**PROJECT 2: MAILBOX IPC**

**GROUP 4**

GEORGE J PALATHINGAL (A20294134)

REKHA BALASUBRAMANIAN (A20274931)

SHILPA RAJAPPA (A20261273)

# Introduction

This document gives the design details of the Mailbox IPC Implementation for a user process.

# Requirements

Design and implement a set of MINIX IPC system calls with the following features:

1) Allow a sender user process to “deposit” a message to a mailbox.

2) Allow a receiver user process to “retrieve” a message from a mailbox.

3) Provide a set of administrative and security related system calls.

4) Administrative system calls will allow an user process to setup and remove a mailbox and to recover from exception conditions.

5) Security system calls will allow the user process that sets up the mailbox to specify the capabilities of other user processes with respect to the mailbox.

The following conditions have to be met while designing the system calls:

1) “Deposit” system call allows the sender to specify the list of receivers and blocks the sender when the mailbox is full.

2) “Retrieve” system call allows the receiver to read messages from the mailbox. Message should be removed from the mailbox after it has been read by all of the receivers.

3) Messages from common senders should be received in the same order.

4) System should support atleast 5 mailboxes at the same time

# High Level Design

We have implemented the necessary system calls under the Process Management (PM) module [/usr/src/servers/pm]. Our wrapper functions are implemented in the /usr/src/lib/libc/sys-minix module. The data structures are maintained in the PM module. The main data structures used in our project are:

* structure for the communicating nodes i.e. sender and receiver maintained as a list of communicating nodes
* structure for the messages i.e. a queue for messages
* structure for the mailbox i.e. a list of mailboxes.

Each communicating node is identified by its process-id. Messages are associated with a sender and list of receivers. Mailboxes have their unique ids and are associated with a list of senders and a list of receivers.

The architecture of the mailbox IPC is as shown below:

User Process 1

User Process

2

Process Manager

grsMailbox system call handlers

do\_deposit & do\_retrieve

User Process

3

User Process

4

Retrieve call

Deposit call

Retrieve call

Deposit call

access mailbox and associated kernel data structures

Fig: Mailbox IPC architecture

In order to satisfy the requirements, we have implemented 5 system calls in the Process Management Module:

1) openMailBox

2) closeMailBox

3) deposit

4) Retrieve

5) getReceiverList

Each if these system calls and their associated data structures are discussed in the following section

# Detailed Design

In this section we discuss the design specifications for:

a) Data Structures.

b) System Calls.

c) List of Error Codes Used

## Data Structure Design

All the different structures used in this project are maintained in the form of a linked list data structure under Process Management itself.

Following are the different structured types used in this project:

#### Sender/Receiver node

Each sender or receiver process is identified by the below structure. Each node structure maintains the process id and a pointer to the next node.

struct node

{

pid\_t \_pid;

struct node \*next;

};

typedef struct node NODE;

#### Mailbox message

Each message structure maintains the sender node, intended receiver node list, message and a pointer to next message in the mailbox.

struct message

{

NODE \_sender;

NODE \*\_receiverList;

char \_text[MAX\_MESSAGE\_LENGTH];

struct message \*next;

};

typedef struct message MESSAGE;

#### Mailbox

The mailbox structure contains a linked list of allowed senders, allowed receivers, owner of the mailbox, number of messages in the mailbox, pointer to the message structure and pointer to the next mailbox. Each mailbox is represented by a unique integer id and a mailbox name.

struct mailbox

{

int \_id;

char \_name[MAX\_MAILBOX\_NAME\_LENGTH];

NODE \*\_owner;

NODE \*\_senderList;

NODE \*\_receiverList;

MESSAGE \*\_messageList;

int \_messageCount;

struct mailbox \*next;

};

typedef struct mailbox MAILBOX;

## System Call Design

#### 1) openMailBox

|  |  |
| --- | --- |
| System Call | openMailBox |
| Prototype | int openMailBox(char\* , int , pid\_t); |
| Purpose | To create a mailbox and to open a mailbox |
| Return Value | Returns a mailbox ID on SUCCESS. Returns negative value on FAILURE. |
| Exceptions | 1. Mailbox is not present and flag is not set to create the mailbox.  2. Same mailbox is opened twice by the same process. |
| Error Code | -1: ERROR\_MAILBOX\_NOT\_PRESENT  -2: ERROR\_ILLEGAL\_OPENMAILBOX\_FLAG |
| Description | a) Lets the user process to open a mail box with mailbox name as argument  b) If the mailbox is not present   * user process will be able to create a mailbox upon specifying the O\_CREAT flag. This process becomes the owner of the mailbox. * if no O\_CREAT flag is specified, system calls returns an error message saying cannot create mailbox.   c) If the mailbox is present   * openMailBox will add the current user process to the mailbox sender’s list, if the flag is mentioned as O\_WRONLY. * openMailBox will add the current user process to the mailbox receiver’s list, if the flag is mentioned as O\_RDONLY. |
| Design Discussion | Designing the system call in this manner will ensure that:   * user process can setup a mailbox in a simple manner and own the mailbox * if mailbox exists user process can just operate on the mailbox. * access control list with respect to the mailbox are provided in the form of READ and WRITE flags. * capability list for the mailbox is managed in the mailbox structure itself. i.e. mailbox keeps track of who are the senders and who are the receivers |

#### 2) closeMailBox

|  |  |
| --- | --- |
| System Call | closeMailBox |
| Prototype | int closeMailBox(int , pid\_t); |
| Purpose | To close a mailbox and to delete a mailbox |
| Return Value | Returns 1 if a mailbox or any node in mailbox is deleted. Returns 0  or negative value otherwise. |
| Exceptions | 1. A mailbox does not exist for the id that is passed as argument.  2. closeMailBox is called by a process which has not opened it. |
| Error Code | -1: ERROR\_MAILBOX\_NOT\_PRESENT |
| Description | a) Removes the user process from the receiver’s and sender’s  list.  b) If the called process is the owner of the mailbox, delete the  mailbox itself.  c) Return error if the mailbox does not exist. |
| Design Discussion | Designing the system call in this manner will ensure that:   * user process is de-linked with the mailbox. * Mailbox structure is updated by removing the calling user process’ id from the sender’s or receiver’s list based on whether the user process has deposited or retrieved * Mailbox is deleted when all the user processes are de-linked from the mailbox |

#### 3) deposit

|  |  |
| --- | --- |
| System Call | deposit |
| Prototype | int depositMessage(int mailboxId, const char \*msg\_text, pid\_t receiverList[]); |
| Purpose | To deposit a message into the mailbox |
| Return Value | returns 0 on success, -1 on failure |
| Exceptions | * mailbox specified by the mailbox id does not exist * user process does not have the permission to deposit msg to the mailbox |
| Error Code | EMSGSIZE, EINVAL, EMSGEMPTY, EINVALIDSENDER |
| Description | a) Lets the user process to deposit a message into the mailbox.  b) Message is deposited into the mailbox. Mailbox is specified by the *mailboxId*, obtained from the openMailBox.  c) If a user process calls a deposit , but has only read permissions on the mailbox, an error is returned  d) Multiple receivers can be specified in the deposit system call  e) deposit blocks when the mailbox is full. |
| Design Discussion | Designing the system call in this manner will ensure that:   * deposit system call works correctly only for processes with correct permissions. * capability for the mailbox is being handled in this scenario. |

#### 3) Retrieve

|  |  |
| --- | --- |
| System Call | retrieve |
| Prototype | int retrieveMessage(int mailboxid, char \*resposenmessage) |
| Purpose | Retrieves the mail from mailbox |
| Return Value | Response message on SUCCESS, NULL on FAILURE |
| Exceptions | * mailbox is not available * receiver process does not have the access rights |
| Error Code | -1 : MAILBOX\_NO\_AVAILABLE  -3 : ILLEGAL\_MAILBOX\_ACCESS |
| Description | 1. The user process can retrieve a mail from the required mailbox by providing the mailbox id 2. The user access permission is verified before accessing the mailbox 3. The messages are retrieved in the FIFO manner 4. The receiver reference is removed from the mail message 5. If this was the last intended receiver the message will be deleted from the mailbox 6. Non-blocking system call |
| Design Discussion | Designing the system call in this manner will ensure that:   * The mailbox referred by the user exists and the also the user has read access privilege to the mailbox * Further the call verifies if the receiver is any intended recipient for the mail * The FIFO data structure maintained ensures that all receivers receive the message in the same order that were sent. * The timely removal of the receiver id from the mail ensures that any stale or outdated mails/messages are removed from the mailbox. * This is a non-blocking system call hence the receiver need not be blocked in case of no mail available. This ensures the operation continues to proceed even in the absence. Since the messages communicated here have no decision condition on the receiver, there is no need for blocking the process. |

#### 3) getRecieverList

|  |  |
| --- | --- |
| System Call | getRecieverList |
| Prototype | int getReceiverList(int mailboxId, int receiverList[]); |
| Purpose | Fetch the list of receivers registered for a given mailbox. |
| Return Value | returns the list of receivers on success, NULL on failure |
| Exceptions | * receiver list for the mailbox is empty * invalid mailbox id |
| Error Code | ERROR\_MAILBOX\_NOT\_PRESENT |
| Description | a) Fetches the list of registered receivers for the mailbox  b) Using the list of receivers returned by this sytem call, the user process can choose a subset of receivers, to send the deposit message. |
| Design Discussion | Designing the system call in this manner will ensure that:   * That the user before can get the list of valid receivers for the mailbox performing a deposit system call. * This system can be extended to incorporate other features in the future |

## Error Codes:

The below table lists the different error codes reflected in the GRS Mailbox:

|  |  |
| --- | --- |
| Error Code | Description |
| -1 | ERROR\_MAILBOX\_NOT\_PRESENT |
| -2 | ERROR\_ILLEGAL\_OPENMAILBOX\_FLAG |
| -3 | ERROR\_ILLEGAL\_RECEIVE\_FLAG |
|  |  |

# Design Considerations

During the course of our design, the following details were taken into consideration:

*Blocking deposit system call*

The deposit system call was made as a blocking system call. When the mailbox gets full, the deposit system call waits until the mailbox has space to accommodate new messages. The internal structure of the mailbox maintains a queue of messages. Messages are deposited from the front of the queue. Once the message is deposited, the sender list is updated with the depositer’s pid. Having the deposit system call blocking on mailbox full ensures that there is no occurrence of lost mail and the sender can be sure of the mail being deposited in mailbox.

*Non-blocking retrieve system call*

The retrieve system call is implemented as a non-blocking system call. The main reason for making retrieve non-blocking is that it ensures that there are no deadlock occurrences. When the retrieve system call is made, and if the mailbox is empty, then the retrieve system call returns with an error message instead of blocking itself. Thus ensuring there is no indefinite wait on the mailbox, enhancing the security aspect of the mailbox ipc. Also considering the message read activity it is not required for the receiver to wait indefinitely until any message has reached the mailbox. The receiver can try to retrieve message again after sometime.

*Choice of data structure*

We have designed our data structures such that each structure distinctly identifies the elements used in the mailbox ipc. We have used “struct” type for this implementation. The ‘node’ structure uniquely identifies each communicating node through its pid and is implemented as a linked list of nodes.

The ‘message’ structure (we have done a typedef to avoid conflicts with the kernel message structure) identifies a message and associates each message with its sender, linked list of receivers, and is implemented as linked list of messages.

The ‘mailbox’ structure identifies the mailbox in the system. Each mailbox has a unique mailbox id, mailbox name, list of messages in the mailbox and the list of senders and receivers who can access the mailbox. Capability and access control lists are achieved through this design. Two type of access is defined for the mailboxes which is READ and WRITE. Hence a process that has the READ access to the mailbox is listed in the receiver list and the process having WRITE access to the mailbox is listed in the sender list. For a READ\_WRITE access to mailbox the process would be independently listed in both the sender and the receiver list.

*Choosing the linked list pattern of storing data*

By using a linked list instead of any static data structure, it is easier and simpler to insert messages into the mailbox in a FIFO (First In First Out) manner. Also, deletion of messages and mailboxes would be faster, since there is no movement of data. Linked lists are implemented using dynamic memory allocation. The FIFO approach for maintaining the messages in the mailbox ensures that all receivers of the message will read the messages in the same sequence.

*Security aspects of mailbox and messages:*

Each mailbox has a defined set of senders and revivers through which the READ and WRITE access is controlled. The deposit system call ensures to check the WRITE access of the user process before adding the message to mailbox. Similarly the retrieve system call verifies at the first level if the user process has access to read the messages from the specified mailbox.

At a per message level access control the sender process of the message needs to specify the intended recipients of the message. This ensures the message level security of any unauthorised read access to the message. Hence at the retrieve system call it is verified if the requesting process is one of the intended recipient for the message. Only if true the message is returned else an appropriate error code indicating illegal access is sent in response.