Q1. Attempt answers in the printed order. Do not waste paper. [2 + 1 + 1 + 2 = 6 marks]
Note: Only the first 5 lines of your answers shall be graded.

a) Suppose two programs, A and B, must work together. A makes data, and B uses it. Both run at the same time. Explain a way for A to give data to B in the following situations: (No drawing. No code) [2]
   i. What if A and B are not running on the same computer?
      A sends data to B over a network using messages [1].
   ii. What if one is A, and many are B?
      A writes data to a shared memory and multiple instances of B read data from the same shared memory concurrently [1].

b) Suppose a process creates many threads. Now, answer the following:
   i. Why does a computation use the same code for all threads? No drawing. No code.
      To efficiently process large amounts of data using multithreading, where each thread executes the same code that acts on different parts concurrently **[0.5]**.
   ii. How does a thread keep its local data separate? Show C code. No text. No drawing. **[0.5]**

```
void* thread_func(void* arg) {
    static __thread int local = *(int*)arg;
    ...
    return NULL;
}
```

c) Which web server consumes less memory? The one that creates a new process for each incoming request or the one that creates threads instead. Give a technical reason for your decision.
   A web server using threads consumes less memory **[0.5]**. **Reason:** Threads share the same process memory space (e.g., code, heap), while a new process for each request allocates the memory in a different area of RAM, thus increasing overhead **[0.5].**

d) Suppose the serial fraction in a parallel code is 10%. Give a technical reason why Amdhal's Law speedup increases with the increase of the number of threads. Show calculations for 25 and 100 threads in support. Hint: Amdahl's Law equation speedup = 1/(S+((1-S)/N)). Calculations and Text only. [2]

Speedup = 1 / (S + (1 - S) / N), where S = serial fraction (0.1), N = number of threads **[0.5]**.

For 25 threads: **[0.5]**
Speedup = 1 / (0.1 + (1 - 0.1) / 25) = 1 / (0.1 + 0.9 / 25) = 1 / (0.1 + 0.036) = 1 / 0.136 ≈ 7.35

For 100 threads:**[0.5]**
Speedup = 1 / (0.1 + (1 - 0.1) / 100) = 1 / (0.1 + 0.9 / 100) = 1 / (0.1 + 0.009) = 1 / 0.109 ≈ 9.17

Reason: (1-S) is the parallel portion. As we increase number of threads (i.e. N >> 1). The value of (1-S) / N decreases resulting in small numerator thus increasing the speedup **[0.5].**

CLO # 2: Understand, design, and implement solutions employing concepts of Processes/Threads.

Q2. Write C snippets. [3 marks x 3 parts = 9 marks in total, estimated time: 10 minutes per question]
Note: Avoid missing computational thinking steps, syntax errors, invalid logic, and input-output statements unless asked. Messy, unreadable, and/or scattered code will get very low scores.

Students must write full logic of their coding solutions:
- No grading of code fragments without logical structure of the full solution.
- Partial weightage allowed, when well defined logical structure (as per slides/textbook) is present.
- No weightage should be given in Q2 part (c) for dry-run if no code is shown.
- 50% of less weightage in Q2 part (b) is shared memory code is not shown.
- 25% or less weightage in Q2 part (a) and part (b) if the threading function is missing or has major logic issues.

a) Sum an array of 100K integers using 10 threads (show Pthread code). Each thread stores its sum in an element of a global array **lsum**. The main() will compute the final sum in **tsum**. [1 + 1 + 0.5 + 0.5=3]

```
#include <stdio.h>
#include <pthread.h>
#define N 100000
#define NUM_THREADS 10

int arr[N], lsum[NUM_THREADS];
void* sum_thread(void* arg) { [1] if code in blue present
    int tid = *(int*)arg,
    start = tid * (N/NUM_THREADS), end = start + (N/NUM_THREADS);
    int local_sum = 0; [0.5] if code in purple present
    for (int i = start; i < end; i++) local_sum += arr[i];
    lsum[tid] = local_sum;
    return NULL; // replaced by pthread_exit(NULL)
}
int main() { [1] if code in blue present
    pthread_t threads[NUM_THREADS];
    int tids[T], tsum = 0;
    // assume arr contains data
    for (int i = 0; i < NUM_THREADS; i++ {
        tids[i] = i;
        pthread_create(&threads[i],NULL,sum_thread,&tids[i]);
    }
    for (int i = 0; i < ; i++) pthread_join(threads[i],NULL);

    // [0.5] if code in purple present
    for (int i = 0; i < NUM_THREADS; i++) tsum += lsum[i];
    // tsum contains global sum of arr
    return 0; // replaced by pthread_exit(NULL)
}
```

b) Assume a process PD has written data in a shared memory named SHEMO (do not show code). Show the code for a detached thread - that exists in another process JD, that reads all shared memory data, displays it on screen, and terminates. How does this thread ensure that it reads data after the PD process has written it? Give a theoretical explanation only. [2 + 1=3]

```
#include <stdio.h> // This is code run by Process JD
#include <pthread.h>
#include <sys/shm.h>

void* read_thread(void* arg) { // [0.5] if code in blue present
    char *shared_data = (char *)arg;
    // each thread read same shared memory data
    // like printf("Thread reads: %s\n", shared_data);
    pthread_exits(NULL);
}

int main() { // [0.5]
    const char *name = "SHEMO";
    int memory_size = 1024, int memory_perm = 0666;
    int sfd = shm_open(name, O_RDONLY, memory_perm); // Read only
    char *sdata = mmap(0,memory_size,PROT_READ,MAP_SHARED,sfd,0);
    // read data if needed in main thread e.g. printf("%s", (char *)sdata);
    pthread_t thread_id;
    pthread_attr_t thread_attr;
    pthread_attr_init(&thread_attr); [0.5]
    pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_DETACHED);
    pthread_create(&thread_id, &thread_attr, read_thread, sdata);
    pthread_attr_destroy(&thread_attr);
    Sleep (5); // wait for detached threads to read data [0.5]
    shm_unlink(name);
    pthread_exit(NULL); // Exit main thread as other threads run independently
}
```

Theoretical explanation [1]:
Because JD is an independently runnable process there are no guarantees that PD has written data before the thread reads it, as shared memory can be written to by both process at any time (asynchronously). Needs synchronization. Memory barriers can be used to ensure write operations are committed to SHEMO before the reader accesses. But JD can be scheduled before PD breaking the guarantee.

c) Suppose two processes (P0 and P1) start simultaneously and try incrementing a global variable to assign a unique number using Peterson's solution. Write C code for this scenario. Also dry-run your code by showing values in a dry-run table with fields: Step#, Process name, Actions, Turn, Flag[0], Flag[1], and Remarks. During the dry run, both processes will execute till they assign two unique values to the respective variables in their memories. [1.5 + 1.5 = 3]

```
#include <stdio.h> [1.5 for flawless code]

int turn, flag[2], global = 0;

void P0() {
    flag[0] = 1; turn = 1; //peterson's variables
    while (flag[1] && turn == 1); // busy wait
    global++; // critical section
    flag[0] = 0;
}

void P1() {
    flag[1] = 1; turn = 0; //peterson's variables
    while (flag[0] && turn == 0); // busy wait
    global++; // critical section
    flag[1] = 0;
}

int main() {
    flag[0] = flag[1] = 0; //peterson's variables init.
    P0(); // no need to write while (true) {}
    P1();
    return 0;
}
```

Dry-Run Table [1.5 for flawless Dry-Run table and Explanation]

Assumptions: P0 starts first, but P1 interleaves. Execution is simulated step-by-step until both assign unique values.

Step #	Process	Action	turn	flag[0]	flag[1]	Remarks
1	P0	flag[0] = 1	0	1	0	P0 expresses interest
2	P0	turn = 1	1	1	0	P0 yields to P1
3	P1	flag[1] = 1	1	1	1	P1 expresses interest
4	P1	turn = 0	0	1	1	P1 yields to P0
5	P0	while(flag[1] && turn == 1)	0	1	1	P0 checks: false (turn = 0)
6	P0	global++	0	1	1	global = 1, P0 assigns 1
7	P0	flag[0] = 0	0	0	1	P0 exits critical section
8	P1	while(flag[0] && turn == 0)	0	0	1	P1 checks: false (flag[0] = 0)
9	P1	global++	0	0	1	global = 2, P1 assigns 2
10	P1	flag[1] = 0	0	0	0	P1 exits critical section

Explanation

- Peterson's solution ensures mutual exclusion. P0 enters its critical section first (step 6) because P1 sets turn = 0, allowing P0 to proceed.
- P0 assigns global = 1, then exits. P1, waiting in its while loop, proceeds once flag[0] = 0, assigning global = 2.
- Final values: P0 gets 1, P1 gets 2, stored in their respective outputs.