page 48, question 3
modification bagha answer madhe

Module 1:

1. What is the difference between fork and vfork? What sequence operation fork does on calling? For example explain, how fork system calls are different from vfork system calls.

Ans:

	fork	vfork
1	In fork() system calls, the child and parent process have separate memory space.	While in vfork() system call, the child and parent process share the same address space.
2	The child process and parent process gets executed simultaneously.	Once the child process is executed then the parent process starts its execution.
3	The fork() system call uses	While vfork() system call
	copy-on-write as an alternative.	does not use copy-on-write.
4	Child process does not suspend parent process execution in fork() system call.	Child process suspends parent process execution in vfork() system call.

5	Page of one process is not affected by the page of the other process.	Page of one process is affected by the page of the other process.
6	There is wastage of address space.	There is no wastage of address space.
7	If the child process alters the page in address space, it is invisible to the parent process.	If the child process alters the page in address space, it is visible to the parent process.

The steps followed by the kernel for fork are:

1.It creates a new entry in the process table.

2.It assigns a unique ID to the newly created process.

3.It makes a logical copy of the regions of the parent process. If a region can be shared, only its reference count is incremented instead of making a physical copy of the region.

4.The reference counts of file table entries and inodes of the process are increased.

5.It turned the child process ID to the parent and 0 to the child.

Example:

a. Copy-on-write behavior:

fork(): When fork() is called, it creates a duplicate of the current process (the parent process). This includes duplicating the entire address space of the parent process. However, the memory pages are not immediately copied. Instead, they are marked as

copy-on-write (COW). This means that the pages are shared between the parent and child processes until one of them modifies the page. At that point, the page is copied for the process of making the modification.

vfork(): When vfork() is called, it also creates a new process like fork(), but it does not immediately create a copy of the parent's address space. Instead, the child process shares the address space of the parent process. This means that any modifications made by the child process directly affect the parent's address space. Because of this, the parent process is typically suspended until the child process either calls exec() or _exit().

b. Parent Process Behavior:

fork(): In the case of fork(), the parent process continues execution independently of the child process. It receives the child's process ID (PID) as the return value of the fork() call.

vfork(): In the case of vfork(), the parent process is typically suspended until the child process either calls exec() or _exit(). This is because the child process shares the parent's address space, and any modifications made by the child affect the parent process. Therefore, it's crucial for the child process to either replace its image with a new program using exec() or exit immediately using _exit() to avoid potential conflicts with the parent.

2) What is an orphan process? Who is the parent of the orphan process? Why?

If the parent process itself dies before the child process, then the child process becomes orphan. When a parent process terminates without properly waiting for its child processes to terminate, these child processes become orphaned. Orphan processes are then adopted by the init process (process ID 1), which is the ancestor of all processes in a Unix-like operating system.
The init process becomes the parent of orphaned processes because it has the responsibility of reaping orphaned child processes

3) Enlist the system calls used in process control.
Ans:wait() :

In some systems, a process may wait for another process to complete its execution. This happens when a parent process creates a child process and the execution of the parent process is suspended until the child process executes. The suspending of the parent process occurs with a wait() system call. When the child process completes execution, the control is returned back to the parent process.

exec() :

This system call runs an executable file in the context of an already running process. It replaces the previous executable file. This is known as an overlay. The original process identifier remains since a new process is not created but data, heap, stack etc. of the process are replaced by the new process.

fork() :

Processes use the fork() system call to create processes that are a copy of themselves. This is one of the major methods of process creation in operating systems. When a parent process creates a child process and the execution of the parent process is suspended until the child process executes. When the child process completes execution, the control is returned back to the parent process.

exit() :

The exit() system call is used by a program to terminate its execution. In a multi-threaded environment, this means that the thread execution is complete. The operating system reclaims resources that were used by the process after the exit() system call.

kill() :

The kill() system call is used by the operating system to send a termination signal to a process that urges the process to exit. However, kill system call does not necessary mean killing the process and can have various meanings.

4) why System calls are required

The following circumstances involve the use of system calls in OS:

- System calls are necessary for reading and writing from files.
- System calls are necessary for a file system to add or remove files.
- New processes are created and managed using system calls.
- System calls are required for packet sending and receiving over network connections.
- A system call is required to access hardware devices like scanners and printers

5) **Kernel is non-preemptive .Does it mean mutual exclusion is there on kernel .If yes/no justify.**

Yes,kernel maintains consistency of data structures thereby avoiding mutual exclusion problem making sure that critical sections of code are executed by at most one process at a time

The kernel allows a context switch only when a process moves from the state "kernel running" to the state "asleep in memory". Processes running in kernel mode cannot be preempted by other processes; therefore the kernel is sometimes said to be non-preemptive.

6) **Why are kernel data Structure are Static**?

Ans:Kernel data structures are often designed to be static for several reasons:

1. Performance: Static data structures can be more efficient in terms of memory and execution time compared to dynamic data structures. Since the kernel is responsible for managing system resources and handling critical tasks, efficiency is crucial.

Static data structures eliminate the overhead of dynamic memory allocation and deallocation, which can be relatively expensive in terms of both time and space.

2. Predictability: Static data structures have deterministic behavior, which can be important for real-time systems or environments where predictability is essential. Dynamic memory allocation introduces uncertainty in terms of allocation time and fragmentation, which may not be desirable in the kernel where timing constraints are critical.

3. Simplicity and Reliability: Static data structures are typically simpler to implement and reason about compared to dynamic ones. They have a fixed size and layout, making it easier to verify correctness and ensure reliability. This simplicity can be crucial in the kernel, where bugs or errors can have severe consequences for system stability and security.

4. Safety: Static data structures are less prone to memory leaks and fragmentation issues that can occur with dynamic memory allocation. In a resource-constrained environment like the kernel, ensuring memory safety and stability is paramount.

Data Structure for my own OS

1.process Management : Doubly Linked List, Process table

2.File Management : Tree, user file descriptor, File table, Inode table

3.Buffer cache Management : Free list, Hash queue

4.region Management : Stack, Region table, Per process region table

7) **Enlist the function of system administrator**.

In short, system administrators:

1. Install and configure hardware and software.

2. Maintain and update systems.

3. Manage user accounts and access.

4. Implement backup and recovery plans.

5. Ensure system security.

6. Monitor and optimize performance.

7. Document procedures and provide training.

8. Enforce compliance with policies and regulations.

8) What is the difference between user mode and Kernel mode and when process moving from user mode to kernel mode.

Ans :User mode and kernel mode are two different states in which a computer's processor can operate:

- User Mode: In user mode, applications and user-level processes run. They have limited access to the computer's hardware and resources, which helps ensure system stability and security. Most programs you use, like web browsers and word processors, run in user mode.

- Kernel Mode: Kernel mode is the privileged state where the operating system's core functions operate. It has unrestricted access to the computer's hardware and can execute critical tasks like managing memory, handling hardware interrupts, and controlling device drivers.

When a process needs to perform a task that requires higher privileges, such as accessing hardware or modifying system settings, it transitions from user mode to kernel mode. This transition is usually triggered by a system call, which is a request from the user mode process to the kernel for a specific service or operation. The kernel then executes the requested operation on behalf of the user mode process in kernel mode, ensuring that critical system functions are performed securely and

efficiently. Once the operation is completed, control returns to the user mode process.

9) Which memory management technique is suitable for multi-user OS?

Ans: time-sharing and batch processing

10)What are the advantages in kernel, when devices are treated as file?

Ans Treating devices as files in the kernel makes it easier for programs to communicate with devices. It provides a consistent way to read from and write to devices, simplifies security and access control, abstracts hardware details, and allows for easier error handling and integration with existing tools. This approach improves usability, security, and development in the operating system.

3 marks
Treating devices as files in the kernel provides several advantages, including:

1. Uniform Interface: By treating devices as files, the kernel can provide a uniform interface for interacting with both devices and regular files. This simplifies the programming model for developers as they can use similar operations (read, write, open, close) on devices as they do on regular files.

2. Simplified I/O Operations: Treating devices as files allows for a consistent way of performing input and output operations. Applications can use standard file I/O functions to communicate with devices, making it easier to manage and control I/O operations.

3. Ease of Access: Devices being represented as files means that they can be accessed using file system APIs, making it easier for applications to interact with devices without needing to understand the intricacies of device-specific APIs.

4. File Permissions and Security: By treating devices as files, the kernel can leverage the file system's permission and security mechanisms to control access to devices. This allows for fine-grained control over which processes can access specific devices.

5. Device Abstraction: Treating devices as files abstracts the underlying hardware details from applications. This abstraction simplifies application development as developers do not need to worry about the specific hardware details of each device.

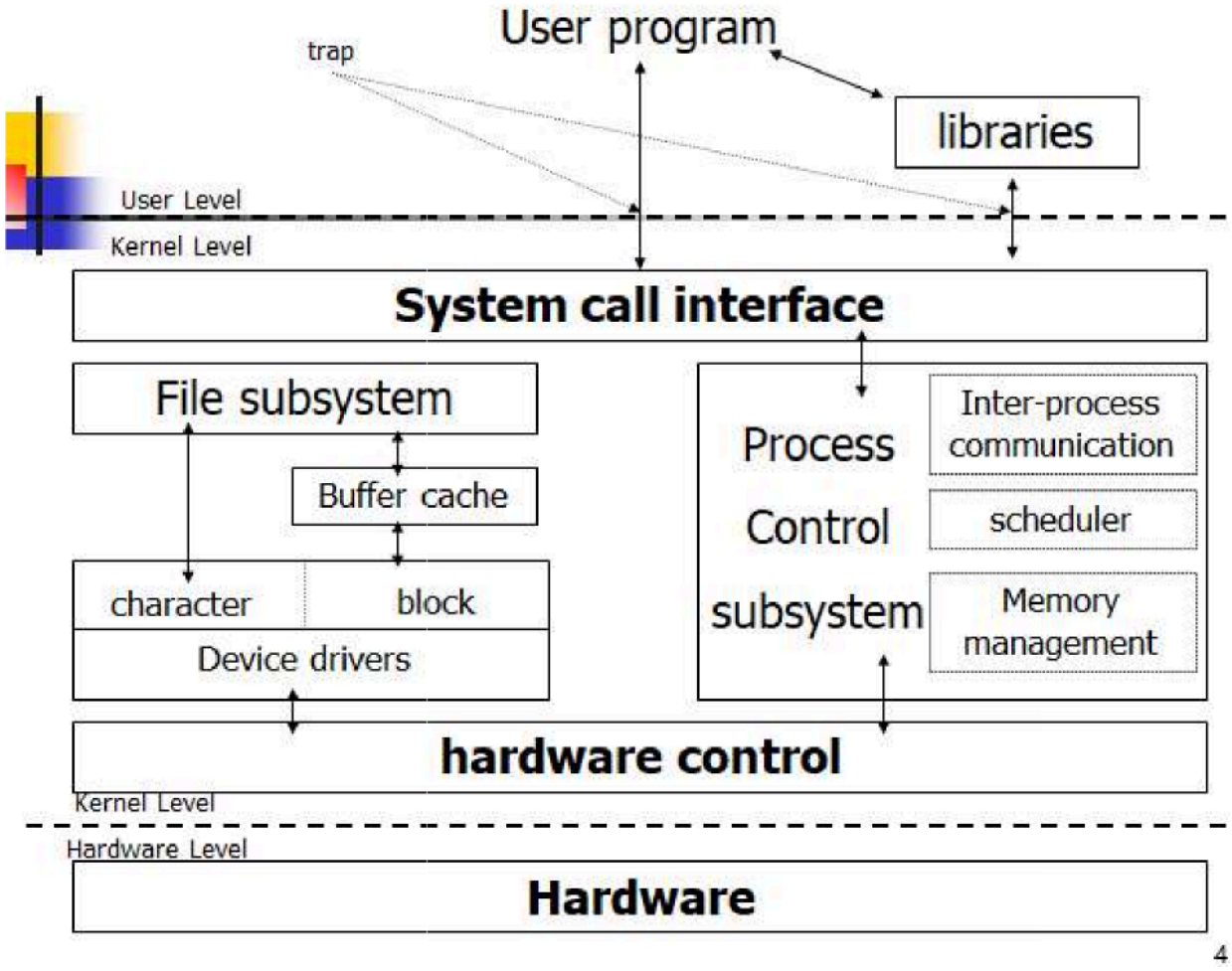
6. Consistent Error Handling: Using the file interface for devices allows for consistent error handling mechanisms. Errors related to device operations can be handled in a similar way to file-related errors, providing a standardized approach to error handling.

7. Integration with Existing Tools: Devices being treated as files means that existing file manipulation tools and utilities can be used to interact with devices. This simplifies device management and troubleshooting tasks.

11)Why processes in kernel mode cannot be preempted?
Justify?

Ans: Processes in kernel mode cannot be preempted because they handle crucial tasks like managing hardware and system resources. Preempting them could lead to data corruption, missed hardware events, and security risks. Keeping them running without interruption ensures the stability, consistency, and security of the operating system.

12)Block diag of system kernel and explain responsibility of processes control and file control subsystem



The process control subsystem and file control subsystem are essential components of an operating system responsible for managing processes and files, respectively. Here's an overview of their responsibilities:

1. .Process Control Subsystem:.
   - .Process Creation and Termination:. It handles the creation and termination of processes. This includes allocating resources, setting up the process environment, and managing the life cycle of processes.

- .Process Scheduling:. It decides which processes to run and when, based on scheduling algorithms. This involves managing CPU time allocation to processes efficiently.
- .Process Communication:. It facilitates communication and synchronization between processes through mechanisms like inter-process communication (IPC), shared memory, semaphores, and message passing.
- .Process Monitoring and Management:. It monitors the status and performance of processes, manages process priorities, handles process states (e.g., ready, running, blocked), and deals with process interruptions (e.g., signals, interrupts).
- .Process Protection:. It enforces security and protection mechanisms to ensure that processes operate within their designated boundaries and do not interfere with each other or with system resources.

2. .File Control Subsystem:.
- .File Creation and Deletion:. It manages the creation, deletion, and renaming of files and directories. This includes allocating storage space and maintaining file metadata (e.g., file attributes, permissions).
- .File Organization:. It determines how files are organized on storage devices, such as sequential, indexed, or hashed organization. This affects file access and retrieval efficiency.
- .File Access Control:. It enforces access control mechanisms to regulate file access based on user permissions and file attributes. This includes authentication, authorization, and file-level security.
- .File Retrieval and Modification:. It handles read, write, append, and update operations on files. This involves managing file buffers, caching, and ensuring data integrity during file operations.
- .File Backup and Recovery:. It may include features for file backup, versioning, and recovery to protect against data loss or corruption. This can involve periodic backups, snapshotting, and recovery utilities.

Both subsystems play crucial roles in the overall functioning of an operating system by managing resources efficiently, ensuring system stability, providing a secure environment for processes and files, and facilitating smooth interaction between users and the system.

Module 4
The System Calls

1) What is a system call? Why are they designed in OS?

Ans: A system call is a request made by a program to the operating system for a specific service, such as reading or writing data to a file, creating a new process, or allocating memory. System calls provide an interface for programs to interact with the underlying operating system and access its resources and services.

System calls are designed in operating systems to provide a controlled and secure way for applications to access system resources and perform privileged operations. By using system calls, programs can interact with hardware devices, manage memory, and perform other low-level operations without compromising the stability and security of the system. System calls also help in abstracting the hardware details from the application, making it easier to develop and port software across different platforms.

2). What is the stat system call? Enlist and explain the various fields shown by stat system call?

Ans: The system calls stat and fstat allow processes to query the status of files.

The syntax for the system calls is stat (pathname, statbuffer); where pathname is a file name and statbuffer This is the address of a data structure where the status information of the file will be stored after the system call completes

Stat returns information such as the file type, file owner, access permissions, file size, number of links, inode number, and file access times.

For 8 marks

The "stat" system call retrieves information about a file identified by its pathname. Here's a detailed explanation of how the "stat" system call works:

1. Input: The input to the "stat" system call is the pathname of the file whose information is to be retrieved.

2. .Path Resolution:. The kernel begins by resolving the provided pathname to locate the file in the filesystem. This involves traversing the directory hierarchy from the root to the specified file.

3.Access Check: Once the file is located, the kernel checks if the process has the necessary permissions to access the file. This includes checking permissions such as read, write, and execute permissions based on the process's user and group IDs.

4. Retrieve File Information:.

- After ensuring access permissions, the kernel retrieves various pieces of information about the file. This information typically includes:

- File type (regular file, directory, symbolic link, etc.)

- File size

- Owner user ID and group ID

- File permissions (mode)

- Timestamps (last access, last modification, last status change)

- Number of hard links to the file

- File system ID and inode number

- Device ID (if the file resides on a separate device)

- Block size and number of blocks allocated to the file

5. .Return File Information:.

- Once all the necessary information is gathered, the kernel returns it to the calling process. This information can be used by the process for various purposes, such as file management, access control, and metadata analysis.

6. .Error Handling:.

- If the file specified by the pathname does not exist or if the process lacks sufficient permissions to access the file, the "stat" system call returns an error code to the calling process. This allows the calling process to handle such situations gracefully.

Overall, the "stat" system call provides a convenient way for processes to retrieve detailed information about files in the filesystem, enabling them to make informed decisions and perform various operations on files based on their attributes.

3) Difference between named and unnamed pipes

Ans:

Named Pipes	Unnamed pipes
Named pipes are given a name and exist as a file in a system, represented by an inode. mkfifo (char *path, mode_t access_mode)	They do not have names, also referred to as anonymous and identified by their two file descriptors. pipe (int fd[2]);
They can be used even among unrelated processes.	They can be used only between related processes.
They are bidirectional, which means the same FIFO can be read from as well as written into.	They are unidirectional, two separate pipes are needed for reading and writing.
Once created, they exist in the file system independent of the process, can be used by other processes.	Un-named pipes vanish as soon as it is closed or one of the related processes terminates.
Named pipes can be used for communication between systems across networks	They are local, they cannot be used across networks.

named pipe vs. unnamed pipe

- open system call / pipe system call
- process access file permission can be given like file/default
- pipe call access/ process descendant access
- used for communication between a child and its parent process/ two unnamed process
- Permanent / transient
- handles one-way or two-way communication between two unrelated processes./handles oneway communication.
- Size 64kb /direct block size i.e10kb: depend on os

4) Difference Between interrupt and Exception

Interrupt	Exception
These are Hardware interrupts.	These are Software Interrupts.
Occurrences of hardware interrupts usually disable other hardware interrupts.	This is not a true case in terms of Exception.
These are asynchronous external requests for service (like keyboard or printer needs service).	These are synchronous internal requests for service based upon abnormal events (think of illegal instructions, illegal address, overflow etc).
Being asynchronous, interrupts can occur at any place in the program.	Being synchronous, exceptions occur when there is abnormal event in your program like, divide by zero or illegal memory location.
These are normal events and shouldn't interfere with the normal running of a computer.	These are abnormal events and often result in the termination of a program

For more details:

https://www.geeksforgeeks.org/difference-between-interrupt-and-exception/

5)Which are the only two processes exist throughout in life time of the system

Ans: 1.process 0 (swapper)

2.process 1 (init)

6) which kernel data structure describe the state of the process

Two kernel data structures describe the state of a process: the process table entry and the u-area. The process table contains information that should be accessible to the kernel and the u-area contains the information that should be accessible to the process only when it's running.

The state field in Processes table identifies the state of the process

7)Why are various fields of the u-area required to access during process execution?

The u-area (user area) is a data structure in the Unix operating system that contains various fields and information related to a process during its execution. These fields are required to be accessed for several reasons:

1. Process Control Block (PCB): The u-area contains the Process Control Block, which includes information about the process, such as the process ID, parent process ID, user and group IDs, program counter, and register values. This information is essential for the operating system to manage and control the execution of the process.

2. File Descriptor Table: The u-area also contains the file descriptor table, which maintains information about the files and devices opened by the process. This table is necessary for the process to read from and write to files, sockets, and other I/O devices.

3. Signal Handling: The u-area stores information about the signals received by the process, such as the signal handlers and pending signals. This information is required for the process to handle signals sent by the operating system or other processes.

4. Memory Management: The u-area includes fields related to memory management, such as the program break and stack pointer. These fields are essential for the process to allocate and deallocate memory dynamically during its execution.

5. Environment Variables: The u-area contains the environment variables set for the process, which provide information about the process environment, such as the working directory, path, and other settings.

Overall, the fields in the u-area are required to be accessed during process execution to provide the necessary information and resources for the process to run efficiently and interact with the operating system and other processes.

8) What are the system calls that support the processing environment in the kernel? How the kernel uses system calls for processing.

System calls are essential for interfacing between user-level processes and the kernel in an operating system. They provide a means for user-level processes to request services from the kernel. Some of the system calls that support the processing environment in the kernel include:

1. Process Control:

- fork(): Creates a new process by duplicating the calling process.
- exec(): Loads a new program into the current process's memory space, replacing the existing program.
- exit(): Terminates the calling process and returns its resources to the system.

2. File Management:

- open(): Opens a file and returns a file descriptor.
- read(): Reads data from a file.
- write(): Writes data to a file.
- close(): Closes an open file descriptor.

3. Memory Management:

- brk(): Sets the end of the data segment to the specified value.
- mmap(): Maps files or devices into memory.
- munmap(): Unmaps files or devices from memory.

4. Interprocess Communication:

-pipe(): Creates a pipe, a unidirectional data channel that can be used for interprocess communication.

-shmget(), shmat(), shmdt(): Functions for shared memory management.

5.Signal Handling:

- signal(): Sets a signal handler for a particular signal.
- kill(): Sends a signal to a process.

The kernel uses system calls to facilitate communication between user-space processes and the operating system. When a process invokes a system call, the kernel validates parameters, executes the requested service, manages system resources, and returns the result to the calling process. System calls provide a controlled mechanism for accessing privileged operations and essential operating system services.

9) What is a pipe? What is a filter? Give ex/ appli of pipe and filter

Ans : A pipe is a form of inter-process communication (IPC) that allows the output of one process to be connected directly as the input to another process. In Unix-like operating systems, a pipe is represented by the "|" symbol and is used to create a temporary communication channel between two processes.

A filter is a program or command that processes input data from a source and produces output data that has been modified or transformed in some way. Filters are typically used in conjunction with pipes to create a data processing pipeline where the output of one filter serves as the input to the next.

Example of Pipe and Filter:

Let's say we have a text file containing a list of numbers, and we want to perform some operations on these numbers:

1. Using a Pipe:
   - We can use the "cat" command to read the contents of the file and then use a pipe to pass the output to another command, such as "grep" to filter specific numbers. For example:
cat numbers.txt | grep "even"

   - In this example, the "cat" command reads the contents of the "numbers.txt" file, and the pipe "|" sends this output to the "grep" command, which filters only the even numbers.

2. Using Filters:
   - We can use multiple commands as filters in a pipeline to perform various operations on the data. For example:

cat numbers.txt | grep "even" | sort | uniq

   - In this example, the output of the "cat" command is filtered by "grep" to select only even numbers. The "sort" command then sorts these numbers, and the "uniq" command removes duplicate entries.

In both cases, pipes are used to connect the output of one command to the input of another command, creating a data processing pipeline. Filters are then applied to the data at each

stage of the pipeline to perform specific operations, such as filtering, sorting, or removing duplicates.

Module 5 -> The structure of a process

1) What are the system level context of the process static and dynamic parts

Ans :

The system-level context of a process has a "static part"and a "dynamic part". A process has one static part of the system-level context throughout its lifetime, but it can have a variable number of dynamic parts. The dynamic part of the system-level context should be viewed as a stack of context layers that the kernel pushes and pops on occurrence of various events.
Static part of system level context of process:

• The process table entry of a process defines the state of a process, and contains control information that is always accessible to the kernel

• The u area of a process contains process control information that need be accessed only in the context of the process. General control parameters such as the process priority are stored in the process table because they must be accessed outside the process context.

• Pregion entries, region tables and page tables, define the mapping from virtual to physical addresses and therefore define the text, data, stack, and other regions of a process. If several processes share common regions, the regions are considered part of the context of each process,

Dynamic part of system level context of a process :

•The kernel stack contains the stack frames the kernel functions. Even if all processes share the kernel text and data, kernel stack needs to be different for all processes as

every process might be in a different state depending on the system calls it executes. The pointer to the kernel stack is usually stored in the u-area but it differs according to system implementations. The kernel stack is empty when the process executes in user mode

•The dynamic part of the system level context consists of a set of layers, visualized as a last-in-first-out stack. Each system level context layer contains information necessary to recover the previous layer, including register context of the previous layer.

The kernel pushes a context layer when an interrupt occurs, when a process makes a system call, or when a process does a context switch. It pops a context layer when the kernel returns from handling an interrupt, when a process returns to user mode after the kernel completes execution of a system call, or when a process does a context switch.

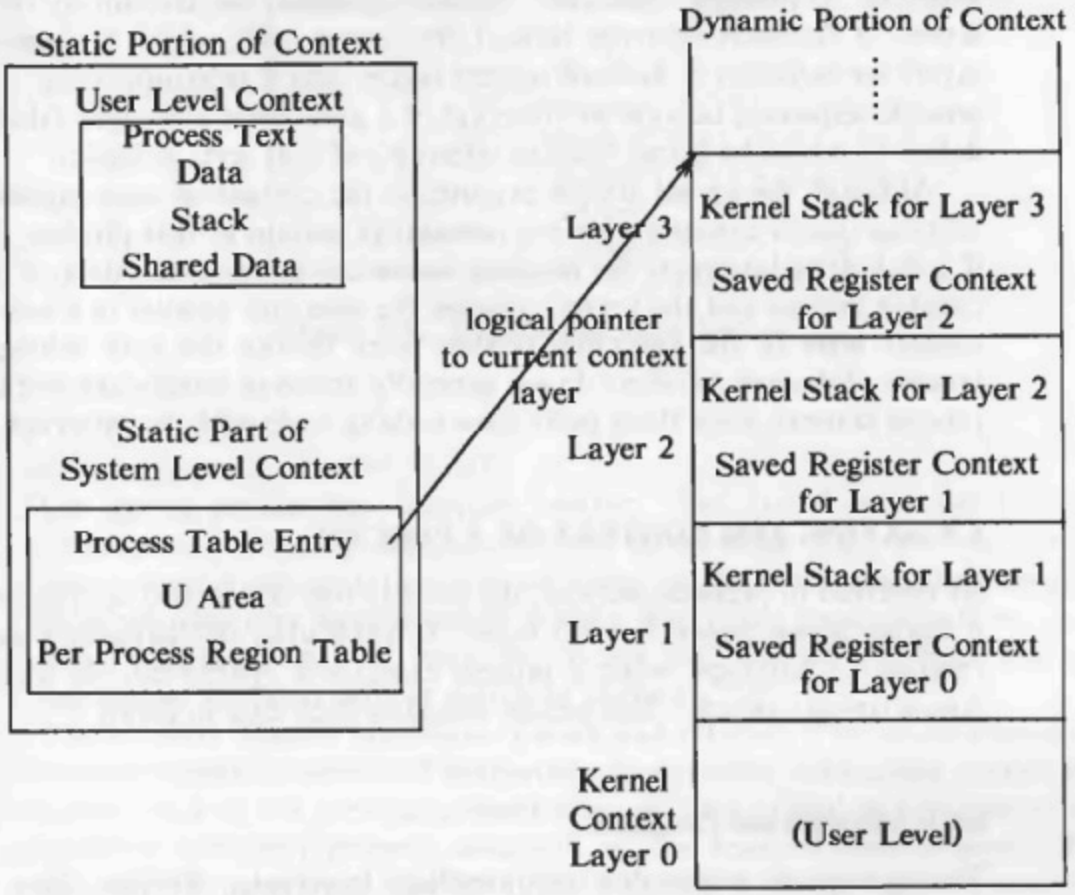


Figure 6.8. Components of the Context of a Process

2) state Transition Diagram:

Ans :

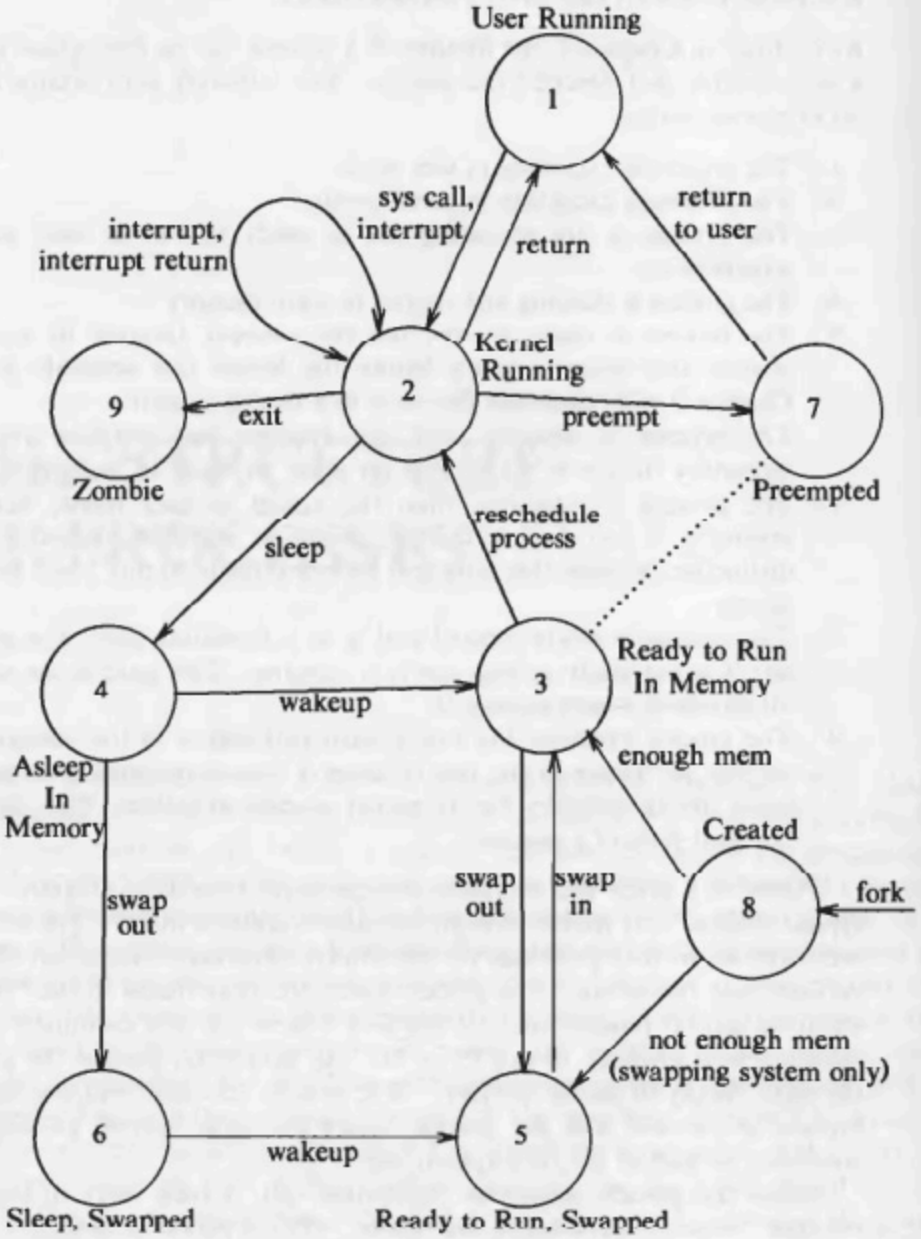


Figure 6.1. Process State Transition Diagram

3) Is the management of regions of process is similar to management of inodes in kernel if yes why is it similar to inode management

Ans:

The management of regions of process is similar to the management of inodes in the kernel in some ways. In both cases, there is a need to keep track of and manage various resources or entities (such as memory regions or files) in an organized and efficient manner.

Inodes are data structures in a file system that contain metadata about files, such as file permissions, ownership, and pointers to the actual data blocks on disk. The kernel manages these inodes to keep track of files and their associated information.

Similarly, in a process, there are different regions of memory that need to be managed, such as the stack, heap, and code segments. The operating system must keep track of these regions and allocate and deallocate memory as needed to ensure that the process runs smoothly.

.The management of regions of process and inodes in the kernel is similar because both involve the **organization and tracking of resources in a structured manner**. In both cases, there is a need to allocate and deallocate resources efficiently, keep track of metadata associated with those resources, and ensure that the resources are accessed and managed in a secure and reliable manner..

Additionally, both process regions and inodes are fundamental components of the operating system that play a crucial role in the execution and management of processes and files, respectively. The operating system needs to manage these resources effectively to ensure the stability and performance of the system as a whole.

Overall, the management of regions of process and inodes in the kernel share similarities in terms of resource organization, allocation, metadata management, and their critical role in the operation of the operating system.

4) What are the advantages to kernel in maintaining the U area in the system?

Ans: Kernel gets to know information about the running processes 1

. 1. pointer to process table

2.Times(time spent by the process in user and kernel mode)

3.Signal handling array

4.real and user effective IDS

5.limit

6.permission field

For 3 marks:

u area Contains following fields

1. a pointer to the process table identifies the entry that Corresponds to the u area
2. Real & effective user IDe : These IDs determine what actions the process is allowed to take like processing files.
3. Timer fields: These keep track of how much time the process Spends running in different modes (user mode and kernel mode)
4. Signal handling Array: How the process wishes to react to signals
5. Control terminal field: If there's a login terminal Connected to the process this field identifies it
6. error field: record errors encountered during System Call
7. return value field: result of a system call
8. I/O parameters: details like how much data to transfer, where it is located in memory
9. Current directory & Current root : describe file system environment of the process

5) When attaching the region to Process ,how can the kernel check that the region does not overlap virtual addresses in regions already attached to the Process?

Ans :

When attaching a region to a process, the kernel can check that the region does not overlap virtual addresses in regions already attached to the process by performing the following steps:

1. The kernel can iterate through the existing regions that are already attached to the process and check the virtual addresses of each region.

2. It can then compare the virtual address range of the new region being attached with the virtual address ranges of the existing regions to ensure that there is no overlap.

3. If there is any overlap detected between the virtual address range of the new region and the virtual address ranges of the existing regions, the kernel can reject the attachment of the new region to the process.

By carefully checking and comparing the virtual address ranges of the regions already attached to the process with the virtual address range of the new region being attached, the kernel can ensure that there are no overlaps and maintain the integrity and consistency of the process's memory layout.

6) Which kernel data structure describes the state of a process?

Ans: Process table entry and U area

7) Suppose a process goes to sleep and the system contain no processes ready to run .What happen when the sleeping process does its context switch

Ans:

When a sleeping process does its context switch and there are no other processes ready to run, the operating system will typically switch to a special idle process or scheduler process. This idle process will simply wait until a new process becomes ready to run, at which point it will switch to that process. This ensures that the CPU is always being utilized and that there is always a process running on the system.

8)What are the system calls that support the processing environment in Kernel?

System calls are essential for interfacing between user-level processes and the kernel in an operating system. They provide a means for user-level processes to request services from the kernel. Some of the system calls that support the processing environment in the kernel include:

1. Process Control:
   - fork(): Creates a new process by duplicating the calling process.
   - exec(): Loads a new program into the current process's memory space, replacing the existing program.
   - exit(): Terminates the calling process and returns its resources to the system.

2. File Management:
   - open(): Opens a file and returns a file descriptor.
   - read(): Reads data from a file.
   - write(): Writes data to a file.
   - close(): Closes an open file descriptor.

3. Memory Management:
   - brk(): Sets the end of the data segment to the specified value.
   - mmap(): Maps files or devices into memory.
   - munmap(): Unmaps files or devices from memory.

4. Interprocess Communication:

-pipe(): Creates a pipe, a unidirectional data channel that can be used for interprocess communication.

-shmget(), shmat(), shmdt(): Functions for shared memory management.

5.Signal Handling:

- signal(): Sets a signal handler for a particular signal.
- kill(): Sends a signal to a process.

9) **Which four circumstances under which kernel permits Context Switch?**

Ans: The Context switching is a technique or method used by the operating system to switch a process from one state to another to execute its function using CPUs in the system.

1. When a process voluntarily yields the CPU: This can happen when a process finishes its time slice or when it explicitly calls a function to yield the CPU, such as sleep() or wait().

2. When a process is blocked: If a process is waiting for an external event, such as I/O operation completion or a signal from another process, the kernel will perform a context switch to allow another process to run while the first process is blocked.

3. When a higher priority process becomes ready: If a higher priority process becomes ready to run, the kernel will perform a context switch to switch to the higher priority process and ensure that it gets CPU time.

4. When a process's time slice expires: In a preemptive scheduling system, each process is given a time slice or quantum to execute. When a process's time slice expires, the kernel will perform a context switch to switch to another process and ensure fair CPU allocation among all processes.

(Acc to book:

the kernel permits a context switch under 4 situations:

1. When a process sleeps
2. When a process exits
3. When a process returns from a system call to user mode but is not the most eligible process to run.
4. When a process returns from an interrupt handler to user mode but is not the most eligible process to run.

)

Module 6: Process Control

1. **Whether signal handling is optional to a process, if yes/no justify? Which system call is used to make it optional?**

   Ans: Yes, signal handling is optional for a process.

   Here's why:

a. By default, most signals cause a process to terminate abnormally.
b. You can choose to handle specific signals by providing a custom signal handler function. This function decides how the process reacts to the signal.
c. If you don't define a signal handler, the default behavior takes effect.

There are three cases for handling signals: the process *exit*s on receipt of the signal, it ignores the signal, or it executes a particular (user) function on receipt of the signal. The default action is to call *exit* in kernel mode, but a process can specify special action to take on receipt of certain signals with the *signal* system call.

old function = signal (signum, function);

where signum is the signal number the process is specifying the action for, function is the address of the (user) function the process wants to invoke on receipt of the signal. The process can pass the values 1 or 0 instead of a function address: The process will ignore future occurrences of the signal if the parameter value is 1 and *exit* in the kernel on receipt of the signal if its value is 0 (default value). The u-area contains an array of signal-handler fields, one for each signal defined in the system. The kernel stores the address of the user function in the field that corresponds to the signal number.

In Unix, the signal() system call as well as sigaction() is indeed used to establish signal handling behavior within a process.

2. **What information does wait find when the child process invokes exit without a parameter? That is, the child process calls exit() instead of exit(n).**

Ans: When a child process invokes exit() without a parameter in Unix-like systems, the information retrieved by the parent process using wait() includes:

Exit Status: Even though no explicit parameter is provided to exit(), the child process exits with a default status code of **0**, which indicates successful termination. The wait() system call can be used to retrieve this exit status.

Here's what wait() retrieves in different scenarios:

1. If the child process terminates normally without explicitly providing an exit status using exit(), wait() retrieves a termination status indicating that the child process exited normally.
2. If the child process is terminated due to a signal, wait() retrieves a termination status indicating that the child process was terminated by a signal.

3. **Draw the nine state diagram and show when the signals are handled and checked? Why are signals handled and checked on those transitions only?**

Ans:

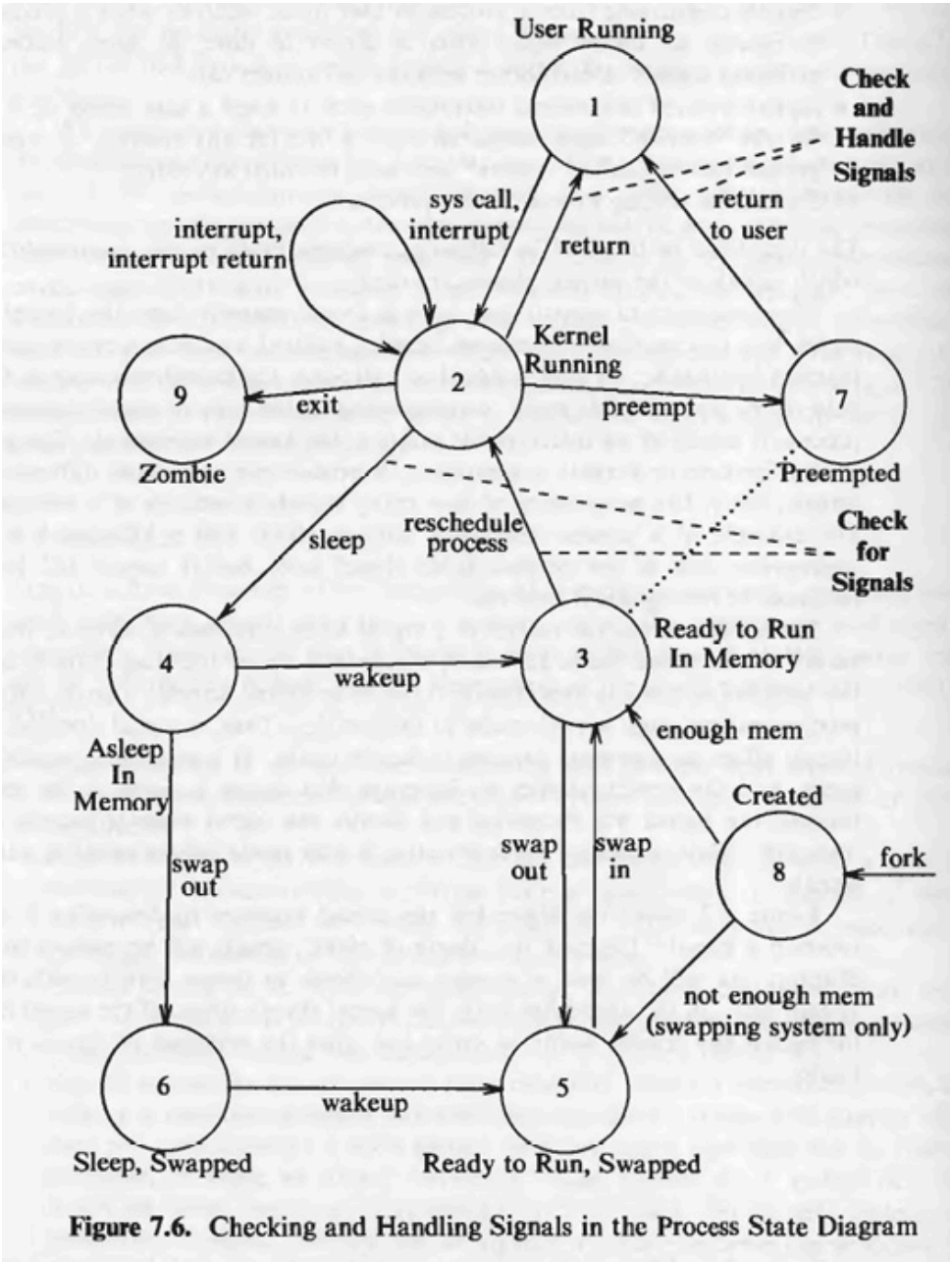


Figure 7.6. Checking and Handling Signals in the Process State Diagram

The kernel checks for receipt of a signal when a process is about to return from kernel mode to user mode and when it enters or leaves the sleep state at a suitably low scheduling priority (see Figure 7.6). The kernel handles signals only when a process returns from kernel mode to user mode. Thus, a signal does not have an instant effect on a process running in kernel mode. If a process is running in user mode, and the kernel handles an interrupt that causes a signal to be sent to the process, the kernel will recognize and handle the signal when it returns from the interrupt. Thus, a process never executes in user mode before handling outstanding signals.

Reasoning:

- Checking signals at state transitions allows the system to find a safe point to interrupt the process without corrupting its state or ongoing operations.
- Blocking states are natural points to check for signals because the process is waiting and less susceptible to disruption.
- Ignoring signals while running helps maintain program flow and avoids unexpected behavior.

4. **What are the functions (algorithm) carried out by exit system call(process termination). Why is the exit system call designed in Kernel for process termination?**

Ans: Processes on the UNIX system exit by executing the *exit* system call. When a process *exit*s, it enters the zombie state, releases all of its resources, and dismantles its context except for its process table entry.

exit (status);

where status is the exit code returned to the parent. The kernel may call *exit* on receiving an uncaught signal. In such cases, the value of *status* is the signal number.

The algorithm for exit is given below:

{

ignore all signals;

```
        if (process group leader with associated control terminal)
        {
                send hangup signal to all members of the process group;
                reset process group for all members to 0;
        }
        close all open files (internal version of algorithm close);
        release current directory (algorithm: iput);
        release current (changed) root, if exists (algorithm: iput);
        free regions, memory associated with process (algorithm: freereg);
        write accounting record;
        make process state zombie;
        assign parent process ID for all child processes to be init process (1);
                if any children were zombie, send death of child signal to init;
        send death of child signal to parent process;
        context switch;
}
```

The `exit` system call is responsible for terminating a process in an operating system. When a process makes an `exit` system call, several functions or steps are typically carried out to ensure a clean and orderly termination. Here is an outline of the functions or algorithm commonly associated with the `exit` system call:

1. .Cleanup and Resource Reclamation:.

- Close open file descriptors: The process closes any files or resources it has opened during its execution. This prevents resource leaks and ensures that files are properly closed before termination.

- Release allocated memory: The process frees any memory it has allocated dynamically during its lifetime. This memory may include heap memory, stack memory, or other resources allocated by the process.

- Release system resources: The process releases any other system resources it has acquired, such as semaphores, mutexes, or network connections.

2. .Exit Status and Signal Handling:.

- Set exit status: The process sets an exit status code to indicate the reason for its termination. This status code can be retrieved by the parent process or by querying the process status.

- Signal handlers: If the process has registered signal handlers for specific signals (e.g., SIGTERM, SIGINT), these handlers may be invoked to perform cleanup or handle the signal before termination.

3. .Process State Update:.

- Update process state: The operating system updates the process control block (PCB) or equivalent data structure to mark the process as terminated. This includes updating process status flags and releasing any resources held by the operating system for the process.

4. .Parent Process Notification:.

- Notify parent process: If the process was created by another process (its parent), the operating system notifies the parent process about the termination of its child. This may involve sending a signal (e.g., SIGCHLD) or updating parent process data structures.

5. .Termination Cleanup:.

- Perform final cleanup: Any additional cleanup tasks specific to the operating system or process environment are performed before the process is completely removed from the system.

- Remove process from process table: The operating system removes the process entry from the process table or similar data structure, freeing up resources associated with the terminated process.

6. .Return Control to Operating System:.

- Return control: After completing the termination process, control is returned to the operating system scheduler or dispatcher, which may schedule another process for execution.

Overall, the `exit` system call ensures that a process terminates gracefully, releases resources, notifies relevant parties, and updates system state accordingly.

why the exit() system call is designed in the kernel for process termination:

1. Control Over Resource Cleanup: By handling process termination in the kernel, the operating system can ensure proper cleanup of these resources to prevent resource leaks and system instability.
2. Consistency and Reliability: Centralizing process termination in the kernel ensures that termination operations are performed consistently across all processes. This helps maintain system reliability and prevents inconsistencies that could arise from processes terminating in different ways.
3. Inter-Process Communication:. By handling this communication in the kernel, the operating system can ensure proper synchronization and coordination between processes.
4. Security and Privilege Separation: Process termination may involve privileged operations, such as releasing system resources or signaling other processes. By handling process termination in the kernel, the operating system can enforce security policies and ensure that only authorized operations are performed during termination.

5. **How many signals are there in system V UNIX? Give the correspondence between PID and set of processes/processes in the kill system call for sending the signal?**

Ans: There are 19 signals in the System V (Release 2) UNIX system that can be classified as follows:

a. Signals having to do with the termination of a process, sent when a process *exit*s or when a process invokes the *signal* system call with the *death of child* parameter.
b. Signals having to do with process induced exceptions such as when a process accesses an address outside its virtual address space,etc
c. Signals having to do with the unrecoverable conditions during a system call, such as running out of system resources during *exec* after the original address space has been released
d. Signals caused by an unexpected error condition during a system call, such as making a nonexistent system call , writing a pipe that has no reader processes,etc.
e. Signals originating from a process in user mode, such as when a process wishes to receive an *alarm* signal after a period of time, or when processes send arbitrary signals to each other with the *kill* system call.
f. Signals related to terminal interaction such as when a user hands up a terminal (or the "carrier" signal drops on such a line for any reason), or when a user presses the "break" or "delete" keys on a terminal keyboard.
g. Signals for tracing execution of a process.

Processes use the *kill* system call to send signals.

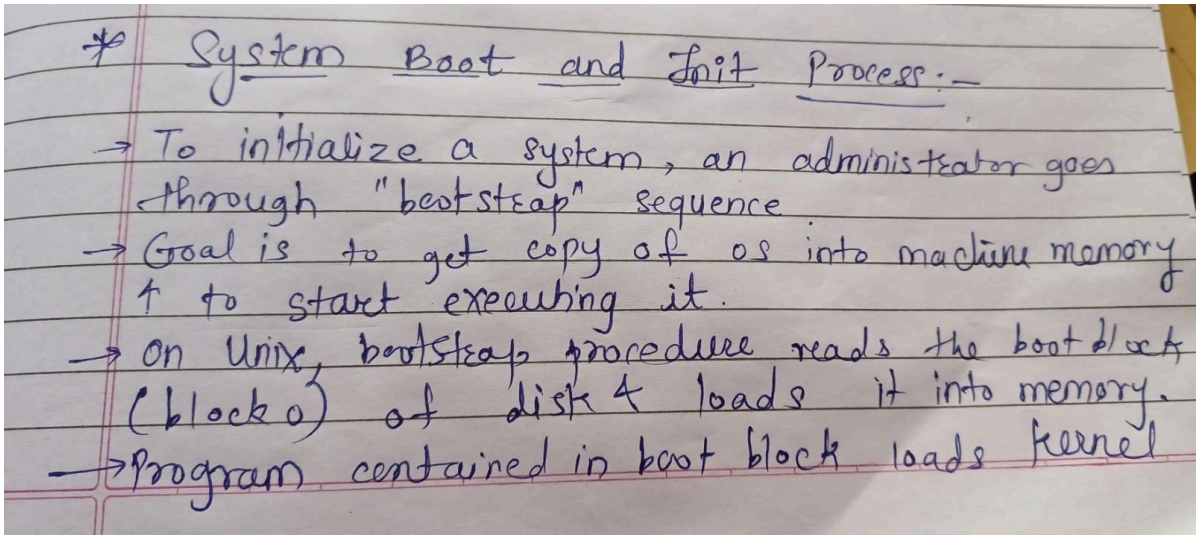
```
kill (pid, signum);
```

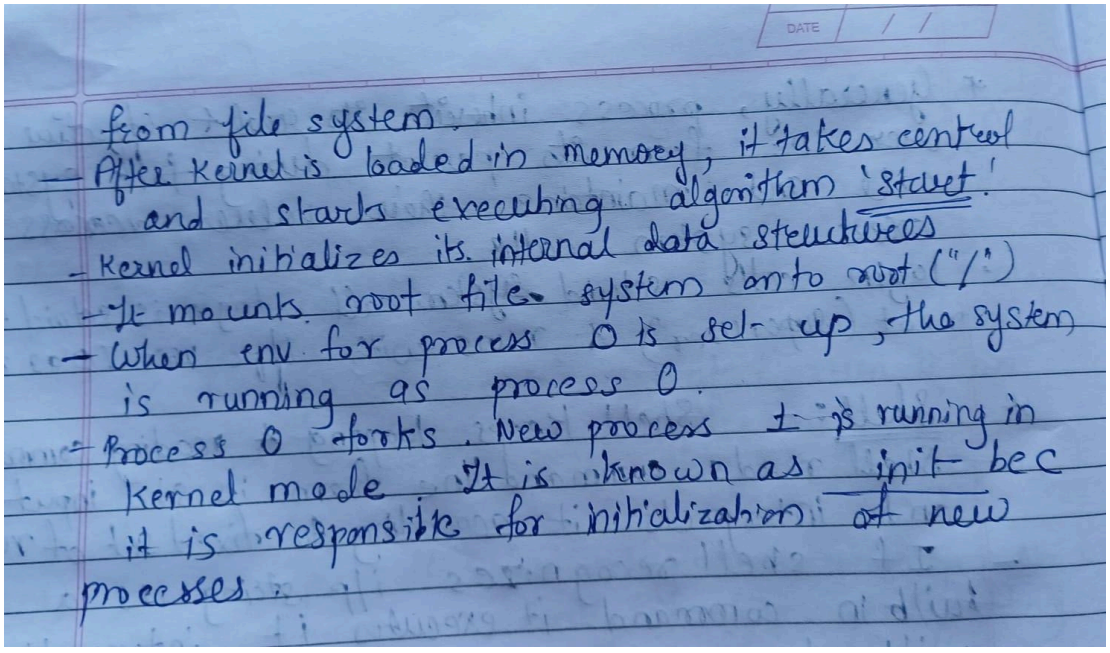
where pid identifies the set of processes to receive the signal, and signum is the signal number being sent. The following list shows the correspondence between values of *pid* and sets of processes.

- If *pid* is a positive integer, the kernel sends the signal to the process with process ID *pid*.
- If *pid* is 0, the kernel sends the signal to all processes in the sender's process group.
- If *pid* is -1, the kernel sends the signal to all processes whose real user ID equals the effective user ID of the sender. If the sending process has an effective user ID of superuser, the kernel sends the signal to all processes except processes 0 and 1.
- If *pid* is a negative integer but not -1, the kernel sends the signal to all processes in the process group equal to the absolute value of *pid*

6. **Explain the system boot steps or (Start) algorithm**

Ans:





Algorithm start:

{

initialize all kernel data structures;

pseudo-mount of root;

```
        hand-craft environment of process 0;
        fork process 1:
        {
                // process 1 here
                allocate region;
                attach region to init address space;
                grow region to accommodate code about to copy in;
                copy code from kernel space to init user space to exec init;
                change mode: return from kernel to user mode;
                /*  init never gets here---as result of above change mode,
                 *  init exec's /etc/init and becomes a "normal" user process
                 *  with respect to invocation of system calls
                 */
        }
        // proc 0 continues here
        for kernel processes;
        /*  process 0 invokes the swapper to manage the allocation of
         *  process address space to main memory and the swap devices.
         *  This is an infinite loop; process 0 usually sleeps in the
         *  loop unless there is work for it to do.
         */
        execute code for swapper algorithm;
}
```

7. **Why is there a need for a wait system call between parent and child?**

   Ans: The wait system call in Unix-like systems serves several crucial purposes in the interaction between parent and child processes:

   1. **Reaping Zombie Processes:**

- When a child process exits using exit() or terminates abnormally, it becomes a "zombie" process. This process is mostly defunct, but it still occupies an entry in the process table until the parent calls wait().
- wait() allows the parent process to acknowledge the child's termination and reclaim the resources associated with the zombie process. This prevents the process table from becoming cluttered with defunct processes and helps maintain system efficiency.

2. **Collecting Exit Status:**

   - The child process can indicate its reason for termination by setting an exit status using exit(n). This status code is often used to convey success (exit(0)) or various error conditions (exit(n) where n represents a specific error).
   - wait() can optionally retrieve the child's exit status. This information allows the parent to understand how the child process terminated and take appropriate actions. For example, the parent might launch a new child process if the previous one failed.

3. **Synchronization:**

   - In some scenarios, a parent process might need to wait for its child process to finish a specific task before proceeding. wait() can be used as a synchronization mechanism. The parent calls wait(), effectively blocking itself until the child terminates. Once the child exits, wait() returns, allowing the parent to continue execution.

4. **Error Handling:**

   - If the child process terminates abnormally or the parent fails to call wait(), the system might eventually send a special signal (SIGCHLD) to the parent. This can be used as an error handling

mechanism, notifying the parent that the child process might have encountered problems.

In summary, wait() acts as a bridge for communication and resource management between parent and child processes. It allows the parent to:

Clean up zombie processes.

Retrieve the child's exit status for informed decision making.

Synchronize execution with the child process.

Handle potential errors during child process termination.

8. **Enlist the system calls used in process control.**

Ans: Here are some of the most common system calls used in process control in Unix-like systems:

fork(): Creates a new process that is a nearly identical copy of the calling process. Both the parent and child process share the same memory space initially, but the child process receives its own unique PID (process ID).

wait() / waitpid(): These system calls allow a parent process to wait for the termination of one or more of its child processes. They also retrieve information about the terminated child process, such as its exit status. **waitpid()** offers more flexibility compared to **wait()**.

exit(): This system call terminates the calling process. It performs necessary cleanup tasks like closing open file descriptors and releasing memory before the process exits. It also sends a notification signal (SIGCHLD) to the parent process (if it exists).

execve() / execvp() / execv() / execl() / execlp(): These system calls are used to execute a new program within the address space of the calling process. Essentially, the calling process is replaced by the new program. The specific variations differ in how they specify the program to be executed.

getpid(): This system call retrieves the process ID (PID) of the calling process.

getppid(): This system call retrieves the parent process ID (PPID) of the calling process.

setsid(): This system call allows a process to create a new session and become the session leader. It's useful for detaching a process from its controlling terminal.

kill(): This system call sends a signal to one or more processes. Signals can be used to terminate processes, pause them, or notify them of specific events.

sigaction() / signal(): These system calls allow a process to define how it wants to handle specific signals. A process can choose to ignore a signal, execute a custom signal handler function, or use the default behavior.

9. State various functions of clock interrupt handler

Ans:

The functions of the clock interrupt handler are to:

- restart the clock
- schedule invocation of internal kernel functions based on internal timers

- provide execution profiling capability for the kernel and for user processes
- gather system and process accounting statistics,
- keep track of time
- send alarm signals to processes on request
- periodically wake up the swapper process
- control process scheduling

10. **Which system call is used to send the signal?**

Ans: Processes use the *kill* system call to send signals.

kill (pid, signum);

where pid identifies the set of processes to receive the signal, and signum is the signal number being sent.

11. Define zombie state. Why is it designed in the life cycle of a process?

Ans: In Unix-like operating systems, a zombie state refers to a state that a process enters after it has terminated, but its parent process has not yet collected its exit status through the wait() system call. The zombie state is the final state of a process.

Zombie processes still occupy an entry in the process table but do not consume any system resources other than the process table entry and a small amount of kernel memory.

The existence of the zombie state in the lifecycle of a process serves several purposes:

a. Exit Status Collection: When a process terminates, its parent process typically needs to collect its exit status to determine the outcome of its execution. By entering the zombie state, the terminated process can retain its exit status until the parent process

collects it using the wait() system call. This allows the parent process to handle the termination of its child processes properly.

b. Prevention of Resource Leaks: If a parent process fails to collect the exit status of its terminated child processes, those processes would remain in the system indefinitely, consuming system resources. By entering the zombie state, terminated processes indicate to the system that they have completed execution but are waiting for their exit status to be collected. This prevents resource leaks and helps maintain system stability.
c. Signaling Parent Process: When a process terminates, it may have important information or resources that need to be communicated to its parent process. By entering the zombie state, the terminated process signals to its parent process that it has terminated and is waiting for its exit status to be collected. This allows the parent process to perform any necessary cleanup or handle the termination event appropriately.
d. Process Accounting: Even though they have terminated, their presence in the process table indicates that they were once active processes in the system. This information can be useful for system administrators and monitoring tools to track process activity and resource usage over time.

12. **When a process terminates, the kernel performs clean-up, assigns any children of the existing process to be adopted by inti, and sends the death of a child signal to the parent process. Why?In what state the init process is? / What is the orphan process? Who is the parent of the orphan process? Why?**

    Ans: If the terminated process had any child processes that were still running, the kernel assigns these orphaned child processes to be adopted by the init process. This ensures that orphaned processes have a parent process responsible for their cleanup and prevents them from becoming zombies, which could consume system resources indefinitely. Init will eventually perform the wait system call for these orphans so they can die. The init process, in its **running** state, acts as a safety net by adopting orphaned children and handling their termination gracefully.

13. **What happens to a situation where the parent itself dies before the child dies and what Kernel does with the such process?**

    Ans: If the parent itself dies before the child, the kernel disconnects the parent from the process tree by making process 1 (init) adopt all its child processes. That is, process 1 becomes the legal parent of all live children that the existing process had created. If any of the children are zombies, the existing process sends init a "death of child" signal so that init can remove them from the process table.

14. **What is a socket ? Which System call is responsible for binding port and process**

    Ans : A socket is a communication endpoint that allows processes to communicate with each other over a network. In Unix-like operating systems, sockets are widely used for inter-process communication (IPC) and network communication.

There are two main types of sockets:

1. Stream Sockets (TCP): Stream sockets provide a reliable, connection-oriented communication channel between processes. They use the Transmission Control Protocol (TCP) for communication, which ensures that data is delivered in the correct order and without errors.
2. Datagram Sockets (UDP): Datagram sockets provide a connectionless, unreliable communication channel between processes. They use the User Datagram Protocol (UDP) for communication, which does not guarantee the order or reliability of data delivery.

Sockets are identified by an address, which consists of an IP address and a port number. The combination of IP address and port number uniquely identifies a socket and determines the source and destination of data communication.

In Unix, the system call responsible for binding a port and process to a socket is the bind() system call. The bind() system call is used to associate a socket with a specific network address (IP address) and port number on the local machine. This allows a process to listen for incoming connections or send data from a specific port.

The bind() system call has the following syntax:

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

- sockfd: The file descriptor referring to the socket to be bound.
- addr: A pointer to a struct sockaddr structure containing the address information (IP address and port number) to bind to the socket.
- addrlen: The size of the address structure.

By using the bind() system call, a process can specify the port on which it will listen for incoming connections (in the case of a server) or from which it will send data (in the case of a client).

Module 3: Internal representation of file structure

1. **Why there is per user or per process file descriptor table?**

Ans: the file descriptor table is a per-process data structure that maintains information about open files and other I/O resources associated with a process.

Entries in the user file descriptor table maintain the state of the file and the user's access to it.The user file descriptor table identifies all open files for a process.The kernel returns a file descriptor for the open and creat system calls, which is an index into the user file descriptor table. Keeping all these records of all the processes in a single global data structure is not possible hence, user file descriptor table is allocated per process.

There are several key reasons why Unix-like systems utilize a per-process file descriptor table instead of a single, system-wide table:

Resource Management and Isolation:

a. Each process has its own set of open files and resources. This ensures that processes don't interfere with each other's open files and helps prevent resource leaks or conflicts.

Security:

b. Processes cannot directly access or manipulate another process's file descriptors. This isolation helps protect the system and user data. A malicious process wouldn't be able to read or write to files opened by another process without proper authorization.

Flexibility:

c. Each process can have a different number of open files depending on its needs. A per-process table allows for this flexibility, as processes don't have to share a limited system-wide pool of file descriptors.

Efficiency:

d. Looking up a file descriptor in a per-process table is generally faster than searching a single, system-wide table, especially for systems with many running processes. This improves performance for file I/O operations.

User-Level Abstraction:

e. The file descriptor table provides a simple integer-based abstraction for processes to interact with open files. Processes don't need to deal with complex file system details; they just use file descriptors for operations like reading, writing, and closing.

2. Bases on 13 entries of table of content for regular file, What is max size of file a file system support?

Ans: If a logical block on the file system holds 1K bytes and that a block number is addressable by a 32 bit integer, then a block can hold up to 256 block numbers. The maximum file size with 13 member data array is:

10 direct blocks with 1K bytes each = 10K bytes

1 indirect block with 256 direct blocks = 256K bytes

1 double indirect block with 256 indirect blocks = 64M bytes

1 triple indirect block with 256 double indirect blocks = **16G bytes**

But the **file size field in the inode is 32 bits**, the size of a file is effectively limited to **4 gigabytes**.

To understand this deeply, go to: https://github.com/suvratapte/Maurice-Bach-Notes/blob/master/4-Internal-Representation-of-Files.md , in structure of regular file section.

3. **Which Block which helps to find Max number of files /directories user can create. Block which gives info about state of file system. Which block is responsible for loading multiple os?**

Ans: Max number of files /directories: superblock. Explanation khalachach yeil

info about state of file system: Superblock:

The superblock is a data structure that contains essential information about the file system, such as its size, block size, inode count, free block count, and free inode count. It also includes information about the file system's state, such as its mount status and the last time it was mounted or modified.

The superblock's information, such as the number of free inodes and free blocks, can be used to determine the current state of the file system, including the amount of available space and the potential for creating new files and directories.

For loading multiple os: Boot Block: Unix systems typically rely on a boot loader residing in a separate partition (often the first sector) of the disk. This program is responsible for loading the kernel of the chosen operating system from the appropriate partition.

4. **With diagram, show that how many direct, single indirect, double indirect and triple indirect blocks are required for file size of 3,50,000 Bytes. (1 logical block-2K bytes, Block number address: a 64 bit (8byte) integer)**

What is maximum size of file a file system support? What is maximum number of files a file system can contain?

Ans: 10 direct + 161 direct = 171 direct blocks and 1 single indirect block

Can refer to https://github.com/suvratapte/Maurice-Bach-Notes/blob/master/4-Internal-Representation-of-Files.md , in structure of regular file section.

(or you can ask Shriya 🙂)

Ans :- first 10 blocks + 161
= 171 direct blocks needed + 1 single indirect block

1B = 8bits

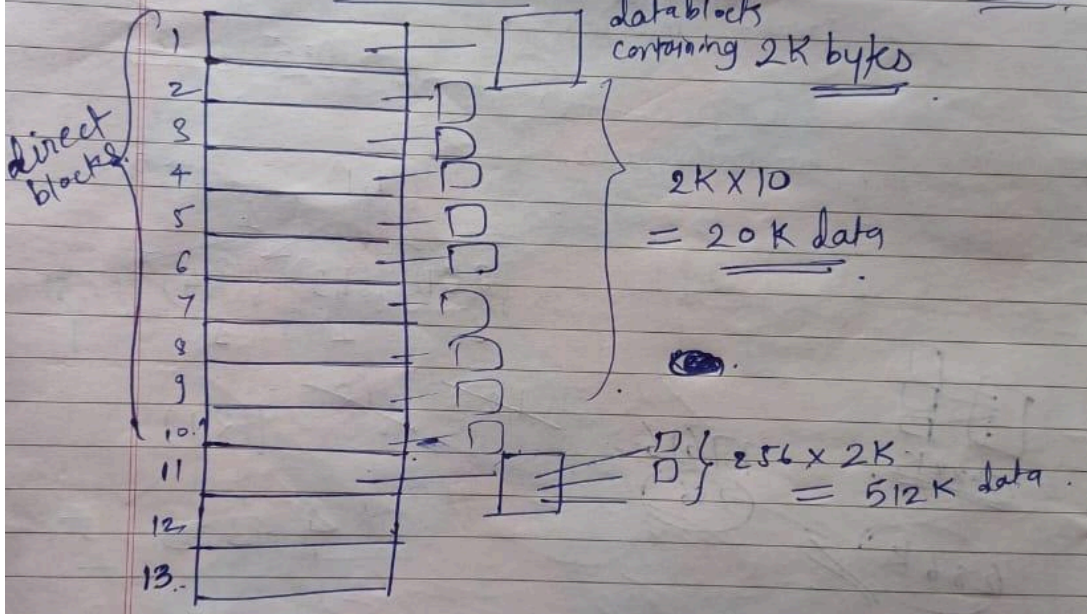
3,50,000 B

1 Block → 2K Bytes 1 add = 64 bit (8 byte)

∴ In one block, total no. of entries = $\frac{\text{size}}{\text{size per entry}} = \frac{2 \times 1024}{8} = \frac{2048}{8}$

= 256

∴ 256 entries



for first 10 Direct block → 20K data.

Remaining → 350000 − 20480 = 329520 B

for single indirect :- 256 × 2K = 512K = 524288 B

∴ Rem. data = 329520 < 512 K.

∴ no need of using double/ triple indirect block.

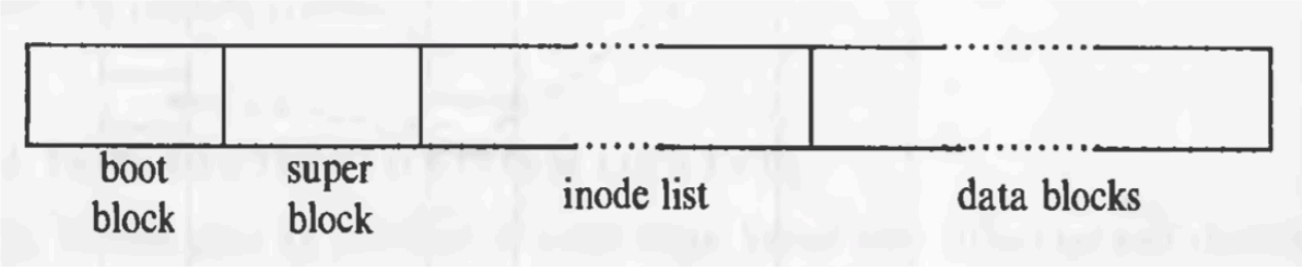
we can store data till single indirect blocks.

- no. of direct blocks needed from single indirect = 329520/2048 = 160.89 ≈ 161.

In 161th block → 329520 % 2048 = 1840th offset

5. **Draw and elaborate the blocks of file system layout. Which block helps to find the maximum number of files/directory users can create? Which block gives information about the state of the file system?**

   Ans:



A file system has the following structure:

- The *boot block* occupies the beginning of a file system, typically the first sector, and may contain the *bootstrap* code that is read into the machine to *boot* , or initialize, the operating system. Although only one boot block is needed to boot the system, every file system has a (possibly empty) boot block.
- The *super block* describes the state of a file system -- how large it is, how many files it can store, where to find free space on the file system, and other information.
- The *inode list* is a list of inodes that follows the super block in the file system. Administrators specify the size of the inode list when configuring a file system. The kernel references inodes by index into the inode list. One inode is the *root inode* of the file system: it is the inode by which the directory structure of the file system is accessible after execution of the *mount* system call.
- The data blocks start at the end of the inode list and contain file data and administrative data. An allocated data block can belong to one and only one file in the file system.

6. **What are advantages to kernel in maintaining the superblock in the file system?**

   Ans: The contents of the super block are:

- size of the file system.
- number of free blocks in the file system.
- list of free blocks in the file system.
- pointer to the next free block in the free blocks list
- size of the inodes list.
- number of free inodes in the file system.

- list of free inodes in the file system.
- pointer to the next free inode in the free inodes list.
- lock fields for the free blocks and free inodes list.
- a field indicating whether the super block has changed.

Maintaining the superblock in the file system provides several advantages to the kernel in Unix-like operating systems:

1. File System Initialization: During file system initialization, the kernel reads the superblock to determine the file system's layout and configuration, allowing it to set up data structures and allocate resources accordingly.
2. File System Integrity: The superblock includes information about the file system's state and integrity, such as the last mount time, mount count, and file system state. This information helps the kernel ensure the consistency and integrity of the file system.
3. Allocation and Deallocation of Resources: The superblock tracks the allocation and deallocation of disk blocks and inodes within the file system. This information allows the kernel to manage the allocation of resources efficiently, ensuring that disk space is allocated appropriately and preventing fragmentation.
4. Performance Optimization: The superblock contains parameters that affect the performance of the file system, such as the block size, inode size, and allocation policies. By maintaining this information in the superblock, the kernel can optimize file system performance based on the characteristics of the underlying storage devices and the workload patterns of the system.
5. Mounting and Unmounting: The superblock contains information about the file system's mount status, including whether it is currently mounted and the mount options used. This information is essential for the kernel to mount and unmount file systems correctly, ensuring that file system operations are performed safely and securely.
6. Backup and Recovery: The superblock serves as a critical reference point for backup and recovery operations. Backup utilities and file system checkers use the information stored in the superblock to create backups, verify file system integrity, and recover from errors or corruptions.

7. Which sys calls are used to change contains of inode header?

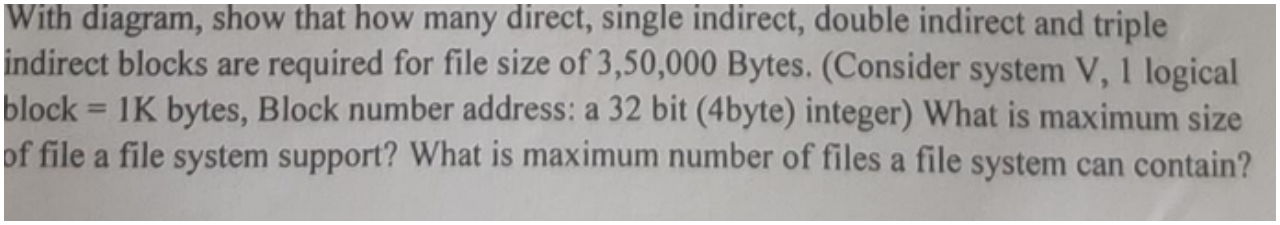
Ans: To change the contents of an inode header, you typically use system calls related to file manipulation and attribute modification

1. open(): This system call is used to open a file. You can specify flags such as `O_RDWR` (read-write) to open the file for both reading and writing.
2. read() and write(): These system calls are used to read from and write to files, respectively. After opening the file, you can use these calls to read the contents of

the inode header into memory, modify it, and then write the modified data back to the file.

3. lseek(): This system call is used to change the current file offset within a file. It allows you to move the file offset to a specific position within the file, which can be useful for positioning the read/write cursor to the beginning of the inode header.
4. chmod: This system call changes the file permissions stored in the inode.
5. chown: Changes the ownership information (user and group) in the inode.

8.


With diagram, show that how many direct, single indirect, double indirect and triple indirect blocks are required for file size of 3,50,000 Bytes. (Consider system V, 1 logical block = 1K bytes, Block number address: a 32 bit (4byte) integer) What is maximum size of file a file system support? What is maximum number of files a file system can contain?

Ans:

1 block → $\frac{1 \times 1024 \times 8}{32}$

= 256 entries

∴ 10 direct blocks

= 10 × 1K = 10K

Remaining = 3,50,000 - 10K

= 339760 bytes

1 single indirect

= 1K × 256 = 256K

Remaining = 339760 − 256K

= 77616 bytes

No. of direct blocks = $\frac{262144}{1024}$

= 256 direct

1 double indirect

= 1K × 256 × 256

= 65536 K bytes > 77616 bytes

with 1 single indirect

= 1K × 256 = 262144 > 77616 bytes

∴ No. of direct blocks = $\frac{77616}{1024} = 75.7 \simeq 76$

∴ No. of direct = 76 + 256 + 10 = 342 direct

2 single indirect

1 double indirect

9.

e)Why there is lock on inode, when there is read or write system calls on a inode but not while opening or creating a file?

Ans: The primary purpose of the inode lock is to ensure data integrity during concurrent read/write access to the same file. By acquiring a lock, the kernel prevents other processes from modifying the file while one process is reading or writing to it. This avoids race conditions and data corruption that could occur if multiple processes try to update the file simultaneously.
Opening a file involves translating the file path into an inode and updating access times. Creating a file requires finding free space, allocating an inode, and setting up the initial data structures. These operations are generally fast and don't require exclusive access to the entire file, so locking the inode wouldn't be as critical here.

10.

iii. When the count field of file table is increased or decreased? Why?
iv. When the count field of i-node table is increased or decreased? Why?
v. What is directory? Why root and parent directory entries are by default in every directory?

Ans:

iii) Count field in the file table:

- Increase: The count field in the file table is typically increased when a new file descriptor is created that refers to the same open file. This can happen, for example, when a process calls `open()` on a file, which increments the count. Subsequent calls to `dup()` or `dup2()` also increment this count, as they create additional file descriptors referring to the same open file.
- Decrease: Conversely, the count field is decreased when a file descriptor is closed. When a process calls `close()` on a file descriptor, the count associated with that file descriptor is decremented. When the count reaches zero, indicating that no more file descriptors are referring to the open file, the operating system can reclaim any resources associated with that open file, such as releasing memory or releasing locks.

iv) Count field in the inode table:

- Increase: The count field in the inode table is typically increased when a new hard link to a file is created. When a hard link is created using `link()` or `ln`, a new directory entry is added pointing to the same inode, and the count in

the inode table is incremented to reflect this additional reference to the inode.

- Decrease: The count in the inode table is decreased when a file is unlinked or removed. When a file is unlinked using `unlink()` or `rm`, the link count associated with the corresponding inode is decremented. When the link count reaches zero, it indicates that no more directory entries point to the inode, and the inode and associated data blocks can be deallocated.

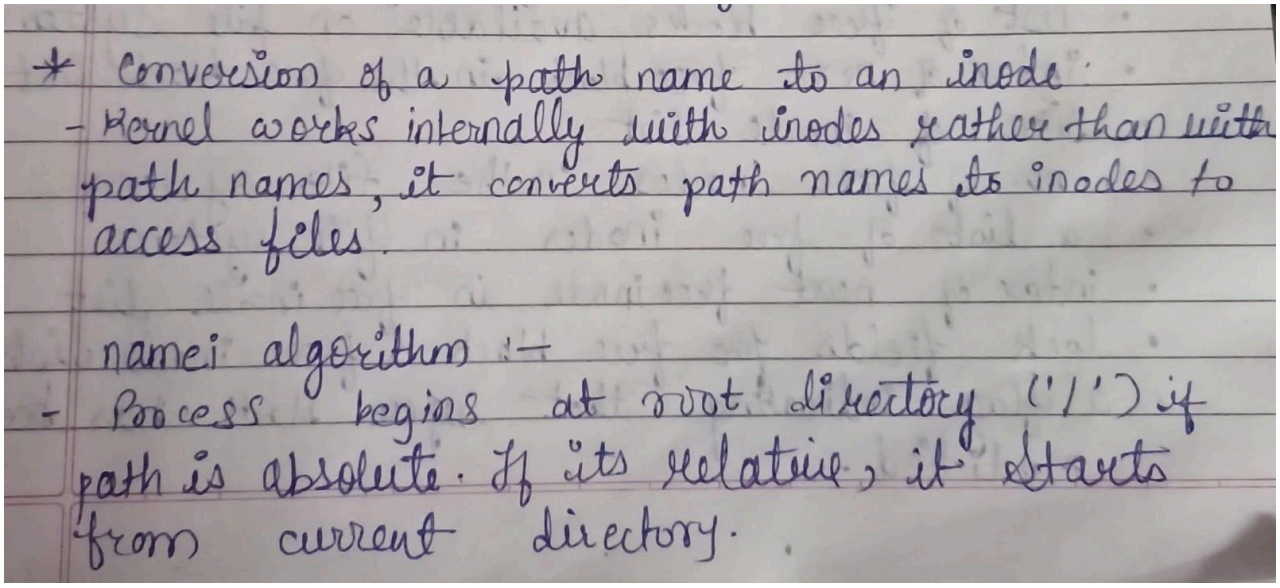
V. A directory, in the context of file systems, is a special type of file that contains a list of file names and their corresponding inode numbers. It serves as a way to organize files into a hierarchical structure for easier navigation and management.

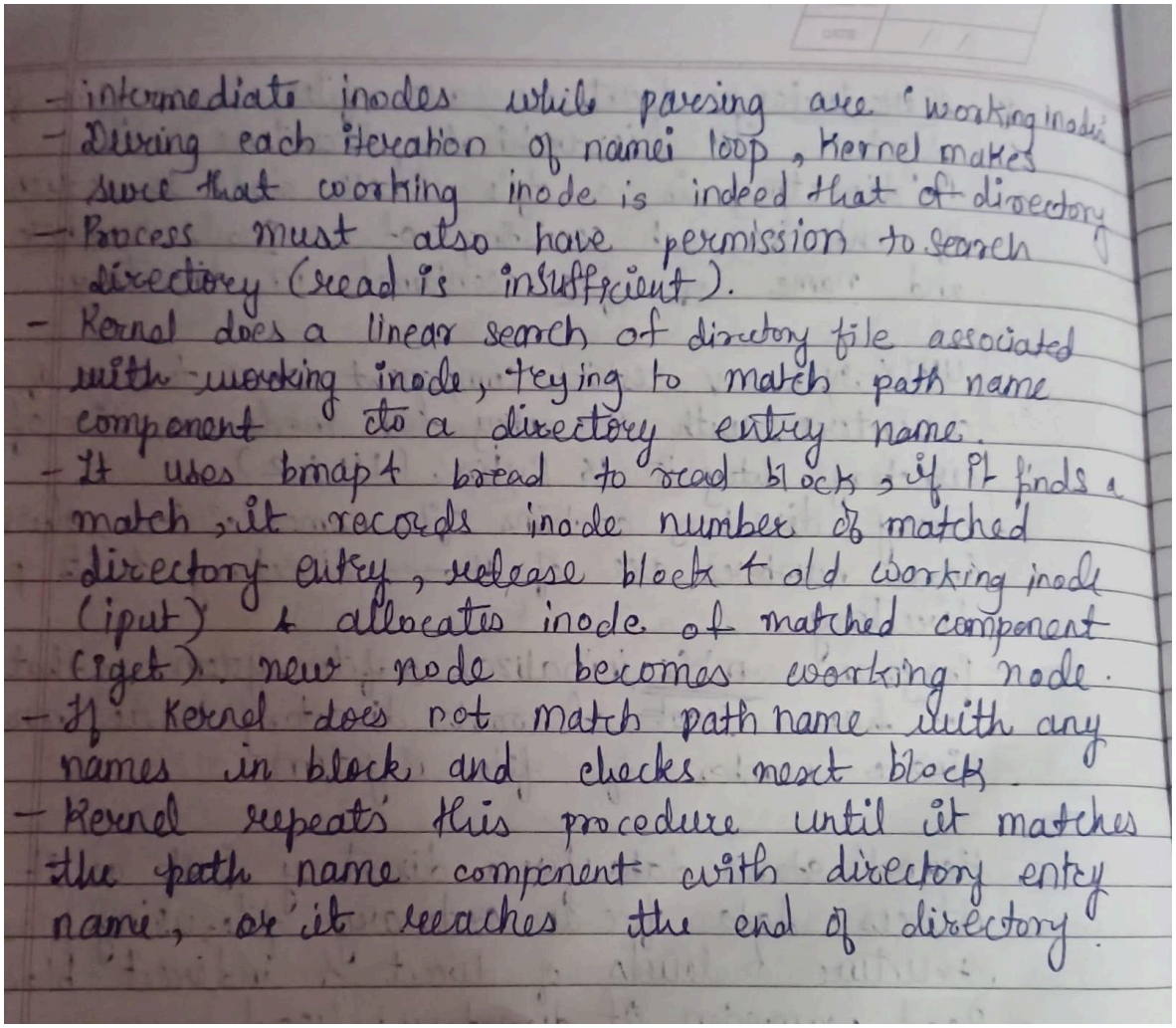
- The entry "." represents the current directory.It serves as a reference to the directory itself.
- The entry ".." represents the parent directory.
  It serves as a reference to the directory containing the current directory

Having root and parent directory entries in every directory ensures the hierarchical structure of the file system. It allows for easy traversal from any directory to its parent directory and ultimately to the root directory, providing a systematic way to organize and access files and directories within the file system. Additionally, these entries facilitate the implementation of various file system operations, such as path resolution and directory navigation.

11. Write an algo for conversion of pathname to inode.

Ans:





Jar tumhala algo form madhe pan lihaychi iccha asel tar he liha:

```
/*  Algorithm: namei
 *  Input: pathname
 *  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 a directory and access permissions are OK;
                if (working inode is of root and component is "..")
                        continue;
                read directory (working inode) by repeated use of algorithms: bmap, bread,
brelse;
                if (component matches an entry in the directory (working inode)
                {
                        get inode number for matched component;
                        release working inode (algorithm: iput);
                        working inode = inode of matched component (algorithm: iget);
                }
                else
                        return (no inode)  // component not in the directory
        }

        return (working inode);
}
```

Module 2: Buffer Cache

1. **Advantages and disadvantages of buffer cache**

   Ans: See Shriya's Notes

2. **Enlist the five scenarios of getblk algorithm ,which scenarios are suffered from race condition**

   Ans: Check details in Shriya's Notes

   a. Search for a buffer block in hash queue, and found that block, it is free so accessed it
   b. Fails to find in the hash queue. So the kernel removes the first buffer from the free list instead.(not marked for delay write). Kernel immediately marks buffer busy and place in new hash queue.
   c. Buffer it removes from free list is marked for 'delayed write'. First write contents to the disk async and then use
   d. No buffer in free list too. Process goes to sleep until some process executes brelse algo and wakes it up(race condition after wake up)
   e. Kernel finds the buffer in the hash queue, but the buffer is busy. Waits till it get available(race condition after wake up)

3. **In the algorithm getblk,if kernel removes a buffer from the free list, it must raise processor priority level to block out interrupts before checking free list, why?**

Ans:

Just as the kernel invokes algorithm *brelse* when a process has no more need for a buffer, it also invokes the algorithm when handling a disk interrupt to release buffers used for asynchronous I/O to and from the disk, as will be seen in Section 3.4. The kernel raises the processor execution level to prevent disk interrupts while manipulating the free list, thereby preventing corruption of the buffer pointers that could result from a nested call to *brelse*. Similar bad effects could happen if an interrupt handler invoked *brelse* while a process was executing *getblk*, so the kernel raises the processor execution level at strategic places in *getblk*, too. The exercises explore these cases in greater detail.

4. **Why free list of buffer in buffer cache is implemented like LRU if it is like MFU what will happen**

   Ans: LRU (Least Recently Used) Strategy:

- The free list prioritizes buffers that haven't been used recently. When a new buffer is needed, the one that hasn't been accessed for the longest time is removed from the free list and made available for use.
- This strategy aims to keep recently accessed data readily available in the cache, assuming recently used data is more likely to be used again soon.

Benefits of LRU:

- Improved Performance: LRU prioritizes data that's likely still relevant, leading to faster disk access times overall.
- Efficiency: LRU is a simple and efficient algorithm to implement, making it suitable for buffer cache management.

Why not MFU (Most Frequently Used)?

While MFU might seem appealing because it prioritizes the most frequently used data, it has drawbacks in the context of buffer cache management:

- Difficult to Implement: Accurately tracking the most frequently used data blocks can be computationally expensive and resource-intensive.

- Inefficiency for Random Access: In real-world workloads, access patterns are often unpredictable, with frequent bursts of random access. MFU might not effectively capture this behavior.
- Caching Less Frequently Used Data: Some data might be critical but accessed less frequently. LRU allows caching such data temporarily, potentially improving overall access times.

Consequences of MFU-like Buffer Cache:

If the buffer cache behaved like MFU, here are some potential issues:

Reduced Performance for Random Access: MFU might constantly remove recently accessed data to keep the most frequently used data, leading to more frequent disk accesses and potentially slower performance.

Inefficient Use of Cache Space: Data that is frequently accessed but only for short bursts might occupy cache space unnecessarily, preventing caching of potentially more valuable data.

5. **In buffer cache management why hash queue and free list needed to maintain separately which are suitable DS for Hash queue and free list?**

   Ans: Hash Queue:

   - Purpose: The hash queue is used to quickly locate disk blocks in the buffer cache based on their block numbers. Each entry in the hash queue corresponds to a hash bucket, and disk blocks with the same hash value are linked together in a hash chain within each bucket.
   - Function: When a disk block needs to be accessed or searched for in the buffer cache, the hash queue allows for fast lookup by computing the hash value of the block number and traversing the corresponding hash chain to find the block.
   - Suitable Data Structure: A suitable data structure for the hash queue is typically an array of hash buckets, with each bucket containing a linked list or some other data structure to manage collisions within the same hash value. This structure allows for efficient insertion, deletion, and lookup operations.

2. Free List:

- Purpose: The free list keeps track of available buffer slots in the buffer cache that can be used to cache disk blocks. Each entry in the free list represents a buffer slot that is not currently occupied by a disk block.
- Function: When a disk block needs to be read from or written to disk, the buffer cache manager checks the free list to find an available buffer slot to store the block.
- Suitable Data Structure: A suitable data structure for the free list is typically a linked list or a pool of buffer slots represented by an array. Linked lists allow for efficient insertion and deletion of buffer slots.

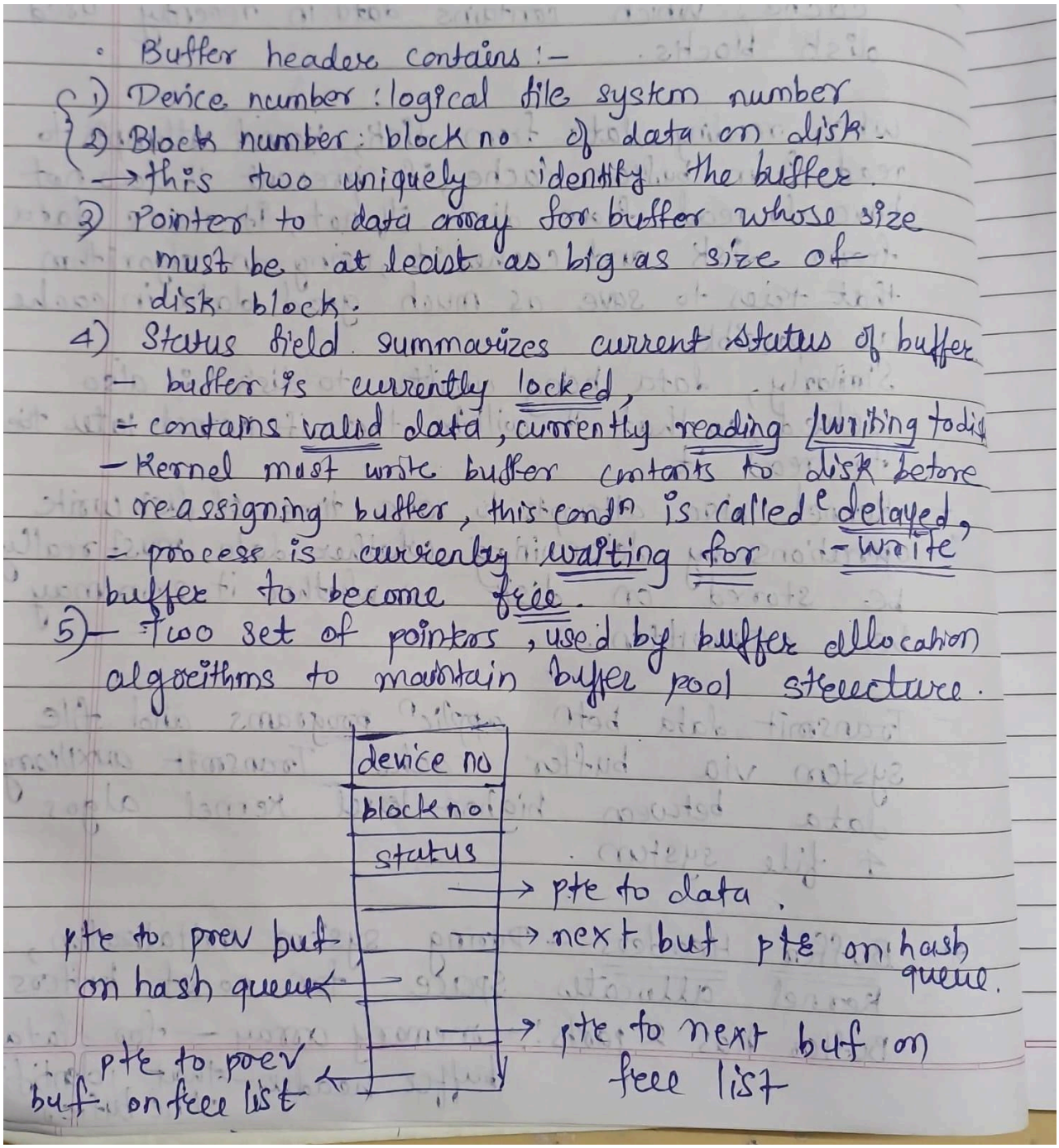
Maintaining the hash queue and free list separately allows for efficient management of disk blocks in the buffer cache:

- The hash queue enables fast lookup of disk blocks based on their block numbers, facilitating quick access to cached blocks for read and write operations.
- The free list allows the buffer cache manager to efficiently track available buffer slots and manage buffer allocation and deallocation as blocks are read from and written to disk.

By using separate data structures for these purposes, the buffer cache manager can optimize cache performance and resource utilization in Unix-like operating systems.

6. With diagram describe various fields of buffer header and structure of buffer header

Ans:



7. **In getblk scenarios which block suffers from race condition?why? A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.**

Ans: In Scenario 4)**Race for free buffer** in Scenario and **Race for a locked buffer** in scenario 5.

8. **What are functions of buffer? What are reasons for buffer cache in kernel for interacting with block oriented devices?**

   Ans: The kernel could read and write directly to and from the disk for all the file system accesses, but system response time and throughput will be poor because of the slow disk transfer rate. The kernel therefore attempts to minimize the frequency of disk access by keeping a pool of data buffers, called the *buffer cache*, which contains data in recently used disk blocks. Functions of buffer cache include:

1. Temporary Storage: Buffers provide temporary storage for data being transferred between different components of a system.
2. Reducing Overhead: Buffers can reduce overhead by allowing data to be transferred in larger, more efficient chunks rather than individually. This can improve overall system performance by reducing the number of individual I/O operations.
3. Asynchronous I/O: Buffers facilitate asynchronous I/O operations by allowing data to be buffered while other tasks are being performed. This allows the system to overlap I/O operations with computation, improving overall system throughput.
4. Caching: Buffers can be used as cache memory to temporarily store frequently accessed data. This helps reduce the latency of accessing data from slower storage devices by providing faster access to frequently used data.

Reasons for having a buffer cache in the kernel for interacting with block-oriented devices:

1. Performance Improvement: Buffer cache helps improve system performance by reducing the number of actual disk reads and writes. By caching frequently accessed blocks in memory, subsequent read requests for the same data can be satisfied from the cache, avoiding the need to access the slower disk storage.
2. Reducing Disk I/O: By caching frequently accessed blocks in memory, buffer cache reduces the number of disk I/O operations, which are typically slower compared to memory accesses.

3. Enhancing System Responsiveness: Buffer cache can improve system responsiveness by reducing the latency of accessing frequently used data. This can lead to faster application startup times, quicker file access, and overall smoother system performance.

In summary, buffer cache in the kernel for interacting with block-oriented devices helps improve system performance, reduce disk I/O operations, and enhance system responsiveness by caching frequently accessed data in memory.

9. Suppose the Kernal does a delayed write of a block. What happens when another process takes that block from the hash queue, from the free list

Ans: Taking the block from the hash queue:

- If another process accesses the block from the hash queue while the delayed write is pending, it means that the process wants to read or modify the data associated with that block.
- Depending on the specific implementation, the kernel may need to handle this situation differently:
    - If the process only wants to read the data, and the delayed write does not modify the data further, the kernel may allow the process to access the data directly from memory.
    - If the process wants to modify the data, the kernel may need to synchronize writting to the disk. This might involve waiting for the delayed write to complete before allowing the process to access the block.

2. Taking the block from the free list:

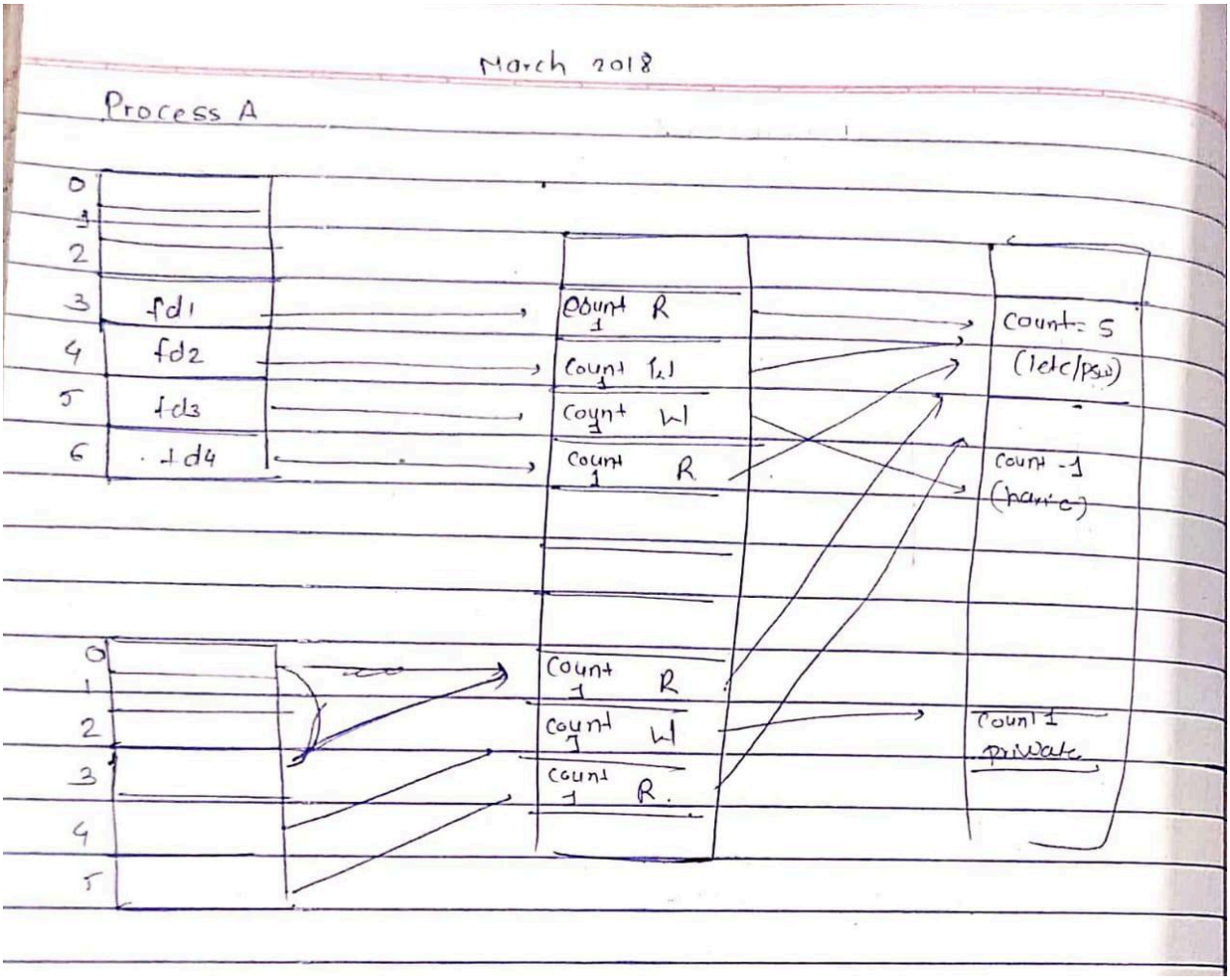
- If another process takes the block from the free list while the delayed write is pending, it suggests that the block was previously allocated but is now being reused for a different purpose.
- Depending on the specific implementation and the state of the delayed write, the kernel may need to handle this situation differently:
    - The kernel starts async write of data to disk.
    - If the delayed write has completed successfully, the kernel may allow the block to be reused for the new purpose, as the data associated with the block has already been written to disk.

Module 4

Diagram questions (if you can't understand diagram then message Ankita)

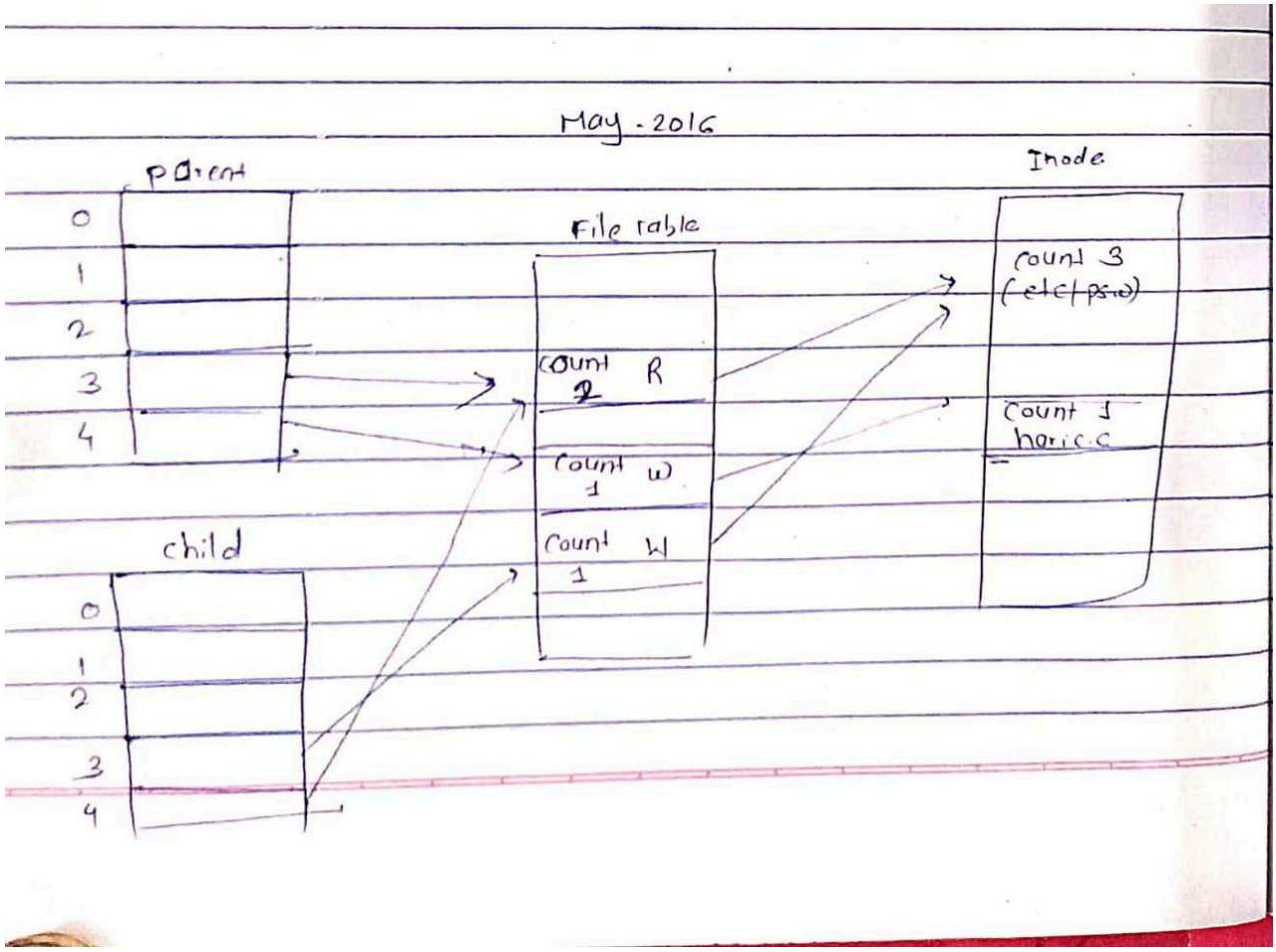
Ans:

Q4	For the following sequence system call, draw the files data structure entries.(per process FDT, File table and inode table)
	Process A
	fd1=open("etc/password", R); fd2 = open("etc/password", W);
	fd3= create("hari.c", W); fd4=open("etc/password", R);
	Process B
	fd1=open("etc/password", R); fd2 = open("private", W); fd3=open("etc/password", R);
		4	CO2



		needs to be done in file system.		

Q2	B)	For the following sequence system call in parent and child, draw the files subsystem data structure entries.
		Parent:{ fd1=open("etc/password", R);
		fd2= open ("hari.c", W);
		pid=fork(); //Child
		fd3 = open("etc/password", W);
		fd4=open("etc/password", R); }
			6	CO3



Q4 A) For the following sequence system call, draw the files data structure entries.(per process FDT, File table and inode table) 4

Process A

fd1=open("etc/password", R); fd2= dup(fd1); fd3 = open("etc/password", W);
fd4= create("hari.c", W); fd5=open("etc/password", R);

Process B

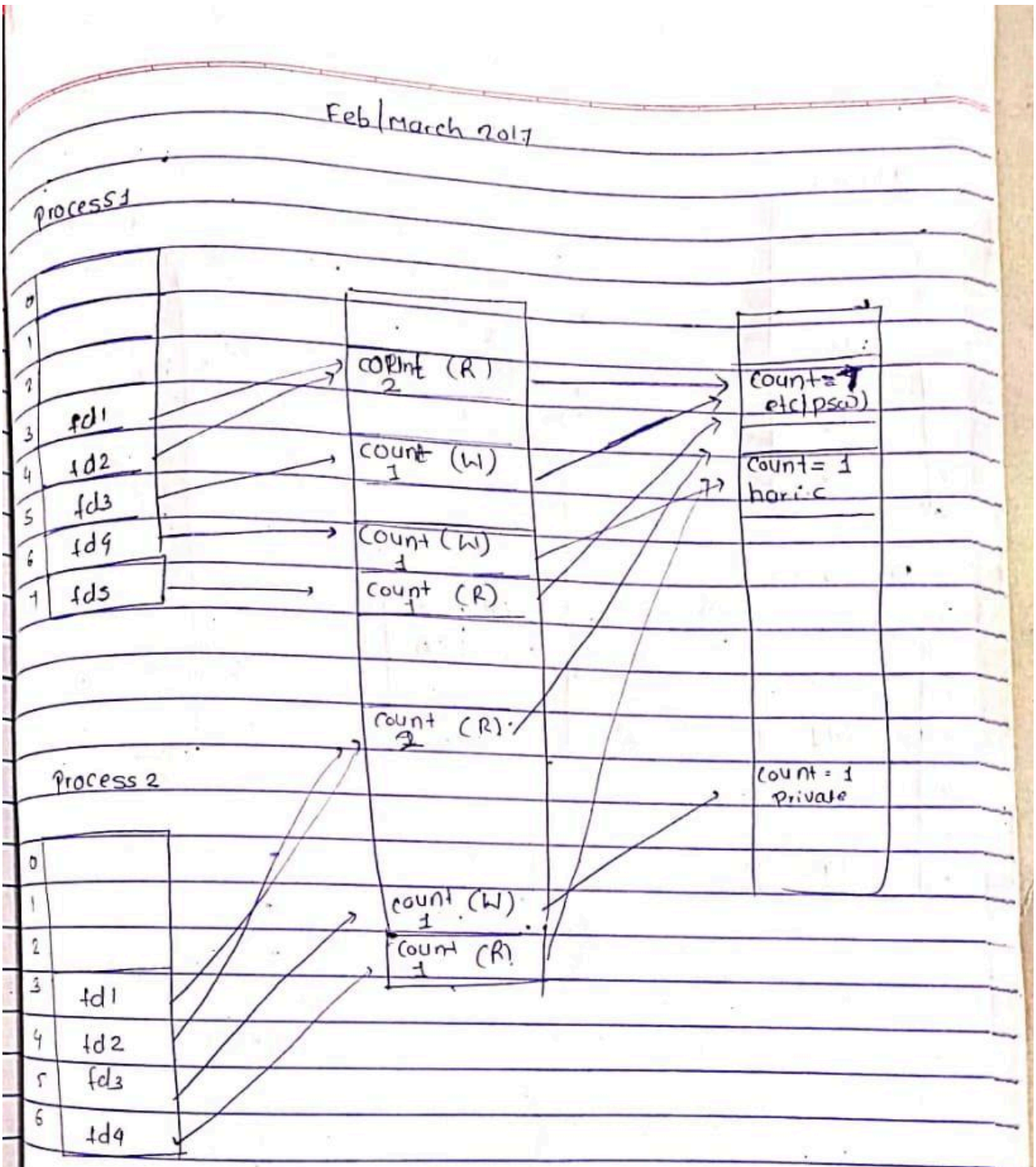
fd1=open("etc/password", R); fd2= dup(fd1); fd3 = open("private", W);
fd4=open("etc/password", R);

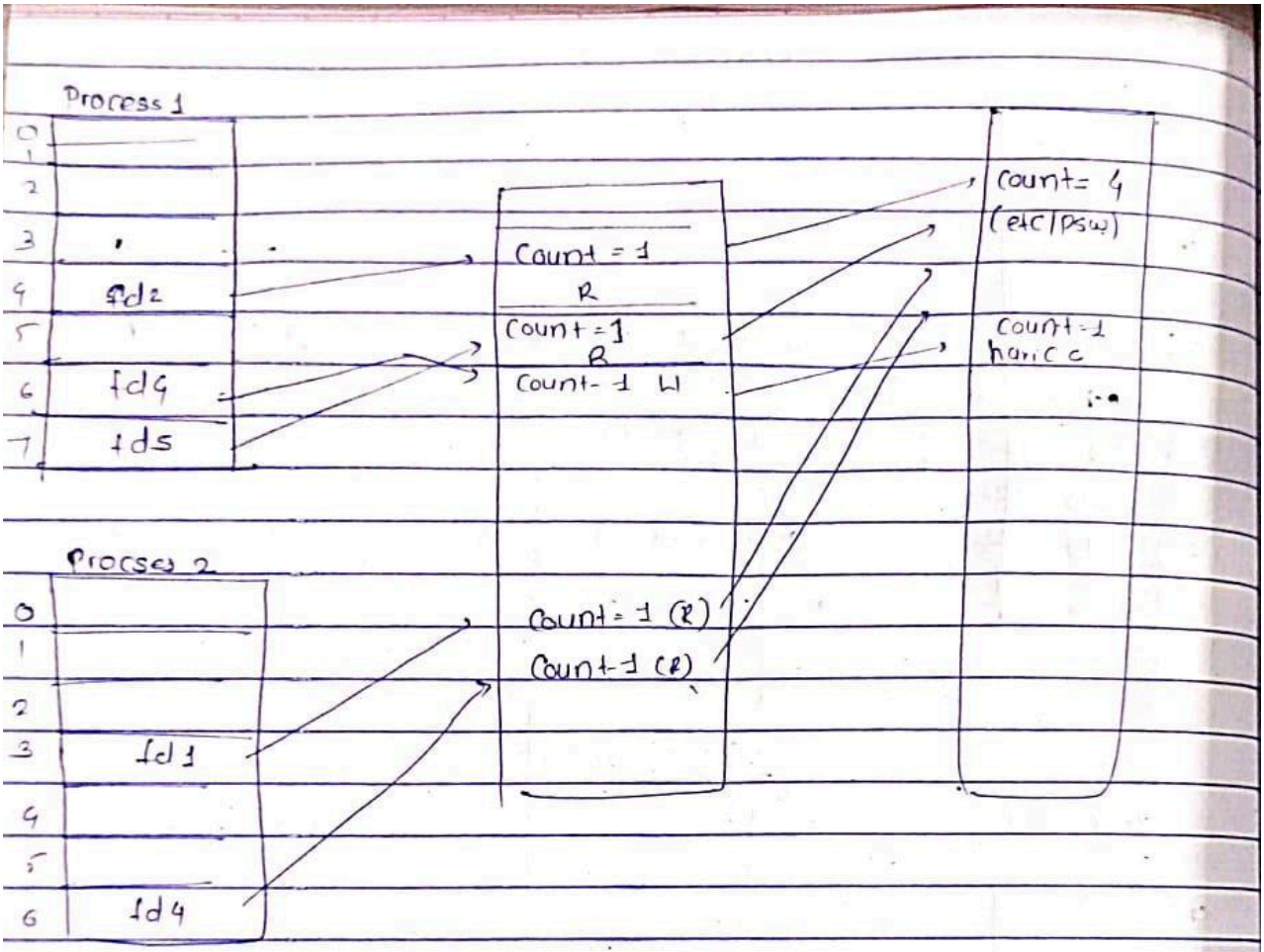
Then again draw the data structure entries for following:

Process A:Close (fd1);Close (fd3);

Process B:Close (fd2);Close (fd3);

Feb/March 2017



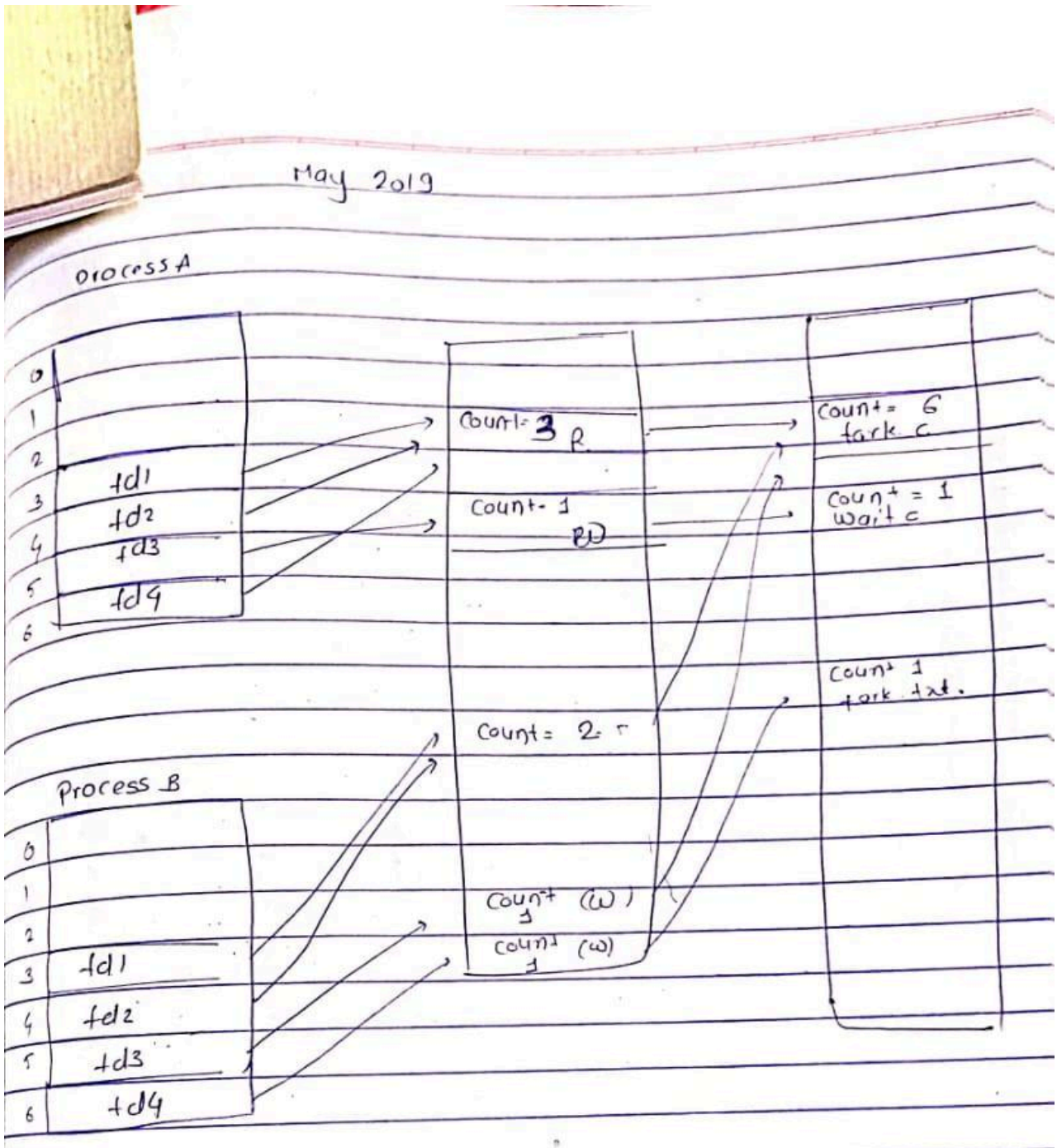

Process 1
0
1
2
3
4
fd2
5
6
fd4
7
fd5
Count = 1
R
Count = 1
R
Count - 1 W
count = 4
(etc/psw)
count = 1
hanic c
Process 2
0
1
2
3
fd1
4
5
6
fd4
Count = 1 (R)
Count - 1 (R)

		(Draw diagrams if necessary)		

Q3	B)	For the following sequence system call, draw the files data structure entries.(per process FDT, File table and inode table)
		Process A
		fd1=open("fork.c", R); fd2= dup(fd1); fd3 = open("wait.c", W);
		fd4= dup(fd2);
		Process B
		fd1=open("fork.c", R); fd2= dup(fd1); fd3 = open("fork.c", W);
		fd4=create("fork.txt", W);
			6	CO2

		...termination).	5	CO2

May 2019


Process A
0
1
2
3 fd1
4 fd2
5 fd3
6 fd4
Count = 3 R
Count = 1 RD
Count = 2 r
Count (W) 1
Count (W) 1
Count = 6 fork c
Count = 1 wait c
Count 1 fork txt.
Process B
0
1
2
3 fd1
4 fd2
5 fd3
6 fd4