    - Return the buffer, whether it was retrieved from the buffer cache or read from the disk.

Read Ahead (Improving Performance):

To enhance performance, a read-ahead strategy is introduced, where an additional block is read before it is requested. The **breada()** function is utilized for this purpose.

Algorithm: breada()

Input:

1. File system block number for immediate read.

2. File system block number for asynchronous read (read ahead).

Output: Buffer containing data for immediate read.

1. **Check and Read First Block:**
    - If the buffer for the first block is not in the cache:
        - Get a buffer for the first block using **getblk()**.
        - If the buffer's data is not valid, initiate a disk read operation for the first block.
2. **Check and Read Second Block (Asynchronous Read):**
    - If the buffer for the second block is not in the cache:
        - Get a buffer for the second block using **getblk()**.
        - If the buffer's data is valid, release the buffer (using **brelse()**).
        - Otherwise, initiate a disk read operation for the second block.
3. **Check Status of First Block:**
    - If the first block was originally in the cache:
        - Read the first block (using **bread()**).
        - Return the buffer containing data.
4. **Sleep on Valid Data (First Buffer):**
    - If the first buffer's data was not valid, sleep until it contains valid data.
5. **Return Buffer:**
    - Return the buffer, whether obtained from the buffer cache or read from the disk.

These algorithms facilitate efficient disk block reading, considering cache presence, asynchronous reads, and read-ahead strategies for improved performance.

Reading Disk Blocks

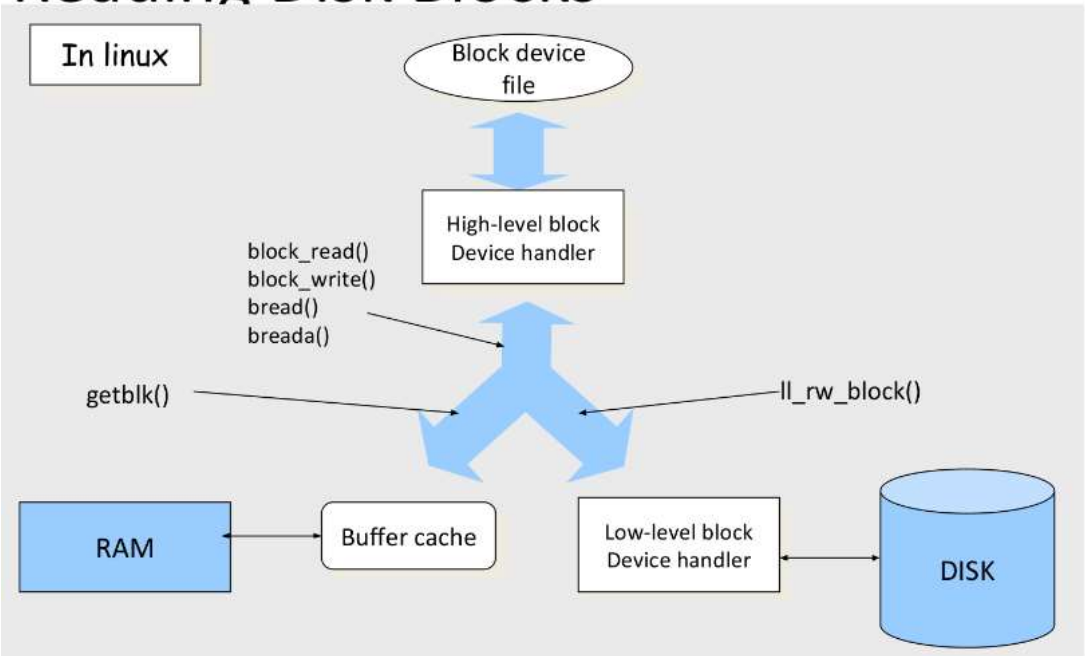


Figure 13-3 block device handler architecture for buffer I/O operation in Understanding the Linux Kernel

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

A buffer in an operating system is a temporary storage area in main memory used to hold data temporarily while it is being transferred between devices or between a device and an application. Buffers are essential for efficient data transfer and processing, helping to manage data flow between different components of the system and preventing bottlenecks.

The structure of a buffer header is crucial in managing the buffer cache in UNIX-based operating systems. Here's an explanation of the components of a buffer header:

1. **Buffer Header:**
    - **Device Number:** Identifies the device associated with the buffer. This information helps in locating the data on the storage device.
    - **Logical File System Number:** Represents the logical file system number, not the physical device number. It helps in organizing and managing files within the file system.
    - **Block Number:** Specifies the block number associated with the data in the buffer. Blocks are the basic units of data storage on storage devices.
    - **Pointer to Data Array:** Points to the memory array containing the actual data from the disk. This pointer enables access to the data stored in the buffer.
2. **Memory Array (Buffer):**
    - Contains the actual data retrieved from or to be written to the disk. The size of the memory array corresponds to the size of the buffer allocated for storing data temporarily.
3. **States of Buffer:**
    - **Locked:** Indicates that the buffer is currently being used or modified and is not available for other operations. Locking ensures data integrity during concurrent access.
    - **Valid:** Denotes that the data in the buffer is valid and consistent with the corresponding disk block. It indicates that the buffer contains accurate data.
    - **Delayed Write:** Signifies that the changes to the buffer are scheduled for delayed writing to the disk. This optimization helps in improving disk write performance by batching write operations.
    - **Reading/Writing:** Indicates that the buffer is actively involved in a read or write operation. It reflects the current I/O operation being performed on the buffer.

- **Waiting for Free:** Implies that the buffer is in a queue, waiting for a free slot in the buffer cache. This state occurs when there are no available buffers for new data and the system needs to wait for a buffer to become available.