Introduction to POSIX Message Queue

In the world of inter-process communication, POSIX message queues play a vital role in enabling processes to exchange data in a reliable and efficient manner. These message queues provide a powerful mechanism for different processes to communicate with each other, allowing for the seamless transfer of messages and data. Unlike traditional message passing methods, POSIX message queues offer a robust and standardized approach, ensuring compatibility across various systems and architectures.

Developers utilize these message queues to implement communication patterns such as request-reply, publish-subscribe, and more, facilitating the development of complex and distributed systems. With a focus on reliability, message ordering, and message priority, POSIX message queues offer a versatile solution for diverse communication requirements in modern software development.

What is a message queue?

A message queue is a form of inter-process communication used in computer systems, allowing processes to communicate with each other without having to coordinate directly. It provides a way for processes to exchange data asynchronously, improving system resilience and enabling better scalability. Message queues are commonly used in distributed systems, where different components need to communicate efficiently and reliably.

One of the key benefits of using a message queue is that it allows for the decoupling of sender and receiver, meaning that the producer and consumer of data do not need to interact with each other in real time. This can lead to more flexible and robust systems, as components can operate independently and handle messages at their own pace.

Moreover, message queues can facilitate load balancing and provide a buffer for managing bursts of data, ensuring that the system remains responsive even during peak periods. By using a message queue, developers can design more modular and scalable architectures, enhancing the overall performance and reliability of the system.

Advantages of using POSIX message queue

1 Inter-process Communication

POSIX message queues provide a robust mechanism for inter-process communication, allowing different processes to exchange data **efficiently and reliably**. This is particularly useful in scenarios where multiple processes need to cooperate and communicate with each other in a **real-time application**. The SYS V message queue is designed for a **server application**.

2 Asynchronous Messaging

Another advantage is the support for asynchronous messaging. Processes can send and receive messages without having to be synchronized in time. This flexibility contributes to the overall responsiveness and performance of the system.

3 Prioritized Messaging

POSIX message queues also support prioritized messaging, where messages can be assigned different priorities. This feature is valuable when certain messages need to be processed with higher precedence, optimizing the system's efficiency.

4 Queue Persistence

Unlike other forms of inter-process communication, POSIX message queues provide queue persistence. This means that messages remain in the queue until they are explicitly consumed, making it possible to recover from system failures and ensuring reliable message delivery.

Differences between System V & POSIX MQ

1 Message Structure

POSIX message queues do not have any specific requirement of the message structure. However, System V message queues require the first element of the structure to be the type of message and it needs to be long integer.

2 Command Line Interface

No standard commands can be used for managing the POSIX IPC objects. For System V objects, there are specific commands, such as ipcs, lsipc and ipcrm for listing and deleting a IPC object.

3 Object Identification

POSIX IPC uses names instead of keys in System V IPC. This provides a consistent method with the traditional UNIX file model.

4 Portability

POSIX IPC is supported by some UNIX and Linux implementations. However, System V IPC is supported by most of UNIX and Linux implementations.

Differences between System V & POSIX MQ

5 Message Handling

a POSIX message can be sent with a priority as an argument in the “mq_send” function instead of selecting type of messages with the “msgrcv” function of System V message queues. Message handling decision is made on the sender for POSIX message queues, not on the receiver for System V message queues.

6 Read Operation

Read operation from a POSIX message queue will return the oldest message with the highest priority. However, a System V message queue will return a message with any desired priority.

7 Time Limit Operation

For POSIX message queues, the timeout can be set for sending and receiving a message. This feature is not supported in System V message queues.

Creating a message queue

In the world of inter-process communication, message queues play a vital role in enabling seamless interaction between different processes. A message queue is a form of communication mechanism that allows processes to exchange data in the form of messages, providing a robust and reliable means of communication.

When creating a message queue, it is essential to consider various aspects such as the size of the queue, the permissions required, and the naming conventions. Each of these factors contributes to the overall functionality and usability of the message queue, ensuring that it meets the specific communication requirements of the processes involved.

Additionally, establishing a message queue involves setting up the underlying infrastructure, defining the message format, and implementing error handling mechanisms to ensure the integrity and consistency of the data being exchanged.

Furthermore, understanding the intricacies of message queue creation empowers developers to design efficient and scalable communication solutions, laying the groundwork for seamless inter-process communication within diverse computing environments.

As developers navigate the process of creating a message queue, they delve into the intricacies of inter-process communication, paving the way for enhanced collaboration and data exchange between different components of a system.

Query for image: "abstract digital representation of interconnected data nodes, vibrant neon lighting, futuristic and dynamic vibe"

MQ Open System Call

The “mq_open” system call is used to create or open the POSIX message queue.

```
#include <fcntl.h>          /* For O_* constants */
#include <sys/stat.h>       /* For mode constants */
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
```

The “name” parameter is the identifier of the message queue that is required to start with ‘/‘ followed by characters.
The “oflag” parameter specifies the operation for the message queue. It must be one of the followings:

- O_RDONLY for receiving messages only.
- O_WRONLY for sending messages only.
- O_RDWR for both sending and receiving messages.
- Additional flags can be combined with the above flags;
- such as, O_CREAT for Creating the message queue if it does not exist.

The “mode” parameter specifies the permissions on the new message queue if the O_CREAT is used.
The “attr” parameter specifies the attributes of the new message queue if the O_CREAT is used.

Message Queue Attributes

The structure of message queue attributes is defined as follows:

```
struct mq_attr {
    long mq_flags;       /* Flags  0 or O_NONBLOCK (ignored for mq_open()) */
    long mq_maxmsg;      /* Max. # of messages on queue */
    long mq_msgsize;     /* Max. message size (bytes) */
    long mq_curmsgs;     /* # of messages currently in queue (ignored for mq_open()) */
};
```

Only the mq_maxmsg and mq_msgsize fields are employed when calling mq_open();
the values in the remaining fields are ignored. If attr is NULL, then the queue is created
with implementation-defined default attributes.

Sending messages to a message queue

When it comes to sending messages to a message queue, it's crucial to ensure that the data is structured correctly and that the queue is open to receive messages. The process involves assemble the data into a format that can be transmitted, and then finally deploying the message into the queue.

It's important to handle any potential errors that may occur during the sending process, such as a full queue.

MQ Send System Call

The “mq_send” system call is used to send a message to a POSIX message queue.

#include <mqueue.h>

int mq_send(mqd_t mqdes, const char msg_ptr[ ], size_t msg_len, unsigned int msg_prio);

The “mqdes” parameter is the descriptor of the message queue.
The “msg_ptr” parameter is the pointer to a message to be sent.
The “msg_len” parameter is the size of a message.
The “msg_prio” parameter specifies the priority of a message. It must be a non-negative integer.

MQ Send System Call with a timeout

The “mq_timedsend” system call is used to send a message to a POSIX message queue with a timeout.

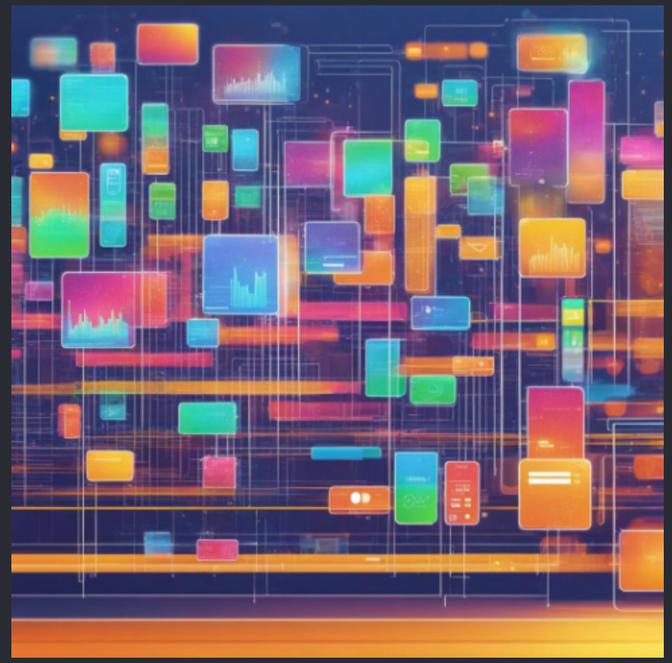
#include <time.h>
#include <mqueue.h>

int mq_timedsend(mqd_t mqdes, const char msg_ptr[ ], size_t msg_len, unsigned int msg_prio, const struct timespec ***abs_timeout);**

The “mqdes” parameter is the descriptor of the message queue.
The “msg_ptr” parameter is the pointer to a message to be sent.
The “msg_len” parameter is the size of a message.
The “msg_prio” parameter specifies the priority of a message. It must be a non-negative integer.
The “abs_timeout” parameter specifies how long the call will be blocked if the message queue is full.

Receiving messages from a message queue

When receiving messages from a POSIX message queue, it is essential to understand the process and mechanisms involved. After successfully creating a message queue and sending messages to it, the receiving end needs to be equipped to handle and process the incoming messages effectively. The receiver should be designed to handle the messages based on the intended use case, ensuring that the data is processed accurately and efficiently. It's also crucial to consider error handling and message queue attributes to ensure seamless reception and processing of messages.



Utilizing the POSIX message queue for receiving messages enables reliable communication between different processes, making it a powerful tool for inter-process communication. Receivers can implement strategies to prioritize certain messages, and ensure that no message is lost or overlooked. By fully understanding the intricacies of receiving messages from a message queue, developers can create robust and responsive systems that leverage the capabilities and advantages of POSIX message queues.

MQ Receive System Call

The “mq_receive” system call is used to receive a message from a POSIX message queue.

#include <mqueue.h>

int mq_receive(mqd_t mqdes, const char msg_ptr[ ], size_t msg_len, unsigned int *msg_prio);

The “mqdes” parameter is the descriptor of the message queue.
The “msg_ptr” parameter is the pointer to a message to be sent.
The “msg_len” parameter is the size of a message.
The “msg_prio” parameter specifies the pointer to a valid space for keeping the priority of the received message.

MQ Receive System Call with a timeout

The “mq_timedreceive” system call is used to receive a message from a POSIX message queue with a timeout.

#include <time.h>
#include <mqueue.h>

int mq_timedreceive(mqd_t mqdes, const char msg_ptr[ ], size_t msg_len, unsigned int *msg_prio, const struct timespec ***abs_timeout);**

The “mqdes” parameter is the descriptor of the message queue.
The “msg_ptr” parameter is the pointer to a message to be sent.
The “msg_len” parameter is the size of a message.
The “msg_prio” parameter specifies the pointer to a valid space for keeping the priority of the received message.
The “abs_timeout” parameter specifies how long the call will be blocked if the message queue is empty.

Message queue limits and restrictions

POSIX Message Queue Limits	POSIX message queues have system-imposed limits on the maximum number of queues, maximum message size, and maximum message priority. These limits are defined by constants like MQ_OPEN_MAX, MQ_PRIO_MAX, and MQ_MSGSIZE_MAX. It's important to be aware of these limits when designing and implementing systems that rely on POSIX message queues.
Message Queue Permissions	POSIX message queues are subject to file system permissions, making it crucial to understand and set appropriate permission settings to control access to the message queues. This includes permissions for reading, writing, and changing the queue's ownership.
Message Queue Restrictions	There are certain restrictions on message queue names, including length limitations and naming conventions. Additionally, message queue attributes such as the maximum number of messages and the maximum message size can impose constraints on the system.

POSIX message queue - Example

```
// Sender.c
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

#define Q_NAME "/my_mq"
#define Q_PERM 0660
#define MAX_MSG 3
#define MAX_MSG_SIZE 256
#define BUF_SIZE 64

struct msgs
{
  int written;
  char data[BUF_SIZE];
};
```

```
int main(void)
{
  int running = 1;
  struct msgs a_msg;
  struct mq_attr q_attr;
  mqd_t mqd;
  unsigned int priority = 0;
  char buffer[BUF_SIZE], *s;

  q_attr.mq_maxmsg = MAX_MSG;
  q_attr.mq_msgsize = MAX_MSG_SIZE;

  mqd = mq_open(Q_NAME, O_WRONLY | O_CREAT, Q_PERM, &q_attr);
  if (mqd == -1)
  {
    perror("mq_open failed ");
    exit(EXIT_FAILURE);
  }
```

```
  while (running)
  {
    printf("Enter data: ");
    s = fgets(buffer, BUF_SIZE, stdin);
    printf("Size of msgs: %lu\n", sizeof(struct msgs));
    strcpy(a_msg.data, buffer);
    if (mq_send(mqd, (void *)&a_msg, sizeof(struct msgs), priority) == -1)
    {
      perror("mq_send failed ");
      exit(EXIT_FAILURE);
    }
    if (strncmp(buffer, "end", 3) == 0)
      running = 0;
  }

  mq_close(mqd);
  exit(EXIT_SUCCESS);
}
```

POSIX message queue - Example (Cont.)

```
// Receiver.c
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

#define Q_NAME "/my_mq"
#define Q_PERM 0660
#define MAX_MSG 3
#define MAX_MSG_SIZE 256
#define BUF_SIZE 64

struct msgs
{
  int written;
  char data[BUF_SIZE];
};
```

```
int main(void)
{
  int running = 1;
  struct msgs a_msg;
  struct mq_attr q_attr;
  mqd_t mqd;
  unsigned int priority = 0;

  q_attr.mq_maxmsg = MAX_MSG;
  q_attr.mq_msgsize = MAX_MSG_SIZE;

  mqd = mq_open(Q_NAME, O_RDONLY | O_CREAT, Q_PERM, &q_attr);
  if (mqd == -1)
  {
    perror("mq_open failed ");
    exit(EXIT_FAILURE);
  }
```

```
while (running)
{
  if (mq_receive(mqd, (void *)&a_msg, MAX_MSG_SIZE, &priority) == -1)
  {
    perror("mq_receive failed ");
    exit(EXIT_FAILURE);
  }
  printf("Received Message: %s", a_msg.data);
  if (strncmp(a_msg.data, "end", 3) == 0)
    running = 0;
}
mq_close(mqd);
if (mq_unlink(Q_NAME) == -1)
{
  perror("mq_unlink failed ");
  exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);
}
```

POSIX message queue with a timeout - Example

```
// Sender.c
#include <time.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

#define Q_NAME "/my_mq"
#define Q_PERM 0660
#define MAX_MSG 3
#define MAX_MSG_SIZE 256
#define BUF_SIZE 64

struct msgs
{
  int written;
  char data[BUF_SIZE];
};
```

```
int main(void)
{
  int running = 1;
  struct msgs a_msg;
  struct mq_attr q_attr;
  mqd_t mqd;
  unsigned int priority = 0;
  char buffer[BUF_SIZE], *s;
  struct timespec to;

  q_attr.mq_maxmsg = MAX_MSG;
  q_attr.mq_msgsize = MAX_MSG_SIZE;

  mqd = mq_open(Q_NAME, O_WRONLY | O_CREAT, Q_PERM, &q_attr);
  if (mqd == -1)
  {
    perror("mq_open failed ");
    exit(EXIT_FAILURE);
  }
```

```
while (running)
{
  printf("Enter data: ");
  s = fgets(buffer, BUF_SIZE, stdin);
  printf("Size of msgs: %lu\n", sizeof(struct msgs));
  strcpy(a_msg.data, buffer);
  clock_gettime(CLOCK_REALTIME, &to);
  to.tv_sec += 5;
  if (mq_timedsend(mqd, (void *)&a_msg, sizeof(struct msgs), priority, &to) == -1)
  {
    perror("mq_timedsend failed ");
    //exit(EXIT_FAILURE);
  }
  if (strncmp(buffer, "end", 3) == 0)
    running = 0;
}

mq_close(mqd);
exit(EXIT_SUCCESS);
}
```

POSIX message queue with a timeout - Example (Cont.)

```
// Receiver.c
#include <time.h>
#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

#define Q_NAME "/my_mq"
#define Q_PERM 0660
#define MAX_MSG 3
#define MAX_MSG_SIZE 256
#define BUF_SIZE 64

struct msgs
{
  int written;
  char data[BUF_SIZE];
};
```

```
int main(void)
{
  int running = 1;
  struct msgs a_msg;
  struct mq_attr q_attr;
  mqd_t mqd;
  unsigned int priority = 0;
  struct timespec to;

  q_attr.mq_maxmsg = MAX_MSG;
  q_attr.mq_msgsize = MAX_MSG_SIZE;

  mqd = mq_open(Q_NAME, O_RDONLY | O_CREAT, Q_PERM, &q_attr);
  if (mqd == -1)
  {
    perror("mq_open failed ");
    exit(EXIT_FAILURE);
  }
```

```
while (running)
{
  clock_gettime(CLOCK_REALTIME, &to);
  to.tv_sec += 15;
  if (mq_timedreceive(mqd, (void *)&a_msg, MAX_MSG_SIZE, &priority, &to) == -1)
  {
    perror("mq_timedreceive failed ");
    //exit(EXIT_FAILURE);
  }
  printf("Received Message: %s", a_msg.data);
  if (strncmp(a_msg.data, "end", 3) == 0)
    running = 0;
}
mq_close(mqd);
if (mq_unlink(Q_NAME) == -1)
{
  perror("mq_unlink failed ");
  exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);
}
```

Conclusion and Key Takeaways

In conclusion, POSIX message queues provide a reliable inter-process communication mechanism in Unix-like operating systems. They offer a convenient way for processes to exchange data and communicate asynchronously. By utilizing message queues, developers can design robust and scalable applications that can handle complex workflows efficiently.