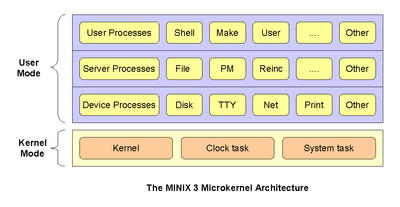
**Design Document**

In this Project we have developed new System calls to send messages from on application to another . The Process manager of the Minix acts like a mailbox i.e the queue where the messages are stored and the queue is in the PM server.



The sender application sends the message to addressed to the receiver to the pm server. The Process Manager stores the message and notifies the receiver if it has requested for notification and delivers the message to the receiver when the receiver requests for a message.

sys\_datacopy is the inbuilt function of Minix that is used to transfer data from the user application to the PM Server and the vice versa and this function is very useful in implementing the message passing between user application.

Some of the key features implemented as part of this Project are:

* MQ\_SEND System call to pass the message from sender application to the receiver application (Intermediate step : stored in the PM server).
* MQ\_RECEIVE System call to pass the message from the PM server to the receiver application.
* MQ\_REQNOTIFY System call to request for notification if a message has been sent to queue by the sender to the receiver who has requested for notification.
* MQ\_OPEN System call to open a queue on the PM Server so that the user application can store or retrieve the messages .
* MQ\_CLOSE System call to close a queue on the PM server so that the user application which closes it cannot access it further until it is reopened again .
* MQ\_SETATTR System call to set the attributes of the queue such as the maximum number of messages that can be stored in the queue .
* MQ\_GETATTR System call to get the attributes of the queue such as the maximum number of messages that can be stored in the queue .
* Priority of message to be delivered : the receiver can request for a message of a particular priority or when no priority value is provided the message with highest priority is retrieved. If two messages of the same priority are present in the queue for the receiver the first message is retrieved by the receiver.

**System Call API’s**

**1. mq\_close**

**syscall(PM\_PROC\_NR, 44, &m)**

This system call can be called using the API :

mq\_close(mqd\_t mqd)

Parameters:

mqd -> is the queue id which is to be closed

Return Value int :1 if the queue is closed successfully -1 if there was some error

2. **mq\_open**

**\_syscall(PM\_PROC\_NR, 44, &m)**

This system call can be called using the API:

mq\_open(const char \*name, int read\_write\_flag, int blocking\_flag, int max\_messages)

Parameters:

name -> name of the queue that is to be opened

read\_write\_flag-> flag whether the application wants to read or write or do both on the queue

max\_messages -> maximum number of messages that is allowed in the queue.

Return value int : queue id

3 **mq\_setattr**

**\_syscall(PM\_PROC\_NR, 56, &m)**

This system call can be called using the function

mq\_setattr(mqd\_t mqd, mq\_attr\_t \*mq\_attr)

Parameters

mqd id the queue id whose parameters are to be set

mq\_attr is an instance of the structure mq\_attr\_t which has queue attributes such maximum

messages , send blocking , receive blocking

1. **mq\_getattr**

**\_syscall(PM\_PROC\_NR, 58, &m)**

This system call can be called using the function

**mq\_getattr(mqd\_t mqd, mq\_attr\_t \*mq\_attr)**

Parameters:

mqd -> id of queue whose parameter are to be set

mq\_attr -> is the structure with null values passed to the system call and after the system call the queue parameters can be retrieved from this.

Return int:

1. **mq\_send**

**\_syscall(PM\_PROC\_NR, 69, &m)**

This system call can be called using the function

mq\_send(mqd\_t mqd, message\_t \*data, size\_t message\_length, unsigned int priority)

Parameters:

mqd id of the queue where the message is to be populated

data -> message to be sent

message\_length – message length

priority -> priority of the message to be sent it can be

1 – highest priority

2- medium priority

3-low priority

1. **mq\_receive**

**\_syscall(PM\_PROC\_NR, 70, &m)**

The system call can be called using the function

mq\_receive(mqd\_t mqd, size\_t buffer\_length, char \*buffer, unsigned int priority)

Parameters

mqd- queue id

buffer\_length -> length of the message to be retrieved

buffer-> character pointer to location of buffer where the retrieved message is stored.

1. **mq\_reqnotify**

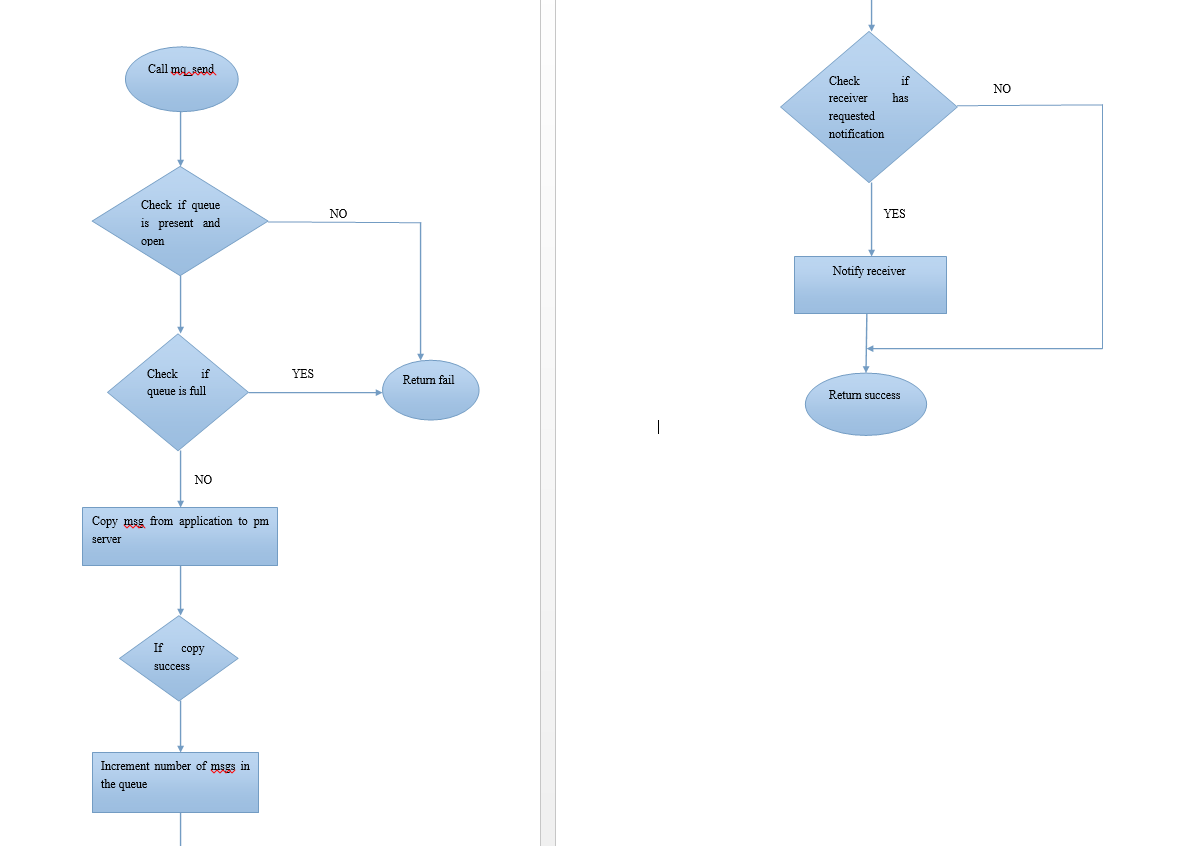
**\_syscall(PM\_PROC\_NR, 79, &m)**

The system call can be called by the function

mq\_reqnotify(int signum)

Flow Diagram of the System call Implementation

**MQ\_SEND**

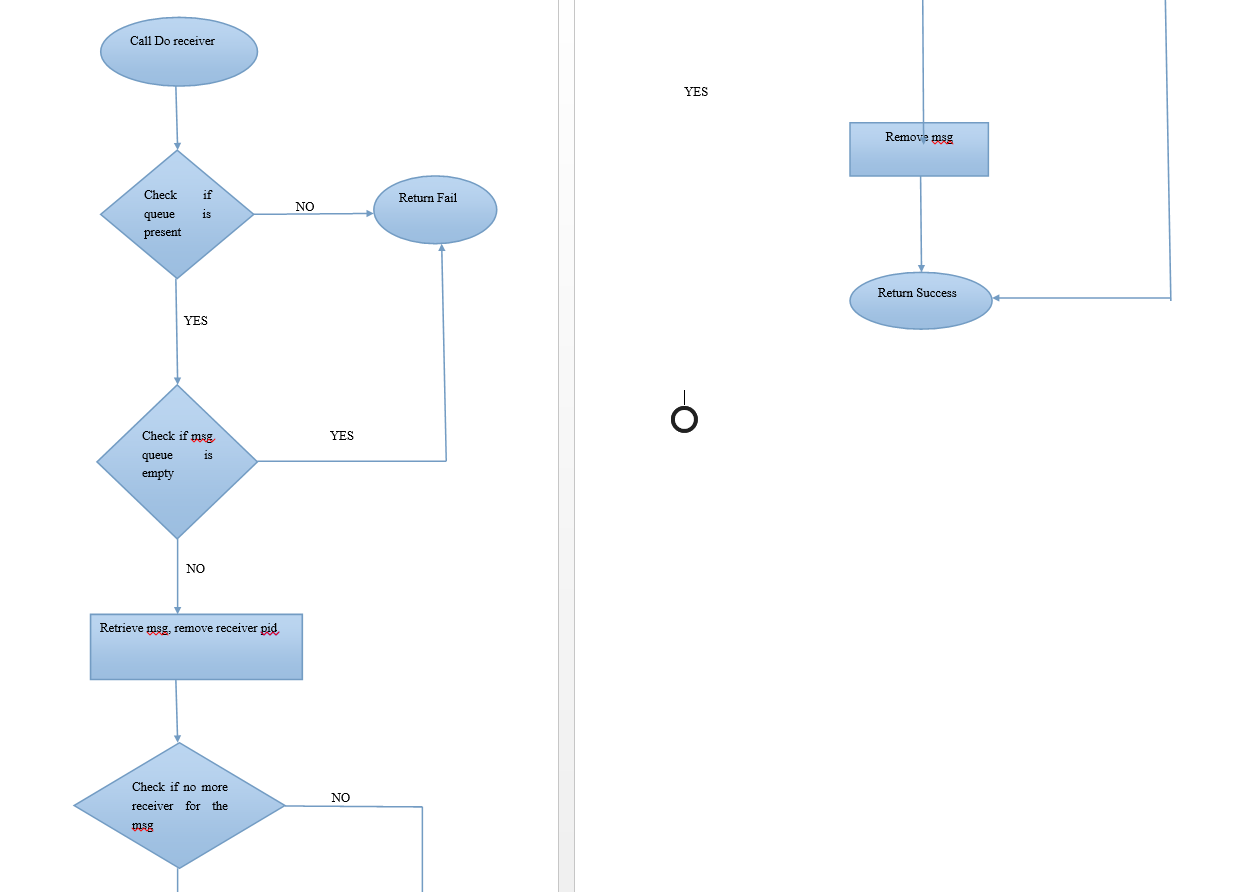


Exception handling :

-> Checking if queue is Present if not throw an error message

-> Checking if queue already full if yes throwing error message.

**MQ\_RECEIVE**

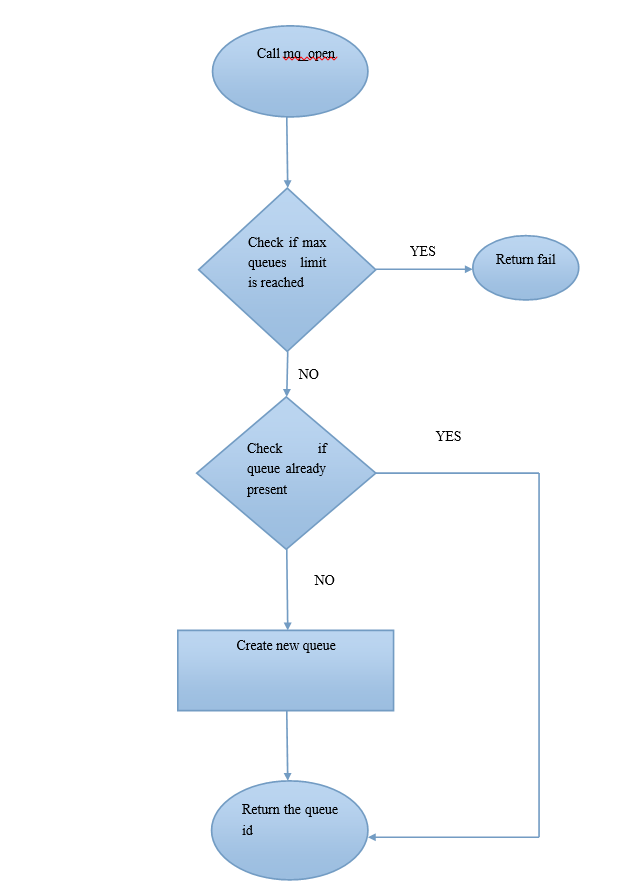


Exception handling

Checking if queue is present while trying to retrieve message from a queue, if not throw an error

Checking if there are no more receivers for a particular message in before removing it from the queue.

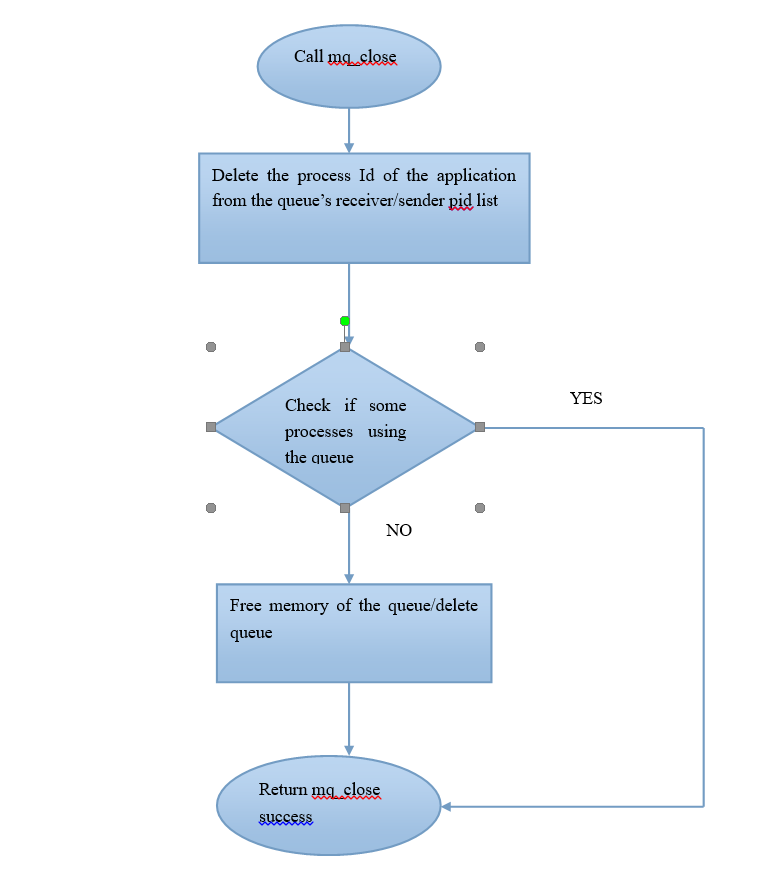
**MQ\_OPEN**



Exception Handling

Check if the maximum number of allowed queues has reached before it is created.

**MQ\_CLOSE**



Exception Handling:

Before freeing up a queue , a check is made if the queue has no receiver or senders using it.

**Deadlock Handling**

This Project is designed as non blocking send and a non blocking receive i.e the sender does not wait for an acknowledgement from the receiver and the receiver does not wait for a message to be sent for it. If there is no message in the queue for a particular receiver then the receiver requests for a message then it is just given an error message stating that there are no messages for it at that time.

Also there is an option to request for notification by the receiver. The notify can be used by receiver to initiate the request for retrieval of message from the queue thus making the receiver the aware of all its incoming messages although it can retrieve it at any time.

Since both send and receive are non blocking the chances of deadlock is minimized. There is no circular wait situation created i.e sender waiting for receiver’s acknowledgement and receiver waiting for the message.

**Sections of code which are considered critical**

* Writing of the messages to and from the queue

Mutex Flags are used to protect these critical regions

* Removing of messages which have 0 number of receivers (removing of messages should not happen at the same time as writing of messages so as to avoid ambiguity in current number of messages in queue).

**Resource management**

Freeing of memory of queues when user wants to remove the queue.