# CS558-Computer-Systems-Lab-IITG

## Section A: Thread Synchronization and Concurrency (30 marks)

### Q1. (10 marks)

Consider the following code snippet:

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

using namespace std;

mutex mtx;
condition_variable cv;
bool ready = false;
int data = 0;

void producer() {
    {
        unique_lock<mutex> lock(mtx);
        data = 42;
        ready = true;
    }
    cv.notify_one();
}

void consumer() {
    unique_lock<mutex> lock(mtx);
    cv.wait(lock, [] { return ready; });
    cout << "Data: " << data << endl;
}

int main() {
    thread t1(consumer);
    thread t2(producer);
    t1.join();
    t2.join();
    return 0;
}
```

a) Explain the purpose of the `unique_lock` in this program. (2 marks)

**Answer:** The `unique_lock` provides RAII-style mutex locking that automatically acquires the mutex when created and releases it when destroyed. It's needed to protect the shared variables (`data` and `ready`) from concurrent access. Unlike `lock_guard`, `unique_lock` is required for condition variables since it allows temporarily releasing the lock during waiting.

b) What would happen if the `notify_one()` call was placed inside the lock scope in the producer function? (3 marks)

**Answer:** If `notify_one()` was placed inside the lock scope, the consumer would be notified while the producer still holds the lock. This isn't optimal because the consumer would wake up but then immediately block again trying to acquire the lock. Placing the notification outside the lock is more efficient since it allows the consumer to acquire the lock immediately after being notified.

c) Modify the program to use a second condition variable to signal back to the producer that the consumer has processed the data. (5 marks)

**Answer:**

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

using namespace std;

mutex mtx;
condition_variable cv_producer, cv_consumer;
bool ready = false;
bool processed = false;
int data = 0;

void producer() {
    {
        unique_lock<mutex> lock(mtx);
        data = 42;
        ready = true;
        cv_consumer.notify_one();

        // Wait for consumer to process
        cv_producer.wait(lock, [] { return processed; });
        cout << "Producer: Consumer has processed the data" << endl;
    }
}

void consumer() {
    unique_lock<mutex> lock(mtx);
    cv_consumer.wait(lock, [] { return ready; });
    cout << "Consumer: Data received = " << data << endl;

    // Signal back to producer
    processed = true;
    cv_producer.notify_one();
}

int main() {
    thread t1(consumer);
    thread t2(producer);
    t1.join();
    t2.join();
    return 0;
}
```

### Q2. (10 marks)

Referring to the bus simulation program (Q1.cpp in Assignment 2):

a) What is the role of the condition variables `cv_gate_open` and `cv_bus_depart`? Explain how they ensure correct synchronization. (3 marks)

**Answer:** The condition variables ensure proper synchronization between the bus and passenger threads:

- `cv_gate_open`: Controls when passengers can board the bus. Passengers wait on this variable until there's space on the bus.
- `cv_bus_depart`: Signals the bus to depart when full. The bus waits on this variable until the `bus_departed` flag is set.

Together, they implement a rendezvous pattern where passengers wait until they can board, and the bus waits until it's full before departing.

b) What would happen if we replaced `cv_bus_depart.notify_one()` with `cv_bus_depart.notify_all()`? (2 marks)

**Answer:** Using `notify_all()` instead of `notify_one()` would unnecessarily wake up all waiting threads that are waiting on the `cv_bus_depart` condition variable. Since only one bus thread exists in the current implementation, there's no need to notify multiple threads. It wouldn't cause incorrect behavior but would be less efficient.

c) Why does the passenger function explicitly call `lock.unlock()` before exiting, while the bus function does not? (2 marks)

**Answer:** The passenger function explicitly unlocks the mutex because it needs to release the lock before exiting, allowing other passengers to enter the critical section. The bus function doesn't explicitly unlock because it uses the RAII principle - the `unique_lock` automatically releases the mutex when it goes out of scope at the end of each loop iteration.

d) Modify the program to support multiple buses, each with capacity N. (3 marks)

**Answer:**

```
// Additional variables
const int NUM_BUSES = 3;
vector<bool> bus_departed(NUM_BUSES, false);
vector<int> boarded_passengers(NUM_BUSES, 0);
condition_variable cv_buses_depart[NUM_BUSES];

void bus(int id) {
    while (true) {
        unique_lock<mutex> lock(mtx);

        cv_buses_depart[id].wait(lock, [id] { return bus_departed[id]; });

        cout << "Bus " << id << " is departing with " << N << " 
passengers.\n";
        this_thread::sleep_for(chrono::seconds(2));

        boarded_passengers[id] = 0;
        bus_departed[id] = false;

        cout << "Bus " << id << " departed. Gate reopens for new 
passengers.\n";
        cv_gate_open.notify_all();
    }
}

void passenger(int id) {
    unique_lock<mutex> lock(mtx);

    waiting_passengers++;
    cout << "Passenger " << id << " is waiting at the gate.\n";

    // Find a bus with available space
    bool found_bus = false;
    int bus_id = 0;

    while (!found_bus) {
        for (int i = 0; i < NUM_BUSES; i++) {
            if (boarded_passengers[i] < N) {
                found_bus = true;
                bus_id = i;
                break;
            }
        }

        if (!found_bus) {
            cv_gate_open.wait(lock);
        }
    }

    cout << "Passenger " << id << " enters bus " << bus_id << ".\n";
    waiting_passengers--;
    boarded_passengers[bus_id]++;

    if (boarded_passengers[bus_id] == N) {
        bus_departed[bus_id] = true;
        cv_buses_depart[bus_id].notify_one();
    }

    lock.unlock();
}
```

## Q3. (10 marks)

Regarding the bidirectional bridge crossing simulation (Q2.cpp in Assignment 2):

a) Explain the difference between the "Strict Directional Priority" and "Fair Direction Switching" modes. Which one prevents starvation and how? (3 marks)

**Answer:** In "Strict Directional Priority" mode, once vehicles from one direction start crossing, they continue until no more vehicles are waiting in that direction. Only then does the bridge switch direction. This can lead to starvation if vehicles continuously arrive from one direction.

"Fair Direction Switching" mode uses the `k` parameter to limit consecutive crossings from one direction to at most `k` vehicles, then forces a direction switch. This prevents starvation by ensuring vehicles from both directions get an opportunity to cross, even under heavy traffic conditions in one direction.

b) What is the significance of the `consecutive_crossings` variable? (2 marks)

**Answer:** The `consecutive_crossings` variable counts how many vehicles have crossed the bridge consecutively in the current direction. When it reaches the threshold `k` in fair mode, the bridge switches direction even if more vehicles are waiting in the current direction. This is key to implementing the fairness policy.

c) Identify a potential deadlock scenario in the code and explain how it's prevented. (2 marks)

**Answer:** A potential deadlock could occur if:

1. The bridge is empty
2. Vehicles are waiting in both directions
3. No direction is given priority

The code prevents this by:

- Always having a current direction (`current_dir`) initialized to NORTH
- Having clear rules for switching direction when the bridge is empty
- Using the fairness parameter `k` to force direction changes
- Having separate condition variables for each direction

d) Modify the code to prioritize emergency vehicles that can cross the bridge regardless of the current direction. (3 marks)

**Answer:**

```
// Add emergency vehicle type
enum VehicleType { NORMAL, EMERGENCY };

// Add emergency vehicle parameters
bool emergency_vehicle_crossing = false;
bool emergency_vehicle_waiting = false;

void enter_bridge(Direction dir, VehicleType type = NORMAL) {
    unique_lock<mutex> lock(bridge_mutex);
    if (dir == NORTH) waiting_north++;
    else waiting_south++;

    if (type == EMERGENCY) {
        emergency_vehicle_waiting = true;
        // Emergency vehicles wait only if other emergency vehicles are
crossing
        while (vehicles_on_bridge > 0 && emergency_vehicle_crossing) {
            if (dir == NORTH)
                north_wait.wait(lock);
            else
                south_wait.wait(lock);
        }
        emergency_vehicle_waiting = false;
        emergency_vehicle_crossing = true;
    } else {
        // Normal vehicles follow regular rules plus wait for emergency
vehicles
        if (fair_mode) {
            while ((current_dir != dir && vehicles_on_bridge > 0) ||
                   (consecutive_crossings >= k) ||
emergency_vehicle_waiting) {
                if (consecutive_crossings >= k && vehicles_on_bridge == 0)
{
                    current_dir = (current_dir == NORTH) ? SOUTH : NORTH;
                    consecutive_crossings = 0;
                    if (current_dir == NORTH)
                        north_wait.notify_all();
                    else
                        south_wait.notify_all();
                }
                if (dir == NORTH)
                    north_wait.wait(lock);
                else
                    south_wait.wait(lock);
            }
        } else {
            while ((vehicles_on_bridge > 0 && current_dir != dir) ||
                   emergency_vehicle_waiting) {
                if (dir == NORTH)
                    north_wait.wait(lock);
                else
                    south_wait.wait(lock);
            }
            current_dir = dir;
        }
    }

    if (dir == NORTH) waiting_north--;
    else waiting_south--;

    vehicles_on_bridge++;
    if (type != EMERGENCY) consecutive_crossings++;
    cout << "Vehicle crossing bridge (Dir: " << (dir == NORTH ? "NORTH" :
```

```
"SOUTH")
              << ", Type: " << (type == EMERGENCY ? "EMERGENCY" : "NORMAL") <<
")" << endl;
}

void exit_bridge(Direction dir, VehicleType type = NORMAL) {
    unique_lock<mutex> lock(bridge_mutex);
    vehicles_on_bridge--;

    if (type == EMERGENCY) {
        emergency_vehicle_crossing = false;
    }

    if (vehicles_on_bridge == 0) {
        if (fair_mode) {
            if (consecutive_crossings >= k || (current_dir == NORTH &&
waiting_south > 0) ||
                (current_dir == SOUTH && waiting_north > 0)) {
                current_dir = (current_dir == NORTH) ? SOUTH : NORTH;
                consecutive_crossings = 0;
            }
        } else {
            current_dir = (current_dir == NORTH) ? SOUTH : NORTH;
        }

        if (current_dir == NORTH)
            north_wait.notify_all();
        else
            south_wait.notify_all();
    }
}
```

## Section B: Networking and Socket Programming (30 marks)

### Q4. (10 marks)

Based on the video streaming simulation in Assignment 4:

a) Compare and contrast TCP and UDP in the context of video streaming. Include specific metrics like throughput, latency, and reliability in your answer. (4 marks)

**Answer:**

| Metric | TCP | UDP |
|---|---|---|
| Throughput | Generally lower due to overhead of reliability mechanisms | Higher for the same network conditions |
| Latency | Higher due to connection setup, ordered delivery, and retransmission | Lower as packets are sent without waiting |

| Metric | TCP | UDP |
|---|---|---|
| Reliability | Guaranteed delivery, no packet loss | Packets may be lost without retransmission |
| Use case | Video-on-demand where quality is prioritized over timeliness | Live streaming where timeliness is critical |

TCP ensures all packets arrive correctly and in order but introduces latency due to retransmissions and head-of-line blocking. UDP provides lower latency but requires application-level handling of packet loss, making it better for real-time applications where occasional frame drops are acceptable.

b) Explain the difference between FCFS and Round-Robin scheduling policies for a streaming server. Which would perform better under high load and why? (3 marks)

**Answer:**

- FCFS (First-Come-First-Serve): Processes client requests in the order they arrive. Each client gets the server's full attention until completion.
- Round-Robin: Serves each client for a fixed time slice in a circular order, cycling through all connected clients.

Under high load, Round-Robin generally performs better because:

- It prevents long-waiting clients from experiencing starvation
- It distributes server resources more evenly, especially when some clients have slow connections
- It provides a more consistent user experience across all clients
- It reduces the maximum response time for any client

c) Write pseudocode for implementing a simple rate-limiting mechanism in a streaming server. (3 marks)

**Answer:**

```
// Token bucket algorithm for rate limiting
MAX_TOKENS = MAXIMUM_BANDWIDTH_MBPS
tokens = MAX_TOKENS
last_update_time = current_time()

function send_data(data_size):
    // Calculate tokens to add based on time elapsed
    current = current_time()
    time_passed = current - last_update_time
    tokens_to_add = time_passed * TOKEN_RATE

    // Update token count (capped at maximum)
    tokens = min(tokens + tokens_to_add, MAX_TOKENS)
    last_update_time = current

    // Check if we have enough tokens
    if data_size <= tokens:
        tokens = tokens - data_size
        send_packet(data)
        return SUCCESS
    else:
        // Not enough bandwidth, delay or reduce quality
        return BANDWIDTH_EXCEEDED
```

### Q5. (10 marks)

Regarding TCP connection establishment:

a) Explain the TCP three-way handshake process. Draw a diagram showing the sequence of packets exchanged. (3 marks)

**Answer:**

1. Client sends SYN packet with initial sequence number (ISN_C)
2. Server responds with SYN-ACK packet with its own ISN_S and acknowledgment of client's sequence number (ISN_C + 1)
3. Client sends ACK packet acknowledging server's sequence number (ISN_S + 1)

Diagram:

```
Client                  Server
  |                       |
  |------- SYN -------→ | Seq=ISN_C
  |                       |
  |←--- SYN-ACK -------| Seq=ISN_S, Ack=ISN_C+1
  |                       |
  |------- ACK -------→ | Seq=ISN_C+1, Ack=ISN_S+1
  |                       |
Connection Established
```

b) What is the purpose of the TCP sequence number? How is it used for reliable data transfer? (3 marks)

**Answer:** TCP sequence numbers serve multiple purposes:

- Track the ordering of packets to allow reassembly at the receiver
- Detect duplicate packets that might be received
- Track which bytes have been successfully received and acknowledged

For reliable data transfer, the sender assigns sequence numbers to each byte of data. The receiver acknowledges receipt by sending back an acknowledgment number equal to the next expected sequence number. If the sender doesn't receive an acknowledgment within a timeout period, it retransmits the unacknowledged packets. This ensures all data arrives at the receiver without loss or duplication.

c) Using Wireshark syntax, write a display filter that would show only the TCP handshake packets for connections to port 443. (2 marks)

**Answer:**

```
tcp.flags.syn == 1 && tcp.port == 443
```

d) Explain how TCP handles packet loss and retransmission. (2 marks)

**Answer:** TCP handles packet loss and retransmission through:

1. Sequence numbers to track sent bytes
2. Acknowledgment numbers to confirm received bytes
3. Retransmission timeout (RTO): If no ACK is received before timeout, packet is retransmitted
4. Fast retransmit: If sender receives 3 duplicate ACKs for the same sequence number, it assumes packet loss and retransmits without waiting for timeout
5. Selective acknowledgment (SACK): Allows receiver to acknowledge non-contiguous blocks of data, reducing unnecessary retransmissions
6. Exponential backoff: Increases RTO exponentially after multiple timeouts

### Q6. (10 marks)

Based on the networking concepts covered in Assignment 3:

a) Explain how adaptive bitrate streaming works. What metrics does a client use to determine the appropriate quality level? (3 marks)

**Answer:** Adaptive bitrate streaming works by:

1. Encoding the same video content at multiple bitrates/resolutions
2. Dividing each encoding into small segments (typically 2-10 seconds)
3. Creating manifest files that list the available qualities and segments
4. Having the client dynamically select the appropriate quality for each segment based on network conditions

The client makes quality decisions based on metrics including:

- Available bandwidth estimation
- Buffer fullness (how much video is pre-loaded)
- Buffer drain rate
- Recent throughput measurements
- Rate of successful segment downloads
- Playback experience (recent stalls/rebuffering)

b) What is a CDN (Content Delivery Network) and how does it improve video streaming performance? (2 marks)

**Answer:** A Content Delivery Network (CDN) is a distributed network of servers that delivers content to users based on their geographic location. CDNs improve video streaming by:

- Reducing latency by serving content from edge servers closer to the user
- Distributing load across multiple servers to handle traffic spikes
- Providing redundancy to improve reliability
- Optimizing delivery paths to minimize network congestion

- Caching popular content to reduce origin server load
- Implementing optimized protocols for faster delivery

c) Explain the role of manifest files in HTTP-based streaming protocols like HLS and MPEG-DASH. (3 marks)

**Answer:** Manifest files are essential configuration files in HTTP-based streaming that:

- List all available quality levels (bitrates, resolutions)
- Provide URLs for each segment of the video at each quality level
- Specify segment duration
- Include codec information and other metadata
- Define alternate audio tracks and subtitles
- May contain DRM information

In HLS, manifest files use the .m3u8 format with a master playlist containing information about different renditions and media playlists for each quality level.

In MPEG-DASH, the manifest is an XML file called the Media Presentation Description (MPD) that hierarchically defines periods, adaptation sets, representations, and segments.

Clients first download the manifest file to understand what content is available, then make decisions about which segments to request based on their capabilities and network conditions.

d) Compare evening vs. morning network performance for video streaming services based on the metrics observed in Assignment 3. (2 marks)

**Answer:** Evening vs. morning network performance:

- Evening performance showed 30-40% lower throughput (2.95 Mbps vs 4.32 Mbps) due to network congestion during peak hours
- Higher average RTT in evenings (68.94ms vs 45.32ms) indicating increased latency
- Higher packet retransmission rates in evenings (0.38% vs 0.12%)
- More frequent quality switches during evening streaming sessions
- Lower average streaming quality achieved during peak hours
- Larger playback buffers typically used in evening to compensate for variability

## Section C: Memory Management (10 marks)

### Q7. (10 marks)

Based on the memory examples in Assignment 1:

a) Identify and explain the memory leak in the following code: (3 marks)

```
void function() {
    int* arr = (int*)malloc(10 * sizeof(int));
    arr[0] = 5;
    arr = (int*)malloc(20 * sizeof(int));
    free(arr);
}
```

**Answer:** The memory leak occurs because the first memory allocation (`int* arr = (int*)malloc(10 * sizeof(int))`) is never freed. When `arr` is reassigned to a new memory allocation (`arr = (int*)malloc(20 * sizeof(int))`), the reference to the original 10-integer block is lost, but the memory remains allocated. Only the second allocation is freed with `free(arr)` at the end of the function.

To fix this, the code should free the first allocation before reassigning the pointer:

```
void function() {
    int* arr = (int*)malloc(10 * sizeof(int));
    arr[0] = 5;
    free(arr);  // Free first allocation
    arr = (int*)malloc(20 * sizeof(int));
    free(arr);  // Free second allocation
}
```

b) What is the difference between stack and heap memory? Where are local variables, global variables, and dynamically allocated memory stored? (3 marks)

**Answer:** Stack vs. Heap memory:

- Stack memory:
    - Automatically managed (allocation/deallocation)
    - Fixed size determined at compile time
    - Fast allocation (just moves stack pointer)
    - LIFO (Last In, First Out) data structure
    - Stores local variables, function parameters, return addresses
    - Limited in size (typically a few MB)
- Heap memory:
    - Manually managed in C/C++ (malloc/free, new/delete)
    - Variable size that can grow during program execution
    - Slower allocation (requires finding free blocks)
    - Can be accessed in any order
    - Stores dynamically allocated data
    - Much larger than stack (limited by physical memory)
- Local variables are stored on the stack
- Global variables are stored in the data segment (static memory)
- Dynamically allocated memory is stored on the heap

c) Explain the concept of memory fragmentation. How can it be minimized? (2 marks)

**Answer:** Memory fragmentation occurs when free memory becomes divided into small, non-contiguous blocks that are too small to satisfy allocation requests, even though the total free memory might be sufficient.

Types of fragmentation:

- External fragmentation: Free memory is fragmented into small blocks between allocated blocks
- Internal fragmentation: Allocated memory blocks are larger than needed

Minimization techniques:

- Use memory pools or object pools for fixed-size allocations
- Implement custom memory allocators for specific usage patterns
- Use compacting garbage collectors (in languages that support them)
- Allocate similar-lifetime objects together
- Avoid frequent allocations and deallocations in performance-critical code
- Use appropriate data structures to minimize fragmentation

d) Write a simple program that demonstrates proper use of `valgrind` to detect memory leaks. (2 marks)

**Answer:**

```c
#include <stdlib.h>
#include <stdio.h>

// Function with a memory leak
void leaky_function() {
    int* array = malloc(10 * sizeof(int));
    // Missing free(array)
}

// Function with proper memory management
void proper_function() {
    int* array = malloc(10 * sizeof(int));
    free(array);
}

int main() {
    // Call both functions
    leaky_function();
    proper_function();

    printf("To detect memory leaks, run with:\n");
    printf("valgrind --leak-check=full --show-leak-kinds=all ./a.out\n");

    return 0;
}
```

Running this with valgrind will show:

```
valgrind --leak-check=full --show-leak-kinds=all ./a.out
```

Output will indicate a memory leak in `leaky_function()` but not in `proper_function()`.

## Short Q&A Items (50 Questions)

### Thread Synchronization and Concurrency

1. **Q:** What happens if a thread calls `wait()` on a condition variable without holding the associated mutex? **A:** Runtime error or undefined behavior. Condition variables require the associated mutex to be locked before calling wait().

2. **Q:** What is the difference between `notify_one()` and `notify_all()`? **A:** `notify_one()` wakes up a single waiting thread, while `notify_all()` wakes up all threads waiting on the condition variable.

3. **Q:** Why is it necessary to check the predicate in a while loop when using condition variables? **A:** To protect against spurious wakeups and handle cases where the condition might be true but change before the thread gets the lock.

4. **Q:** What is a race condition? **A:** A situation where multiple threads access shared data concurrently, and the outcome depends on the relative timing of their execution.

5. **Q:** How does the `detach()` method differ from `join()`? **A:** `detach()` separates the thread from its thread object letting it run independently, while `join()` waits for the thread to complete execution.

6. **Q:** What is thread starvation? **A:** A situation where a thread is unable to gain regular access to shared resources and is unable to make progress.

7. **Q:** What is a deadlock? **A:** A situation where two or more threads are unable to proceed because each is waiting for the other to release a resource.

8. **Q:** What is a mutex used for? **A:** To provide exclusive access to a shared resource, preventing data races and ensuring thread safety.

9. **Q:** When would you use `unique_lock` instead of `lock_guard`? **A:** When you need more flexibility like manually unlocking, working with condition variables, or using deferred locking.

10. **Q:** What does the C++ `std::thread::hardware_concurrency()` function tell you? **A:** It provides a hint about the number of concurrent threads supported by the implementation/hardware.

11. **Q:** What's the difference between logical and physical concurrency? **A:** Logical concurrency is having multiple threads in a program, while physical concurrency is actually executing multiple threads simultaneously on multiple CPU cores.

12. **Q:** What is a critical section? **A:** A segment of code that accesses shared resources and must not be executed by more than one thread at a time.

13. **Q:** In the bus simulation program, what would happen if the bus capacity (N) was set to 0? **A:** The program would deadlock because passengers would wait indefinitely for space on the bus, and the bus would never depart.

14. **Q:** What is the purpose of a latch/barrier synchronization primitive? **A:** It allows multiple threads to wait until all reach a certain point before any proceed.

## Networking and Socket Programming

15. **Q:** What is the key difference between blocking and non-blocking sockets? **A:** Blocking socket operations don't return until the operation completes, while non-blocking operations return immediately with an error code if the operation would block.

16. **Q:** Why would you set the `SO_REUSEADDR` socket option? **A:** To allow binding to a port that's in TIME_WAIT state, useful for quick server restarts.

17. **Q:** What happens if you try to connect to a port where no server is listening? **A:** The connection attempt fails with a connection refused error.

18. **Q:** What is the purpose of the `listen()` system call? **A:** It marks a socket as passive and specifies the maximum number of pending connections in the queue.

19. **Q:** What does the `htons()` function do? **A:** Converts a 16-bit integer from host byte order to network byte order (big-endian).

20. **Q:** How does TCP ensure reliable data transfer? **A:** Through sequence numbers, acknowledgments, checksums, retransmission of lost packets, and flow control mechanisms.

21. **Q:** What is the MTU (Maximum Transmission Unit) and why is it important? **A:** The largest size packet that can be transmitted over a network. Data larger than the MTU must be fragmented, affecting performance.

22. **Q:** What is the main difference between `select()` and `epoll()` for handling multiple connections? **A:** `select()` checks all file descriptors in each call (O(n) complexity), while `epoll()` uses a callback mechanism for active descriptors only (O(1) complexity).

23. **Q:** What is a socket backlog? **A:** The maximum number of pending connections that can be queued up before the server accepts them.

24. **Q:** How does UDP packet loss affect video streaming compared to file transfer? **A:** In video streaming, packet loss may cause momentary glitches but playback continues; in file transfers, it results in corrupted files.

25. **Q:** What is the purpose of the Nagle algorithm in TCP? **A:** To reduce network congestion by buffering small outgoing packets until acknowledgments for outstanding packets are received.

26. **Q:** Why might a TCP connection hang in the `CLOSE_WAIT` state? **A:** The application has not closed its side of the connection after the remote peer closed its side.

27. **Q:** What is TCP flow control? **A:** A mechanism to prevent the sender from overwhelming the receiver by dynamically adjusting the transmission rate based on the receiver's capacity.

28. **Q:** What is the difference between congestion control and flow control? **A:** Flow control prevents the sender from overwhelming a receiver, while congestion control prevents overwhelming the network itself.

29. **Q:** Why do streaming applications often prefer UDP over TCP? **A:** Lower latency, no retransmission delays, and the ability to continue playback even with some packet loss.

### Network Analysis and Protocols

30. **Q:** What information does Wireshark capture when analyzing network traffic? **A:** Packet contents, timestamps, protocol information, source and destination addresses/ports, and various protocol-specific fields.

31. **Q:** How does TLS establish a secure connection? **A:** Through handshake phases including cipher suite negotiation, key exchange, server/client authentication, and session key establishment.

32. **Q:** What is QUIC and how does it improve over TCP+TLS? **A:** QUIC combines transport and encryption layers, reducing connection setup latency, handling multiplexing without head-of-line blocking, and improving connection migration.

33. **Q:** Why does HTTP/2 use a single TCP connection with multiple streams? **A:** To reduce overhead from multiple TCP connections, improve parallelism, and eliminate redundant headers.

34. **Q:** How does HLS (HTTP Live Streaming) handle adaptive bitrate streaming? **A:** By creating multiple renditions of the content at different bitrates and using short segment files with manifest files that let clients switch between renditions.

35. **Q:** What does a CDN edge server do to improve streaming performance? **A:** Caches content closer to end users, reducing latency and improving throughput by avoiding long-distance network routes.

36. **Q:** What does RTT (Round Trip Time) indicate in network performance? **A:** The time it takes for a packet to travel from source to destination and back, indicating network latency.

37. **Q:** How does video streaming services handle network jitter? **A:** By using client-side buffers that store several seconds of video to absorb variations in packet arrival time.

38. **Q:** What's the significance of the TCP window size? **A:** It determines how much data can be in flight (sent but not acknowledged) at once, affecting throughput, especially over high-latency connections.

39. **Q:** Why does packet capture show more DNS queries than expected web domains? **A:** Modern websites often include content from multiple domains, CDNs, tracking services, and ad networks.

### Memory Management

40. **Q:** What happens when `malloc()` cannot allocate the requested memory? **A:** It returns a NULL pointer, indicating the allocation failed.

41. **Q:** What is the difference between `malloc()` and `calloc()`? **A:** `malloc()` allocates uninitialized memory, while `calloc()` initializes all bytes to zero.

42. **Q:** What causes a double free error? **A:** Attempting to call `free()` on the same memory address that has already been freed.

43. **Q:** What is a dangling pointer? **A:** A pointer that references memory that has been deallocated or is out of scope.

44. **Q:** How does `valgrind` detect memory leaks? **A:** By tracking all memory allocations and deallocations, and reporting any allocations that were never freed when the program terminates.

45. **Q:** What is heap fragmentation? **A:** The condition where the heap contains many small free blocks interspersed between allocated blocks, making it difficult to allocate larger chunks.

46. **Q:** What is the purpose of `realloc()`? **A:** To resize a previously allocated memory block, possibly moving it to a new location if necessary.

47. **Q:** In C++, what is RAII and how does it help with resource management? **A:** Resource Acquisition Is Initialization - a pattern where resource acquisition occurs in the constructor and release in the destructor, ensuring proper cleanup even with exceptions.

48. **Q:** What happens if you allocate too much memory on the stack? **A:** Stack overflow, which typically crashes the program with a segmentation fault.

49. **Q:** What is the advantage of using smart pointers over raw pointers in C++? **A:** Automatic memory management that prevents leaks by handling deallocation when the pointer goes out of scope.

50. **Q:** What is the difference between `new` and `malloc()`? **A:** `new` is a C++ operator that constructs objects, while `malloc()` is a C function that only allocates raw memory without initialization. Additionally, `new` throws exceptions on failure while `malloc()` returns NULL.

## Additional Q&A Items

### Thread Synchronization Advanced Concepts

51. **Q:** What is a semaphore and how does it differ from a mutex? **A:** A semaphore is a synchronization primitive that maintains a count and allows multiple threads to access a resource simultaneously up to a specified limit, while a mutex allows only one thread exclusive access.

52. **Q:** What is the monitor pattern in concurrent programming? **A:** A synchronization construct that encapsulates shared data with procedures that provide exclusive access to that data, using mutual exclusion and condition variables.

53. **Q:** Explain the difference between a binary semaphore and a mutex. **A:** While both allow only one thread at a time, a binary semaphore can be released by any thread, whereas a mutex should only be released by the thread that acquired it.

54. **Q:** What is the readers-writers problem? **A:** A synchronization problem where multiple readers can access shared data simultaneously, but writers need exclusive access. The challenge is balancing throughput and preventing starvation.

55. **Q:** What is priority inversion and how can it be addressed? **A:** A scenario where a high-priority thread is indirectly preempted by a lower-priority thread. Solutions include priority inheritance, priority ceiling protocols, or avoiding priority-based scheduling.

56. **Q:** In the bridge crossing simulation (Assignment 2), what ensures that vehicles from both directions eventually get to cross? **A:** The fair mode implementation with the parameter k that limits the number of consecutive crossings from one direction before switching.

57. **Q:** What is the dining philosophers problem and what does it illustrate? **A:** A classic synchronization problem illustrating challenges in resource allocation and deadlock avoidance, where philosophers (threads) need two resources (forks) simultaneously to eat.

58. **Q:** What is the difference between coarse-grained and fine-grained locking? **A:** Coarse-grained locking uses fewer locks to protect larger sections of code or data, while fine-grained locking uses many locks for smaller sections, offering better concurrency but increased complexity.

59. **Q:** How does a C++ `std::atomic` differ from using a mutex? **A:** Atomics provide lock-free operations on individual variables with guarantees about visibility and ordering, while mutexes provide exclusive access to code regions that may operate on multiple variables.

60. **Q:** What is a thread barrier? **A:** A synchronization primitive that blocks a group of threads until a specified number of threads have reached the barrier, then lets all threads proceed.

### Memory Management Extended

61. **Q:** What is the difference between shallow copy and deep copy? **A:** Shallow copy duplicates only the pointers to objects, while deep copy creates new copies of the pointed-to objects themselves.

62. **Q:** How does the C++ `std::shared_ptr` work internally? **A:** It uses reference counting to track how many shared pointers point to an object, and automatically deallocates the object when the count reaches zero.

63. **Q:** What are memory-mapped files and what advantages do they offer? **A:** Files mapped directly to memory addresses, allowing file I/O to be treated as memory access. Benefits include potential performance improvements and simplified code.

64. **Q:** What is the difference between stack unwinding and memory leaks? **A:** Stack unwinding is the process of removing function frames from the call stack during exception handling, while memory leaks occur when allocated memory is never freed.

65. **Q:** How does the C++ move semantics help with memory management? **A:** Move semantics allows resources to be transferred between objects without copying, reducing overhead and preventing unnecessary duplication of resources.

### Networking and Protocols Deep Dive

66. **Q:** What are TCP keepalive packets and why are they used? **A:** Special packets sent periodically to verify if an idle connection is still active, used to detect disconnected peers and clean up dead connections.

67. **Q:** In the context of networking, what is a socket pair? **A:** A unique 4-tuple consisting of source IP, source port, destination IP, and destination port that uniquely identifies a connection.

68. **Q:** How does TCP slow start work? **A:** A congestion control algorithm where a new connection starts with a small congestion window that doubles in size with each successful acknowledgment, until reaching the slow start threshold.

69. **Q:** What is the difference between HTTP pipelining and HTTP multiplexing? **A:** HTTP pipelining (HTTP/1.1) sends multiple requests without waiting for responses but suffers from head-of-line blocking, while multiplexing (HTTP/2) sends multiple requests asynchronously over a single connection with independent streams.

70. **Q:** What is the purpose of ICMP? **A:** Internet Control Message Protocol is used for error reporting and operational information about IP packet processing, including ping and traceroute functionalities.

71. **Q:** How does a CDN handle content routing decisions? **A:** Through DNS-based redirection, anycast routing, HTTP redirection, and load balancing algorithms that consider factors like user location, server load, and network conditions.

72. **Q:** What challenges does adaptive bitrate streaming solve compared to progressive downloading? **A:** It adapts to changing network conditions, reduces buffering, conserves bandwidth, supports live streaming, and provides faster startup times.

73. **Q:** How does a TCP reset (RST) packet differ from normal connection termination? **A:** A RST immediately aborts a connection without the normal four-way termination handshake, used for error conditions or abnormal termination.

74. **Q:** What causes TCP retransmissions and how do they affect streaming performance? **A:** Packet loss, timeout expirations, or duplicate ACKs cause retransmissions, which increase latency and may lead to buffering in streaming applications.

75. **Q:** What is the difference between multicast and broadcast in networking? **A:** Broadcast sends packets to all devices on a network segment, while multicast sends to a group of interested receivers, conserving bandwidth.

### Video Streaming and Real-time Applications

76. **Q:** What is jitter buffer in streaming applications? **A:** A buffer that collects packets and delays their processing to compensate for variable network delays (jitter), ensuring smooth playback.

77. **Q:** Why do video streaming services typically use HTTP-based streaming despite TCP's overhead? **A:** HTTP-based streaming leverages existing web infrastructure (CDNs, caches, proxies), simplifies firewall traversal, and provides reliability for video segments.

78. **Q:** How does Forward Error Correction (FEC) help in real-time streaming? **A:** It adds redundant data to transmitted packets allowing receivers to reconstruct lost packets without retransmission, reducing latency.

79. **Q:** What is the difference between live latency and VOD (Video on Demand) latency requirements? **A:** Live streaming typically requires latency under 30 seconds (ideally 5-10 seconds for interactive experiences), while VOD can tolerate much higher initial buffering times.

80. **Q:** How does MPEG-DASH handle encryption and DRM? **A:** Through the Common Encryption (CENC) specification, which allows content to be encrypted once but played using different DRM systems specified in the manifest.

### Systems Programming

81. **Q:** What is the difference between blocking I/O and non-blocking I/O? **A:** Blocking I/O operations do not return until completed, potentially pausing thread execution, while non-blocking I/O returns immediately with available data or an error code.

82. **Q:** What is the purpose of `fork()` and how does it relate to `exec()`? **A:** `fork()` creates a new process by duplicating the calling process, while `exec()` replaces the current process image with a new one. They're often used together to spawn new programs.

83. **Q:** What is the difference between user space and kernel space? **A:** Kernel space is where the OS kernel executes with full hardware access privileges, while user space is where applications run with limited privileges and controlled hardware access.

84. **Q:** What is the purpose of the `/proc` filesystem in Linux? **A:** A virtual filesystem that provides access to information about system processes, hardware configuration, and kernel parameters in a file-like interface.

85. **Q:** What is a system call and how does it differ from a library function call? **A:** A system call is a request for service from the kernel that requires a context switch from user mode to kernel mode, while a library function executes entirely in user space.

86. **Q:** How does `select()` work for I/O multiplexing? **A:** It monitors multiple file descriptors, waiting until one or more become ready for I/O operations, allowing a program to handle multiple I/O sources with a single thread.

87. **Q:** In the context of socket programming, what does the `bind()` function do? **A:** Associates a socket with a specific local address and port number, necessary for server sockets that need to listen on a known port.

88. **Q:** What happens during a context switch between threads? **A:** The CPU saves the state of the current thread, including register values and instruction pointer, then loads the saved state of the next thread to be executed.

89. **Q:** What is the purpose of the `setsockopt()` function? **A:** Allows manipulation of options for a socket, such as buffer sizes, timeouts, address reuse, and keep-alive settings.

90. **Q:** How does a zero-copy operation improve network performance? **A:** By avoiding data copying between kernel and user space buffers, reducing CPU usage, memory bandwidth consumption, and latency.

### Advanced Networking

91. **Q:** What is the difference between TLS 1.2 and TLS 1.3? **A:** TLS 1.3 reduces handshake latency (1-RTT vs 2-RTT), removes support for legacy algorithms, improves security, and adds features like 0-RTT resumption and encrypted SNI.

92. **Q:** How does IPv6 address the limitations of IPv4? **A:** Provides vastly more addresses (128-bit vs 32-bit), simplifies header format, improves routing efficiency, offers built-in security, and eliminates NAT requirements.

93. **Q:** What is a STUN server used for? **A:** Session Traversal Utilities for NAT helps devices behind NATs discover their public IP addresses and port mappings to facilitate peer-to-peer connections.

94. **Q:** How does a sliding window protocol improve network throughput? **A:** By allowing multiple packets to be in transit simultaneously before requiring acknowledgment, utilizing bandwidth more efficiently, especially on high-latency links.

95. **Q:** What is the concept of opportunistic encryption? **A:** A security approach where communications are encrypted when possible without requiring explicit configuration, falling back to unencrypted if necessary.
96. **Q:** What is the difference between symmetric and asymmetric encryption? **A:** Symmetric encryption uses the same key for encryption and decryption (faster but key distribution is challenging), while asymmetric uses different public/private keys (slower but solves key distribution).
97. **Q:** How does DNS over HTTPS (DoH) improve privacy? **A:** By encrypting DNS queries inside HTTPS traffic, preventing ISPs and network observers from seeing which domains are being requested.
98. **Q:** What is the difference between connection pooling and persistent connections? **A:** Connection pooling maintains multiple reusable connections for different clients, while persistent connections (keep-alive) reuse the same connection for multiple requests from the same client.
99. **Q:** How does a load balancer perform health checks? **A:** By periodically sending test requests to backend servers and measuring response times, status codes, or application-specific metrics to determine server health.
100. **Q:** What is the WebRTC protocol stack and what applications use it? **A:** A collection of protocols (RTP/RTCP, DTLS, ICE, etc.) enabling real-time communication directly between browsers without plugins, used for video conferencing, voice calls, file sharing, and gaming.