System Programming - Linux
Week 11 - Message Queues

Lecturer: Dr. Aggrey Obbo (PhD)

June 11, 2025

Contents

1 Introduction

Welcome to this exploration of Linux Message Queues, an effective and flexible tool for inter-process communication (IPC). In concurrent programming, facilitating the exchange of information between independent processes in an efficient and reliable manner is inevitable. Message queues offer a manner for processes to communicate that is asynchronous, typed, and persistent, allowing them to operate independently and improving the resilience of the system. The objectives of this course is as listed below. By the conclusion of this lecture, one should be prepared to effectively employ message queues in Linux applications for resilient and scalable communication.

Objectives

- To understand the fundamental concepts of Linux message queues as a mechanism for asynchronous inter-process communication
- To gain proficiency in using the core system calls (msgget, msgsnd, msgrcv, msgctl) especially in controlling Linux message queues.

- To develop practical skills in implementing producer-consumer patterns and other communication scenarios using message queues.
- To learn how to monitor and manage Linux message queues using command-line tools.
- To identify and mitigate common pitfalls related to Linux message queue usage in inter-process communication.

2 Message Queues

In concurrent systems, several processes must operate as if they are running simultaneously, making effective and adaptable inter-process communication (IPC) essential. Although synchronous communication techniques, which require processes to wait for one another to continue, are useful, asynchronous IPC presents considerable benefits, especially in improving responsiveness and decoupling components.

Asynchronous communication permits a sending process to convey data or a signal without needing an immediate reply from the receiver. The sender can proceed with its operations, executing other tasks while the message is stored or processed independently by the receiving process at a later time. This non-blocking characteristic is especially crucial in concurrent systems for several reasons. First, it ensures that processes do not remain idle while waiting for others to respond, enhancing resource usage and overall system performance. An example could be a web server that manages several client requests; if every request paused the server until the complete processing of a response, the server's performance would be greatly hampered. Asynchronous communication allows the server to manage numerous requests at the same time, thus increasing responsiveness for all clients.

Secondly, asynchronous communication encourages improved modularity and fault tolerance. When processes communicate in an asynchronous manner, they are less interconnected, which means that the breakdown or temporary unavailability of one process is less likely to disrupt others. A producer process can keep producing data even if the consumer is momentarily engaged or offline. This separation increases the resilience and manageability of intricate systems.

Linux offers several inter-process communication (IPC) mechanisms, with message queues being a key option for facilitating asynchronous communication. Message queues are data structures managed by the kernel that serve as intermediaries, allowing processes to exchange messages without needing a direct or simultaneous connection (Figure 1 below).

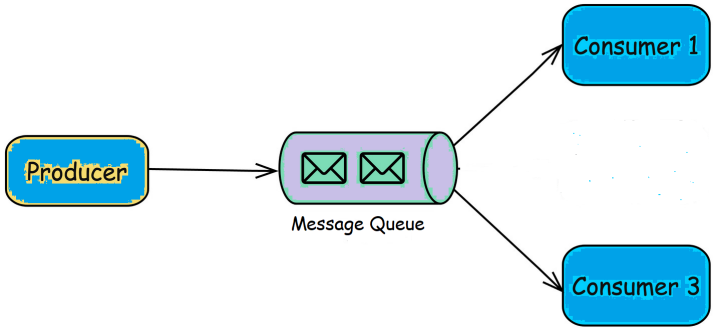

Figure 1: Message Queues in Linux.

A sender puts a message into the queue, while a receiver can retrieve it at a later time. The kernel guarantees the reliable delivery and ordering of these messages, which is typically done in a first-in, first-out (FIFO) manner. Additionally, message queues support message typing, which allows receivers to selectively access messages based on their specific content category. This blend of asynchronicity, buffering, and message typing makes Linux message queues an effective mechanism for developing responsive, scalable, and robust concurrent applications.

2.1 Message Queue Mechanism

Kernel-managed message passing serves as a key Inter-Process Communication (IPC) method in which the operating system kernel functions as a middleman for processes to share data. Rather than communicating directly, processes send messages to the kernel, which subsequently forwards them to the designated recipient process.

Message passing presents multiple benefits. Firstly, the kernel guarantees protection and security by allowing processes to communicate solely through specified channels, preventing interference with each other's memory areas.

Secondly, it takes care of the intricate aspects of message routing and buffering, enabling developers to concentrate on the content of the messages instead of the underlying transmission specifics. The kernel is also capable of overseeing message queues, offering short-term storage for messages in case the recipient process is temporarily inaccessible. Nevertheless, message passing managed by the kernel may lead to overhead due to the context switching between user space (where processes operate) and kernel space for every message that is sent and received. This can affect performance in comparison to other IPC methods like shared memory, which allows processes to access a shared memory region directly. Additionally, the kernel must oversee the message queues and guarantee equitable access, increasing its responsibilities. In spite of this overhead however, its dependability and security features make it an essential and widely used means of inter-process communication in various operating systems.

2.2 Key Message Passing Characteristics

The key message passing characteristics that make it an essential and widely used inter process mechanism are as follows:

- *Asynchronous Nature:* In asynchronous message passing, the sender process doesn't wait for the receiver to acknowledge message delivery. Once the message is sent to the kernel, the sender can continue its execution. The receiver can retrieve the message at a later time. This non-blocking behavior allows for greater concurrency and responsiveness, as processes aren't held up waiting for communication to complete. However, it introduces complexity in managing message delivery confirmation and potential out-of-order processing if not carefully handled at the application level.

- *Message Typing for Selective Retrieval:* Message typing allows senders to associate a type identifier with each message. Receivers can then selectively retrieve messages based on their type, enabling them to prioritize or filter incoming communication. This mechanism enhances flexibility by allowing processes to handle different categories of information differently. Instead of processing messages in a strict FIFO order, a process can specifically look for messages of a certain type, improving efficiency and organization in complex communication scenarios.

- *Persistence Within the Kernel:* Messages sent via a kernel-managed system can often persist within the kernel's memory until explicitly removed by the receiving process or until a system reboot. This persistence ensures that messages are not lost if the receiver is temporarily unavailable or busy. It provides a reliable communication channel, guaranteeing message delivery even if there are transient issues with the receiving process. However, it also necessitates careful management of kernel memory to prevent resource exhaustion from unread messages.

- *Ordered Delivery (Typically FIFO):* Many kernel-managed message passing systems guarantee ordered delivery of messages from a specific sender to a specific receiver. Typically, this order is First-In, First-Out (FIFO). This characteristic simplifies the design of applications that rely on the sequential processing of information. Receivers can expect to process messages in the same order they were sent, maintaining the logical flow of communication and reducing the complexity of managing message sequences at the application level.

- Flexible Message Structure: Kernel-managed message passing often supports flexible message structures. This means that the content of a message is generally treated as a sequence of bytes by the kernel, allowing applications to define their own data formats and structures within the message payload. This flexibility enables the exchange of diverse types of information, from simple control signals to complex data structures, catering to the varied communication needs of different applications without imposing rigid data format constraints at the kernel level.

These core characteristics—asynchronous operation, selective retrieval via typing, kernel persistence, ordered delivery, and flexible structures—collectively define robust kernel-managed message passing. They enable efficient, reliable, and adaptable inter-process communication, balancing concurrency with data integrity and allowing diverse application designs to leverage the operating system's communication backbone.

3 Message Queues - Use Cases

3.1 Improving Producer-Consumer Scalability

Message queues serve as intermediaries, effectively distinguishing the data-generating components (Producers) from those that handle processing (consumers).

Producers send messages to the queue without needing to be aware of whether or when a consumer is available. This non-blocking nature allows producers to continue their tasks uninterrupted, even if consumers are slow or temporarily offline. On the other hand, consumers fetch messages from the queue at their own speed. They can thus scale independently, increasing their numbers during high demand or decreasing them when demand drops, without affecting producers. This approach enhances resilience because if a consumer fails, messages stay securely in the queue until another consumer can take over, preventing data loss and ensuring ongoing operation hence reinforcing the entire system's robustness and flexibility.

3.2 Implementing task queues for background processing

Message queues implement task queues in background processing, enabling applications to maintain responsiveness by offloading non-critical operations.

When the main application, acting as a producer, encounters a task that doesn't require an immediate user response. Instead of executing it directly, the producer creates a "message" containing all necessary task data and sends it to a dedicated message queue. Concurrently, independent worker processes (consumers) continuously monitor this queue. And upon detecting a new task message, an available worker retrieves it from the queue and proceeds to execute the background operation. This execution is entirely decoupled from the main application's user interface, ensuring a seamless user experience. Once a worker successfully completes a task, it sends an acknowledgment back to the message queue, prompting the message's removal.

Crucially, message queues incorporate robust error handling: should a worker fail, the task message can be requeued for another attempt or redirected to a "dead-letter queue" for diagnostic purposes, preventing data loss.

This architecture offers significant advantages. It dramatically improves responsiveness by offloading intensive computations from the foreground. Scalability is enhanced as you can easily adjust the number of workers to match varying loads. Reliability and durability are core benefits, as tasks persist in the queue even if workers or parts of the system go offline, ensuring eventual processing. The inherent decoupling fosters modularity, allowing independent development and deployment of producers and consumers. Finally, message queues inherently provide load balancing, distributing tasks efficiently across available workers and preventing bottlenecks.

3.3 Building event notification systems where order and type matter

Message queues are fundamental in event notification systems, particularly When order and type of events are significant, message queues play an important role of separating event producers from consumers, which facilitates reliable and organized communication.

And when it comes to order, message queues by default support First-In, First-Out (FIFO) processing. Events are stored in the order they are received. In scenarios with a single consumer, strict ordering is maintained. For distributed systems, partitioning enables parallel processing while preserving order within each partition (for instance, all events related to a particular user are directed to the same partition). Producers may also incorporate sequence numbers or timestamps to enhance validation. Message acknowledgments guarantee that an event is completely processed before the next one is handled, preventing out-of-order execution due to failures.

And concerning type, message queues handle various events through different methods. Events generally contain metadata or a header that specifies their eventType (such as user.created, order.updated). Consumers can then check this type field to implement specific processing logic. Systems might utilize dedicated queues for particular event types, ensuring only pertinent events reach designated consumers. Alternatively, a publish-subscribe model permits producers to publish to a broad "topic," while consumers subscribe and filter for specific event types, allowing multiple services to respond to the same event in various ways.

This blend of ordered delivery and type-aware routing renders message queues essential for developing resilient, scalable, and reliable event-driven architectures where both the sequence and nature of actions are critical.

3.4 Communication between distinct modules within complex applications

Message queues play an essential role in enabling communication between different modules in intricate applications. They act as intermediaries that separate the modules.

In asynchronous communication modules do not need to operate simultaneously. A "producer" module sends a message to the queue and continues with its tasks. A "consumer" module retrieves and processes the message when it is prepared. This prevents bottlenecks and enhances the overall responsiveness of the system.

Decoupling makes modules autonomous. They do not need to know the exact location or availability of other modules. If one module encounters an issue, the others can keep functioning without direct impact, as messages are

held in the queue.

And when a module faces high demand, additional instances of consumer modules can be introduced to handle messages from the queue concurrently, spreading the workload evenly.

Because messages are stored in the queue until they are successfully processed, reliability and durability are enhanced. This is because data loss is minimized even if a consumer goes offline or crashes temporarily. Thus by implementing message queues, complex applications become more robust, scalable, and simpler to develop and maintain, as each module can progress independently while communicating efficiently.

4 Management of Message Queues

The kernel is essential for overseeing message queues, serving as the foundation for inter-process communication. It is responsible for allocating and freeing memory for message queues, managing message storage, and controlling synchronization mechanisms to guarantee secure concurrent access. Additionally, the kernel provides system calls that enable user-space processes to transmit and receive messages, thus functioning as a central intermediary for the exchange of queued data.

4.1 Message Queues as Kernel Objects

The kernel considers each message queue to be a separate kernel object. This implies:

1. Dedicated Data Structures: Each message queue that is created has a dedicated data structure allocated and managed by the kernel in kernel space. The structure structure contains all the essential metadata for the queue, such as:

   - Queue ID: A unique integer identifier that the kernel assigns the queue.
   - Message Buffer: Pointers to the specific memory areas where messages are stored.
   - Message Count and Size: Data regarding the number of messages in the queue and the total memory they occupy.
   - Synchronization Primitives: Tools such as semaphores or mutexes to ensure safe concurrent queue access by multiple processes, preventing race conditions during message sending or receiving.
   - Timestamps: Documentation of creation, last send, and last receive times.
   - Kernel-Managed Lifecycle: The kernel oversees the complete lifecycle of a message queue. Processes engage with the queue through system calls (msgget() for creation/access, msgsnd() for sending, msgrcv() for receiving, msgctl() for control actions), while the kernel takes care of the actual memory allocation, message buffering, and deallocation when the queue is explicitly deleted or the system restarts.
   - Persistent Storage: The messages in the queue remain in kernel memory until a receiving process fetches them. This ensures durability against temporary process failures, as messages stay accessible even if the sender or receiver crashes.

2. Identification using Unique Keys Generated by ftok() To enable multiple distinct processes to access the same message queue, System V IPC mechanisms utilize a universally unique identifier known as a key_t value that is commonly produced by the ftok() function

   The function ftok() is used to create a unique identifier known as a "key." By providing ftok() with a standard file path and a single-character project ID, it generates a distinct number (the unique key) based on this information. All applications that utilize the same file path and project ID with ftok() will receive the identical unique key. This key serves as the "address" for the shared resource (such as a shared memory segment or a message queue), enabling various programs to locate and interact with one another through it.

3. File-like Permissions Model for Access Control The kernel implements access control on message queues through a method similar to that used for regular files. This mechanism ensures that only permitted processes can send, receive, or manage a message queue:

   Each message queue is linked to a struct msqid_ds that contains an ipc_perm member. This ipc_perm structure holds the permission settings for access. Permissions are categorized into three groups:
   Owner: The user ID of the process that established the message queue.
   Group: The group ID of the process that established the message queue.

Others: All other system users.
And for permissions, each category usually consist of:
Read (0400 octal): Grants a process the ability to receive messages from the queue.
Write (0200 octal): Enables a process to send messages to the queue.
Note: Execute permission does not apply to message queues.
When a process creates a message queue using msgget(), it defines the initial permissions using a flag (e.g., 0666 | IPC_CREAT for read/write access for everyone). While the msgctl() system call permits the owner or superuser to alter these permissions or delete the message queue.

Security: This permission framework is vital for security, as it stops unauthorized processes from disrupting inter-process communication or accessing sensitive information transmitted via message queues.

5 Anotomy of a Message

Like a postal system, message queue mechanism exchange "messages" via a central hub known as the message queue. The messages contain the information that one section of an application needs to relay to another or that one application needs to share with a different one. Similar to a physical letter needs an address, a stamp, and the actual message, the digital "message" in a message queue also consists of:

1. The "Envelope" (Metadata): This serves as the outer layer of the message. It doesn't contain the actual content you wish to convey but rather information regarding the data. Some key elements you would find on this "envelope" include:
   - Message ID: A distinct tracking identifier used in recognizing individual messages.
   - Correlation ID: This identifier connects related messages, and is useful when engaging in a back-and-forth exchange between programs.
   - Timestamp: This indicates when the message dispatched?
   - Expiration Time (Time-to-Live): Shows how long should this message exist in the queue before it's deemed outdated and discarded if not picked up.
   - Priority: The priority indicates to what extent the message requires immediate attention.
   - Reply To: Specifies where the recipient should send the response.
   - Content Type: Shows What type of data is enclosed? whether it Is text, image, or a specific kind of computer code. This helps the recipient to understand how to interpret the message's contents.
2. The "Address" (Routing Information): This information helps determine how the message queue route the message. Although not always be a distinct part of the message itself.
3. The "Actual Letter" also called the Payload/Body: This is the most crucial component—the actual information to be sent. It constitutes the data, the instructions, the information, or the event specifics that the sending application seeks to convey to the receiving application. This could be a simple text string such as "new order received" or even a complex segment of data representing an entire customer profile.

6 Key System Calls

msgget() is like setting up a special mailbox system for different parts of a computer program to send messages to each other. msgget()'s job is to either create a new mailbox or find an existing one so you can start putting messages in it or taking them out.

key_t key is a unique number that identifies the mailbox under consideration. For communication between different programs the same key must be used.

int msgflg: These are special instructions for msgget() that specifies

- IPC_CREAT: This flag tells msgget(): "If there's no mailbox at this key address, please create one for me."
- IPC_EXCL: Use this with IPC_CREAT. It means: "Create a new mailbox, but only if one doesn't already exist. If it does, tell me there's an error." This is useful if you want to be the only one creating that specific mailbox.

Permission bits: These are like setting who can access your mailbox (e.g., read-only, write-only, read and write, for different users).

Return value: If msgget() successfully creates or finds a mailbox, it gives you back a special number called the "message queue ID" (msqid). This msqid is what you'll use for all future operations with that specific mailbox. If something goes wrong (e.g., you try to create an exclusive mailbox that already exists), it returns -1.

Practical Examples:

1. Creating a new queue

int msqid = msgget(1234, IPC_CREAT | 0666);
This tries to create a new message queue with the address 1234. The 0666 means anyone can read and write to it. If it already exists, msgget() will just connect to it.

Creating an exclusive queue:
int msqid = msgget(5678, IPC_CREAT | IPC_EXCL | 0600);
This tries to create a new queue at address 5678. It will only succeed if no queue with that address exists. The 0600 means only the owner can read and write.

Connecting to an existing queue:
int msqid = msgget(1234, 0);
This tries to connect to the queue at address 1234. If it doesn't exist, it will fail (because IPC_CREAT wasn't used).

In essence, msgget() is the first step in setting up a communication channel using message queues.

2.Sending Messages to a Queue (msgsnd())

Syntax and parameters: 'int msqid', 'const void *msgp', 'size_t msgsz', 'int msgflg'.
The structure of the message pointer ('msgp').
Specifying the message size ('msgsz').
Flags: 'IPC_NOWAIT' (non-blocking send) and its implications.
Blocking behavior and potential for blocking indefinitely.

3. Receiving Messages from a Queue (msgrcv())

Syntax and parameters: 'int msqid', 'void *msgp', 'size_t msgsz', 'long msgtyp', 'int msgflg'.
The role of 'msgtyp' in selective reception:
'0': Receive the first message in the queue.
'> 0': Receive the first message of that specific type.
'< 0': Receive the first message with a type less than or equal to the absolute value of 'msgtyp'.
Setting the maximum receive size ('msgsz') and the 'MSG_NOERROR' flag.
Flags: 'IPC_NOWAIT' (non-blocking receive) and its implications.
Blocking behavior and waiting for specific message types.

4. Controlling Message Queues (msgctl())

Syntax and parameters: 'int msqid', 'int cmd', 'struct msqid_ds *buf'.
Detailed look at crucial commands:
'IPC_STAT': Retrieving the current status and attributes of the queue (using 'struct msqid_ds').
'IPC_SET': Modifying queue attributes (permissions, maximum message bytes).
'IPC_RMID': Marking the queue for removal (immediate or after all processes detach).
Understanding the 'struct msqid_ds' and its members.

7 Practical Implementation - Codes for the Producer Consumer App using Message Queues.

Instructions:
Below are producer and consumer side programs. Compile the different programs seperately before running then. Observe messages being sent by the producer.

7.1 producer.c File

```
cat > producer.c << EOF
#include <stdio.h>
#include <string.h>
```

```
#include <sys/msg.h>
#include <sys/ipc.h>
#include <unistd.h> // For sleep

    struct message  long mtype; char mtext[20]; ;
#define MSG_KEY 1234

    int main()
int qid = msgget(MSG_KEY, IPC_CREAT — 0666);
struct message msg = 1; // Message type 1
sprintf(msg.mtext, "Hello");
msgsnd(qid, &msg, strlen(msg.mtext) + 1, 0);
printf("Producer sent: %s", msg.mtext);
sleep(1);
sprintf(msg.mtext, "World");
msgsnd(qid, &msg, strlen(msg.mtext) + 1, 0);
printf("Producer sent: %s", msg.mtext);
return 0;

EOF
```

7.2 Consumer.c File

```
cat > consumer.c << EOF
#include <stdio.h>
#include <string.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <errno.h>

    struct message  long mtype; char mtext[20]; ;
#define MSG_KEY 1234

    int main()
int qid = msgget(MSG_KEY, 0);
struct message msg;
msgrcv(qid, &msg, sizeof(msg.mtext), 0, 0);
printf("Consumer received: %s", msg.mtext);
msgrcv(qid, msg, sizeof(msg.mtext), 0, 0);
printf("Consumer received: %s", msg.mtext);
msgctl(qid, IPC_RMID, NULL); // Remove queue
return 0;

EOF
```

8 Conclusion

This lecture notes have delved into Linux Message Queues, an effective System V IPC mechanism that facilitates asynchronous, decoupled communication among processes. These queues offer a dependable method for producers to transmit structured messages and for consumers to retrieve them, serving as a mediator buffer. Although they are conceptually straightforward, grasping functions such as msgget(), msgsnd(), msgrcv(), and msgctl() is essential for their successful implementation. Message queues play a crucial role in situations that require organized data exchange and task distribution between independent processes, positioning them as a key asset in the Linux programmer's IPC toolkit.

List of References

1. Silberschatz, A., & Galvin, P. B. (2013). Operatings System Concepts.
2. Stevens, W. R., & Rago, S. A. (2013). Advanced programming in the UNIX environment. Addison-Wesley.
3. Love, R. (2013). Linux system programming: talking directly to the kernel and C library. " O'Reilly Media, Inc.".
4. Kerrisk, M. (2010). The Linux programming interface: a Linux and UNIX system programming handbook. No Starch Press.