1) a) i) F
In round robin scheduling, all processes have a common and fixed priority value, but they are only given a fixed duration (quantum) to run every time they are scheduled to the CPU, and are interrupted by the scheduler once the quantum is up. This is unlike priority-based scheduling, where every process is allowed to run to completion once they are set to run (in absence of an interrupt)

ii) F
Even though there is a cycle, there are two resources of R1, allowing P1 to continue execution

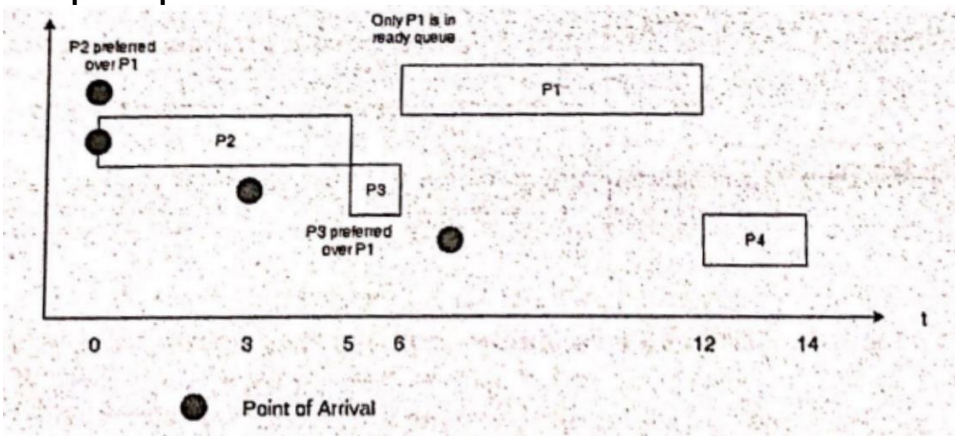
iii) F
Abort() is called by parent process to terminate a child process

iv) F
DMA still uses interrupt, just that they copy a large block of data per interrupt, while interrupt driven I/O has much more interrupts where it copies byte per interrupt

v) F
Multiprogramming is still possible within each core (e.g. modern-day CPUs)

vi) T
I/O protection all I/O instructions go through kernel to ensure correctness and legality

b) i) **Non-preemptive SJF** results in the execution order: P2 P3 P1 P4



ii) To ensure the same schedule,
Q = 6 CPU Burst for round robin
To ensure that every process is ran to completion the first time it is executed.

Resulting Priorities:
P1: 3rd
P2: 1st
P3: 2nd
P4: 4th

Note: When a process is completed before its allocated time is up/an event interrupt occurs (e.g. I/O), the scheduler moves on to the next job scheduled.

c) No, it does not satisfy mutual exclusion.
Initially, flag = false, turn = 0

```
Process P1
while (1) {
        turn = 0;
        while (flag and (turn == 0);
        critical-section
        flag = false;
        remainder-section
}
```

Start off with Process P1 first.
While flag = true and turn = 0, P1 will be stuck in 'busy wait'.
However, as flag = false, it can enter the **critical section**.
Suppose **context switch** occurs here. Now P0 runs.

```
Process P0
while (1) {
        flag = true;
        while (turn == 1);
        critical-section
        turn = 1;
        remainder-section
}
```

While turn = 1, P0 will be stuck in busy wait.
But currently, turn = 0.
P0 can enter the **critical section.**
Thus, both processes are now in the critical section.
Hence the conditions are insufficient to ensure mutual exclusion.

2) a) i) T
Semaphores ensure mutual exclusion. Value = 1 means only one instance of that resource is available.

ii) F
Non-preemptive scheduling: scheduler not allowed to interrupt running processes when a new process is inserted into the queue.

iii) T

iv) T
Code is stored in the Text section

v) F
4 conditions must be satisfied for deadlock to occur: mutual exclusion, circular wait, hold and wait, no pre-emption.

vi) T
Message passing results in overhead.

b) i) A:3 B:6 C:3
Rule: Available >= Need

				Available		
				A	B	C
				0	2	0
Process	Allocation	Need				
P4	1,1,1	0,1,0	+	1	1	1
				1	3	1
P2	1,1,0	0,1,1	+	1	1	0
				2	4	1
P3	1,1,1	2,0,0	+	1	1	1
				3	5	2
P1	0,1,1	1,1,0	+	0	1	1
				3	6	3

ii) It is safe, there exists at least 1 safe sequence as shown above: P4 > P2 > P3 > P1

c)

Writer	Reader

while(1) {
wait(w_mutex);
w_count++;
signal(w_mutex);
 	
wait(res);
update D;
signal(res);
 	
wait(w_mutex)
w_count--;
signal(w_mutex);
 	
}
	while(1) {
	wait(w_mutex);
	if (w_count == 0) {
	wait(r_mutex);
	readcount++;
	if (readcount == 1) wait(res);
	signal (r_mutex);
	read D;
	wait(r_mutex);
	readcount--;
	if (readcount == 0) signal(res);
	signal(r_mutex);
	}
	signal(w_mutex);

We introduce a 'w_count' variable that represents the number of writers, and protect it using the binary semaphore 'w_mutex'. Reader has to check that 'w_count' = 0 before it can subscribe to read.

3) a) i) **T**
Compile time binding results in the same logical and physical address, which is why the program instruction address is absolute and not relocatable in memory.

ii) **F**
External fragmentation: due to poor allocation of programs in memory, there are unused memory blocks in between programs that when combined would have been able to accommodate more programs.
Solution: Compaction – to make these unused memory blocks contiguous.

Internal fragmentation: unused memory space as program size is smaller than allocated memory block size.
Solution: Segmentation, to ensure allocated space = program size.

iii) **F**
Read/Write from secondary storage (e.g. HDD) is the largest bottleneck due to slow read/write speed compared to RAM.

iv) **F**
Not always the case, example: Thrashing.

v) **T**
Fixed frame allocation policy can only use local replacement policy.

b) i) 32 frames = 2^5
Page size is 256 bytes = 2^8
$2^8 \times 2^5 = 2^{13}$

ii) Physical Address: | 5 bits | 8 bits |
Frame bits: 5, Address bits: 8

Logical Address: | 2 bits | 8 bits |
Page bits: 2, Address bits: 8

iii) Page 10_2 □ 2_{10}
Corresponds to 16_{10} in page table □ Frame number: $\underline{10000}_2$

Address: $\mathbf{01001010_2}$ (Keep the same)
Answer: Physical address $\mathbf{\underline{10000}\ 01001010_2}$

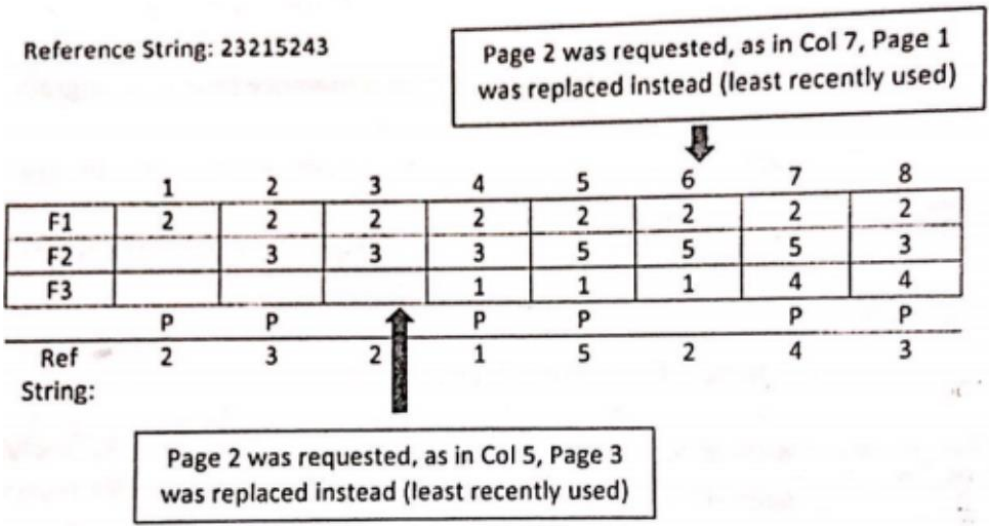
c) i) Yes. Page is found in both TLB and page table

ii) No. The TLB is a cache of the page table. If there is a TLB hit, typically the page is already in the page.

iii) Yes, TLB miss but page is still cached in memory and available via accessing the page table, resulting in no page fault.

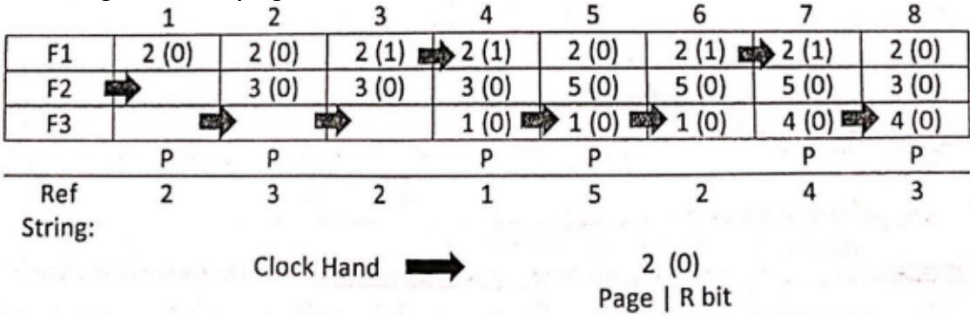
iv) Yes, TLB miss and page fault, if the page is absent from both the TLB and page table (e.g. at the start of the table, when the page has not been loaded from secondary storage yet, or page entry has been overwritten in both the TLB and page table).

d) i)



When there is a page fault, the page that appears in the next column is the page that was just brought in. When there isn't a page fault, it means the page requested is already in the page table. Check the next few columns to determine the LRU order.

ii) Clock algorithm: 6 page faults.



4) a) i) **T**
Symbolic link: A directory entry link, which contains the absolute/relative path name of the file.
Access the file via its absolute path for the symbolic link.

ii) **F**
Two groups of uses have read permission: Owner and Group.

iii) **F**
Linked file allocation allows file blocks to be distributed and allocated anywhere in memory, especially in unused pockets of memory and reducing external fragmentation.

iv) **F**
Contiguous allocation method would be the best, for fastest performance since the access is in a sequential manner.

b) Contiguous: 1R (Support random access)
Linked: 5R (Has to read from Block 6 to Block 10)
Indexed: 1R (Supports random access via the inode)

c) *In SCAN, the head moves from one end to the other end, reverses direction and repeats. In Circular-SCAN, the head moves from one end to the other end, and jumps black to the original position, and repeats.*

i) SCAN: |102 – 100| + |105 – 102| + |150 – 105| + |199 – 150| + |89 – 199| + |85 – 89| + |63 – 85| = 235
C-SCAN: |102 – 100| + |105 – 102| + |150 – 105| + |199 – 150| + |0 – 199| + |63 – 0| + |85 – 63| + |89 – 85| = 387

ii) For C-SCAN, max delay for any request is lesser as it skips servicing requests on the return trip as compared to SCAN. Thus, C-SCAN results in more uniform waiting time for requests.

--End of Answers--