Section A

1. (a) Differentiate between Orphan process and zombie process giving one difference each.
    - **Orphan Process:** An orphan process is a process whose parent process has terminated without waiting for its child to terminate. The orphan process is then adopted by the *init* process (or systemd in modern Linux systems).
    - **Zombie Process:** A zombie process (also known as a defunct process) is a process that has completed its execution but still has an entry in the process table. This entry is kept to allow the parent process to read its child's exit status. If the parent does not call *wait()* or *waitpid()*, the zombie process persists.
2. (b) Explain the convoy effect exhibited by FCFS scheduling algorithm with an example.
    - The convoy effect occurs in the First-Come, First-Served (FCFS) CPU scheduling algorithm. It happens when a CPU-bound process comes first in the ready queue, followed by several I/O-bound processes.
    - **Explanation:** The CPU-bound process will execute for its entire burst time, keeping the CPU busy. During this time, the I/O-bound processes, even if they become ready for a short CPU burst, will be stuck waiting behind the long CPU-bound process. Once the CPU-bound process finally releases the CPU, all the I/O-bound processes will rapidly execute their short CPU bursts and then go back to waiting for I/O. When they return to the ready queue, they might again encounter another long CPU-bound process, leading to a "convoy" of processes waiting for the CPU.
    - **Example:** Consider three processes P1, P2, P3 with burst times 10, 2, 2 milliseconds respectively. If P1 (burst time 10ms) arrives first, followed by P2 (burst time 2ms) and P3 (burst time

2ms), then P1 will run for 10ms, during which P2 and P3 wait. After P1 completes, P2 and P3 will execute quickly. If P1 frequently returns to the ready queue, it will cause P2 and P3 to continuously wait in a "convoy".

3. (c) What will be the output of the parent and child processes in the following code?

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
   int pid;
   cout<<"Hello\n";
   pid = fork();
   if (pid == 0)
   {
      cout << "World\n";
   }
   return 0;
}
```


   - **Output:**
     - "Hello" will be printed once.
     - Then, after *fork()*, two processes (parent and child) will be created.
     - The child process (*pid == 0*) will print "World".
     - The parent process (*pid > 0*) will not enter the *if* block, so it will not print "World".

- The order of "Hello" and "World" might vary. Typically, "Hello" is printed first, and then "World" is printed by the child.
- Example output: Hello World

4. (d) Discuss the problem of cache coherency with an example.
    - **Problem of Cache Coherency:** In a multiprocessor system, each CPU typically has its own local cache. When multiple CPUs access and modify the same shared data, copies of that data might exist in different caches. If one CPU modifies its cached copy, other CPUs might continue to work with their stale (outdated) copies, leading to data inconsistency. This problem is known as cache coherency.
    - **Example:**
        - Consider two CPUs, CPU1 and CPU2, and a shared variable *X* with an initial value of 10 in main memory.
        - Both CPU1 and CPU2 read *X*, and its value (10) is loaded into their respective caches.
        - CPU1 then executes an instruction to update *X* to 20. CPU1's cache now holds *X=20*, while CPU2's cache still holds *X=10*. Main memory might or might not be updated immediately depending on the write policy (write-through or write-back).
        - If CPU2 now reads *X* from its cache, it will retrieve the outdated value of 10 instead of 20, leading to an incorrect result. This demonstrates the cache coherency problem.
5. (e) Why page sizes in memory management scheme are always chosen as power of 2?

- Page sizes in memory management schemes are always chosen as powers of 2 (e.g., 2KB, 4KB, 8KB, etc.) primarily for efficient address translation.
- **Efficient Address Translation:** When a logical address is translated into a physical address, it is divided into a page number and an offset. By using a page size that is a power of 2, this division can be performed very efficiently using bitwise operations (bit shifting and masking) instead of computationally expensive integer division and modulo operations.
- For example, if the page size is 2^k bytes, then the last k bits of the logical address represent the offset within the page, and the remaining higher-order bits represent the page number. This simplifies the hardware implementation of the Memory Management Unit (MMU) and speeds up address translation significantly.

6. (f) Give any one difference between Asymmetric multiprocessing and Symmetric multiprocessing.
   - **Asymmetric Multiprocessing (AMP):** In AMP, one processor (the master server) handles all operating system tasks and controls the system, while other processors (slaves) only execute user processes.
   - **Symmetric Multiprocessing (SMP):** In SMP, all processors are peers and can run any process (including operating system code and user processes). They share a common main memory and are controlled by a single operating system.
7. (g) Which of the following components of program state are shared across threads in a multithreaded process? Answer with yes or no.
   - (i) Register values: No
   - (ii) Heap memory: Yes
   - (iii) Global variables: Yes

- (iv) Stack memory: No

8. (h) Give an example for each of the following:
    - (i) **Privileged Instruction:** An instruction that can only be executed when the CPU is in kernel mode (or supervisor mode).
        - Example: *HALT* (to stop the CPU), *LOAD_TIMER* (to set the system timer), *I/O instructions* (e.g., *READ_PORT*, *WRITE_PORT*).
    - (ii) **Instruction that can be run in user mode:** An instruction that can be executed by user applications without requiring special privileges.
        - Example: *ADD* (addition), *MOVE* (data transfer between registers), *JUMP* (control flow), *READ_MEMORY* (accessing user-space memory).

9. (i) Define data parallelism with suitable example.
    - **Data Parallelism:** Data parallelism is a parallelization strategy where the same operation or task is applied concurrently to different subsets of data. The data is divided into multiple parts, and each part is processed by a different processing unit (e.g., a core, a thread, a GPU stream) simultaneously.
    - **Example:**
        - **Image Processing:** Applying a filter (e.g., blur, sharpen) to a large image. The image can be divided into smaller blocks, and each block can be processed by a separate thread or processor concurrently.
        - **Vector Sum:** Calculating the sum of two large vectors, A and B, to produce a result vector C. Each element $C[i]$ can be computed independently as $A[i] + B[i]$. Different

processing units can work on different ranges of indices simultaneously.

10. (j) Consider a process P1 with values of relocation and limit registers as 600 and 250 respectively. Assume that the process tries to reference the logical address 200. What will be the corresponding physical address and how is it computed?

    - **Relocation Register:** 600
    - **Limit Register:** 250
    - **Logical Address:** 200
    - **Computation:**
        i. **Check against Limit Register:** The logical address (200) must be less than the limit register value (250). Since $200 \< 250$, the address is valid.
        ii. **Calculate Physical Address:** The physical address is computed by adding the logical address to the relocation register value. Physical Address = Relocation Register + Logical Address Physical Address = 600 + 200 = 800
    - **Corresponding Physical Address:** 800

11. (k) Differentiate between Monolithic and Microkernel approach to operating system design. Give any one difference.

    - **Monolithic Kernel:** In a monolithic kernel, all operating system services (process management, memory management, file system, device drivers, etc.) run in a single, large address space in kernel mode.
    - **Microkernel:** In a microkernel, only the absolute essential operating system services (e.g., inter-process communication, basic memory management, process scheduling) run in kernel mode. Other services (like file systems, device drivers, and network protocols) run as user-level processes.

12. (l) In the given process state transition diagram for a uniprocessor system, assume that there are always some processes in the ready state. Which of the following statements are true? Justify.

Process State Transition Diagram:

B
*/ *
A D

Start -> Ready -> Running -> Terminated \ / E F \ / Blocked

(Diagram represents states and transitions: Start to Ready (A), Ready to Running (B), Running to Ready (C), Running to Terminated (D), Blocked to Ready (E), Running to Blocked (F))

- (i) **Termination of a process (Transition D) will result in invoking the long term scheduler to bring another process in the memory (Transition A).**
  - **False.** Transition D (Running -> Terminated) means a process has finished its execution. While termination of a process might create an opportunity for the long-term scheduler to bring a new process into memory (Transition A), it does not **directly** invoke the long-term scheduler as a necessary consequence of every termination. The long-term scheduler is typically invoked less frequently to manage the degree of multiprogramming, not on every process termination. Termination mainly affects the short-term scheduler as it frees up the CPU.
- (ii) **Consider a process P2 that is currently in the "Blocked state". Process P2 can make transition E to the ready state while another process P1 is in running state.**
  - **True.** Transition E (Blocked -> Ready) occurs when an event that P2 was waiting for (e.g., I/O completion) occurs. This event allows P2 to move to the Ready state.

In a uniprocessor system, only one process can be in the Running state at a time (P1 in this case). The transition of P2 from Blocked to Ready does not require the CPU to be idle; P2 simply becomes eligible to run. It will wait in the ready queue until the scheduler picks it, potentially after P1 yields the CPU or terminates.

13. (m) Fill in the blanks.

- (i) **Long-term** Scheduler controls the degree of multi-programming.
- (ii) Privileged instructions are executed in **kernel/supervisor** mode.
- (iii) The time it takes disk head to reach the required cylinder during a disk access is termed as **seek time**.
- (iv) Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch**.

Section B

2. (a) In a demand paging system with the page table stored in memory, if the memory reference time is 200 nanoseconds and the page fault service time is 5 milliseconds, what should be the maximum page fault rate to achieve an effective access time of 350 nanoseconds?

- Given:
  - Memory reference time (m) = 200 ns
  - Page fault service time (pfs) = 5 ms = $5 \\times 10^6$ ns (since 1 ms = 10^6 ns)
  - Desired effective access time (EAT) = 350 ns
  - Let P be the page fault rate.

- The formula for Effective Access Time (EAT) in a demand paging system where the page table is in memory is: $EAT = (1 - P) \\times (m + m) + P \\times pfs$ (Here, $m + m$ indicates two memory accesses: one for the page table entry, one for the actual data. If TLB is used, it would be different, but the problem states "page table stored in memory" implying a direct memory access for page table.)
- Let's assume the question implicitly implies that the total memory access for a hit (including page table lookup) is just 200ns, and if there is a page fault, it's 200ns + 5ms. If the page table lookup itself costs 200ns and data access costs 200ns, then a page hit costs 400ns. Let's clarify the "memory reference time". If it means the time to access data *after* address translation: $EAT = (1 - P) \\times (Memory Reference Time + Page Table Access Time) + P \\times Page Fault Service Time$ Assuming "memory reference time" (200 ns) includes the page table lookup if it's a hit. This is a common interpretation to avoid ambiguity. If it means *only* the data access, then the page table access would be separate.

  Let's consider two common interpretations: **Interpretation 1: Memory reference time (200 ns) is for data access. Page table is also in memory, so it's an additional memory access.** $EAT = (1 - P) \\times (200 \\text{ ns (for page table)} + 200 \\text{ ns (for data)}) + P \\times pfs$ $EAT = (1 - P) \\times 400 \\text{ ns} + P \\times 5 \\times 10^6 \\text{ ns}$ $350 = (1 - P) \\times 400 + P \\times 5 \\times 10^6$ $350 = 400 - 400P + 5 \\times 10^6 P$ $350 - 400 = (5 \\times 10^6 - 400)P$ $-50 = (4999600)P$ $P = \\frac{-50}{4999600}$ (This gives a negative page fault rate, which is impossible. This implies my interpretation of 'memory reference time' is likely incorrect for the intended question.)

  Interpretation 2: Memory reference time (200 ns) is the time for a successful page access (including page table lookup).

This means $m\_{hit} = 200 \\text{ ns}$. $EAT = (1 - P) \\times m\_{hit} + P \\times pfs$ $350 = (1 - P) \\times 200 + P \\times (5 \\times 10^6)$ $350 = 200 - 200P + 5000000P$ $150 = 4999800P$ $P = \\frac{150}{4999800}$ $P \\approx 0.00003000$ To express as a percentage: $P \\approx 0.003%$

- **Final Answer (based on Interpretation 2, which is more common for this type of problem):** The maximum page fault rate should be: $P = \\frac{150}{4999800} \\approx 0.00003$ or 0.003

3. (b) Explain diagrammatically many-to-many multithreaded model giving one advantage and disadvantage.
    - **Many-to-Many Multithreaded Model:**
        - In this model, many user-level threads are multiplexed to a smaller or equal number of kernel threads.
        - This model allows the operating system to create a sufficient number of kernel threads for the application to achieve concurrency on a multiprocessor.
        - It also allows for multiple user threads to run in parallel on multiple processors. If a user thread performs a blocking system call, the kernel can schedule another user thread on the same kernel thread.
    - **Diagram (Conceptual):**
        - Imagine a layer of "User Threads" on top.
        - Below that, a layer of "Kernel Threads".
        - Arrows go from "many" user threads pointing to "fewer or equal" kernel threads, showing a many-to-one mapping initially.

- Then, arrows from kernel threads point to "CPUs/Processors" at the bottom, indicating scheduling.

User Threads (e.g., U1, U2, U3, U4) Arrows down to Kernel Threads (e.g., K1, K2) Arrows down to CPUs (e.g., CPU1, CPU2)

(Since diagrams are disallowed, this text description serves as the explanation.)

- **Advantage:**
  - It provides the flexibility of both the many-to-one and one-to-one models. The kernel can create more kernel threads as needed, allowing true parallelism on multiprocessor systems. It overcomes the blocking system call issue of the many-to-one model, as one user thread blocking doesn't block the entire process.
- **Disadvantage:**
  - It is more complex to implement than the many-to-one or one-to-one models, as it requires sophisticated synchronization and scheduling between user-level and kernel-level threads.

4. (c) Consider the following set of processes, with the length of CPU burst time given in milliseconds as below: Table: Process Information Process | Arrival Time | Burst Time | Priority P1 | 0 | 6 | 3 P2 | 2 | 3 | 1 (Highest) P3 | 4 | 2 | 2 P4 | 8 | 9 | 4 (Lowest)

   (i) Draw the Gantt Charts for priority scheduling (Pre-emptive), and Round Robin (Quantum = 3 ms) scheduling algorithms

   - **Priority Scheduling (Pre-emptive) Gantt Chart:** (Lower priority number means higher priority)

Time	0	2	4	6	9	18
CPU	P1	P2	P2	P3	P1	P4

- 0-2: P1 runs (Priority 3).
- 2-4: P2 arrives (Priority 1). P2 pre-empts P1. P2 runs for 2ms. P1 remaining burst = 4ms.
- 4-6: P3 arrives (Priority 2). P2 (Priority 1) is still highest. P2 runs for 1ms (total 3ms, completes).
- 6-9: P2 completes. P3 (Priority 2) is higher than P1 (Priority 3). P3 runs for 2ms (completes).
- 9-13: P3 completes. P1 (Priority 3) is higher than P4 (Priority 4). P1 runs for its remaining 4ms (completes).
- 13-22: P1 completes. P4 (Priority 4) runs for its 9ms (completes).

(The Gantt chart will look like a horizontal timeline with segments for each process.)

- **Round Robin (Quantum = 3 ms) Gantt Chart:**

Time	0	3	6	9	12	15	18	21	22
CPU	P1	P2	P3	P1	P4	P2	P1	P4	P4

  - **Ready Queue at various times:**
    - T=0: Q={P1}
    - 0-3: P1 runs (burst 6 -> 3).
    - T=2: P2 arrives.
    - T=3: P1 quantum expires. Q={P2, P1}. P2 starts.
    - 3-6: P2 runs (burst 3 -> 0). P2 completes.

- T=4: P3 arrives.
- T=6: P2 completes. Q={P1, P3}. P3 starts.
- 6-8: P3 runs (burst 2 -> 0). P3 completes.
- T=8: P4 arrives.
- T=9: P3 completes (burst 2 -> 0). P3 quantum expires. Q={P1, P4}. P1 starts.
- 9-12: P1 runs (burst 3 -> 0). P1 completes.
- T=12: P1 completes. Q={P4}. P4 starts.
- 12-15: P4 runs (burst 9 -> 6).
- T=15: P4 quantum expires. Q={P4}. P4 starts.
- 15-18: P4 runs (burst 6 -> 3).
- T=18: P4 quantum expires. Q={P4}. P4 starts.
- 18-21: P4 runs (burst 3 -> 0). P4 completes.
- T=21: All processes complete.

(The Gantt chart will look like a horizontal timeline with segments for each process.)

(ii) Calculate waiting time for processes P1 and P3 in case of priority scheduling (Pre-emptive)

- **Priority Scheduling (Pre-emptive) Waiting Times:**
  - P1:
    - Waited from 2 to 9 (P2 and P3 ran).
    - Waiting Time for P1 = (9 - 2) = 7 ms
  - P3:

- Waited from 4 to 6 (P2 ran).
- Waiting Time for P3 = (6 - 4) = 2 ms

(iii) Calculate turnaround time for processes P2 and P4 in case of Round Robin (Quantum = 3 ms) scheduling algorithm.

- **Round Robin (Quantum = 3 ms) Turnaround Times:**
  - P2:
    - Completion Time = 6 ms
    - Arrival Time = 2 ms
    - Turnaround Time for P2 = Completion Time - Arrival Time = 6 - 2 = 4 ms
  - P4:
    - Completion Time = 21 ms
    - Arrival Time = 8 ms
    - Turnaround Time for P4 = Completion Time - Arrival Time = 21 - 8 = 13 ms

5. (a) Consider the program given below. Assume that the actual PIDs of the parent and child processes are 1500 and 1567 respectively. Identify the output of the lines A, B, C, and D.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <iostream> // Added for cout
using namespace std; // Added for cout

int main()
{
   pid_t pid;
```

```
    pid=fork();
    if (pid==0)
    {
       cout<<pid;          //Line A
       cout<<getppid();     //Line B
    }
    else if (pid>0)
    {
       cout<<pid;          //Line C
       cout<<getpid();      //Line D
    }
    return 0; // Added return statement
}
```

- **Explanation:**
  - When *fork()* is called, it creates a new child process.
  - In the **child process**, *fork()* returns *0*.
  - In the **parent process**, *fork()* returns the PID of the child process.
  - *getpid()* returns the PID of the calling process.
  - *getppid()* returns the PID of the parent of the calling process.
  - Assumed Parent PID = 1500, Child PID = 1567.
- **Output:**
  - **Line A (Child Process):** *cout<<pid;*
    - Output: 0 (The value of *pid* in the child process is 0).
  - **Line B (Child Process):** *cout<<getppid();*
    - Output: 1500 (The parent's PID for the child is 1500).

- **Line C (Parent Process):** *cout<<pid;*
    - Output: 1567 (The value of *pid* in the parent process is the child's PID).
- **Line D (Parent Process):** *cout<<getpid();*
    - Output: 1500 (The parent's own PID is 1500).
- The exact interleaving of the output from parent and child processes is not guaranteed due to concurrent execution. However, the individual values printed will be as specified. For example, a possible output could be: *0150015671500*.

6. (b) Consider the following processes P1 and P2. P1: counter ++ P2: counter -- Assume that the initial value of counter is 10. Show how P1 and P2 can exhibit race condition while executing concurrently. State three requirements that a solution to critical-section problem must satisfy.

- **Race Condition Example:**
    - *counter*++ typically translates to three machine instructions:
        1. *R1* = *counter* (Load counter into register R1)
        2. *R1* = *R1* + *1* (Increment R1)
        3. *counter* = *R1* (Store R1 back to counter)
    - *counter*-- typically translates to three machine instructions:
        4. *R2* = *counter* (Load counter into register R2)
        5. *R2* = *R2* - *1* (Decrement R2)
        6. *counter* = *R2* (Store R2 back to counter)

- Initial *counter* = *10*.
- **Scenario for Race Condition:**
    7. P1 executes *R1* = *counter*. (R1 = 10)
    8. P1 executes *R1* = *R1* + *1*. (R1 = 11)
    9. Context switch occurs. P2 starts executing.
    10. P2 executes *R2* = *counter*. (R2 = 10, *stale value*)
    11. P2 executes *R2* = *R2* - *1*. (R2 = 9)
    12. P2 executes *counter* = *R2*. (counter = 9)
    13. Context switch occurs. P1 resumes.
    14. P1 executes *counter* = *R1*. (counter = 11)
- **Result:** The final value of *counter* is 11, instead of the expected 10 (10 + 1 - 1 = 10). This is a race condition because the outcome depends on the specific order of execution of instructions from P1 and P2. Other interleavings could result in *counter* being 9.

- **Three Requirements for a Solution to Critical-Section Problem:**
    ii. **Mutual Exclusion:** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
    iii. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

iv. **Bounded Waiting:** There must be a limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

7. (c) For the given sets P (processes), R (resources) and E (edges) P= {P1, P2, P3, P4} R={R1, R2} E= {P1->R1, R1->P2, R1->P3, P3->R2, R2->P1, R2->P4} There are two instances each of resources R1 and R2.

   (i) Draw Resource allocation graph. (Please note: Since diagrams are not allowed, I will describe the graph verbally.)

   ** **Nodes:***
   ** Processes: P1, P2, P3, P4 (represented by circles)*
   ** Resources: R1, R2 (represented by squares, with two dots inside each square to indicate two instances)*
   ** **Edges:***
   ** `P1 -> R1`: P1 requests an instance of R1 (Request Edge: P1 -> R1 square)*
   ** `R1 -> P2`: R1 is allocated to P2 (Assignment Edge: R1 square -> P2)*
   ** `R1 -> P3`: R1 is allocated to P3 (Assignment Edge: R1 square -> P3)*
   ** `P3 -> R2`: P3 requests an instance of R2 (Request Edge: P3 -> R2 square)*
   ** `R2 -> P1`: R2 is allocated to P1 (Assignment Edge: R2 square -> P1)*
   ** `R2 -> P4`: R2 is allocated to P4 (Assignment Edge: R2 square -> P4)*

   ** **Visual Representation (mental image):***
   ** P1 is a circle, R1 is a square with two dots, R2 is a square with two dots.*
   ** From P1, an arrow points to R1's square.*
   ** From R1's square, an arrow points to P2's circle (one dot from R1*

connects).

** From R1's square, an arrow points to P3's circle (the other dot from R1 connects).*

** From P3's circle, an arrow points to R2's square.*

** From R2's square, an arrow points to P1's circle (one dot from R2 connects).*

** From R2's square, an arrow points to P4's circle (the other dot from R2 connects).*

(ii) Is the given system in deadlock state? Justify your answer.

- **Analyze for Deadlock:**
  - **Resource R1:** Has two instances. Both instances are held by P2 and P3 (R1->P2, R1->P3). No free instances of R1.
  - **Resource R2:** Has two instances. Both instances are held by P1 and P4 (R2->P1, R2->P4). No free instances of R2.
  - **Process P1:** Holds R2. Requests R1 (P1->R1). P1 is blocked waiting for R1.
  - **Process P2:** Holds R1. Requests no resources. P2 is currently holding a resource and not waiting. (Can complete if it only holds R1 and does not request anything else).
  - **Process P3:** Holds R1. Requests R2 (P3->R2). P3 is blocked waiting for R2.
  - **Process P4:** Holds R2. Requests no resources. P4 is currently holding a resource and not waiting. (Can complete if it only holds R2 and does not request anything else).
  - **Check for cycles and unfulfilled requests:**

- P1 is waiting for R1. R1 is held by P2 and P3.
- P3 is waiting for R2. R2 is held by P1 and P4.

- If P2 can finish and release R1, then P1 or P3 could potentially get R1.
- If P4 can finish and release R2, then P3 could potentially get R2.
- Let's see if any process can complete:
  - P2 is holding R1 and not requesting anything. P2 can complete and release R1.
  - P4 is holding R2 and not requesting anything. P4 can complete and release R2.
- **Safety sequence (if exists):**

  1. P2 completes, releases R1. Now, R1 has one free instance.
  2. P4 completes, releases R2. Now, R2 has one free instance.
  3. Now, P1 is requesting R1, and there is an instance of R1 available. P1 can get R1.
  4. P1 completes, releases R1 and R2. Now, R1 has two free instances and R2 has two free instances.
  5. P3 is requesting R2, and there is an instance of R2 available. P3 can get R2.
  6. P3 completes, releases R1 and R2.
- **Conclusion:**
  - No, the given system is **not** in a deadlock state.

- **Justification:** While there are resource dependencies and potential cycles, all processes are not mutually blocked. Specifically, processes P2 and P4 are holding resources but not requesting any others. They can complete their execution, release their held resources, and allow the other blocked processes (P1 and P3) to proceed. This indicates that a safe sequence exists for the system to eventually complete all processes.

8. (a) Consider a system with logical address space comprising of 256 pages having 1024 bytes each, mapped to physical address space of 128 frames.

   (i) How many bits are there in the logical address?

   - Number of pages = 256
   - Page size = 1024 bytes
   - Logical address space size = Number of pages $\\times$ Page size = $256 \\times 1024$ bytes = $2^8 \\times 2^{10}$ bytes = 2^{18} bytes
   - The number of bits in the logical address is $\\log\_2(2^{18}) = 18$ bits.

   (ii) How many bits are there in the physical address?

   - Number of frames = 128
   - Frame size = Page size = 1024 bytes (in a paging system, page size = frame size)
   - Physical address space size = Number of frames $\\times$ Frame size = $128 \\times 1024$ bytes = $2^7 \\times 2^{10}$ bytes = 2^{17} bytes
   - The number of bits in the physical address is $\\log\_2(2^{17}) = 17$ bits.

9. (b) Name any two operations which can be performed on a directory. List any one disadvantage of Two-level directory structure. How can this disadvantage be overcome using Tree Structured directory structure?

   - **Two Operations on a Directory:**

     v. **Create File:** Add a new file entry to the directory.

     vi. **Delete File:** Remove an existing file entry from the directory. (Other operations: Search for a file, Rename a file, List contents of a directory, Create directory, Delete directory).

   - **Disadvantage of Two-level Directory Structure:**

     - **Lack of Structure/Grouping:** Users cannot create subdirectories or group files in a logical hierarchical manner. This makes it difficult to organize files, especially for users with many files or in complex projects. It also makes sharing and protection more cumbersome.

   - **How Tree Structured Directory Overcomes this Disadvantage:**

     - A Tree Structured directory allows users to create subdirectories within their own directories (and within other subdirectories). This forms a hierarchical structure where files can be organized into a logical tree. This provides:

       - **Better Organization:** Users can create a logical grouping of related files and subdirectories.
       - **Easier Sharing and Protection:** Permissions can be applied to directories, affecting all files and subdirectories within them, simplifying sharing and access control.

- **Avoids Name Collisions:** Each user's directory provides a separate namespace, reducing the chance of name collisions for common file names (e.g., "program.c").

10. (c) Consider the following page reference string: 9, 5, 3, 6, 5, 8, 2, 1, 9, 9, 0, 7 In case of demand paging, assuming four frames, how many page faults would occur for following page replacement algorithms:

    - (i) **FIFO (First-In, First-Out) replacement:**

        - Frames: 4
        - Reference String: 9, 5, 3, 6, 5, 8, 2, 1, 9, 9, 0, 7

Ref	F1	F2	F3	F4	Fault?	Notes
9	9				Y	
5	9	5			Y	
3	9	5	3		Y	
6	9	5	3	6	Y	
5	9	5	3	6	N	5 is in
8	8	5	3	6	Y	9 replaced by 8
2	8	2	3	6	Y	5 replaced by 2
1	8	2	1	6	Y	3 replaced by 1
9	8	2	1	9	Y	6 replaced by 9
9	8	2	1	9	N	9 is in
0	0	2	1	9	Y	8 replaced by 0
7	0	7	1	9	Y	2 replaced by 7

- Total Page Faults (FIFO): 10

- (ii) **Optimal replacement:**
  - Frames: 4
  - Reference String: 9, 5, 3, 6, 5, 8, 2, 1, 9, 9, 0, 7

Ref	F1	F2	F3	F4	Fault?	Notes (Future References)
9	9				Y	(5,3,6,5,8,2,1,9,9,0,7)
5	9	5			Y	(3,6,5,8,2,1,9,9,0,7)
3	9	5	3		Y	(6,5,8,2,1,9,9,0,7)
6	9	5	3	6	Y	(5,8,2,1,9,9,0,7)
5	9	5	3	6	N	5 is in
8	9	5	8	6	Y	Replace 3 (furthest future: 3 is not referenced)
2	9	5	8	2	Y	Replace 6 (furthest future: 6 is not referenced)
1	9	1	8	2	Y	Replace 5 (furthest future: 5 is only referenced next at index 9 (value 9, then 9). Oh, 5 is referenced far away. Replace 5 (later at 9). No, 5 is not referenced again for a long time. The next 5 is actually the next 9. Let's trace it properly: Future of 9 (at 8), 5 (not again), 8 (not again), 2 (not again). The 5 is the first to be replaced. Oh wait, the reference string is *9, 5, 3, 6, 5, 8, 2, 1, 9, 9, 0, 7*. The next 5 is

Ref	F1	F2	F3	F4	Fault?	Notes (Future References)
						at index 4, which is current. Next 9 is at index 8.

Let's redo Optimal for 8, 2, 1, 9, 9, 0, 7 from 9,5,3,6,5. Current frames: [9, 5, 3, 6] (after first 4 faults) Next ref: 5 (Hit) Frames: [9, 5, 3, 6]

Next ref: 8 (Fault) Frames: [9, 5, 3, 6]. Check future uses: 9: used at ref index 8 (next 9). 5: used at ref index 4 (current). Not used later. 3: not used later. 6: not used later. Replace 3 or 6. Let's replace 3. Frames: [9, 5, 8, 6]

Next ref: 2 (Fault) Frames: [9, 5, 8, 6]. Check future uses: 9: used at ref index 8 (next 9). 5: not used later. 8: not used later. 6: not used later. Replace 5 (or 8 or 6, choose any of them as they are not used). Let's replace 5. Frames: [9, 2, 8, 6]

Next ref: 1 (Fault) Frames: [9, 2, 8, 6]. Check future uses: 9: used at ref index 8 (next 9). 2: not used later. 8: not used later. 6: not used later. Replace 2 (or 8 or 6, choose any). Let's replace 2. Frames: [9, 1, 8, 6]

Next ref: 9 (Hit) Frames: [9, 1, 8, 6]

Next ref: 9 (Hit) Frames: [9, 1, 8, 6]

Next ref: 0 (Fault) Frames: [9, 1, 8, 6]. Check future uses: 9: not used later. 1: not used later. 8: not used later. 6: not used later. Replace any, say 9. Frames: [0, 1, 8, 6]

Next ref: 7 (Fault) Frames: [0, 1, 8, 6]. Check future uses: 0: not used later. 1: not used later. 8: not used later. 6: not used later. Replace any, say 0. Frames: [7, 1, 8, 6]

- Count of faults: 9 (Y), 5 (Y), 3 (Y), 6 (Y) = 4 faults 5 (N) = 0 faults 8 (Y) = 1 fault (total 5) 2 (Y) = 1 fault (total 6) 1 (Y)

= 1 fault (total 7) 9 (Y) = 1 fault (total 8) - No, wait. 9 is in the frame already. Let's re-do Optimal carefully:

Ref	F1	F2	F3	F4	Fault?	Page to Replace (furthest use)
9	9				Y	
5	9	5			Y	
3	9	5	3		Y	
6	9	5	3	6	Y	
5	9	5	3	6	N	
8	9	5	8	6	Y	Replace 3 (3 is not in future string)
2	9	5	8	2	Y	Replace 6 (6 is not in future string)
1	9	1	8	2	Y	Replace 5 (5 is not in future string)
9	9	1	8	2	N	
9	9	1	8	2	N	
0	0	1	8	2	Y	Replace 9 (9 is not in future string)
7	0	1	7	2	Y	Replace 8 (8 is not in future string)

- Total Page Faults (Optimal): 9

(This seems consistent with typical optimal performance. My previous trace errors show how tricky optimal can be.)

The program shown below uses the Pthreads API. What would be the output from the program at LINE X and LINE Y respectively? Justify your answer.

```
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h> // Corrected for pid_t, but not directly used in this snippet
#include <iostream>
using namespace std;

int value = 10; // Global variable

void *printing (void *param);

int main(int argc, char *argv[])
{
   pthread_t tid;
   pthread_attr_t attr;

   pthread_attr_init(&attr);

   pthread_create(&tid, &attr, printing, NULL);

   pthread_join(tid, NULL);

   cout<<value;       // LINE X
   return 0; // Added explicit return
}

void printing (void *param)
{
   int value = 20; // Local variable to this thread function
   value +=10;
   cout<< value;      // LINE Y
}
```

- **Output at LINE Y:** 30

- **Justification:** Inside the *printing* function, *int value = 20;* declares a *local* variable named *value*. This local variable is distinct from the global *value*. The line *value += 10;* increments this local *value* to 30. Then, *cout<<value;* prints this local *value*.

- **Output at LINE X:** 10

  - **Justification:** The *main* function (the main thread) declares a global variable *int value = 10;*. The *pthread_join(tid, NULL);* ensures that the main thread waits for the *printing* thread to complete. After *printing* completes, the main thread proceeds to *cout<<value;*. This *value* refers to the *global* variable, which was never modified by the *printing* function (because *printing* worked on its own local *value*). Therefore, the global *value* remains 10.

11. (b) Define external fragmentation? Does paging suffer from external fragmentation? Justify your answer.

- **External Fragmentation:**

  - External fragmentation occurs in memory management when there is enough total free space to satisfy a request, but the available free space is not contiguous. Instead, it is scattered in small, non-adjacent blocks throughout the memory, making it unusable for a process that requires a large contiguous block.

  - It is a problem primarily associated with variable-partition memory allocation schemes (like dynamic partitioning) and segmentation.

- **Does paging suffer from external fragmentation?**

  - No, **paging does not suffer from external fragmentation**.

- **Justification:** Paging divides both the physical memory (into frames) and logical memory (into pages) into fixed-size blocks. Processes are allocated frames in physical memory, which do not need to be contiguous. A logical page can be placed into any available physical frame. Since any free frame can be used, there is no issue of scattered free space being unusable due to non-contiguity. The entire physical memory is effectively a pool of available frames. Paging can suffer from internal fragmentation (where a page might not be fully utilized by a process), but not external fragmentation.

12. (c) Consider a disk drive of 2000 cylinders, numbered from 0 to 1999. Drive is currently serving a request at cylinder 500, and the previous request was at cylinder 250. The queue of pending request is 150, 1500, 900, 1700, 800, 1900. Starting at current head position, show the head movement and calculate the total distance in cylinders that the disk arm moves to satisfy all pending requests for each of following disk scheduling algorithms:

- Current Head Position: 500
- Previous Request: 250 (Implies head moving upwards towards higher cylinder numbers)
- Queue: 150, 1500, 900, 1700, 800, 1900

(i) **Shortest Seek Time First (SSTF):** * SSTF chooses the request that is closest to the current head position.

1. ***Start:** 500*
2. *Requests remaining: {150, 1500, 900, 1700, 800, 1900}*
3. *Closest to 500:*

** |500 - 150| = 350*
** |500 - 1500| = 1000*
** |500 - 900| = 400*
** |500 - 1700| = 1200*

*   *|500 - 800| = 300 (Smallest)*
*   *|500 - 1900| = 1400*

4. *Move to **800** (Distance = 300)*
5. *Current: 800. Requests remaining: {150, 1500, 900, 1700, 1900}*
6. *Closest to 800:*
    *   *|800 - 150| = 650*
    *   *|800 - 1500| = 700*
    *   *|800 - 900| = 100 (Smallest)*
    *   *|800 - 1700| = 900*
    *   *|800 - 1900| = 1100*
7. *Move to **900** (Distance = 100)*
8. *Current: 900. Requests remaining: {150, 1500, 1700, 1900}*
9. *Closest to 900:*
    *   *|900 - 150| = 750*
    *   *|900 - 1500| = 600*
    *   *|900 - 1700| = 800*
    *   *|900 - 1900| = 1000*
10. *Move to **1500** (Distance = 600)*
11. *Current: 1500. Requests remaining: {150, 1700, 1900}*
12. *Closest to 1500:*
    *   *|1500 - 150| = 1350*
    *   *|1500 - 1700| = 200 (Smallest)*
    *   *|1500 - 1900| = 400*
13. *Move to **1700** (Distance = 200)*
14. *Current: 1700. Requests remaining: {150, 1900}*
15. *Closest to 1700:*
    *   *|1700 - 150| = 1550*
    *   *|1700 - 1900| = 200 (Smallest)*
16. *Move to **1900** (Distance = 200)*
17. *Current: 1900. Requests remaining: {150}*
18. *Move to **150** (Distance = 1750)*

*   *Total Distance (SSTF) = 300 + 100 + 600 + 200 + 200 + 1750 = **3150 cylinders***

** Head Movement Order: 500 -> 800 -> 900 -> 1500 -> 1700 -> 1900 -> 150*

(ii) **LOOK Scheduling:** * LOOK is a version of SCAN. It moves in one direction, servicing requests until it reaches the last request in that direction, then reverses direction. The previous request was 250, current is 500, so the head is currently moving towards higher cylinder numbers.

1. ***Start:** 500*
2. *Queue: {150, 1500, 900, 1700, 800, 1900}*
3. *Sort requests in order of direction:*
   ** Higher than 500: 800, 900, 1500, 1700, 1900*
   ** Lower than 500: 150*

4. *Move towards higher cylinders first:*
   ** 500 -> **800** (Distance = 300)*
   ** 800 -> **900** (Distance = 100)*
   ** 900 -> **1500** (Distance = 600)*
   ** 1500 -> **1700** (Distance = 200)*
   ** 1700 -> **1900** (Distance = 200)*

5. *Reverse direction (since 1900 is the highest request in the queue):*
   ** 1900 -> **150** (Distance = 1750)*

** Total Distance (LOOK) = 300 + 100 + 600 + 200 + 200 + 1750 = **3150 cylinders***
** Head Movement Order: 500 -> 800 -> 900 -> 1500 -> 1700 -> 1900 -> 150*

(Note: In this specific example with these numbers, SSTF and LOOK ended up with the same total distance. This is not always the case.)

13. (a) Consider the following segment table Table: Segment Table Segment | Base | Length 0 | 519 | 500 1 | 1800 | 95 2 | 170 | 300 3 | 1920 | 780 4 | 1860 | 50

What are the physical addresses for the following logical addresses? (Logical Address = (Segment Number, Offset))

- (i) **0,240**
  - Segment 0: Base = 519, Length = 500
  - Offset = 240
  - Check validity: Offset (240) < Length (500) -> Valid.
  - Physical Address = Base + Offset = 519 + 240 = **759**
- (ii) **2,320**
  - Segment 2: Base = 170, Length = 300
  - Offset = 320
  - Check validity: Offset (320) < Length (300) -> **Invalid (Trap: addressing error)**
  - (If it were valid, Physical Address = 170 + 320 = 490)
- (iii) **3,670**
  - Segment 3: Base = 1920, Length = 780
  - Offset = 670
  - Check validity: Offset (670) < Length (780) -> Valid.
  - Physical Address = Base + Offset = 1920 + 670 = **2590**
- (iv) **4,10**
  - Segment 4: Base = 1860, Length = 50
  - Offset = 10
  - Check validity: Offset (10) < Length (50) -> Valid.
  - Physical Address = Base + Offset = 1860 + 10 = **1870**

14. (b) For a paged system, Translation Lookaside Buffer (TLB) hit ratio is 80%. Let memory access time be 150 ns and TLB buffer access time be 10 ns. Calculate the following:

- (i) **Effective memory access time without TLB**
  - Without TLB, every memory access requires two memory lookups: one for the page table entry (in memory) and one for the actual data.
  - Effective Access Time = Memory access time for page table + Memory access time for data
  - Effective Access Time = 150 ns + 150 ns = **300 ns**
- (ii) **Effective memory access time with TLB**
  - Hit Ratio (H) = 80% = 0.80
  - Miss Ratio = 1 - H = 0.20
  - Memory access time (m) = 150 ns
  - TLB access time (t) = 10 ns
  - Formula: $EAT = H \\times (t + m) + (1 - H) \\times (t + m + m)$ (If TLB hit, TLB access + one memory access for data. If TLB miss, TLB access + one memory access for page table + one memory access for data).
  - $EAT = 0.80 \\times (10 \\text{ ns} + 150 \\text{ ns}) + 0.20 \\times (10 \\text{ ns} + 150 \\text{ ns} + 150 \\text{ ns})$
  - $EAT = 0.80 \\times (160 \\text{ ns}) + 0.20 \\times (310 \\text{ ns})$
  - $EAT = 128 \\text{ ns} + 62 \\text{ ns}$
  - $EAT = **190 ns**

15. (c) Consider the following: (i) TLB miss with no page fault (ii) TLB miss and page fault (iii) TLB hit For each of the above cases, state the situation in which they occur. Also show how the frame number is determined in each case.

- (i) **TLB miss with no page fault**
  - **Situation:** This occurs when the required page number is not found in the Translation Lookaside Buffer (TLB), but the corresponding page *is* present in main memory (i.e., its valid bit in the page table is set). This often happens when a page is accessed for the first time or after it has been evicted from the TLB.
  - **Frame Number Determination:**
    1. The CPU tries to find the page number in the TLB (TLB miss).
    2. The operating system's memory management unit (MMU) then accesses the **page table in main memory** using the page number from the logical address.
    3. It finds the page table entry, verifies the valid bit is set, and extracts the corresponding **frame number**.
    4. The frame number is then used with the offset to form the physical address.
    5. Optionally, the (page number, frame number) pair is then added to the TLB for faster future accesses.
- (ii) **TLB miss and page fault**
  - **Situation:** This occurs when the required page number is not found in the TLB, AND when the MMU then looks up the page table entry in main memory, it finds that the

page is *not* currently in main memory (i.e., its valid bit is not set, indicating it's on disk).

- **Frame Number Determination:**

6. The CPU tries to find the page number in the TLB (TLB miss).
7. The MMU accesses the page table in main memory.
8. It finds that the valid bit for the requested page is clear, triggering a **page fault**.
9. The operating system's page fault handler is invoked.
10. The OS finds a free frame in physical memory (or uses a page replacement algorithm to evict an existing page).
11. The required page is loaded from disk into the allocated free frame.
12. The page table entry for that page is updated with the new **frame number** and its valid bit is set.
13. The (page number, frame number) pair is then added to the TLB.
14. The instruction that caused the page fault is restarted.

- (iii) **TLB hit**

  - **Situation:** This occurs when the required page number is found directly in the Translation Lookaside Buffer (TLB), meaning the corresponding page is actively being used and its mapping is in the fast cache.
  - **Frame Number Determination:**

15. The CPU presents the logical address to the MMU.
16. The MMU extracts the page number and simultaneously searches for it in the TLB.
17. If the page number is found in the TLB (TLB hit), the corresponding **frame number** is retrieved directly from the TLB.
18. This frame number is then combined with the offset from the logical address to directly form the physical address, bypassing the slower main memory page table lookup.

16. (a) Assuming a 2-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

- Page size = 2 KB = $2 \\times 1024$ bytes = 2048 bytes
- Since page size is 2^{11} bytes ($2048 = 2^{11}$), the offset will be the lower 11 bits of the logical address.
- Page Number = $\\lfloor \\text{Logical Address} / \\text{Page Size} \\rfloor$
- Offset = $\\text{Logical Address} \\pmod{\\text{Page Size}}$
- (i) **3085**
  - Page Number = $\\lfloor 3085 / 2048 \\rfloor = \\lfloor 1.506... \\rfloor = 1$
  - Offset = $3085 \\pmod{2048} = 1037$
  - Page Number: 1, Offset: 1037
- (ii) **42095**

- Page Number = $\\lfloor 42095 / 2048 \\rfloor = \\lfloor 20.554... \\rfloor = 20$
- Offset = $42095 \\pmod{2048} = 1119$
- Page Number: 20, Offset: 1119

17. (b) Describe Belady's anomaly in context of Demand paging. Which of the following algorithms suffer from Belady's anomaly:

- **Belady's Anomaly:**
  - Belady's Anomaly is a phenomenon observed in some page replacement algorithms, particularly FIFO (First-In, First-Out). It states that increasing the number of available memory frames can sometimes lead to an *increase* in the number of page faults for a given page reference string.
  - This counter-intuitive behavior means that providing more resources (more frames) to a system does not always guarantee better performance (fewer page faults).
- **Which algorithms suffer from Belady's anomaly:**
  - (i) **FIFO (First-In, First-Out): Yes**, FIFO is known to suffer from Belady's anomaly. This is the classic example where it is observed.
  - (ii) **Least Recently Used (LRU) algorithm: No**, LRU does not suffer from Belady's anomaly. LRU is a stack algorithm, and it has been mathematically proven that stack algorithms do not exhibit Belady's anomaly. For LRU, increasing the number of frames will always result in either fewer or the same number of page faults.

18. (c) Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order). How would the following

scheduling algorithms place processes of size 115 KB, 500 KB and 375 KB (in order)?

- Available Partitions: 300KB, 600KB, 350KB, 200KB, 750KB, 125KB
- Processes to place (in order): P1 (115 KB), P2 (500 KB), P3 (375 KB)
- (i) **First-fit algorithm:**
  - **Rule:** Allocate the first partition that is large enough.

vii. **Place P1 (115 KB):**

- Checks 300 KB (fits).
- P1 goes into 300 KB partition.
- Remaining partitions: 185KB (from 300KB), 600KB, 350KB, 200KB, 750KB, 125KB
- (For First-Fit, the 300KB slot becomes 185KB, it remains at its original position in the list, unless the list is re-ordered, which is not typical for First-Fit's operation)
- Let's denote allocated partitions explicitly:
  - Partitions: 300 (used by P1, 185 left), 600, 350, 200, 750, 125

viii. **Place P2 (500 KB):**

- Checks 185 KB (does not fit).
- Checks 600 KB (fits).
- P2 goes into 600 KB partition.
- Partitions: 300 (used by P1, 185 left), 600 (used by P2, 100 left), 350, 200, 750, 125

ix. **Place P3 (375 KB):**

- Checks 185 KB (does not fit).
- Checks 100 KB (from 600KB, does not fit).
- Checks 350 KB (does not fit).
- Checks 200 KB (does not fit).
- Checks 750 KB (fits).
- P3 goes into 750 KB partition.
- Partitions: 300 (used by P1, 185 left), 600 (used by P2, 100 left), 350, 200, 750 (used by P3, 375 left), 125

▪ **Placement for First-Fit:**

- P1 (115 KB) $\\to$ 300 KB partition
- P2 (500 KB) $\\to$ 600 KB partition
- P3 (375 KB) $\\to$ 750 KB partition

○ (ii) **Best-fit algorithm:**

▪ **Rule:** Allocate the smallest partition that is large enough. Requires searching the entire list of available partitions.

▪ Available Partitions (sorted for easier check, but algorithm searches unsorted list): 125KB, 200KB, 300KB, 350KB, 600KB, 750KB

x. **Place P1 (115 KB):**

- Suitable partitions: 200KB, 300KB, 350KB, 600KB, 750KB.
- Smallest suitable: 200 KB.

- P1 goes into 200 KB partition.
- Partitions: 300, 600, 350, 200 (used by P1, 85 left), 750, 125

xi. **Place P2 (500 KB):**

- Suitable partitions: 600KB, 750KB.
- Smallest suitable: 600 KB.
- P2 goes into 600 KB partition.
- Partitions: 300, 600 (used by P2, 100 left), 350, 200 (used by P1, 85 left), 750, 125

xii. **Place P3 (375 KB):**

- Suitable partitions: 750KB. (350KB is too small)
- Smallest suitable: 750 KB.
- P3 goes into 750 KB partition.
- Partitions: 300, 600 (used by P2, 100 left), 350, 200 (used by P1, 85 left), 750 (used by P3, 375 left), 125

- **Placement for Best-Fit:**
  - P1 (115 KB) $\\to$ 200 KB partition
  - P2 (500 KB) $\\to$ 600 KB partition
  - P3 (375 KB) $\\to$ 750 KB partition