Systems Programming HW4

Saiprakash Nalubolu

B01037579

Q1.

Message queues are a mechanism used for exchanging information between processes, often in a server/client application framework. They provide an asynchronous communication method where messages are stored in a queue until they are retrieved by the receiving process.

When a malicious process reads a message from a message queue that is being used by a server and several clients here is what is expected.

(1) **To the server/client:**

1. If the malicious process reads a message from the queue, that message is removed from the queue and the intended recipient (client or server) will not receive it. This can cause data loss or the failure of operations that depend on that message.

2. The server or clients might be waiting for specific messages to proceed with their tasks. The absence of expected messages can lead to timeouts, retries, or even application crashes. Critical operations might be delayed or disrupted, leading to degraded performance or denial of service.

3. If the malicious process alters the message before re-inserting it (if allowed), it can lead to corrupted data being processed by the server or clients. This can have serious consequences, especially in systems handling sensitive or critical data.

4. Sensitive information might be exposed if the messages contain confidential data. The malicious process could leak, misuse, or exploit this information, leading to security vulnerabilities and breaches.

5. The server and clients might become out of sync if messages are lost or altered, leading to inconsistent states and potential errors in processing.

(2) **Criteria for a Malicious Process to Read the Message Queue:**

POSIX message queues allow processes to exchange messages in a standardized way. They are identified by a name (for example /my_queue) and accessed via system calls such as mq_open, mq_send, and mq_receive. Each POSIX message queue has associated permissions similar to file permissions in Unix-like systems. These permissions determine who can read from or write to the queue. When a message queue is created using mq_open, it is assigned attributes including permissions. The permissions are set using a mode argument, which specifies read, write, and execute permissions for the owner, group, and others.

Criteria for Access:

1. Process Identity: The malicious process must have the appropriate user or group identity to match the permissions set on the message queue.
2. Permissions: The message queue must be created with permissions that allow read access to the malicious process. This could happen if:
   - The queue is created with overly permissive settings (e.g., world-readable).
   - The malicious process runs under the same user or group identity that has read access.
3. Capability: On systems with fine-grained security controls, the malicious process might need specific capabilities (for example, CAP_SYS_ADMIN) to bypass standard permissions.

Some common Vulnerabilities: Developers might create message queues with default or insecure permissions. Running server and client processes with the same user identity might inadvertently grant access to unintended processes. Absence of mandatory access control policies (e.g., SELinux, AppArmor) that could enforce stricter controls over inter-process communication.

An example of Insecure Permissions:

mqd_t mq = mq_open("/my_queue", O_CREAT | O_RDWR, 0666, NULL); // World-readable and writable

In this example, the mode 0666 allows any process to read and write to the message queue.

An example of Secure Permissions:

mqd_t mq = mq_open("/my_queue", O_CREAT | O_RDWR, 0600, NULL); // Only owner can read and write

In this example, the mode 0600 restricts access to the owner of the message queue.

Conclusion:

If a malicious process reads a message from a message queue used by a server and clients, it can cause message loss, service disruption, data corruption, security breaches, and operational inconsistency. For a malicious process to read a POSIX message queue, it must have the necessary permissions, which can be due to overly permissive settings, shared user/group identities, or inadequate security policies. By understanding these potential impacts and criteria, one can better appreciate the importance of securing inter-process communication in distributed systems.

Q2.

FIFOs (First-In-First-Out) and POSIX message queues are both inter-process communication (IPC) mechanisms that allow processes to exchange data. However, they have distinct characteristics and usage scenarios. Below are the detailed similarities and differences between FIFOs and POSIX message queues:

Similarities:

1. Both FIFOs and POSIX message queues are used for IPC, allowing processes to communicate and synchronize with each other.
2. Both mechanisms typically support unidirectional communication, though bidirectional communication can be achieved by using multiple FIFOs or message queues.
3. Both are managed by the operating system kernel, which handles the underlying implementation and data storage.
4. Both FIFOs and POSIX message queues support blocking and non-blocking operations, allowing processes to wait for data or continue execution if no data is available.
5. Both can have access permissions set (read, write) to control which processes can access the IPC mechanism.

Differences:

1. FIFOs operate on a byte stream basis, meaning data is read in the same order it was written without any message boundaries. Whereas POSIX message queues operate on discrete messages, where each message has a defined boundary and can be read individually.
2. FIFOs are identified by a name in the filesystem and created using the mkfifo command or the mkfifo system call. POSIX message queues are identified by a name starting with a / and created using the mq_open system call.
3. FIFOs do not support prioritization; data is read in the order it was written. But POSIX message queues support message prioritization, allowing messages to be inserted into the queue with different priority levels.
4. With FIFOs the capacity is typically limited by the system buffer size, and there are no explicit message limits. But POSIX message queues have configurable message size and queue size limits set during creation with mq_open.
5. FIFOs use standard file I/O operations (open, read, write, close) for communication. Whereas POSIX Message Queues use specialized message queue operations (mq_open, mq_send, mq_receive, mq_close, mq_unlink) for communication.
6. FIFOs do not support asynchronous notifications natively. But POSIX message queues: support asynchronous notifications via signals, allowing a process to be notified when a message is available.
7. With FIFOs error handling is similar to regular file I/O, using return values and errno. But with POSIX message queues error handling is also done through return values and errno, but specific errors related to message queue operations are provided.

8. With FIFOs security is managed through file system permissions. While POSIX message queues' Security is managed through queue-specific permissions set during creation.

Overall, FIFOs are simpler and use the familiar file I/O operations, suitable for stream-based communication. POSIX Message Queues offer more advanced features like message prioritization, discrete message handling, and asynchronous notifications, making them more flexible for complex IPC needs.

Q3.

POSIX binary semaphores and Pthread mutexes are both synchronization mechanisms used in concurrent programming to control access to shared resources and prevent race conditions. While they have some similarities, they serve different purposes and have distinct characteristics.

Similarities:

1. Both are used to synchronize access to shared resources and ensure mutual exclusion.
2. Both are supported by the OS kernel and provide efficient mechanisms for thread synchronization.
3. Both can block a thread if the resource is not available, causing the thread to wait until it can proceed.
4. Both are thread-safe and designed to be used in multi-threaded applications.

Differences:

1. A POSIX binary semaphore can only have two states, 0 or 1. It can be used to signal the availability of a resource. A Pthread mutex (mutual exclusion) is a locking mechanism that ensures only one thread can access a resource at a time.
2. Binary Semaphores are typically used for signaling between threads. They are useful for scenarios where you need to notify one thread that an event has occurred. Mutexes are primarily used for protecting critical sections of code, ensuring that only one thread can execute the protected section at a time.
3. API and operations performed by both are different.
   Binary Semaphores:
   - Initialization: sem_init
   - Wait (lock): sem_wait
   - Post (unlock): sem_post
   - Destroy: sem_destroy

   Pthread Mutexes:
   - Initialization: pthread_mutex_init
   - Lock: pthread_mutex_lock
   - Unlock: pthread_mutex_unlock
   - Destroy: pthread_mutex_destroy

4. Binary Semaphores do not have the concept of ownership. Any thread can signal (post) or wait (decrement) the semaphore. While mutexes have the concept of ownership. A mutex must be unlocked by the thread that locked it, ensuring proper usage and avoiding deadlocks.
5. Binary Semaphores do not handle priority inversion directly. But mutexes can handle priority inversion if configured to do so (e.g., using priority inheritance protocol).
6. Binary Semaphores are not directly associated with condition variables. Mutexes are often used in conjunction with condition variables (pthread_cond_wait, pthread_cond_signal) for more complex synchronization patterns.
7. Binary Semaphores are more prone to mismanagement leading to deadlocks since any thread can post or wait. Mutexes are designed to minimize the risk of deadlocks through proper usage patterns and ownership rules.
8. Binary Semaphores are generally more efficient for signaling since they only involve changing a flag state. Mutexes may involve more overhead due to ownership checks and potential priority adjustments.
9. Binary Semaphores can be initialized to 0 or 1, providing immediate control over resource availability. Mutexes are initialized in an unlocked state, ready to be locked by a thread.
10. Binary Semaphores are simpler and less prone to misuse, but lack the robust features of mutexes. Mutexes provide robust error checking and handling, making them suitable for more complex synchronization needs.

As a whole, POSIX Binary Semaphores are the best for simple signaling tasks between threads, with no ownership concept and very efficient for on/off signaling. And Pthread Mutexes are ideal for protecting critical sections and managing resource access, with features to handle priority inversion and robust error handling.

Q4.

(1)The program sets up a POSIX message queue notification system where a thread is created to handle incoming messages asynchronously. Here's a breakdown of program's functionality.

Main Function (main):

1. Opens a message queue specified by the first command-line argument in non-blocking read-only mode (mq_open).
2. Sets up a notification mechanism on the message queue using notifySetup.
3. Calls pause to wait indefinitely for signals, specifically the notification signal which will trigger the threadFunc.

Notification Setup (notifySetup):

1. Configures a sigevent structure to specify that the notification should create a new thread (SIGEV_THREAD) to execute the threadFunc.
2. Registers this notification with the message queue using mq_notify.

Thread Function (threadFunc):

1. Retrieves the message queue descriptor from the sigevent structure.
2. Gets the attributes of the message queue (including the maximum message size) using mq_getattr.
3. Allocates a buffer to hold incoming messages.
4. Re-registers the notification to ensure continuous notification handling after each message reception.
5. Enters a loop to receive messages from the queue (mq_receive) and prints the number of bytes read.
6. Frees the allocated buffer before exiting.

(2) Line 13 (notifySetup(mqdp)) is used for re-registering notification. That is when a message is received and the notification is triggered, the registration for notification is automatically removed. Calling notifySetup again ensures that the message queue is re-registered for the next notification. This way, the program can continuously handle incoming messages without missing any notifications.

(3) Making Buffer a Global Variable is possible but not recommended. It is technically possible to make the buffer a global variable and allocate its memory once in the main program. However, this approach has several downsides.

1. The current design ensures that each thread has its own buffer, avoiding concurrent access issues. Making the buffer global would require synchronization mechanisms (e.g., mutexes) to prevent race conditions.
2. Allocating and freeing memory within the thread function ensures proper management of resources. A global buffer might lead to complexities in managing the buffer's lifecycle, especially in a multi-threaded environment.
3. If multiple threads are supposed to handle messages simultaneously, each thread needs its own buffer to operate independently. A single global buffer would become a bottleneck and limit the program's ability to handle multiple messages concurrently.

While making the buffer global and allocating its memory in the main program is possible, it introduces potential issues related to thread safety, resource management, and scalability. The current approach of using a local buffer for each thread is more robust and aligns with good concurrent programming practices.

Q5.

(1) Consequences of Removing listen():

The listen() function marks the socket referred to by listenfd as a passive socket, meaning it will be used to accept incoming connection requests using accept().

If the listen() call is removed, the program will not be able to properly prepare the socket to accept connections. As a result, the subsequent accept() call (line 22) will fail because the socket is not in a listening state. The accept() function will return -1 and set errno to EOPNOTSUPP or a similar error, indicating that the operation is not supported because the socket is not a listening socket.

The program will enter the infinite loop, repeatedly calling accept(), which will fail each time. This will likely lead to error handling issues or an infinite loop without proper connection handling.

(2) Consequences of Removing bind():

The bind() function assigns a local address to the socket. In this case, it assigns the IP address and port number to the socket (listenfd).

If bind() is removed, the socket will not have a specific IP address and port number assigned to it. The listen() function will still be called, and it might succeed depending on the system and socket type. Some operating systems may automatically assign an ephemeral port if bind() is not called explicitly. However, since the program intends to run a daytime server on port 13, this automatic assignment would not fulfill that requirement.

If bind() is omitted and the OS does not assign a port, the program might fail at listen() or accept() with an error indicating that the socket is not properly bound. If the OS assigns an ephemeral port, the server will not be running on the intended port 13. Clients expecting the service on port 13 will not be able to connect to it. The daytime service will not be available on the standard port, rendering the server effectively useless for its intended purpose.

So, Removing listen() prevents the server from accepting connections, leading to failures and potential infinite looping without handling connections. And Removing bind() results in the server not being bound to the intended port (port 13), either causing accept() to fail or the server to listen on an incorrect, ephemeral port, making it inaccessible on the expected port.