**CHAPTER 10**

**1a)**

* **SSDs have no mechanical moving parts:**

They don’t suffer from seek time or rotational latency caused by physically moving a read/write head.

* **Uniform Access Time**:

Every memory cell in an SSD can be accessed in roughly thesame amount of time.

* **Lower Overhead:**

FCFS is a simple and fair algorithm with very low scheduling overhead. Since performance gain from complex algorithms is minimal for SSDs.

**1b)**

* **FCFS:**

Add absolute difference between each cylinder. i.e. |current-next| and we start from cylinder 2150.

Total = 81+857+1084+504+2256+1074+1262+1167+3442+1284

= **13,011 cylinders**

* **SSTF:**

Calculate distances from the current head and always move to the closest request.

Head movement = |current head- Closest Request | e.g. From 2150, the closest is 2069

**distance=81**

Total = 81+227+504+881+1284+3347+95+311+668+188

= **7586 cylinders**

* **SCAN:**

Current head=2150,

Head movement = |current-next|,

Upward head movement:(2150-2296=146, 2296-2800=504, 2800-3681=881, 3681-4965=1284).

Reverse = (4965- 2069 = 2896, 2069-1618=451, 1618-1523 = 95, 1523-1212 = 311, 1212-544 = 668, 544-356=188)

Total = 146+504+881+1284+2896+451+95+311+668+188

= **7,424 cylinders**

* **LOOK:**

|2150-1618|+|1618-1523|+|1523-1212|+|1212-544|+|544-356|+|356-2069|+|2069-2296|+|2296-2800|+|2800-3681|+|36814965|532+95+311+668+188+1713+227+504+881+1284

= **6403 cylinders**

* **C-SCAN:**

Summing the absolute differences between consecutive serviced requests, but with a "wrap-around" to the opposite end of the disk after reaching the maximum cylinder.

|2150-2296|+|2296-2800|+|2800-3681|+|3681-4965|+|4965-4999|+|0-356|+|356-544|+|544-1212|+|1212-1523|+|1523-1618|+|1618-2069|= 146+504+881+1284+34+356+188+668+311+95+451

= **9917 cylinders**

* **C-LOOK:**

Goes to the last request then jumps to lowest request(left) and continues.

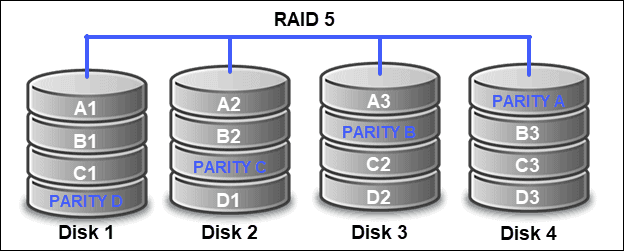
|2150-2296|+|2296-2800|+|2800-3681|+|3681-4965|+|4965-356|+|356-544|+|544-1212|+|1212-1523|+|1523-1618|+|1618-2069| = 146+504+881+1284+4609+188+668+311+95+451

= **9137 cylinders**

**2a)**

1. RAID 5 is considered the most secure and most common RAID implementation. It combines striping and parity to provide a fast and reliable setup. Such a configuration gives the user storage usability as with RAID 1 and the performance efficiency of RAID 0.

This RAID level consists of at least three hard drives (and at most, 16). Data is divided into data strips and distributed across different disks in the array. This allows for high performance rates due to fast read data transactions which can be done simultaneously by different drives in the array.



From the diagram above, for a single-block write, RAID 5 requires:

* **Reading the old data block (1 read)**
* **Reading the old parity block (1 read)**
* **Writing the new data block (1 write)**
* **Writing the updated parity block (1 write)**

**Total blocks accessed = 4 (2 reads + 2 writes)**

ii) If we assume that the write is stripe-aligned (that is begins at the beginning of a physical block), writing seven blocks’ results in the following block accesses:

* **First four blocks:** five writes (four data blocks and a parity block),
* **Last three blocks:** one read (for the unchanging block), plus four writes (three new data blocks plus new parity block) The total is 10 block accesses.
* **If the write is not stripe-aligned then the write requires 15 disk accesses:** A total of 10 writes for the modified data and parity blocks, plus 5 reads for the non-modified blocks on the three stripes which nevertheless must be read in order to calculate parity.

**2b)**

#### 1. **Efficient Disk Access:**

#### Understanding the physical layout of blocks helps the OS reduce disk **seek time** and **rotational delay**, which are major contributors to I/O latency.

#### 2. **Optimal File Allocation:**

The OS can place files or file blocks in **sequential or nearby blocks**, improving read/write performance and reducing fragmentation.

#### 3. **Intelligent Scheduling:**

With block location data, the OS can implement disk scheduling algorithms (like **elevator or SCAN**) to service I/O requests in an order that minimizes disk head movement.

#### 4. **Effective Caching and Prefetching:**

When the OS knows that blocks are stored sequentially, it can pre-load blocks into memory before they’re requested (read-ahead), improving application responsiveness.

#### 5. **Reduced Fragmentation:**

Accurate knowledge helps in minimizing both internal and external fragmentation, keeping the file system clean and efficient.

6. **Enhanced Performance for Large Files:**

Large files like videos or databases benefit from contiguous block placement, leading to **smoother streaming** and **faster access**.

**2c)**

**Sector Sparing:**

Sector sparing is the practice of setting aside one or more sectors as backup or reserve. These sectors remain inactive during normal operation and are only utilized in cases of failure, overload, or scheduled maintenance in the active sectors.

**Advantages:**

* **Increased Reliability:** Sparing provides redundancy, ensuring the system remains operational even if a sector fails.
* **Maintenance Flexibility:** Reserved sectors can replace active ones during maintenance, minimizing downtime.
* **Improved System Longevity:** Since some sectors are not always in use, their operational lifespan can be extended.
* **Controlled Load Management:** Spare sectors can be activated during peak loads to balance performance.

**Disadvantages:**

* **Resource Underutilization:** Idle sectors represent unused capacity, leading to inefficiency.
* **Higher Costs:** Additional infrastructure must be developed and maintained for backup purposes.
* **Management Complexity:** Systems need logic to monitor and activate spare sectors appropriately.

**Sector Slipping:**

Sector slipping refers to dynamically reallocating or shifting workloads, capacity, or functions between sectors in response to changes in demand or performance issues. Rather than keeping spare capacity idle, the system continuously balances loads across all sectors.

**Advantages:**

* **Efficient Resource Utilization:** All sectors are actively used, maximizing operational capacity.
* **Dynamic Adaptability:** Can respond in real time to shifting demands or sector issues.
* **Cost-Effective:** No need to maintain idle backup sectors, reducing infrastructure expenses.
* **Balanced Load Distribution:** Helps prevent bottlenecks by redistributing workload dynamically.

**Disadvantages:**

* **Potential Instability:** Frequent adjustments may cause temporary performance degradation.
* **Implementation Complexity:** Requires real-time monitoring and intelligent control systems.
* **Risk of Overload:** If shifting is not well-managed, it could lead to other sectors being overburdened.
* **Latency or Transition Delays:** Transferring load may introduce delays or disruptions.

**3a)**

**MTBF of a single drive = 750,000hours**

**Number of drives = 1000**

**Failure rate per drive failures per hour**

**Failure rate for 1000 drives = 1000 × failures per hour**

**= = failures per hour**

**MTBF for the entire system = 750 hours (since a failure occurs every 1/failure hour)**

**750 hours/24hours per day = 31.25 which is approximately a month.**

**A disk failure will occur approximately once a month.**

**CHAPTER 12**

**1a)**

**1. Criticality of the Event**

* Interrupts triggered by catastrophic or irrecoverable errors (like hardware failures or memory faults) must be given the highest priority, as these events need immediate attention to prevent system instability or failure.

**2. Time Sensitivity**

* Interrupts related to real-time operations, such as device controllers signaling that they are ready for data transfer or the completion of a critical process, should have higher priority to ensure time-sensitive tasks are handled quickly without delay.

**3. Resource Management and Contention**

* When multiple interrupts arise simultaneously, the CPU must ensure that resources are allocated appropriately. For example, a high-priority interrupt that requires access to the same resources as a low-priority interrupt may need careful management to avoid deadlocks or resource starvation.

**4. Interrupt Source**

* A network interface card (NIC) may need quicker handling than a keyboard interrupt, due to the nature of data flow and real-time communication requirements.

**5. Preemption Behavior**

* The priority scheme should consider how one interrupt can preempt another. Higher priority interrupts should interrupt lower priority ones.

**6. Interrupt Handling Overhead**

* More complex or time-consuming interrupt handlers might need to be assigned lower priorities.

**7. System Throughput and Efficiency**

* System throughput can be impacted by the way interrupts are prioritized. If too many high-priority interrupts are allowed to dominate the CPU.

**8. Frequency of Interrupts**

* Devices that generate interrupts frequently but are not time-critical (e.g., periodic background tasks) may need lower priority to prevent the system from being overwhelmed by constant interrupt handling.

**9. Interdependencies Between Interrupts**

* Some interrupts may depend on the handling of others before they can be serviced.

**10. Impact on User Experience**

* Interrupts related to user interface actions (like keyboard or mouse input) may be assigned higher priority..

**11. Interrupt Vector Table**

* The interrupt vector table directly influences how the system will locate and dispatch the correct interrupt handler, which can affect the handling order, and thus indirectly the priority of interrupts.

**1b)**

**The performance overhead associated with servicing an interrupt includes:**

* Saving the state of the currently executing process (program counter, registers, etc.).
* Mode switching from user mode to kernel mode.
* Context switching to the interrupt handler, which may involve cache invalidations or TLB flushes.
* Execution of the interrupt handler, which takes CPU cycles.
* Restoring the process state **and resuming the interrupted process.**

**Virtual versus physical addressing:**

* In most multi programmed systems, user programs use virtual addresses, while the operating system uses physical addresses. This separation ensures security and isolation between processes. However, this introduces complexity in I/O operations:
* User programs cannot initiate I/O operations directly using virtual addresses, because I/O devices operate using physical addresses. Hence, the operating system must translate virtual addresses to physical addresses before starting any I/O operation. This means that the OS must intervene in every I/O request to ensure correct addressing and protection.

**1c)**

**1. Standard Contiguous Allocation:**

* **How it works**: The file is stored in a single, contiguous block on disk.
* **Advantages:**
  + Fast sequential and direct access.
  + Simple to implement.
* **Disadvantages:**
  + Requires knowing the file size ahead of time.
  + May lead to external fragmentation.
  + Not suitable for dynamic file growth—requires copying or relocation if the file outgrows its space.

**2. Standard Linked Allocation:**

* **How it works:**
  + Each file is a linked list of disk blocks scattered anywhere on the disk.
* **Advantages:**
  + No need for knowing file size in advance.
  + No external fragmentation.
  + Easy to grow files.
* **Disadvantages:**
  + No direct access—only sequential.
  + Overhead due to pointers in each block.
  + More complex to recover from disk errors (loss of one pointer can break the chain).

**3. Hybrid (Contiguous with Overflow Areas):**

* **How it works:**
  + Starts with a contiguous block of a predefined size.
  + If more space is needed, overflow blocks are allocated and linked to the original.
  + Overflow areas can themselves be chained if needed.
* **Advantages:**
  + Fast access within the initial block (like contiguous allocation).
  + Allows for file growth (like linked allocation).
  + Reduces initial overhead of storing pointers in each block.
* **Disadvantages:**
  + Slightly more complex to manage.
  + Accessing overflow areas may reduce performance, especially with multiple overflows.
  + Potential internal fragmentation in initial block if actual file size is much smaller than the preallocated size.

**2a)**

**UNIX Approach:**

***Pros:***

* **Fast Access -** Kernel components directly access memory structures enabling quicker execution
* **Low Overhead –** Fewer abstractions reduce performance penalties from data conversion
* **Precise Control –** Developers have more direct control over I/O handling at the kernel level.

***Cons:***

* **Synchronization Issues –** Shared structures mean multiple components might access them simultaneously, risking data corruption.
* **One part depends too much on the other –** Kernel modules are less modular and more interdependent making it difficult to update without breaking things.
* **Harder to Debug -** Low-Level handling without encapsulation makes error tracking harder.

**WINDOWS Approach**

***Pros:***

* **Separation of Parts (Modularity) –** Components are isolated and communicate through messages, supporting abstraction and reuse.
* **Easier Maintenance -** Object-oriented design allows simpler upgrades to components.
* **Errors don’t spread –** Errors in one module are less likely to affect others since the error can’t spread.

***Cons:***

* **Higher Overhead –** Message Passing systems can be slower due to conversion and queue handling.
* **Increased complexity –** Managing objects and messages requires more system resources
* **Memory Usage –** Message objects and queues require memory beyond what shared data might use.

**2b)**

***Advantages to the user*.**

* Elimination of Artificial limits (fixed-size tables)
* Improved System Utilization
* Supports multiprogramming
* Increases Flexibility.

***Penalties to the operating system***

* Increased complexity since it requires more sophisticated algorithms to handle memory management.
* Higher overhead caused by the processes of allocating and deallocating table entries which dynamically introduces additional overhead.
* Potential for fragmentation overtime. Where memory may be divided into smaller contiguous blocks and this degrade performance and lead to inefficient memory access.
* Resource Exhaustion when there Is no Strict limits, where memory leaks or overuse may occur.
* Unpredictable Performance and delays may occur under heavy load.

**2c)**

**Pure Polling –** This is more efficient in environments where I/O devices are extremely fast and data transfer is continuous such as in Network Interface Cards (NICs) in high performance computing.

**Pure Interrupts –** More efficient in environments where I/O devices are relatively slow such as with Keyboards, mice or printers.

**Hybrid Strategy –** This works best in environments where I/O device response times are unpredictable i.e. Sometimes fast, Sometimes Slow. For example, Wireless Network Interfaces.

**2d)**

**Access time**

* If a file is accessed sequentially, linked allocation is best because blocks are connected in order allowing the system to read one after the other easily.
* If the file needs random/direct access, linked allocation is preferred since it allows direct lookup.
* Contiguous also supports direct access but finding space for a new file is difficult

**File size or growth pattern**

* Accessing a file that is known in advance and unlikely to change is best done contiguously.
* If the file grows overtime or is dynamic in nature, linked or indexed allocation is better.

**Performance requirements**

* Contiguous allocation has the best-read performance (especially for sequential access), but it suffers from external fragmentation.
* Linked allocation avoids external fragmentation but has slower access time because pointers are followed when finding block of a file.
* Indexed allows efficient direct access but uses more index blocks.

**Fragmentation Handling**

* Avoid contiguous allocation if minimizing external fragmentation is important.
* If internal fragmentation is more acceptable, indexed or linked allocation is better

**Disk space management:**

* Contiguous allocation needs large continuous free space which also causes external fragmentation.
* Linked allocation uses scattered blocks, making it easier to utilize free space but wastes space due to pointer overhead.
* Indexed allocation allocates all pointers in one index block, saving space during access.

**2e)**

**Performance Benefits:**

* **Large files can be can be stored in larger blocks e.g. 4kb blocks to reduce managing smaller blocks and small files can be stored in small blocks ed 512- byte blocks reduce internal disk space.**
* **Reducing fragmentation hence making disk space management more efficient.**
* **Efficient disk I/O by using larger blocks for large files minimizes the number of disk I/O operations required to read or write data improving performance**

**Modifications to Free-Space Management:**

* **The file system must decide when to use small/ large blocks based on file size, access patterns and performance consideration.**
* **Efficient block merging i.e If a file grows, the system should efficiently merge small blocks into large ones to optimize performance.**
* **Metadata changes i.e File metadata must store information about block sizes allocated to a file instead of assuming a fixed block size.**
* **Bitmap i.e The file space management system needs to track both large and small blocks possibly using multiple bitmaps for different block sizes.**