1 Implementing Shared Files

Shared files allow multiple directory entries to refer to the same file. There are three main approaches to implement shared files:

1.1 1. Copying the Directory Entry

In this approach, the entire directory entry (including file attributes) is copied for each user.

Disadvantages:

- Updates to the shared file are not visible to all users
- Leads to inconsistency
- Not useful for true file sharing

1.2 2. Sharing Attributes Using Hard Links

In this method, file attributes are stored separately in an **i-node**. Multiple directory entries (names) point to the same i-node.

Advantages:

- Updates are visible to all users
- Efficient access

Disadvantages:

- If one link is removed, the other still exists
- Ownership and protection management can be complex

1.3 3. Symbolic Links (Soft Links)

A special **LINK** file is created that contains the pathname of the original file.

Advantages:

- Can point to files outside the current file system
- Can transparently replace the file with another

Disadvantages:

- File removal leaves dangling (invalid) links
- Less efficient than hard links

2 Comparison Summary

- Copying directory entries does not provide true sharing
- Hard links provide efficient and consistent sharing
- Symbolic links are flexible but less efficient

3 File Sharing and Access Rights

3.1 File Sharing

File sharing allows multiple users or processes to access the same file.

Issues in file sharing:

- Consistency: concurrent access may lead to inconsistent data
- Protection: unauthorized access must be prevented
- Synchronization: coordination among processes

Approaches:

- Shared files with read/write locks
- File versioning

3.2 Access Rights

Access rights specify what operations a user can perform on a file.

Common access rights:

- Read (r)
- Write (w)
- Execute (x)
- Append
- Delete
- List

4 UNIX File Protection Mechanism

UNIX uses a simple protection model based on three user categories:

1. Owner (User)
2. Group
3. Others

Each category has permissions:

Read (r), Write (w), Execute (x)

Example:

```
-rw-r--r--
```

- Owner: read, write
- Group: read
- Others: read

5 Shortcomings of UNIX Permissions

- Only three coarse-grained user classes
- Cannot assign different permissions to individual users

Problematic Example:

- Joe owns file `foo.bar`
- Joe wants:
  - No access to the public
  - Bill: read and write access
  - Peter: read-only access

Problem: UNIX permissions cannot represent this requirement using only owner, group, and others.

6 Access Control Lists (ACLs)

ACLs provide fine-grained access control by allowing permissions to be specified per user.

ACL example:

- Joe (owner): read, write
- Bill: read, write
- Peter: read
- Others: no access

Advantages of ACLs:

- Fine-grained permission control
- More flexible than UNIX permissions

Disadvantages:

- More complex to manage
- Increased storage overhead

7 Directories

A directory is a special file that maps file names to file identifiers.

7.1 Directory Operations

- Create
- Delete
- Open
- Read
- Rename

7.2 Directory Structures

- Single-level directory
- Two-level directory
- Tree-structured directory
- Acyclic graph directory

8 Quotas

Disk quotas limit the amount of disk space or number of files a user can consume.

Types of quotas:

- Soft quota: warning issued
- Hard quota: no further allocation allowed

Purpose of quotas:

- Prevent disk space exhaustion
- Ensure fair resource usage

9 Summary (Exam-Oriented)

- UNIX permissions are simple but coarse-grained
- ACLs solve the limitation of per-user access control
- File sharing requires protection and synchronization
- Directories organize files hierarchically
- Quotas control disk resource usage

User-Level vs Kernel-Level Threads

Key Differences Summary

- **User-Level Threads:** Fast but cooperative, limited parallelism
- **Kernel-Level Threads:** Slower but preemptive, true parallelism
- **Trade-off:** Speed vs. Functionality

Stack Frame Placement in Context Switching

Where Does the Stack Frame Reside?

Context Switch Data Structures

Example: Task Table Structure (ARM)

```c
typedef struct {
    void *sp;       // Stack pointer (PSP) - points to stack frame
    int flags;      // Status flags: activity, parent task, etc.
} task_table_t;

int current_task;
task_table_t task_queue[MAX_TASKS];  // Array in kernel space
```

Memory Layout During Context Switch

```
KERNEL SPACE:
+-- PCB/TCB Table (Process Headers)
    +-- Process 0: {sp=0x2000FFF0, state=READY, ...}
    +-- Process 1: {sp=0x2001FFF0, state=RUNNING, ...}
    +-- Process 2: {sp=0x2002FFF0, state=BLOCKED, ...}

PROCESS ADDRESS SPACE (Process 1):
+-- Code Segment
+-- Data Segment
+-- Heap
+-- Stack  <-- STACK FRAME STORED HERE
    +-- Local variables
    +-- Return addresses
    +-- [STACK FRAME/TRAP FRAME]  <-- Saved registers
        +-- xPSR, PC, LR
        +-- r0-r3, r12
        +-- r4-r11 (saved by OS)
        +-- FPU registers (if active)
```

Important Notes

1. **Stack Frame in Process Stack:** The register save area (stack frame) resides in the process's own stack, NOT in kernel stack
2. **PCB in Kernel:** Only the process control information (PCB/TCB header) with pointer to stack is in kernel space
3. **Efficiency:** This design avoids copying large register sets between kernel and user space

4. **Hardware Support:** Modern processors (ARM, x86) automatically push registers to process stack on exception/interrupt
5. **Naming:** Stack frame and trap frame are the same concept - saved execution context on the stack

Context Switch Flow

1. **Interrupt/Exception occurs**
2. **Hardware automatically:** Pushes xPSR, PC, LR, r0-r3, r12 to process stack
3. **OS Scheduler:** Pushes remaining registers (r4-r11, FPU) to process stack
4. **OS:** Saves current stack pointer (SP) to PCB in kernel
5. **OS:** Loads next process's stack pointer from its PCB
6. **OS:** Pops r4-r11 from new process's stack
7. **Hardware automatically:** Pops xPSR, PC, LR, r0-r3, r12 from new process's stack
8. **New process resumes execution**

Sparse Files

The number of disk blocks used by a file can be much less than what is expected from its file size. This is because modern file systems support **sparse files**.

A **sparse file** is a file in which large regions contain no actual data (all zeros), and the file system does not allocate physical disk blocks for those empty regions. Instead, it stores only the locations of the blocks that contain real data along with metadata indicating the holes in between.

Example

```
write(f, "hello");        // writes 5 bytes
lseek(f, 1000000);        // moves file offset forward
write(f, "world");        // writes 5 bytes
```

In this example:

- `"hello"` is written at the beginning of the file.
- `lseek` creates a large gap (hole) of empty bytes.
- `"world"` is written at position 1,000,000.

Analysis

- Logical file size = 1,000,005 bytes
- Disk blocks used = 2 blocks (for `"hello"` and `"world"`) + metadata overhead
- Empty blocks between the two writes are not stored on disk

Conclusion

Sparse files allow efficient disk space usage by allocating disk blocks only for regions that contain actual data, while logically maintaining a large file size.

Efficient File Access Using Inodes

Consider a UNIX-style inode structure with the following assumptions:

- Block size = 1 KB
- Block number (pointer) size = 4 bytes
- Inode contains 10 direct pointers and 1 single indirect pointer

Maximum File Size Addressable

Direct blocks:

$$10 \times 1 \text{ KB} = 10 \text{ KB}$$

Single indirect block:

$$\frac{1 \text{ KB}}{4 \text{ bytes}} = 256 \text{ block pointers}$$

$$256 \times 1 \text{ KB} = 256 \text{ KB}$$

Total file size covered:

$$10 \text{ KB} + 256 \text{ KB} = 266 \text{ KB}$$

Access Efficiency

For the large majority of files whose size is less than 266 KB:

- Direct blocks require only one disk access
- Indirect blocks require at most two disk accesses

Conclusion

Since most files are smaller than 266 KB, only one or two disk accesses are required to read any block, making inode-based file allocation efficient for common workloads.

Interrupt Handler Execution Steps

When an Interrupt Occurs

Critical Rules for Interrupt Handlers

1. **CANNOT BLOCK:** Interrupt handlers must never wait for anything
    - They run in context of whatever process was running
    - Blocking would freeze that unlucky process
    - May cause deadlock
2. **Must be FAST:** Minimize time spent in handler
    - Do minimal processing
    - Defer complex work to later
3. **Must be NON-BLOCKING:** Cannot call sleep() or wait()
4. **Runs on kernel stack:** Usually on current process's kernel stack

I/O Buffering Strategies

Overview of Buffering

Purpose: Deal with speed mismatches between I/O devices and CPU/processes.

Detailed Buffering Strategies

1. No Buffering

Implementation:

- Process must read/write device one byte/word at a time
- Each I/O operation requires a system call

Problems:

- Each individual system call adds significant overhead
- Process must wait until each I/O completes
- Blocking/interrupt/waking adds overhead
- Many short runs of process = poor CPU cache locality

Performance: Very poor, rarely used

2. User-Level Buffering

Implementation:

- Process specifies a memory buffer
- Incoming data placed in buffer until full
- Filling done by interrupt service routine
- Only single system call and block/wakeup per buffer

Advantages:

- Much more efficient than no buffering
- Reduces number of system calls

Problems:

- **Buffer paged out:** If buffer swapped to disk, could lose data while paging in
- **Lock buffer in memory?** Reduces available RAM, can cause deadlock
- **Write case:** When is buffer available for reuse?
  - Process blocks until device drains buffer, OR
  - Deal with asynchronous signals (complex)

3. Single Buffer (Kernel)

Implementation:

- OS assigns a buffer in kernel memory for I/O request
- **Stream-oriented:** Used a line at a time (e.g., terminal input)
- **Block-oriented:** Transfer in fixed-size blocks (e.g., disk)

Block-Oriented Operation:

1. Input transfers made to kernel buffer
2. Block moved to user space when needed
3. Another block moved into kernel buffer (read-ahead)
4. User process can process one block while next is being read
5. Swapping can occur since input is in kernel memory

Advantages:

- Process and I/O can overlap
- Safer than user-level buffering (in kernel memory)
- OS tracks buffer assignments

Problems:

- If kernel buffer full, user buffer swapped out, and more data arrives → **Data Loss**
- Drop characters or network packets

4. Double Buffer (Kernel)

Implementation:

- Use TWO system buffers instead of one
- Process can transfer data to/from one buffer
- OS simultaneously empties or fills the other buffer

Operation:

```
Time T:
  Buffer A: OS is filling      (reading from device)
  Buffer B: Process is using   (processing data)

Time T+1:
  Buffer A: Process is using   (swap roles)
  Buffer B: OS is filling      (swap roles)
```

Advantages:

- Better overlap of processing and I/O
- Reduces chance of data loss
- More efficient than single buffer

Problems:

- May be insufficient for really bursty traffic
- Long periods of computation while receiving data
- Limited read-ahead for disk (only one block)

5. Circular Buffer (Multiple Buffers)

Implementation:

- More than two buffers used (typically 3-16)
- Buffers arranged in circular queue
- Producer (device) fills buffers sequentially
- Consumer (process) empties buffers sequentially

Operation:

```
[Buffer 0] --> [Buffer 1] --> [Buffer 2] --> [Buffer 3]
     ^                                            |
     |                                            |
     +--------------------------------------------+
     (circular connection)

Producer writes to: Buffer[write_index]
Consumer reads from: Buffer[read_index]
```

Advantages:

- Best for continuous high-speed I/O
- Handles bursty traffic well
- Maximum overlap of I/O and computation
- Can accommodate varying processing rates

When to Use:

- I/O operation must keep up with process
- High-speed network interfaces
- Video/audio streaming
- Disk with extensive read-ahead

Single vs Double Buffer: Speed Comparison

Single Buffer Time:

$$T_{single} = \max(C, T) + M$$

where:

C = Computation time per block

T = Transfer time per block

M = Memory copy time (buffer to user space)

Double Buffer Time:

$$T_{double} = \max(C + M, T)$$

Speedup: Double buffering allows computation and I/O to overlap completely.

I/O Software Layers

Layered Architecture

The OS organizes I/O handling into layers (from top to bottom):

Benefits of Layered Design

1. **Portability:** Upper layers don't need device-specific knowledge
2. **Modularity:** Each layer has clear responsibilities
3. **Simplified Application Development:** Applications use uniform interface
4. **Device Independence:** Same code works for disk, network, keyboard, etc.

Example: Reading from a File

```
User Application:
    read(fd, buffer, size);
    |
    v
User-Level I/O Library:
    Converts to system call
    |
    v
Device-Independent Layer:
    Check permissions, manage buffers
    |
    v
Device Driver:
    Issue disk-specific READ command
    |
    v
Hardware:
    Disk controller reads data
    |
    v
Interrupt Handler:
    Signals completion, wakes driver
    |
    v
(Data flows back up through layers)
```

Operating System Design Issues

1. Efficiency Problem

Challenge:

- Most I/O devices are MUCH slower than main memory and CPU
- Memory: nanoseconds (10^{-9} s)
- Disk: milliseconds (10^{-3} s)
- Speed difference: **1,000,000 times!**

Solutions:

1. **Multiprogramming:**
   - Allow processes to execute while others wait for I/O
   - Keep CPU busy
   - Even with multiprogramming, I/O often cannot keep up
2. **Swapping:**
   - Bring additional ready processes into memory
   - Increased swapping $\rightarrow$ more I/O operations
3. **Optimize I/O Efficiency:**
   - Focus on disk I/O (slowest)
   - Focus on network I/O (variable speed)
   - Use buffering, caching, read-ahead

2. Generality/Uniformity Problem

Challenge:

- Wide diversity of I/O devices
- Different access methods (random access vs. stream-based)
- Vastly different data rates (keyboard: 10 bytes/s, Ethernet: 125 MB/s)
- Different error handling
- Different data formats

Goal:

- Handle all I/O devices in the same way

- Both in OS internals and user applications

Solution:

- **Hide device-specific details** in lower-level routines (device drivers)
- **Present uniform interface** to processes and upper levels
- Generic operations: `open()`, `read()`, `write()`, `close()`
- Applications don't care if reading from disk, network, or keyboard

Trade-off:

- Generality often compromises efficiency
- Must balance uniform interface with performance

OS Design Strategies Summary

Key Formulas and Concepts for Exam

Important Relationships

1. Effective I/O Time with Buffering:

$$T_{no\_buffer} = n \times (T_{transfer} + T_{process})$$
$$T_{single\_buffer} = n \times \max(T_{transfer}, T_{process})$$
$$T_{double\_buffer} \approx n \times \max(T_{transfer}, T_{process}) \text{ (better overlap)}$$

2. System Call Overhead:

$$\text{Total Time} = \text{Useful Work} + \text{System Call Overhead}$$
$$\text{Overhead} = \text{Number of Calls} \times \text{Cost per Call}$$

3. Buffering Efficiency:

$$\text{Efficiency} = \frac{\text{Useful Data Transfer Time}}{\text{Total Time (including waits)}}$$

Must-Know Points for Exam

1. **Interrupt Handlers:**
   - Must be fast and non-blocking
   - Run in context of current process
   - Cannot call sleep() or wait()
   - Save/restore all registers
2. **Buffering Hierarchy:**
   - No buffer ¡ User buffer ¡ Single buffer ¡ Double buffer ¡ Circular buffer
   - Each level improves efficiency but adds complexity
3. **I/O Software Layers (Top to Bottom):**
   - User I/O → Device-Independent → Drivers → Interrupt Handlers → Hardware
4. **OS Goals:**
   - Efficiency: Use multiprogramming and buffering
   - Generality: Uniform interface via layering
5. **Key Trade-offs:**
   - Generality vs. Efficiency
   - Simplicity vs. Performance
   - Memory usage vs. Speed

Common Exam Questions

1. **Explain interrupt handler steps** (3 main phases)
2. **Why can't interrupt handlers block?** (context issue)
3. **Compare buffering strategies** (advantages/disadvantages)
4. **Calculate time with different buffering** (formula application)
5. **Explain I/O software layers** (5 layers and their roles)
6. **OS design issues** (efficiency and generality problems/solutions)

Device-Independent I/O Software

Device-independent I/O software is the part of the operating system that provides a uniform interface to I/O devices, hiding hardware-specific details from user programs and higher-level OS components.

Purpose

- Hide device-specific details
- Provide uniform I/O operations such as `open`, `read`, `write`, and `close`
- Improve portability and maintainability

Without vs With Standard Driver Interface

- Without a standard interface, applications must handle device-specific details
- With a standard interface, all devices are accessed in a uniform manner

Driver–Kernel Interface

The driver–kernel interface defines how the operating system communicates with device drivers. It ensures:

- Standardized communication
- Hardware independence
- Ease of adding new device drivers

Buffering in I/O Systems

Buffering is the temporary storage of data during I/O transfers to improve efficiency and reduce waiting time.

Types of Buffering

1. Unbuffered I/O
2. User-level buffering
3. Single buffering in the kernel
4. Double buffering in the kernel
5. Circular buffering

No Buffering

- Process reads or writes one byte/word at a time
- Each I/O operation requires a system call
- Process must wait until each I/O completes

Disadvantages:

- High system call overhead
- Wastes CPU cycles
- Poor CPU cache performance

User-Level Buffering

- Process provides a buffer in its own address space
- Data is collected in the buffer until it is full
- Only one system call per buffer

Advantages:

- Efficient
- Fewer context switches

Problems:

- Buffer may be paged out to disk
- Data loss may occur
- Buffer locking reduces available RAM
- Deadlock possible if memory is exhausted

Single Buffering

The operating system allocates a single buffer in main memory.

Stream-Oriented Devices

- Input and output handled one line at a time
- Example: terminal I/O

Block-Oriented Devices

- Data transferred in blocks
- Block copied from kernel buffer to user space
- Read-ahead possible

Advantages:

- Overlap of I/O and computation
- Better CPU utilization

Problem:

- Data loss if kernel buffer is full and user buffer is swapped out

Double Buffering

- Uses two kernel buffers
- One buffer used by process, the other by OS

Advantages:

- Higher throughput
- Reduced waiting time
- Better overlap of computation and I/O

Limitations:

- Insufficient for highly bursty traffic
- Limited read-ahead capability

Circular Buffering

- Uses more than two buffers
- Buffers arranged in a circular queue
- Suitable when I/O must continuously keep up with the process

Is Buffering Always Good?

- Buffering increases memory usage
- May increase latency
- Not always beneficial for fast networks

Buffering in Fast Networks

- Excessive buffering increases delay
- Can cause congestion
- Modern networks often prefer minimal buffering

I/O Software Summary

- Device-independent I/O hides hardware details
- Buffering improves performance but has trade-offs
- Multiple buffering strategies exist for different workloads
- Proper buffering is essential for efficient OS design

Virtual File System (VFS)

The Virtual File System (VFS) is an abstraction layer in the operating system that provides a uniform interface to different file system implementations. It allows multiple file systems to coexist and be accessed transparently using common system calls.

Motivation for VFS

Different file systems (e.g., EXT4, FAT, NTFS, NFS) have different internal structures and access methods. Without VFS:

- Each file system would require separate system calls
- Applications would become file-system dependent

VFS solves this problem by hiding file-system-specific details.

Goals of Virtual File System

- Provide a uniform file access interface
- Support multiple file system types
- Improve portability and extensibility
- Separate file system implementation from user applications

VFS Architecture

VFS sits between user-level system calls and the actual file system implementations.

User Processes
↓
System Calls (`open, read, write, close`)
↓
Virtual File System (VFS)
↓
Specific File Systems (EXT4, FAT, NFS, etc.)

Key VFS Data Structures

Superblock Object

- Represents a mounted file system
- Contains information such as file system type, size, and status
- One superblock per mounted file system

Inode Object

- Represents a file or directory
- Stores metadata: permissions, owner, size, timestamps
- Each file has a unique inode

Dentry (Directory Entry) Object

- Maps filenames to inodes
- Speeds up pathname lookup
- Stored in the dentry cache

File Object

- Represents an open file
- Maintains file offset and access mode
- Created when a file is opened

VFS Operations

VFS defines a standard set of operations that every file system must implement.

Common File Operations

- `open()`
- `read()`
- `write()`
- `close()`
- `lseek()`

Directory Operations

- `mkdir()`
- `rmdir()`
- `readdir()`

Mounting in VFS

- A file system is mounted at a mount point
- VFS connects the superblock of the mounted file system to a directory
- All mounted file systems appear as part of a single directory tree

Advantages of Virtual File System

- Applications are independent of file system type
- Easy to add support for new file systems
- Uniform access to local and remote file systems
- Improved system organization and modularity

Disadvantages of Virtual File System

- Additional abstraction layer adds slight overhead
- Increased kernel complexity

Conclusion

The Virtual File System provides a powerful abstraction that allows different file systems to coexist while presenting a consistent interface to user applications, making modern operating systems flexible and extensible.

10 BAtch 26

Q1(a) Caching vs Multiple Layers of Memory

Caching significantly improves system performance, yet modern systems use multiple layers of memory instead of caching everything due to the following reasons:

- **High Cost:** Cache memory (SRAM) is much more expensive than main memory (DRAM).
- **Limited Size:** Cache memory is small and cannot store all programs and data.
- **Power Consumption:** Faster memory consumes more power, which is undesirable.
- **Management Complexity:** Larger caches increase coherence and replacement overhead.

- **Diminishing Returns:** Increasing cache size beyond a point gives little performance gain.

Therefore, systems use a **memory hierarchy** consisting of registers, cache, main memory, and secondary storage to balance speed, cost, and capacity.

Choice: A layered memory hierarchy is preferred as it provides optimal performance with reasonable cost and scalability.

Q1(b) Suitable I/O Buffering Techniques

- **Networked Machines:** Double or circular buffering is suitable to support continuous data streams and avoid packet loss.
- **Keyboard Devices:** Line buffering is appropriate due to low data rate and line-based input processing.
- **Machine-Display Devices:** Double buffering is used to prevent flickering and ensure smooth screen updates.

Q1(c) Device-Independent Drivers

Advantages

- Improved portability across hardware platforms.
- Code reuse and simplified OS design.
- Uniform interface for I/O operations.
- Easier maintenance and extensibility.

Interaction with Device-Dependent Drivers

Applications issue system calls that are handled by the device-independent layer, which performs generic functions such as buffering and protection. The requests are then forwarded to device-dependent drivers that directly control the hardware.

Q2(a) Disk Read and Write Operations

Given average disk access time = 2 ms.

Read Operations

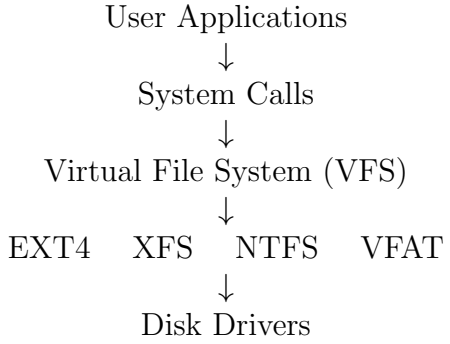
Block Type	Disk Accesses	Time
Direct Block	1	2 ms
Single Indirect Block	2	4 ms
Double Indirect Block	3	6 ms

Write Operations

Write operations require the same number of disk accesses as read operations, excluding allocation overhead.

Q2(b) Virtual File System (VFS)

A Virtual File System provides a uniform interface to users while supporting multiple file systems.



The VFS allows file sharing, access control, and seamless integration of multiple file systems.

Q2(c) Block Size vs Disk Performance

Small Block Size

- Better disk space utilization
- Reduced internal fragmentation
- Increased metadata overhead

Large Block Size

- Improved I/O performance
- Fewer disk accesses

- Increased internal fragmentation

Conclusion: Small blocks improve space utilization, while large blocks enhance performance. Modern systems typically use an optimal block size such as 4 KB.

Question 2

2(a) Speeding up Paging using TLB

Paging can be slow because each memory reference may require two memory accesses: one to access the page table and another to access the actual data.

A **Translation Lookaside Buffer (TLB)** is a small, fast associative memory that stores recently used page table entries.

Working:

- On every memory reference, the CPU first checks the TLB.
- If the page number is found (*TLB hit*), the frame number is obtained immediately.
- If not found (*TLB miss*), the page table is accessed and the entry is loaded into the TLB.

Advantage:

- Reduces the number of memory accesses.
- Improves effective memory access time.
- High hit ratio (90–99%) significantly speeds up paging.

Conclusion:

TLB speeds up paging by caching frequently used page table entries.

—

2(b) Banker's Algorithm: Safe State Analysis

Total Resources

$$R_{max} = [6,\ 8,\ 6,\ 5]$$

Maximum Requirement Matrix (P_{need})

	R_1	R_2	R_3	R_4
P_1	3	4	1	2
P_2	4	5	2	2
P_3	2	3	2	3
P_4	4	5	2	2
P_5	3	4	2	3

Allocation Matrix (A)

	R_1	R_2	R_3	R_4
P_1	0	2	0	0
P_2	2	0	1	1
P_3	0	0	1	1
P_4	2	3	0	1
P_5	0	1	1	0

Available Resources

$$\text{Allocated} = [4,\ 6,\ 3,\ 3]$$

$$\text{Available} = R_{max} - \text{Allocated} = [2,\ 2,\ 3,\ 2]$$

Need Matrix

$$Need = P_{need} - A$$

	R_1	R_2	R_3	R_4
P_1	3	2	1	2
P_2	2	5	1	1
P_3	2	3	1	2
P_4	2	2	2	1
P_5	3	3	1	3

Safe Sequence

A safe execution sequence exists:

$$P_4 \to P_3 \to P_1 \to P_2 \to P_5$$

The system is in a SAFE state.

Additional Request by P_5

$$Request = [2,\ 2,\ 1,\ 2]$$

Since the request exceeds the available resources,

The OS must NOT allocate resources immediately.

—

2(c) Critical Section Identification and Solution

The variable `count` is a global shared variable accessed by multiple threads. This leads to a **race condition**.

Critical Section

```
value += s;
count = value;
```

Corrected Code using Mutex

```
pthread_mutex_t lock;

void show_hit_count(void)
{
    uint32_t value = 0;

    while (1)
    {
        uint32_t s = getHttpHitRequest();
        if (s == 0) break;
        value += s;
    }

    pthread_mutex_lock(&lock);
    count += value;
    printf("Current HTTP hit count %d\n", count);
    pthread_mutex_unlock(&lock);
}
```

Why this Solution Works

- Ensures mutual exclusion
- Prevents race conditions
- Deadlock-free (single mutex)
- Satisfies all four synchronization requirements

The synchronization problem is correctly resolved.

Question 4

4(a) Variable-size Dynamic Partition (Base–Limit Addressing)

Given that a logical address is divided into a partition number and an offset, address translation is performed using the base–limit scheme.

Rule:

$$\text{If offset} < \text{limit}, \quad \text{Physical Address} = \text{base} + \text{offset}$$

Otherwise, the address is invalid.

Allocation Table

Process	Base	Limit
P_1	1000	200
P_2	2000	300
P_3	2400	500

Address Translation

Process P_1:

$$150 < 200 \Rightarrow 1000 + 150 = 1150$$
$$250 \geq 200 \Rightarrow \text{Invalid}$$
$$130 < 200 \Rightarrow 1000 + 130 = 1130$$

Process P_2:

$$160 < 300 \Rightarrow 2000 + 160 = 2160$$
$$230 < 300 \Rightarrow 2000 + 230 = 2230$$
$$350 \geq 300 \Rightarrow \text{Invalid}$$
$$500 \geq 300 \Rightarrow \text{Invalid}$$

Process P_3:

$$200 < 500 \Rightarrow 2400 + 200 = 2600$$
$$400 < 500 \Rightarrow 2400 + 400 = 2800$$
$$350 < 500 \Rightarrow 2400 + 350 = 2750$$
$$501 \geq 500 \Rightarrow \text{Invalid}$$

—

4(b) Page Replacement (4 Frames)

Reference string:

$$1, 2, 4, 5, 8, 3, 4, 10, 2, 3, 6, 7, 8, 2, 6, 8, 9$$

Optimal Page Replacement

Optimal replaces the page whose next use is farthest in the future.

$$\text{Total page faults using Optimal} = 10$$

LRU Page Replacement

LRU replaces the page that was least recently used in the past.

$$\text{Total page faults using LRU} = 12$$

Comparison

Algorithm	Page Faults
Optimal	10
LRU	12

—

4(c) Working-Set Model

Given:

$$\Delta = 5 \quad \text{and total available frames} = 10$$

Working Sets

$$WS(P_1) = 5$$
$$WS(P_2) = 5$$
$$WS(P_3) = 3$$
$$WS(P_4) = 4$$

Total Working-Set Size

$$\sum WS = 5 + 5 + 3 + 4 = 17$$

Thrashing Condition

$$\sum WS > \text{Available Frames}$$
$$17 > 10$$

Conclusion

$$\boxed{\text{The system will experience thrashing.}}$$

To handle thrashing, the operating system should reduce the degree of multiprogramming by suspending one or more processes.

11 Qs 7

RAID Levels (Exam Ready Summary)

RAID (Redundant Array of Independent Disks) uses multiple disks to improve performance, reliability, or both, while presenting a single logical disk to the file system.

RAID Level	Description	Advantages	Disadvantages
RAID 0	Block-level striping across multiple disks, no redundancy	• Very high read-/write performance • Full disk capacity usable • Avoids disk hotspots	• No fault tolerance • Failure of one disk causes total data loss • Less reliable than a single disk
RAID 1	Mirroring: each block written to two disks	• High reliability • Can tolerate one disk failure • Read performance improves	• Requires double the disks • High storage cost • Write performance same as single disk
RAID 0+1 / 1+0	Combination of striping and mirroring	• High performance • High fault tolerance • Better reliability than RAID 0	• Very expensive • Requires many disks
RAID 2	Bit-level striping with Hamming code (ECC) disks	• Can correct single-bit errors	• Requires synchronized spindles • High overhead • Not used in practice
RAID 3	Bit-level striping with single parity disk	• Can recover from one disk failure • Only one extra disk needed	• Synchronized spindles required • Poor performance for small I/O requests
RAID 4	Block-level striping with dedicated parity disk	• Parallel read possible • Single disk failure tolerance	• Parity disk bottleneck • Slow small write updates

RAID	Description	Pros	Cons
RAID 5	Block striping with distributed parity	No parity bottleneck; good read performance; single extra disk	Write penalty; complex recovery

Exam-Oriented Summary

- RAID 0: RAID 0 provides performance improvements, but no availability improvement
- RAID 1:RAID 1 (01,10) provides performance and availability improvements but expensive to implement (double the number of disks)
- RAID 5: RAID 5 is cheap (single extra disk), but has poor write update performance
- RAID 2 and RAID 3: Rarely used

12 Final 27

Q1(a) Essential Components of an Operating System

The essential components of an operating system along with at least one service provided by each are listed below:

- **Process Management:** Creation, scheduling, and termination of processes.
- **Memory Management:** Allocation and deallocation of main memory to processes.
- **File System Management:** Creation, deletion, reading, and writing of files.
- **I/O Device Management:** Control and coordination of input/output devices through device drivers.
- **Secondary Storage Management:** Management of disk space and disk scheduling.
- **Protection and Security:** Access control and protection of system resources.
- **User Interface:** Provides command-line or graphical interface for user interaction.

Q1(b) Process States and State Transitions

Process States

A process can be in one of the following states:

- **New**

- **Ready**
- **Running**
- **Waiting (Blocked)**
- **Terminated**

Assume the process is currently in the **Running** state.

State Transitions

I. **Page Fault Occurs**
The process must wait until the required page is brought into memory.
Next State: Running → Waiting

II. **Request to Secondary Storage (File Write Operation)**
Writing "I print it" to `tt.txt` requires disk I/O.
Next State: Running → Waiting

III. **Interrupted by Another Process**
CPU is preempted and given to another process.
Next State: Running → Ready

Q1(c) Handling SVC Exception in User Mode

When a system exception occurs due to the execution of an SVC (Supervisor Call) instruction in user mode, the following steps are performed:

1. The CPU detects the SVC instruction.
2. The processor switches from **user mode** to **kernel mode**.
3. The current process context (PC, registers, status) is saved.
4. Control is transferred to the SVC handler routine.
5. The operating system executes the requested system service.
6. After completion, the saved context is restored.
7. The CPU switches back to **user mode**.
8. Execution resumes from the instruction following the SVC call.

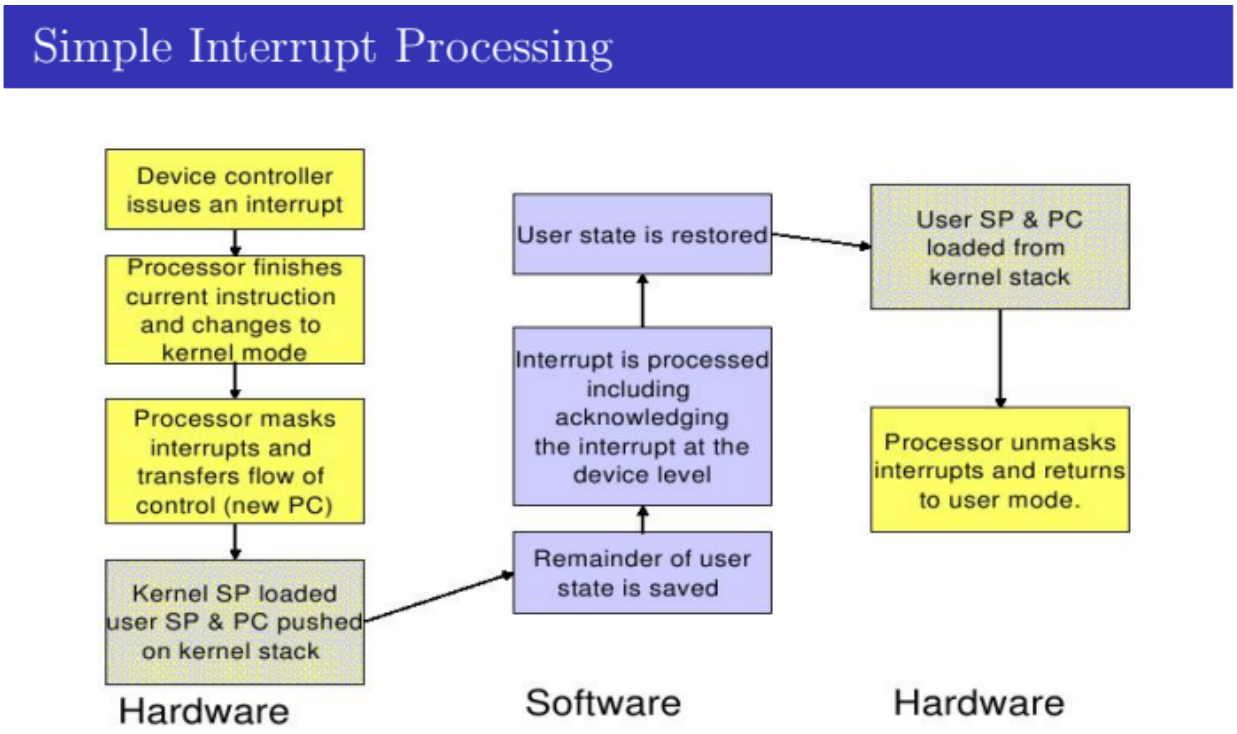


Figure 13: Interrupt Processing

Figure 1: Caption

13 qs 2

Given

- Heap size = 20 MB
- Heap start address = `0x20000000`
- Stack size per process = 200 KB
- Architecture = 32-bit

Data Type Sizes (32-bit system)

Type	Size
int	4 B
float	4 B
double	8 B
pointer	4 B
char	1 B

Memory Rules

- Local variables and pointers are stored in **stack**
- Dynamically allocated memory using `malloc()` is stored in **heap**

Step (i): Process X allocates 64 bytes for `*a`

- Pointer `a` on stack: 4 B
- Heap memory allocated: 64 B

Step (ii): Process X declarations

Stack usage (Process X)

$$\texttt{int b} = 4 \text{ B}, \quad \texttt{float *f} = 4 \text{ B}, \quad \texttt{double *d} = 4 \text{ B}$$

$$\textbf{Total stack usage (X)} = 4 + 4 + 4 = 12 \text{ B}$$

Heap usage (Process X)

$$\texttt{*f} : 10 \times 4 = 40 \text{ B}$$

$$\texttt{*d} : 32 \times 8 = 256 \text{ B}$$

$$\textbf{Total heap (X)} = 64 + 40 + 256 = 360 \text{ B}$$

Step (iii): Process Y allocates 32-byte character array

- Pointer `c` on stack: 4 B
- Heap allocation: $32 \times 1 = 32$ B

Step (iv): Process X frees memory of `*f`

$$\text{Freed heap memory} = 40 \text{ B}$$

$$\textbf{Updated heap (X)} = 360 - 40 = 320 \text{ B}$$

Step (v): Process Z allocates 20 floats

$$20 \times 4 = 80 \text{ B}$$

- Pointer `z` on stack: 4 B
- Heap usage (Z): 80 B

Final Heap Usage

Process	Heap Used
X	320 B
Y	32 B
Z	80 B
Total	**432 B**

Final Stack Usage

Process	Stack Used
X	12 B
Y	4 B
Z	4 B

UART Interrupt Stack Usage (Process X)

Assuming four general-purpose registers are saved:

$$4 \times 4 = 16 \text{ B}$$

$$\text{PC} = 4 \text{ B}, \quad \text{Status Register} = 4 \text{ B}$$

$$\textbf{Total interrupt stack usage} = 16 + 4 + 4 + 12 = \boxed{36 \text{ B}}$$

14 qs 6

Physical Address Calculation using LRU

Given

- Page size = 512 bytes = $0x200$
- Physical memory start address = $0x10000000$
- Three physical frames are numbered: $5, 9, 2$

Frame Mapping

$$\text{LRU slot } 0 \rightarrow \text{Physical frame } 5$$
$$\text{LRU slot } 1 \rightarrow \text{Physical frame } 9$$
$$\text{LRU slot } 2 \rightarrow \text{Physical frame } 2$$

Frame Base Addresses

$$\text{Frame 5 base} = 5 \times 512 = 2560 = 0x0A00$$
$$\text{Frame 9 base} = 9 \times 512 = 4608 = 0x1200$$
$$\text{Frame 2 base} = 2 \times 512 = 1024 = 0x0400$$

Physical Address Formula

$$\text{Physical Address} = 0x10000000 + \text{Frame Base} + \text{Offset}$$

Address Translation Table

Step	Logical Addr	Offset	Frame	Physical Address (Hex)
1	230	230	5	0x10000AE6
2	520	8	9	0x10001218
3	1000	488	9	0x100013E8
4	300	300	5	0x10000B2C
5	3000	440	2	0x100005B8
6	2000	464	9	0x100013D8
7	500	500	5	0x10000BF4
8	270	270	5	0x10000B0E
9	2400	352	2	0x10000560
10	1600	64	9	0x10001240
11	750	238	5	0x10000AEE
12	350	350	2	0x1000055E
13	2000	464	9	0x100013D8
14	340	340	2	0x10000554
15	269	269	2	0x1000050D
16	200	200	2	0x100004C8
17	1020	508	5	0x10000BFC

Conclusion

Each logical address is translated using LRU-selected frames. The physical address is obtained by adding the frame base address and offset to the memory start address.

14.1 Qs 7

Solution

Given:

- Number of direct pointers = 10
- One single, one double, and one triple indirect pointer
- Block size, $B = 512$ bytes
- Block pointer size, $P = 4$ bytes
- File system size = 4 GB

(a) Average Read/Write Access Operations

Number of block pointers in one indirect block:

$$N = \frac{B}{P} = \frac{512}{4} = 128$$

Number of addressable data blocks:

$$\begin{aligned}\text{Direct blocks} &= 10\\ \text{Single indirect blocks} &= 128\\ \text{Double indirect blocks} &= 128^2 = 16{,}384\\ \text{Triple indirect blocks} &= 128^3 = 2{,}097{,}152\end{aligned}$$

Total number of blocks:

$$10 + 128 + 16{,}384 + 2{,}097{,}152 = 2{,}113{,}674$$

$$2{,}113{,}674 \times 512 = 1{,}082{,}201{,}088 \text{ bytes} \approx 1.08 \text{ GB}$$

Disk access cost per block:

- Direct block: 1 access
- Single indirect block: 2 accesses
- Double indirect block: 3 accesses
- Triple indirect block: 4 accesses

Average number of disk accesses:

$$\frac{(10 \times 1) + (128 \times 2) + (16{,}384 \times 3) + (2{,}097{,}152 \times 4)}{2{,}113{,}674} \approx 3.99 \approx 4$$

Average read/write access operations $\approx \boxed{4}$.

—

(b) Average Access Time for Reading a Byte

Given:

- 60% on disk with access time = 10 ms
- 5% in processor cache with access time = 10 ns
- 35% in RAM with access time = 100 ns

Convert all times to nanoseconds:

$$10 \text{ ms} = 10{,}000{,}000 \text{ ns}$$

Average access time:

$$T = 0.60(10{,}000{,}000) + 0.05(10) + 0.35(100)$$

$$T \approx 6{,}000{,}036 \text{ ns} \approx 6 \text{ ms}$$

Average access time $\approx \boxed{6 \text{ ms}}$.

—

(c) Strategy for Accessing Files in Heterogeneous File Systems

Efficient file access strategies for heterogeneous file systems such as VFAT, EXT4, and XFS include:

- Aggressive use of caching (inode cache, page cache)
- Minimizing multi-level indirection
- Using extent-based allocation (EXT4, XFS)
- Exploiting spatial and temporal locality
- Applying read-ahead and write-behind techniques
- Choosing file systems based on workload characteristics

Conclusion: Efficient access is achieved by reducing disk accesses, improving cache utilization, and selecting suitable file systems for specific workloads.

15 Final 25

Question 1

(a) (i) A monolithic kernel provides convenient access to operating system data structures because all kernel services run in the same address space, allowing direct access without inter-process communication.

(ii) A microkernel design is better suited for adding or modifying operating system components since most services run in user space and can be updated independently without changing the core kernel.

(iii) Microkernels provide stronger security and reliability due to their minimal kernel size, fault isolation, and strict separation between kernel and user-level services.

(b) (i) Disabling all interrupts is a kernel-level operation because it requires privileged instructions that must not be accessible from user mode.

(ii) The `scanf()` function is a user-level operation. It is a library function that executes in user mode and internally invokes system calls for input operations.

(c) A user program must pass the system call number, input parameters, memory addresses of data buffers, and control information to the kernel.

To multiply two 64×64 matrices using a DSP unit that is not accessible from user mode, the user program invokes a system call with pointers to the input matrices and output buffer. The kernel switches to kernel mode, validates the parameters, copies the data to kernel or DSP-accessible memory, configures and invokes the DSP to perform the computation, copies the result back to user space, and finally returns control to the user program.

Question 2

(a) (i) Synchronous communication blocks the sender until the receiver is ready, making it simpler to design but reducing concurrency and performance. Asynchronous communication allows the sender to continue execution, improving performance and scalability at the cost of increased complexity.

(ii) Automatic buffering simplifies programming by letting the operating system manage buffers, but offers less control and may introduce overhead. Explicit buffering provides greater control and efficiency but increases programming complexity and risk of errors.

(iii) Fixed-size messages are easy to manage and fast to process but can waste memory due to internal fragmentation. Variable-size messages utilize memory efficiently but require more complex buffer management.

(b) (i) If the number of kernel threads is less than the number of processing cores, some cores remain idle, resulting in underutilization of system resources.

(ii) When the number of kernel threads equals the number of processing cores, each core can execute one kernel thread, achieving maximum parallelism and optimal performance.

(iii) If the number of kernel threads is greater than the number of processing cores but less than the number of user-level threads, all cores are utilized but additional context switching overhead is introduced, resulting in slightly reduced performance.

(c) In a multithreaded process, register values and stack memory are private to each thread, while heap memory and global variables are shared among all threads.

Question 3

(a) (i) Shortest Job First (SJF) can be considered a special case of Priority scheduling if process priority is defined as the inverse of the CPU burst time.

(ii) Multilevel Feedback Queue scheduling reduces to FCFS when it is configured with a single queue, no feedback between queues, and FCFS scheduling within the queue.

(iii) Priority scheduling degenerates to FCFS when all processes are assigned the same priority.

(iv) Round Robin and SJF have no direct relationship, as no choice of time quantum can make Round Robin behave like SJF.

(b) (i) Maximizing CPU utilization often conflicts with minimizing response time, as improving response time favors short or preemptive jobs, which may reduce CPU utilization.

(ii) Minimizing average turnaround time may increase maximum waiting time, since long jobs can suffer starvation, while limiting maximum waiting time requires fairness.

(iii) Maximizing I/O device utilization can conflict with CPU utilization, as balancing CPU and I/O overlap may introduce idle periods for the CPU.

Question 4

(a) (i) The critical sections are the parts of code where shared variables `count` and `inc_count` are accessed and modified:

```
value = count;   value++;   count = value;   inc_count++;
```

These sections are critical because the operations are not atomic. In a multiprocessor environment, concurrent execution can cause race conditions where the value of `count` is modified by another processor between read and write operations, leading to inconsistent results. SVC calls may also cause context switches, further increasing the likelihood of race conditions.

(ii) The synchronization issue can be resolved using a mutex, which avoids busy waiting and spin locks:

```
uint32_t value;
uint32_t inc_count = 0;
mutex m;

while (1) {
    lock(m);

    value = count;
    value++;

    if (value != count + 1) {
        printf("Error %d != %d\n", value, count + 1);
    } else {
        count = value;
        inc_count++;
    }

    unlock(m);

    if (count >= 10000000) {
        uint16_t task_id = getpid();
        printf("Total increments done by task %d is: %d\n",
               task_id, inc_count);
        break;
    }
}
```

(b) The system has five resource types with available resources:

$$E = [5, 10, 7, 9, 6]$$

At time t_1, the request matrix R_{t1} can be satisfied since all requests are less than or equal to the available resources. After allocation, a safe sequence exists, so the system is not in a deadlock state.

At time t_2, the new request matrix R_{t2} contains requests that exceed the remaining available resources. No safe sequence exists, and the system enters a deadlock state.

Question 5

(a) Physical memory size is 2^{32} bytes and page size is 2^{24} bytes.

$$\text{Offset bits} = \log_2(2^{24}) = 24$$

$$\text{Number of frames} = \frac{2^{32}}{2^{24}} = 2^8 \Rightarrow \text{Frame number bits} = 8$$

Logical address space is 2^{48} bytes.

$$\text{Page number bits} = 48 - 24 = 24 \Rightarrow \text{Number of pages} = 2^{24}$$

Each page table entry contains 8 frame bits and 4 control bits.

$$\text{Page table size} = 2^{24} \times 12 = 201{,}326{,}592 \text{ bits}$$

(b) Since the working set contains 20 pages, the TLB should contain at least 20 entries. Each TLB entry contains:

$$24 \text{ page bits} + 8 \text{ frame bits} + 4 \text{ control bits} = 36 \text{ bits}$$

$$\text{Total TLB size} = 20 \times 36 = 720 \text{ bits}$$

(c) Effective Access Time (EAT) is given by:

$$\text{EAT} = h(T + M) + (1 - h)(T + (L + 1)M)$$

Substituting the values:

$$160 = 0.8(20 + 100) + 0.2(20 + (L + 1)100)$$

$$160 = 96 + 24 + 20L \Rightarrow L = 2$$

Thus, a two-level page table is required.

(d) The EIS web server uses a thread-per-request model, where each client request is handled by a separate thread.

When thousands of requests arrive simultaneously, a very large number of threads are created. This leads to excessive context switching overhead, high memory consumption due to per-thread stack space, and increased contention for shared resources such as CPU, locks, and I/O devices.

As a result, the operating system spends more time managing threads than processing requests, causing increased response time and degraded overall server performance.

Question 6

(a) A Unix i-node contains 16 direct block addresses and one each of single, double, and triple indirect block addresses. Each indirect block contains 128 disk addresses. The disk block size is 2 KB.

Solution:

Number of blocks addressed:

$$\text{Direct blocks} = 16$$

$$\text{Single indirect blocks} = 128$$

$$\text{Double indirect blocks} = 128 \times 128 = 16{,}384$$

$$\text{Triple indirect blocks} = 128 \times 128 \times 128 = 2{,}097{,}152$$

Total number of data blocks:

$$16 + 128 + 16{,}384 + 2{,}097{,}152 = 2{,}113{,}680$$

Each block is 2 KB, so maximum file size:

$$2{,}113{,}680 \times 2 = 4{,}227{,}360 \text{ KB}$$

Converting to GB:

$$\frac{4{,}227{,}360}{1024 \times 1024} \approx 4.03 \text{ GB}$$

$$\boxed{\text{Maximum file size} \approx 4.03 \text{ GB}}$$

(b) The file `/home/user1/test/test1.html` is stored in 3 disk blocks. The root directory inode is already in memory. Each directory file and each inode occupies 2 disk blocks.

Solution:

To read the file, the system must traverse each directory in the path.

- Read `/` directory file to find `home` : 2 disk accesses
- Read `/home` directory file to find `user1` : 2 disk accesses
- Read `/home/user1` directory file to find `test` : 2 disk accesses
- Read `/home/user1/test` directory file to find `test1.html` : 2 disk accesses

Total directory accesses:

$$2 \times 4 = 8$$

Reading the inode of `test1.html`:

2 disk accesses

Reading the file data (3 blocks):

3 disk accesses

Total disk accesses:

$$8 + 2 + 3 = \boxed{13}$$

(c) The system uses a Virtual File System (VFS). To deploy a new file system `csefs`, the kernel developer must:

- Implement the required VFS interfaces (inode operations, file operations, directory operations, and superblock operations)
- Register the new file system with the VFS layer
- Provide mount and unmount routines

No changes are required to user programs or system calls, as VFS automatically routes requests to the new file system.

Question 7

(a) A disk has 2000 cylinders (0–1999). The current head position is 143 and the previous position was 125. The request queue is:

$$86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130$$

(i) SSTF

Service order:

$$143 \to 130 \to 86 \to 913 \to 948 \to 1022 \to 1470 \to 1509 \to 1750 \to 1774$$

Total head movement:

$$\boxed{1745 \text{ cylinders}}$$

(ii) SCAN

Service order:

$$143 \to 913 \to 948 \to 1022 \to 1470 \to 1509 \to 1750 \to 1774 \to 1999 \to 130 \to 86$$

Total head movement:

$$\boxed{3769 \text{ cylinders}}$$

(iii) C-SCAN

Service order:

$$143 \to 913 \to 948 \to 1022 \to 1470 \to 1509 \to 1750 \to 1774 \to 1999 \to 0 \to 86 \to 130$$

Total head movement:

$$\boxed{3985 \text{ cylinders}}$$

(b) To ensure data safety at a low cost, RAID 5 is the most suitable choice.

RAID 5 provides fault tolerance using parity while requiring only one additional disk. It can tolerate a single disk failure and offers a good balance between cost, performance, and reliability.

Aspect	User-Level Threads	Kernel-Level Threads
Implementation	Runtime library in user space	OS kernel manages threads
Thread Management	User-level TCB, ready queue, blocked queue, dispatcher	Kernel maintains TCBs and queues
Kernel Awareness	Kernel sees only single process	Kernel aware of all threads
System Calls	No system calls for thread operations	Thread operations require system calls
ADVANTAGES		
Performance	**Fast:** Thread switching at user level (no kernel trap)	Slower: Requires kernel entry/exit
Portability	Can run on any OS (thread or non-thread aware)	Requires OS support
Scalability	Can support massive numbers of threads per application	Limited by kernel memory
Customization	Dispatcher algorithm can be tuned to application (e.g., custom priorities)	Fixed kernel scheduling policy
Overhead	Low memory overhead (uses normal application virtual memory)	Higher overhead (kernel data structures)
Multithreading Type	N/A	**Preemptive multithreading** supported
Multiprocessor Support	N/A	**Can utilize multiple CPUs** for true parallelism
I/O Overlap	N/A	**Can overlap blocking I/O with computation**
DISADVANTAGES		
Multithreading Type	**Cooperative only:** Threads must yield() manually	N/A
CPU Monopolization	**Single bad thread can monopolize CPU** (no preemption)	Timer interrupts prevent monopolization
Multiprocessor Support	**Cannot utilize multiple CPUs** (single process to kernel)	N/A
Blocking System Calls	**If one thread blocks, entire process blocks** (all threads stop)	N/A
I/O Overlap	**Cannot overlap I/O with computation**	N/A
Workarounds Needed	**Requires wrappers** for blocking calls (e.g., select() before read()) - adds overhead	N/A

Component	Location and Details
PROCESS/THREAD CONTROL BLOCK (PCB/TCB)	
PCB/TCB Header	**Location:** Kernel space (in scheduling queue) **Contains:** • Stack pointer (SP/PSP) of the process • Process/Thread ID • State (Running, Ready, Blocked) • Priority • Statistics • Owner information • Pointer to next process in queue
PCB Storage	Forms a process table in kernel memory Entry deleted when process terminates New entry inserted when process created
STACK FRAME (TRAP FRAME)	
Stack Frame	**Location:** Process/Thread stack (in process address space) **NOT in kernel stack** (common misconception)
Stack Frame Contents	Saved register values for the suspended process: **Automatically saved by hardware:** • xPSR (Program Status Register) • PC (Program Counter) • LR (Link Register) • r12, r0-r3 (ARM registers) • D0-D15 (FPU registers if FPU active) **Saved by OS scheduler (e.g., PendSV):** • r4-r11 (remaining ARM registers) • Other FPU registers if needed
Why Process Stack?	• Each process has its own stack in its address space • When process resumes, stack is automatically accessible • Kernel stack would require extra copying • More efficient memory management

Table 2: Stack Frame and PCB Placement

Step	Actions
STEP 1: SAVE EVERYTHING	
1	Save CPU registers (so we can return to interrupted process)
2	Set up stack for interrupt handler
3	Save processor state not already saved by hardware
4	Optionally set up execution context (address space)
STEP 2: HANDLE THE INTERRUPT	
5	Acknowledge the interrupt (tell device "I received it")
6	Figure out what caused the interrupt: - Disk read/write completed? - Network packet arrived? - Keyboard key pressed? - UART transmit queue empty? - Timer tick?
7	Run interrupt service procedure (device-specific handling)
8	Wake up any driver that was waiting for this event
9	If needed, signal blocked device driver
10	Possibly wake up higher-priority blocked thread
STEP 3: RETURN	
11	Choose which process/thread to schedule next
12	Set up MMU/address-space context for selected process
13	Restore registers of new or original process
14	Re-enable interrupts
15	Continue execution (may be different process!)

Table 3: Complete Interrupt Handler Execution Flow

Strategy	How it Works	Problems	Efficiency
No Buffering	Process reads/writes one byte at a time	Too many system calls, Very inefficient	Very Low
User-Level Buffer	Process provides buffer, OS fills it	Buffer may be swapped to disk → data loss	Medium
Single Buffer	One kernel buffer, Process works on one block while next is read	If full and more data arrives → data loss	Good
Double Buffer	Two buffers: one filling, one processing	May be insufficient for bursty traffic	Better
Circular Buffer	Multiple buffers in circular arrangement	None (best solution)	Excellent

Table 4: Comparison of I/O Buffering Strategies

Layer	Components	Responsibilities
5. User-Level I/O	I/O libraries, System call interfaces	Provides functions: open(), read(), write(), close()
4. Device-Independent OS	Uniform naming, Protection, Buffering	Error handling, Naming, Access control, Buffer management (independent of device type)
3. Device Drivers	Device-specific control code	Translate generic commands to device-specific commands, Interface between OS and hardware
2. Interrupt Handlers	Interrupt service routines	Respond to device interrupts, Minimal time-critical processing, Wake up blocked drivers
1. Hardware	Physical devices and controllers	Actual I/O devices (disks, network cards, keyboards, etc.)

Table 5: I/O Software Layers

Issue	Challenge	OS Strategy
Efficiency	I/O much slower than CPU	Multiprogramming, Buffering, Caching, Optimize disk/network I/O
Generality	Device diversity (speed, access methods, formats)	Layered design, Uniform API, Device drivers hide specifics
Reliability	Devices fail, lose data	Error detection/correction, Timeouts, Retry mechanisms
Fairness	Multiple processes competing for I/O	Scheduling algorithms, Request queues, Priorities

Table 6: OS Design Issues and Solutions